Reflective Embedding of Domain-Specific Languages

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

vorgelegt von

Dipl.-Inform. Tom Dinkelaker

geboren in Frankfurt am Main, Deutschland

Referent:	Prof. Dr.-Ing. Mira Mezini
Korreferent:	Prof. Dr.-Ing. Robert Hirschfeld
Datum der Einreichung:	26.11.2010
Datum der mündlichen Prüfung:	10.12.2010

Erscheinungsjahr 2011

Darmstadt D17

Abstract

In the recent years, there is an increasing interest for new programming languages that are special for certain problem domains – so-called *domain-specific languages* [MHS05]. A *domain-specific language* (DSL) provides a syntax and semantics for a particular class of problems. When writing code in a DSL, its concrete syntax enables higher end-user productivity [MHS05, KLP$^+$08] because the syntax is closer to the problem domain than the syntax of a *general-purpose language* (GPL). However, implementing a new programming language is expensive, because language developers need to design and develop a new syntax and semantics. Because of these high initial costs, stakeholders desist from implementing a new DSL from scratch.

To lower the initial costs of developing a DSL, there is the idea to *embed* a language into an existing language [Lan66, Hud96, Hud98, Kam98]. The existing language—often a general-purpose language—acts as a *host language* for the *guest language* that is embedded into the host as a library. Such an embedded language reuses the general-purpose features of the host language. Consequently, this reduces the development costs of the embedded language by reusing these features and by reusing the existing host language infrastructure [Hud98, Kam98, KLP$^+$08], i.e., its development tools, parser, compiler, and/or virtual machine. Furthermore, new language features can be incrementally added to the embedded language by "simply" extending its library [Hud96]. Indeed, there is a comparative study [KLP$^+$08] that shows that embedding requires significant lower investments to implement a new language compared to traditional non-embedded approaches.

However, existing language embedding approaches have several important limitations w.r.t. the features that are usually supported by non-embedded approaches. First and foremost, most embedding approaches and host languages lack support for designing a concrete syntax for an embedded language. Second, current approaches do not allow embedding special language semantics that the embedded language needs but that conflict the host semantics. Third, there is no support for embedding multiple languages that have interactions in their syntax and semantics. The lack of support for these features in current embedding approaches mostly stem from their restrictions to adapt constructs in embedded languages and host languages.

To address these limitations, this thesis proposes the *reflective embedding architecture* that embeds languages into reflective host languages. A *reflective language* is a special programming language with support for *reflection* that enables programs to reason about their structure and behavior. Specifically, there is support for *introspection* that allows analyzing programs, and support for *intercession* that allows transforming programs. Using reflection helps overcoming current limitations of language embedding, since it enables evolution of embedded languages by analyzing and transforming their libraries.

The reflective embedding architecture itself defines a process how to embed a language's syntax and semantics as a set of artifacts. Every language is an encapsulated component that has a set of well-defined interfaces. The classes of the embedding implement these interfaces to define the details of the execution semantics for that language component. In addition, the architecture enables embedding of language compositions, scoping strategies, domain-specific analyses and transformations. What is special in the architecture is that every language component implicitly has a meta level. When necessary, the language developers can use this meta level to evolve embeddings, which enables a number of benefits.

First, in contrast to other embedded approaches, the reflective embedding architecture enables support for concrete syntax for embedded languages without high initial syntax costs. For an embedded language, developers can annotate its library with meta data for rewriting DSL programs in concrete syntax into a corresponding executable form. By using a special technique for agile grammar specification, called *island grammars* [Moo01], developers only need to specify those parts of the embedded language's grammar that deviate from the grammar of the host language. Island grammars help significantly reducing the initial syntax costs, while supporting the full class of context-free grammars.

Second, the meta level enables support for composition of languages with interactions in their syntax and semantics. On the one hand, the architecture provides special *language combiners* that analyze and compose the syntax of all embedded languages in a composition. On the other hand, there are language combiners that transform embedded languages so that they can interact at well-defined points in their evaluation logic. By allowing interacting compositions, the architecture enables embedding special language abstractions that facilitate end user programmers to implement more modular programs.

Third, the meta level enables embedding languages that interact with their host languages. The meta level uses reflective features to interact with the evaluation of host language constructs in order to change the host's semantics. Only when the host semantics can be changed, developers can embed sophisticated DSLs.

To review the reflective embedding architecture, this thesis conducts a qualitative and quantitative evaluation of the proposed concepts. First, to qualitatively evaluate the concepts of this thesis, the evaluation reviews the support of the reflective embedding architecture for supporting a set of desirable properties and compares its support to related work. Second, to quantitatively evaluate the concepts, the evaluation compares to embedded and non-embedded approaches using benchmarks. The results show that the architecture's flexibility only comes with moderately increased runtime costs.

Zusammenfassung

In den letzten Jahren gibt es ein zunehmendes Interesse an neuen Programmiersprachen, die auf bestimmte Problemdomänen spezialisiert sind, so genannte *domänenspezifische Sprachen* [MHS05] (engl. domain-specific languages, DSLs). DSLs stellen eine spezielle Syntax und Semantik zur Lösung einer bestimmten Klasse von Problemen zur Verfügung. Wenn Programme mit einer DSL entwickelt werden, erlaubt ihre *konkrete* Syntax eine höhere Produktivität des Endanwenders [MHS05, KLP+08], da ihre Syntax näher an der Problemdomäne ist als die Syntax einer *allgemeinen Sprache* (engl. general-purpose language, GPL). Jedoch ist die Implementierung einer neuen Programmiersprache kostenintensiv, da Sprachentwickler eine Syntax und Semantik entwerfen und umsetzen müssen. Wegen dieser hohen initialen Kosten sehen viele davon ab eine neue DSL zu implementieren und einzusetzen.

Um die initialen Kosten für DSLs zu minimieren, gibt es den Ansatz eine DSL in eine vorhandene Sprache *einzubetten* [Lan66, Hud96, Hud98, Kam98]. Dabei wird die Syntax und Semantik der *eingebetteten Sprache* in Form einer Bibliothek in dieser Sprache beschrieben. Die vorhandene Sprache fungiert dabei als eine *Wirtssprache* für die DSL, die als *Gastsprache* in die Wirtssprache eingebettet wird. Durch die Einbettung können Gastsprachen die allgemeinen Sprachkonstrukte der Wirtssprache wieder verwenden, was Entwicklungskosten einspart [Hud98, Kam98, KLP+08]. Zusätzlich erlaubt die Einbettung die Wiederverwendung der bestehenden Sprachinfrastruktur, wie z.B. den Entwicklungswerkzeugen, dem Parser, dem Compiler und der virtuellen Laufzeitumgebung. Neue Sprachkonstrukte können bei Bedarf durch Erweiterungen der entsprechenden Bibliotheken [Hud96] hinzugefügt werden. Eine vergleichende Studie [KLP+08] zeigt, dass durch Einbettung die Entwicklungskosten für die Implementierung neuer Sprachen, im Vergleich zu traditionellen nicht eingebetten Sprachentwicklungsansätzen, signifikant reduziert werden können.

Trotz dieser Vorteile gibt es für bestehende Ansätze zur Spracheneinbettung wichtige Einschränkungen bezüglich der Unterstützung für Spracheigenschaften, die normalerweise von nicht eingebetten Ansätzen unterstützt werden. In erster Linie gibt es bei bestehenden Ansätzen zur Einbettung in den meisten verwendeten Wirtssprachen keine Unterstützung für die freie Gestaltung der konkreten Syntax für die eingebettete Gastsprache [KLP+08]. Beim Einbetten der Gastsprache in die Wirtssprache müssen die Ausdrücke der Gastsprache mit der Syntax der Wirtssprache übereinstimmen. Zweitens ist die Sprachsemantik, die eine Gastsprache haben kann, durch die vorgegebene Sprachsemantik der Wirtssprache eingeschränkt. Drittens gibt es keine Unterstützung für spezielle Kompositionen von mehreren DSLs, die Interaktionen in ihrer Syntax und Semantik haben. Diese Einschränkungen sind vorallem auf die mangelnde Anpassbarkeit von Konstrukten in eingebetteten Sprachen und Hostsprachen zurückzuführen.

Um diese Probleme anzugehen, schlägt die vorliegende Arbeit die *Reflective Embedding Architecture* vor, die es ermöglicht Gastsprachen in reflexive Wirtssprachen einzubetten. *Reflexive Sprachen* sind Programmiersprachen, die spezielle Sprachkonstrukte besitzen, um die Struktur und das Verhalten von Programmen während ihrer Auswertung zu verändern. Die Verwendung einer reflexiven Sprache ermöglicht es die obigen Einschränkungen zu überwinden, da eingebettete Sprachen durch Analyse und Transformation ihrer Sprachbibliotheken angepasst werden können.

Die *Reflective Embedding Architecture* definiert eine neue Vorgehensweise zum Einbetten von DSLs und beschreibt wie man für eine neue DSL die Syntax und Semantik als eine Menge von Artefakten einbetten kann. Jede Sprache ist eine eingekapselte *Komponente*, die eine Menge von wohldefinierter Schnittstellen festlegt. Die Klassen der Einbettung realisieren diese Schnittstellen, um die Einzelheiten der Ausführungssemantik für die Sprachkomponente festzulegen. Weiterhin unterstützt die Architektur das Einbetten von Sprachkompositionen, Scoping-Strategien, domänenspezifischen Analysen und Transformationen. Das Besondere an der Architektur zur Einbettung ist, dass jede Sprache automatisch eine Meta-Ebene besitzt, die Entwickler benutzen können, um die Einbettung anzupassen. Anpassungen auf der Meta-Ebene haben folgende Vorteile:

Erstens ermöglicht die Reflective Embedding Architecture die Unterstützung von konkreter Syntax für eingebettete Sprachen. Für jede eingebettete Sprache können die Entwickler die zugehörige Bibliothek mit *Annotationen* versehen, um Meta-Daten für das Umwandeln von DSL Programmen mit konkreter Syntax in eine entsprechende ausführbare Form zu beschreiben. Durch die Verwendung einer speziellen Form von Agilen Grammatiken, die *Insel-Grammatiken* [Moo01] genannt werden, müssen nur die Teile der eingebetteten Sprache in einer Grammatik spezifiziert werden, die tatsächlich von der Grammatik der Wirtssprache abweichen. Insel-Grammatiken helfen dabei die initialen Kosten signifikant im Vergleich zu anderen eingebetteten Ansätzen zu reduzieren, wobei die volle Unterstützung für kontextfreie Sprachen gewährleistet bleibt.

Zweitens ermöglicht die Meta-Ebene Unterstützung für Sprachkompositionen, deren Teilsprachen Interaktionen in ihrer Syntax und Semantik haben. Auf der einen Seite ermöglicht es die Meta-Ebene die Syntax mehrerer Teilsprachen zu analysieren und zu einer zusammengesetzten Sprachsyntax zu komponieren. Auf der anderen Seite ist es ermöglicht die eingebetteten Teilsprachen so anzupassen, dass sie an wohl definierten Punkten innerhalb ihrer Auswertungslogik miteinander interagieren können. In dem Sprachkompositionen mit Interaktionen unterstützt werden, ermöglicht die Architektur die Einbettung von speziellen Sprachabstraktionen, die es dem Endanwender erleichtern, modulare Programme zu entwickeln.

Drittens ermöglicht die Meta-Ebene das Einbetten von Sprachen, die mit ihrer Wirtssprache interagieren, in dem sie reflexive Mechanismen verwenden, um die Auswertung der Sprachkonstrukte des Wirts zu beeinflussen. Nur wenn die Auswertung von Konstrukte der Wirtssprache geändert werden kann, können DSLs mit speziellen Modularisierungskonzepten eingebettet werden.

Zur Evaluation der Reflective Embedding Architecture werden die in dieser Arbeit vorgeschlagenen Konzepte qualitativ und quantitativ überprüft. Zum einen wird eine Bewertung für die

Unterstützung der Reflective Embedding Architecture für wünschenswerte Spracheigenschaften erstellt und mit verwandten Arbeiten verglichen, um die Konzepte qualitativ zu bewerten. Zum anderen wird eine Sprachimplementierung auf Basis der Reflective Embedding Architecture mit Implementierungen der selben Sprache quantitativ verglichen, die auf eingebetteten Ansätze und nicht eingebetten Ansätze aus den verwandten Arbeiten basieren. Die Evaluation zeigt im Vergleich, dass die Reflective Embedding Architecture für einbettete Sprachen erstmals eine umfassende Unterstützung der wünschenswerten Eigenschaften für eingebettete Sprachen ermöglicht und dass es dabei lediglich zu moderat erhöhten Laufzeitkosten kommt.

Acknowledgements

This dissertation would not have been possible without the help of many people, whom I would like to dedicate a few lines to.

First and foremost, I would like to thank my advisor Mira Mezini for always having an open mind for visions and new ideas, asking the right questions, giving feedback, and for patient discussions. I also would like to thank my co-advisor Robert Hirschfeld for valuable feedback and comments.

This work would not have been possible without the support of my collegues at the Software Technology Group of the Technische Universität Darmstadt, who continuously gave me feedback on my papers and presentations. I would like thank Gudrun Harris for supporting me and for having always a honest opinion. I would like to thank my collegues Martin Monperrus, Michael Eichberg, Eric Bodden, Ralf Mitschke, Andreas Sewe, Vaidas Gasiunas, Roman Knöll, Marcel Bruch, Jan Sincheck, and Lukas Satabin. Anis Charfi, Christoph Bockisch, Slim Kallel, Vasian Cepa, Ivica Aracic, Sven Kloppenburg, Shadi Rifai, Thorsten Schäfer, Tobias Schuh, Michael Krebs, Klaus Ostermann, Michael Achenbach, and Christian Hofer are former collegues, which I would like to thank them for good team work.

There are two people that in particularly helped me with improving my skills. Yuecel Karabulut was my supervisor when I was an intern at SAP Research Center in Palo Alto, CA, USA. My intership was very helpful to gain insight into the special requirements of business applications, which helped motivating my subsequent work, for which I am very thankful. Michael Haupt has supervised my Diploma Thesis, and even after I had completed my Diploma, he was remaining a mentor of mine.

Special thanks to the people around the domain-specific aspect languages (DSAL) workshop, Johan Fabry, Eric Tanter, and Jacques Noye. I would like to Thank them for inviting me into the organization committee of the DSAL workshop. Without their continuous feedback, this work would not have been possible.

The AOSD-Europe Network of Excellence (EU grant no. FP6-2003-IST-2-004349) and the feasiPLe project *feasiPLe* project (Federal Ministry of Education and Research, BMBF, Germany) have partially funded part of this work. Special thanks to Mohammed Erradi, Sebastian Oster, Karin Fetzer, Christian Wende, Henrik Lochmann, and Alisdair Johnstone, who I have worked with in various projects.

Thanks to all students, who have contributed to the implementation of the open source projects of POPART, TIGERSEYE, the DSL benchmark contest, or other projects, namely Kamil Erhard, Joscha Drechsler, Oliver Rehor, Jan Stolzenburg, Olga Gusyeva, Yassine Essadraoui, Alexander

Look, Philipp Zühlke, Fuest Christian, Siyu Yang, Leonid Melnyk, Eugen Fanshil, David Marx, Thorsten Peter, Kim David Hagedorn, Christopher Mann, Florian Jakob, and Hasan Ibne Akram.

In particular, I would like to thank all people that have supported me while writing this text, namely Martin Monperrus, Michael Eichberg, Eric Bodden, Benjamin Schmeling, Vaidas Gasiunas, Sebastian Oster and Michael Haupt. My brother, Peter Dinkelaker, and his wife, Stephanie Dinkelaker, who have convinced me to graduate after I had finished my German Diploma. I would also like to thank my best friend Joris Hensen for his valuable feedback on this text and just for being there, when I needed a helping hand.

This work is dedicated to my parents, Edith and Albrecht Dinkelaker, and my wife, Anja Kirschning, who gave me all the support, energy and confidence I needed to complete this big paper.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

To cope with the increasing complexity in software systems, programming paradigms and various language features have been proposed, e.g., object-oriented languages to encapsulate data and procedures into classes. While these languages are most frequently used in software development today, there are several situations in which they fail to implement software modularly. To address modularity problems in object-oriented software, academic research proposes new paradigms with special language concepts that extend traditional object-oriented concepts. These new concepts help developers to implement more modular object-oriented software.

Object-oriented programming languages are also used for implementing new programming languages in form of a language infrastructure. Usually, a language infrastructure consists of several components, namely a parser, an abstract syntax tree (AST), analyses on this AST, e.g., for verification, and transformations on this AST, e.g., for compilation. In general, there are large initial costs associated with implementing the infrastructure.

When languages evolve and new language features need to be added, it is important to reuse and refine parts of an existing language infrastructure. Such a reuse is only possible with an adequate technology that helps to extend existing infrastructures.

1.1. Context of this Thesis

Most modern programming languages follow the paradigms of *object-oriented programming* with classes, single inheritance, and polymorphic calls. To handle the increasing complexity, research advocates using extensions to single inheritance. *Multiple inheritance* [BI82, Mey88, MMP89], *traits* [SDNB03], and *mixins* [BC90] all allow composing modular classes that encode diverse facets of software. *Meta-classes* [Coi87], *computational reflection* [Smi82, Mae87], and *meta-object protocols* [KRB91] propose to isolate concerns as separate entities at the meta-level. These extensions have in common that they strive for better modularity in OO software that helps to manage its complexity.

Still, even with those extensions, software suffers from modularity problems. One of the most prevalent problems with these paradigms is that there are concerns that are hard to localize because they *crosscut* several modules, which is why Kiczales et al. call them *crosscutting concerns* [KLM$^+$97]. The code of crosscutting concerns is *scattered* across multiple types or classes, which makes it hard to modularize them with hierarchical modularization techniques. Additionally, the code of these concerns is *tangled* with code of other concerns and functional code, which makes it hard to maintain the concerns' code fragments.

To separate crosscutting concerns, Kiczales et al. suggest using special modules called *aspects*. Aspects can *quantify* [FF00] over the execution in other modules and augment these modules with new logic that implement crosscutting concerns. Aspects improve the modularity in OO software at the implementation level, since they can localize the code of a crosscutting concern into one module, while the above paradigms would require providing code in several modules. Most existing realizations of this *aspect-oriented programming* (AOP) paradigm extend existing OO languages with aspects, however aspects are not exclusive to OO programming. To improve the modularity of other programming languages, the language concept of aspects would be interesting for them, too.

To support new paradigms or language concepts in an existing language, sometimes developers have to completely re-implement the language. The problem with this is that implementing a new language is expensive, because developers often lose a significant part of the investments of its ancestor language.

In recent years, there has been an increasing interest in new languages that provide special syntax and semantics for certain problem domains, so called *domain-specific languages* (DSLs) [vDKV00, MHS05]. DSLs provide a *concrete syntax* that is closer to its problem domain than a general-purpose language (GPL), thus their syntax allows higher end-user productivity [MHS05, KLP$^+$08]. They provide domain-specific (DS) constraints, which provide opportunities for analysis and optimizations [vDKV00, MHS05]. Famous examples of DSLs are BNF, SQL, and HTML.

Many DSLs are virtually indispensable tools for implementing software artifacts in a certain problem domain. But, such DSL artifacts need to be integrated with other artifacts written in GPLs. Hence, there are increasing requests to compose DSLs with existing GPLs [ME00, BV04, TFH09], e.g. SQLj [ME00] composes SQL [DD89] and *Java*™ [LY99]. The increasing numbers of requests create a challenge for traditional language implementation approaches that have high initial costs [BV04, Cor06, HM03].

To manage the increasing requests for new special languages, there are techniques that aim at a smoother transition by allowing a language to *grow* from one version to the next [Ste99], e.g., by adding new syntax and semantics to an existing language. Those special techniques evolve around establishing modularity, extensibility, composability, and adaptability of languages implementations. The research for modularity in language implementations strives for building language infrastructures in a modular way (i.e. modular parsers and compilers) for the sake of a better *separation of concerns* [Par72]. There are special concepts for extensible language implementation, e.g., *attribute grammars* [Knu68, EH07b], *tree walkers* or *extended visitor patterns* [Par93, NCM03]. Those concepts enable language developers to extend the artifacts of an existing language infrastructure. For example, to extend a language's syntax, there is support for extending grammars with *grammars inheritance* [AMH90, KRV08] and *grammar imports* [HHKR89, Vis97b, Par08]. Composability of multiple languages has been addressed by special extension and composition mechanisms. For example, there is support for composition of grammars [Vis97c, For04] and of semantic *language components* [HM03, Aßm03, Hud96]. Adaptability of language are addressed by special component-oriented language approaches [EH07b] and by composition of *gray-box* language components [dB00, DDMW00, Aßm03, EVI05, KL07b, Cle07, GWDD06]. To consolidate the constructs of several languages, there are several approaches that support composing the syntax and semantics of existing languages. Composing syntax of several languages can lead to conflicts at the lexical level [vdBSVV02]. Composing the semantics of several languages may lead to complicated conflicts at the semantic level [KL05, KL07b, HBA08]. Despite there is a principle support for composition, in particular when composing semantics, there are various challenges with respect to the completeness, correctness, complexity, and implementation efforts when using existing approaches for language compositions [BV04, Cor06, Tra08].

An interesting idea that addresses parts of the aforementioned challenges is to *embed* a language into an existing language [Lan66, Hud96, Hud98, Kam98], which serves as a *host* for implementing the new language. This idea goes back to Landing [Lan66]. Hudak [Hud96, Hud98] proposes a concrete approach for embedding DSLs into general-purpose *host languages* by means of libraries written in this language. Such an embedded language can reuse the general-purpose features of the host language. Consequently, this reduces the development costs of the embedded language by reusing the existing host language infrastructure [Hud98, Kam98, KLP$^+$08], i.e., existing development tools, parsers, compilers, and virtual machines. New language features can be added incrementally by "simply" extending the corresponding libraries [Hud96]. More importantly, embeddings are said to be easier to compose in contrast to languages that are implemented with traditional approaches, such as pre-processors [MHS05].

Various embedding approaches propose using different host languages, namely languages that allow *pure functional programming* [Hud98, CKS09, ALY09], *dynamic programming* [Pes01, TFH09, KG07, KM09, RGN10, AO10], *staging* and *meta-programming* [COST04, SCK04, Tra08], *strong-typing* for object-oriented programming [Eva03, Fow05, Gar08, Dub06, HO10], *generative programming* [Kam98, EFDM03, CM07], and *source code transformations* [BV04, Cor06].

Using these approaches, there are existing embedded DSL implementations for various domains, such as for *mathematical calculations* [Hud98, COST04, Dub06, HORM08, CKS09, ALY09], *query languages* [LM00, Cor06, Dub06], *image processing* [Kam98, EFDM03, SCK04], *user interfaces* [BV04, TFH09, KG07, Gar08], *code generation* [BV04, CM07, Tra08], *simulations* [Pes01, OSV07] and *testing* [FP06, AO10]. However, so far the embedding approach has been applied only for implementing languages with a rather small set of language constructs.

The prospect of a light-weight but powerful language implementation approach has brought forth language embedding as a major area of research in programming language implementation.

1.2. Problem Statement

There are three key problems of current embedding approaches with respect to the syntax and semantics of embedded languages.

First and foremost, there is a missing support for concrete syntax in many embedding approaches [MHS05, KLP$^+$08]. This is because the host and the embedded language share the same homogeneous syntax. Although sharing the host syntax saves the initial costs for a special parser, still with a shared syntax embedded programs have a compromised encoding. That means there is a syntactically compromised representation of a program running on an embedded language in comparison to a freely designed and therefore ideal encoding when implementing the language with a tailored parser. This is particularly unfortunate since comparative studies [KLP$^+$08] indicate that this missing support for concrete syntax significantly lowers the end user's productivity in writing DSL code. Consequently, it is not recommended to follow the embedding approach for implementing a language that requires a special syntax [MHS05].

Second, there is a missing support for special semantics in embedded approaches. When sharing the host's semantics, the host semantics confine the semantics of the embedded language. That means end users can only express in embedded program what is expressible in the host language. For example, in case the host language's composition mechanisms cannot express certain concerns in a modular way, when fine-granular composition semantics are not available, such as aspects, end users programs that implement such concern suffer from bad modularity. Consequently, when implementing a language that requires support for special semantics, such as fine-grained composition mechanisms, language developers cannot use a host language that disallows embedding such semantics, which requires using a more powerful host language.

Third, there is a missing support for implementing and composing multiple languages that have interactions. There can be two forms of interactions. First, an embedded language can have syntactic and semantic conflicts with the host language. Second, multiple embedded languages can have syntactic and semantics interactions among each other. Without having special mechanisms to detect and resolve such interactions, developer cannot implement or compose such embeddings.

The aforementioned problems in current embedding approaches disallow implementing contemporary programming languages that have complicated language constructs with a special syntax and semantics. To overcome the limitations of current embedding approaches, the goal of this thesis is to propose a new embedding approach that addresses these problems. Having special support for solving these open issues could open new grounds for new types of languages, and therefore, it would mature the concept of embedding to compete with traditional language implementation approaches.

1.3. Thesis

The research method of this thesis is organized in four steps. First, it put focus on related work and identifies important problems in its body. Second, it proposes a novel architecture for language embedding in which every language has a designated meta-level. Third, it presents extensions to the core architecture that provide special language concepts and generic tools for embeddings. These concepts and tools facilitate the language developer to implement new languages and language compositions that have special requirements. Fourth, it evaluates the architecture support for addressing the identified problems and compares the resulting solutions to related work. The results of those four steps are summarized in the following subsections.

1.3.1. State-of-the-Art in Language Embedding Approaches

This thesis gives an overview of architectures used in traditional non-embedded approaches for language implementation, such as compilers, interpreters, and pre-processors. It identifies seven desirable properties that traditional non-embedded approaches claim to be important [HM03, Par10], namely (1) extensibility, (2) composability of languages, (3) special means for composition, (4) support for concrete syntax, (5) means for controlling the scope of language constructs, as well as (6) support for analyses and (7) transformations. Since DSLs are themselves programming languages, they are also subject to the requirement to support these important properties [vDKV00, MHS05]. For each of these properties, the thesis proposes concrete scenarios that demonstrate support for the corresponding property and that can be used later on for evaluation of the corresponding property. In the context of this thesis, a review on the support for the proposed properties by related work has been conducted. The review shows that there is certain support for the properties by existing embedding approaches, but it also reveals that all related embedding approaches have important limitations.

1.3.2. Core Architecture for Reflective Embedding

To address the current limitations in related work, this thesis proposes to use reflective host languages for embedding and to exploit reflective features for improving the quality of embeddings. Most importantly, when using a reflective language for embedding, the embedded language implicitly yields a meta-level for the embedding. A key property of the meta-level is that embedded language constructs remain *first-class objects* until and during their execution. The meta-level helps language developers to expose the internals of their embeddings to other parties. With exposed internals, embedded analyses and transformations can access the internal information to reason about an embedded language, its language constructs, and objects in embedded programs.

The embedding approach is based on a disciplined architecture that defines for language developers how to embed a language's syntax and semantics. The architecture uses reflective features for creating language embeddings and is therefore called the *reflective embedding architecture*. The architecture consists of a language independent core that provides a set of *accompanying language key mechanisms* that facilitates implementing and composing languages. If needed, language developers can extend the core architecture with new language mechanisms that better fit special requirements.

In this architecture, every language defines a *language component* that has a set of well-defined interfaces of classes. By default, these classes are well encapsulated and implement the details of the execution semantics for that language component. Whenever necessary the language developers can break up encapsulated languages and use them as *gray-box language components*. Additionally, the language developer can define abstractions that allow language end users to manipulate language internals in a controlled way.

1.3.3. Extensions to the Core Architecture

To demonstrate the applicability of the core architecture, this thesis presents special embeddings and extensions on top of the reflective embedding architecture. The architecture has been used to implement extensible languages for *computer graphics* [New73], *workflows* [WfM], and *SQL queries* [DD89]. In particular, this thesis elaborates on how to embed sophisticated general-purpose language constructs that help to improve modularization, such as *functional* and *structural abstraction* [AS96], *scoping* [AS96, Tan09], and *aspects* [KLM$^+$97]. Further, the thesis presents embedding of language prototypes for formal models, such as *state machines* [OMG04], *feature models* with dependencies [Pre97], and *grammar definitions* (BNF).

To enable implementations of contemporary languages with complicate language constructs that have a special syntax and semantics, the thesis presents several extensions of the core architecture with designated concepts, frameworks, and tools that enable the evolutions of embedding to support these special requirements.

To enable language constructs that have a special semantics, the architecture allows the embedding of special semantics on top of it. As an example for special semantics, this thesis elaborates how to embed aspects that can localize crosscutting concerns in DSL programs. For this, the architecture provides means for composing special DSLs in which the evaluation of DSL program interacts with the evaluation of a DSL program in another DSL. The reflective embedding architecture uses those means to enables crosscutting composition of aspects into DSL programs. general means for *domain-specific join points*, as requested by [RMH$^+$06, CNF$^+$08, RMWG09]. First, it uses DS join points to enable an aspect-oriented version of the Logo language for *computer graphics* [New73]. Second, it enables aspects for *workflow* languages similar to [CM04]. Third, it enables composing dynamic aspects for *SQL queries*.

Concerning the special syntax of DSLs, the architecture provides a generic pre-processor that enables concrete syntax. Language developer can incrementally add meta-data to libraries of embedded languages that specifies the concrete syntax of them. To make DSL programs executable, the pre-processor then uses this meta-data to transform a program in a concrete syntax to an executable form in the host language syntax.

1.3.4. Evaluation

To evaluate the reflective embedding architecture, this thesis both qualitatively and quantitatively reviews the proposed concepts. The evaluation compares to related work on embedded languages. It focuses on evaluating of the support of the provided language tools for language developer and end user at the language level, therefore evaluating the quality of *integrated development environments* (IDEs) for language end user is out of the scope of the evaluation.

To qualitatively evaluate the concepts, in the context of this thesis, the evaluation reviews the support of the reflective embedding architecture for supporting the identified properties and compares its support to related work.

The results of the qualitative evaluation show the following advantages of the reflective embedding architecture. First, language developers can extend and compose language from existing language components for execution, analysis, and transformation. Second, it reduces the effort to implement domain semantics, since one can reuse available OO libraries in their embeddings. Third, they can distribute their compiled embeddings to other developers who can again provide extension and abstraction on top of existing embeddings. While existing approaches have only limited support for the desirable properties, the proposed architecture has practicable no limitations for five of the seven desirable properties. Although the evaluation shows that developers can compose basic analyses and transformations of different languages, it remains unclear whether the embedding approach and related embedding approaches support complicated analyses and transformations.

To quantitatively evaluate the concepts, the evaluation compares to traditional approaches and embedded approaches with benchmarks. It evaluates the implementation costs of embeddings, its compares the basic performance of programs running on an embedding to a traditional language implementations, and the runtime overhead when using special extensions on top of the core architecture.

The results of the quantitative evaluation show that developers can save the initial investments for DSLs compared to traditional approaches. A developer can grow a DSL without concrete syntax into a DSL with concrete syntax. With those means for flexible language implementation, the evaluation measured that there is a moderate performance overhead compared to related approaches. When exploiting the architecture's possibilities for embedding specific analyses and optimizations, one can significantly reduce the performance overhead due to the indirections of the meta-level. For example, when executing DSL programs in an optimized version of an embedded aspect language, the execution overhead of the aspect mechanism is less than an order of magnitude.

1.4. Summary of Contributions

The contributions of this thesis can be classified w.r.t. which field in programming language research they contribute to: (I) embedded domain-specific languages, (II) language architectures, and (III) language methodologies, (IV) benchmarks for DSLs.

(I) Contributions to the research on embedded domain-specific languages: This thesis advances the research of embedded domain-specific languages by:

- **(a) Identifying desirable properties for language embedding approaches**
- **(b) Proposing the concept of the reflective embedding architecture:** This thesis extends the concept of embedded domain-specific languages by Hudak [Hud96] with a meta-level.
- **(c) Systematically studying reflective host languages for embedding**

(II) Contributions to the research on language architectures: This thesis is inspired by existing traditional language architectures, it consolidates these architectures for new applications, and it contributes back generalizing their applicability and specializing them for new domains.

- **(a) An approach to grow a language into a reflective language:** This thesis proposes a new approach to systematically implement a *reflective language* [Smi82, Mae87]. It presents how a language developer can expose parts of an embedded languages to the users, e.g., growing a DSL into a reflective DSL. With a reflective DSL, a user can implement reflective programs that reason about themselves using *structural* and *behavioral reflection* [Smi82, Mae87].
- **(b) Generalization of meta-protocols:** This thesis generalizes the concept of *meta-object protocols* [KRB91] from OO languages to meta-protocols for DSLs.

(III) Contributions to the research on language methodologies: This thesis implements technologies and prototypes that contribute to language methodologies.

- **(a) Composition of languages with syntactic and semantic interactions**
- **(b) Composing crosscutting concerns in domain-specific languages**
- **(c) Controlling the scope of aspects in domain-specific languages:** This thesis provides generic concepts and technology for controlling the *scope* and *activation* of crosscutting concerns in domain-specific languages. With the technology, programs can load and dynamically activate/deactivate aspects at runtime.

(IV) Benchmarking for Embedded DSLs: This thesis proposes an open-source benchmark to measure and compare the costs of different DSL approaches.

In the context of this thesis, the author has published the following papers:

Publications in Conference Proceedings:

- Tom Dinkelaker, Michael Eichberg, and Mira Mezini. **Incremental Concrete Syntax for Embedded Languages.** In *Proceedings of the 26th ACM Symposium on Applied Computing—Technical Track on Programming Languages (PL at SAC)*. Taichung, Taiwan, March 21-25. ACM, NY, USA, 2011.

- Tom Dinkelaker, Michael Eichberg, and Mira Mezini. **An Architecture for Composing Embedded Domain-Specific Languages.** In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD)*. Rennes and Saint Malo, France, March 17-19. ACM, NY, USA, 2010.

- Slim Kallel, Anis Charfi, Tom Dinkelaker, Mira Mezini, and Mohamed Jmaiel. **Specifying and Monitoring Temporal Properties in Web Services Compositions.** In *Proceedings of the 7th IEEE European Conference on Web Services (ECOWS)*. Eindhoven, The Netherland. IEEE Computer Society Press, 2009.

- Tom Dinkelaker, Mira Mezini, and Christoph Bockisch. **The Art of the Meta-Aspect Protocol.** In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development (AOSD)*, Charlottesville, VA, March 2-6. ACM, NY, USA, 2009.

- Anis Charfi, Tom Dinkelaker, and Mira Mezini. **A Plug-in Architecture for Self-Adaptive Web Service Compositions.** In *Proceedings of the 7th IEEE International Conference on Web Services (ICWS)*. Los Angeles, CA, USA, July 6-10. IEEE Computer Society Press, 2009.

- Tom Dinkelaker, Alisdair Johnstone, Yuecel Karabulut, and Ike Nassi. **Secure Scripting Based Composite Application Development: Framework, Architecture, and Implementation.** In *Proceedings of the 3rd ICST/IEEE International Conference on Collaborative Computing*, White Plains, NY, November 12-15. IEEE Computer Society Press, 2007.

- Christoph Bockisch, Matthew Arnold, Tom Dinkelaker, and Mira Mezini. **Adapting Virtual Machine Techniques for Seamless Aspect Support.** In *Proceedings of the 21st Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, OR, October 22-26. ACM, NY, USA, 2006.

- Michael Haupt, Mira Mezini, Christoph Bockisch, Tom Dinkelaker, Michael Eichberg, and Michael Krebs. **An Execution Layer for Aspect-Oriented Programming Languages.** In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, Chicago, IL, June 11-12. ACM, NY, USA, 2005.

Demonstration:

- Tom Dinkelaker. **Demo: A Dynamic Software Product Line Approach using Aspect Models at Runtime.** In *4th Workshop on Domain-Specific Aspect Languages (DSAL).* at the *9th ACM International Conference on Aspect-Oriented Software Development (AOSD).* Rennes and Saint Malo, France, March 17-19. `http://www.dsal.cl/2010/`, 2010.

Publications in Workshop Proceedings:

- Tom Dinkelaker, Johan Fabry, Anne-Francoise Le Meur, Jacques Noye, and Eric Tanter. **Proceedings of the 4th Workshop on Domain-Specific Aspect Languages (DSAL).** at the *9th ACM International Conference on Aspect-Oriented Software Development (AOSD).* Rennes and Saint Malo, France, March 17-19. `http://www.dsal.cl/2010/`, 2010.

- Tom Dinkelaker, Ralf Mitschke, Karin Fetzer, and Mira Mezini. **A Dynamic Software Product Line Approach using Aspect Models at Runtime.** In *Proceedings of the 1st International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines*, at the International Conference on Aspect-Oriented Software Development, Rennes and Saint Malo, France, March 17-19. CEUR-WS.org, ISSN 1613-0073, Vol. 564, `CEUR-WS.org/Vol-564/`, 2010.

- Tom Dinkelaker, Martin Monperrus, and Mira Mezini. **Supporting Variability with Late Semantic Adaptations of Domain-Specific Modeling Languages.** In *Proceedings of the 1st International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines*, at the International Conference on Aspect-Oriented Software Development, Rennes and Saint Malo, France, March 17-19. CEUR-WS.org, Vol. 564, `CEUR-WS.org/Vol-564/`, 2010.

- Tom Dinkelaker, Martin Monperrus, and Mira Mezini. **Untangling Cross-Cutting Concerns in Domain-Specific Languages with Domain-Specific Join Points.** In *Proceedings of the 4th Workshop on Domain-Specific Aspect Languages (DSAL)*, at the International Conference on Aspect-Oriented Software Development, Charlottesville, VA, USA, March 2-6. ACM, NY, USA, 2009.

- Tom Dinkelaker, Mira Mezini. **Dynamically linked Domain-Specific Extensions for Advice Languages.** In *Proceedings of the 3rd Workshop on Domain-Specific Aspect Languages (DSAL)*, at the International Conference on Aspect-Oriented Software Development. ACM, NY, USA, 2008.

Technical Reports:

- Tom Dinkelaker. **Review of the Support for Modular Language Implementation with Embedding Approaches.** TUD-CS-2010-2396, Technische Universität Darmstadt, Germany, November, 2010.

- Tom Dinkelaker, Christian Wende, and Henrik Lochmann. **Implementing and Composing MDSD-Typical DSLs.** TUD-CS-2009-0156, Technische Universität Darmstadt, Germany, October, 2009.

- Mira Mezini, Danilo Beuche, Ana Moreira (Editors). **1st International Workshop on Model-Driven Product Line Engineering (MDPLE'09)**, at the 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA). Centre for Telematics and Information Technology, University of Twente, Enschede, The Netherlands, 2009.

- Tom Dinkelaker and Michael Haupt **Inventory of Aspect-Oriented Execution Models AOSD-Europe.** Deliverable D40, AOSD-Europe-TUD-4, Technische Universität Darmstadt, Germany, February 28, 2006.

1.5. Organization of the Thesis

The structure of the thesis has five parts. The first part discusses the state-of-the-art. Section 2 gives an overview on existing language embedding approaches and identifies their advantages and limitations. Then, Section 3 identifies properties that are often available in non-embedded DSL approaches, but that have a limited support by embedding approaches.

The second part presents the concept, which begins with presenting the reflective embedding architecture in Section 4. Next, Section 5 elaborates the implementation of the architecture in one example host language.

The third part presents the application of the proposed concepts. Section 6 shows how language developer can use the core architecture to implement a DSL in which a DSL program can reason about its domain and customize the language semantics. Section 7 shows how to use the core architecture for composing interacting of domain-specific languages. Section 8 shows how to incrementally improve the concrete notation of programs using an embedded language.

The fourth part presents the evaluation of the proposed concepts. Section 9 compares to related work. Section 10 reviews the support for the identified properties. Section 11 presents the quantitative evaluation and compares to related work.

The fifth and last part concludes this thesis, it summarizes the thesis results and its contributions in Section 12 and discusses future work.

The thesis assumes that the reader has a fluent understanding of *Java* [GJSB00] and a principle understanding of *AspectJ* [KHH$^+$01].

Part I.

State of the Art

2. State of the Art in Language Implementation Approaches

This chapter introduces terminology and gives an overview over the state of the art relevant to this thesis. Starting in Section 2.1 with discussing the importance of DSLs in general, Section 2.2 defines the scope of this thesis on embedded DSLs. Next, Section 2.2.1 elaborates on the problems of existing embedded DSL approaches. After that, Section 2.3 reviews architectures that traditional approaches frequently employ for structuring language implementations.

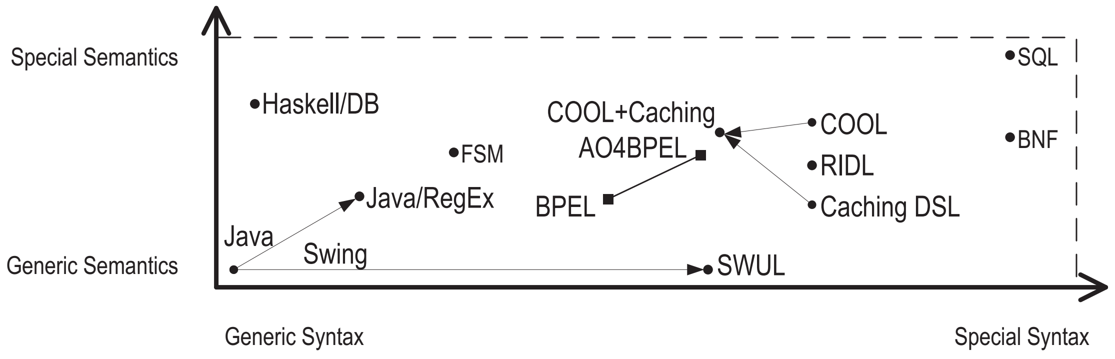
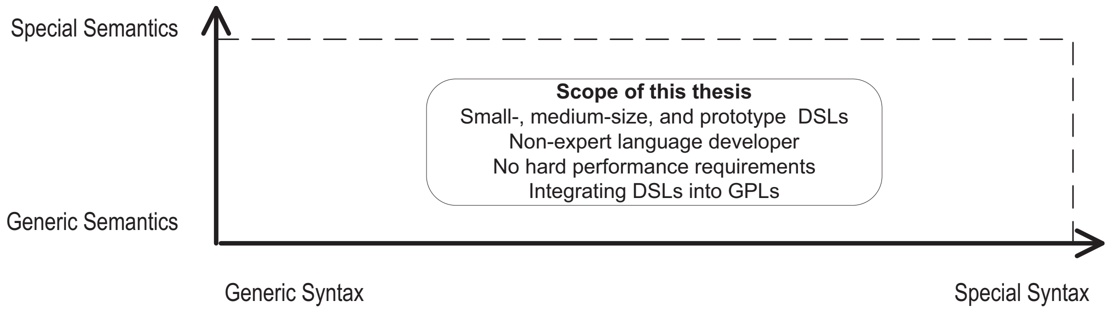


Figure 2.1.: Example DSLs in the design space of DSLs



Scope of this thesis
Small-, medium-size, and prototype DSLs
Non-expert language developer
No hard performance requirements
Integrating DSLs into GPLs

Figure 2.2.: The focus of this thesis in the DSL design space

2.1. Domain-Specific Languages

There are two important surveys [vDKV00, MHS05] that define domain-specific languages and analyze their role in modern software development.

In the first survey paper *"Domain-Specific Languages: An Annotated Bibliography"* [vDKV00], Van Deursen, Klint and Visser proposes the following DSL definition:

> "*A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.*" [vDKV00]

The paper mentions several advantages of DSLs. They have a special syntax that allows a developer to express a solution at a higher level of abstraction that is closer to a problem domain. DSL programs tend to be more concise and self-documenting. Thus, domain experts can easier understand, validate, modify, and often even non-programmer end users can develop DSL programs. Having a more concise encoding of program improves productivity, and more concise programs enhance reliability, maintainability and portability. A DSL implementation directly embodies the domain knowledge and conserves this for reuse. Further, DSLs allow validation and optimization at the domain level.

In their survey, they also discuss disadvantages of DSLs. There are high costs for designing, implementing, and maintaining a DSL, on top of which come the costs of education for DSL end users. While there are many problem domains, the availability of DSLs is limited. For DSL developers, it is often difficult to find the proper scope for a DSL and to balance between domain-specific and general-purpose language constructs in its syntax and semantics. Due to the higher level of abstractions, DSL programs are potentially less efficient compared to hand-coded software.

In the second survey paper with the title *"When and How to Develop Domain-Specific Languages"* [MHS05], Mernik, Heering, and Sloane define DSLs as follows:

> "*DSLs trade generality for expressiveness in a limited domain. By providing notations and constructs tailored toward a particular application domain, they offer substantial gains in expressiveness and ease of use compared with GPLs for the domain in question, with corresponding gains in productivity and reduced maintenance costs. Also, by reducing the amount of domain and programming expertise needed, DSLs open up their application domain to a larger group of software developers compared to GPLs.*" [MHS05]

They discuss developing DSLs in 3 phases: analysis, design, and implementation.

In the *analysis phase*, DSL developers identify the requirements, which have important implications on the DSL design. A DSL design is only successful if it meet those requirements of the language end users. Therefore, the decisions in the design phase of a DSL should be based on those requirements.

In the *design phase*, according to Mernik et.al., DSL design can be characterized along two orthogonal dimensions: (1) the relationship of the DSL design to designs of existing languages, and (2) the formal nature of the design. For the first dimension, they state that it easiest way is to design a DSL is to use part of the design of an existing GPL or DSL, which they call *language exploitation*. Language developers can either (a) *piggy-back* the DSL design by using parts of an existing language design without special restrictions, (b) they can design a *specialization* by restricting an existing language design to a certain domain, or (c) they can design an *extension* to an existing language design by adding new domain concepts to it. For the second dimension, they distinguish how formal the language developer designs a DSL. There is the *informal design* that uses specifications given in natural languages or a set of illustrative DSL example programs that can be later used as tests. Conversely, there is the *formal design* that uses an existing semantics definition method, such as attribute grammars, rewrite rules, or abstract state machines. The decisions made in the design phase of a DSL have important implications for the costs and maintenance of its implementation.

For the *implementation phase*, Mernik et al. give an overview of existing approaches:

Interpreters: They interpret DSL constructs using a standard *fetch-decode-execute cycle*. They support DSLs with a rather dynamic nature. They often have a greater simplicity and control over the execution compared to other approaches. Yet, usually interpreters are implemented in a monolithic way, and therefore not extensible.

Compilers or Generators: These approaches rewrite high-level programs to a target language that is usually at a lower level of abstraction, e.g. assembler or machine code. Compilers often support semantic analysis, e.g. for static analysis and optimization. Similarly to interpreters, most compilers are monolithic and not extensible.

Pre-processing: DSL constructs are translated to constructs in an existing GPL. There are different forms of pre-processing. Macro processing that usually expands DSL expressions from syntactic macro definitions. Source-to-source transformations rewrite DSL programs to another language, which is similar to compilers, but the transformation usually produces code in a high-level language not machine code.

Embedding: In line with Hudak, Mernik et al. define embedding as libraries implemented into an existing GPL. For them, even GPL application libraries are a basic form of embedding. Mernik et al. state that because of the limitations of existing embedding approaches, embedding can only be used if there is no need for special requirements for a DSL, such as there can be a need for concrete syntax, analysis, verification, optimization, parallelization, or transformation. Because existing embeddings are currently confined by their host languages, they do not recommend to follow the embedding approach, if there is a need for such a special requirement.

Extensible Compilers and Interpreters: These are special compilers/interpreters that can be extended with domain-specific (DS) optimization rules and/or code generation. Most existing extensible compilers/interpreters are for one particular language, which is in most cases a GPL. They only allow extension to GPLs. Usually, for a new DSL, an extensible compiler needs to be implemented from scratch. Because the initial cost for an extensible compiler is even higher than for a monolithic compiler, there are only a few extensible compilers available for DSLs [ADDH10].

Commercial Off-The-Shelf (COTS): COTS-based approaches use standard tools and/or standard syntax that is applied to a specific domain. In COTS-based approaches like UML [OMG04] or XML, there are standard mechanisms for syntactic extensions and transformation (rule-based transformation).

Hybrid approaches: This can be any combination of the above approaches, e.g., a combination embedding and compilers [EFDM03]. However, there are only a few examples that actually combine several techniques. Mernik et al. mention two concrete DSLs, but there is no framework approach for hybrid DSL implementation.

The two DSL survey papers by Van Deursen et al. and by Mernik et al. are widely considered as the standard references for DSLs. Still, both definitions are very vague, as the authors themselves admit. Since this thesis focus are embedded DSLs, it is out of the scope of this thesis to provide a new definition of DSLs in general.

This thesis uses the definition of Van Deursen et al. and of Mernik et al., and therefore inherits the vagueness in these definitions. To address this vagueness, this thesis discusses different kinds of DSLs. Indeed there is a large range in the design space of DSLs that comply to that definition. Figure 2.1 (p. 15) gives an overview[1] of DSLs and points out five different kinds of DSLs: (1) DSLs with highly specialized tools that perform analysis, verification, optimization, parallelization, or transformation, such as SQL [DD89]. SQL has a special syntax for implementing database queries, and it has special semantics with support for analysis and transformation for an optimized execution of such queries. (2) DSLs that only define syntactic sugar for domain-concepts but no new semantics, such as SWUL [BV04] with syntactic sugar for Java Swing. (3) languages that are merely defined as GPL application libraries, such as Haskell/DB [LM00] for embedding SQL queries into the Haskell programming language. (4) embedded DSLs that encode domain objects and operations that define a DS vocabulary and execution semantics, such as BPEL [AAB$^+$07] that defines business processes with a workflow DSL embedded into XML; and (5) DSLs that mix DS and general-purpose language constructs, such as Java/RegExp [BV04]. Unfortunately, except embedding and pre-processing with macros, all approaches assume that the language developer is an expert who has special skills.

Figure 2.2 (p. 15) illustrates the focus of this thesis with respect to the DSL design space. This thesis focuses on providing methods and technology for the DSL implementation phase.

[1]The following DSL examples from Figure 2.1 are from the following papers: Swing, SWUL [BV04], Haskell/DB [LM00], Java/RegExp [BV04], FSM (Finite State Machines) [DWL09], BPEL [AAB$^+$07], AO4BPEL, COOL [Lop97], RIDL [Lop97], Caching DSL [HBA08], COOL+Caching [HBA08], BNF (Backus Naur Form), and SQL [DD89].

For a successful development of new DSLs, the proposed technologies in this thesis need to be used together with requirement analysis and design approaches for DSLs. The readers who are interested in further details on the other phases are referred to the survey paper of Mernik et al. [MHS05].

2.2. Embedded Domain-Specific Languages

The idea to embed a language into another language goes back to Landing [Lan66] and Hudak [Hud96, Hud98]. The literature often uses the term *embedded domain-specific language*[2] (embedded DSL or EDSL for short) to refer to such language embedding, therefore this term is used throughout the thesis. Unfortunately both Landing and Hudak do not give a clear definition of what an embedded DSL is. Hudak states that an embedded DSL "*is designed within an existing, higher-order and typed, programming language such as Haskell or ML*". This definition is vague and restricted to functional languages, but it roughly defines an embedded DSL as being a library in a GPL.

Kamin [Kam98] defines embedding a DSL as "*the process of implementing a language by defining functions in an existing host language; the host language with these added functions is the new language [...]*". Further, he states that this method can be used in any language. Therefore, Kamin's definition of an embedded DSL is not restricted to functional languages. Although he states that functions facilitate embedding. According to Kamin, embeddings do not have to be compiled and executed by the host language compiler. Embedding can generate a version of the DSL program into a version in a target language. Next, an external tool can compile the generated version of program into an executable version. Although this definition of embedded DSLs is accurate, it is not used often in the literature because it does not define a precise scope. This imprecise scope also applies to non-embedded approaches (e.g. as parser generators) that do have a clear intention to reuse the host language features for implementing a DSL.

Fowler [Fow05] defines embedded DSLs as *internal DSLs*. An internal DSL is an embedding that uses the *internal* host compiler to compile DSL programs. He considers all language implementations that use tools that are *external* to the host language, i.e. a compiler of another language, as *external DSLs*. While Fowler's definition is close to the one of Hudak, unfortunately his internal DSL definition excludes Kamin's embedding approach.

Mernik et al. points out in [MHS05] that "*with an application library, any GPL can act as a DSL*". However, the expressiveness of application libraries—i.e. their syntax and semantics—is confined by the language they are implemented in. Therefore, DSL constructs cannot always be mapped in a straightforward way to functions or objects in a library. This definition is broader than the previous definitions, as it roughly defines that any library in a GPL is an embedded DSL. Such a broad definition is problematic since it does not take into account the developer's intention when implementing a DSL. But for embedded DSLs, it is important that developers

[2]To refer to a language embedding, Hudak uses the term *domain-specific embedded language* [Hud96]. Other authors [Fow05] propose to refer to embedded languages as *internal DSLs*.

design them with the intention to be used as an embedded DSL, otherwise embedding cannot be expected to have an appropriate notation.

Tratt [Tra08] defines two different styles of embeddings. Language embeddings can be distinguished with respect to what particular language infrastructure is used to compile and execute the embedding: there are *homogeneous* and *heterogeneous* approaches. On the one hand, there are homogeneous embeddings that have a homogeneous syntax with their host, so one can mix embedded DSL expressions with the host expressions. Homogeneous embeddings have a uniform compile- and/or runtime with their host language, which allows objects to *flow* between DSL and host programs back and forth. On the other hand, there are heterogeneous embeddings that have a *source language* (also *meta language*) and a *target language*. Because artifacts of the source and the target language are compiled by different compilers, there is no uniform compile- and/or runtime runtime, hence objects and language features cannot be shared. Tratt's definition is useful to classify existing embedding approaches. Unfortunately, his definition does not define the process of embedding itself.

Since the above DSL definitions are too vague, this thesis consolidates the existing definitions of Hudak, Kamin, Fowler, and Tratt:

> An embedded domain-specific language is a *virtual* language that is implemented as library in an existing host language. A *virtual language* has an implementation that reuses parts of an existing language infrastructure in order to give end users the *illusion* that they use a *real language* with specialized tools for the domain. The virtual language enables special concepts and abstractions for a certain problem domain on top of the reused infrastructure. This infrastructure can encompass tools such as parsers, compilers, interpreters, virtual machines, and developer tools, whereby the embedding can reuse the infrastructure both in a homogeneous or heterogeneous way.

This definition consolidates all previous definitions, but it refines them by two important objectives. First, it requires the embedding to be *intended*. An application library is not seen as a full-fledged embedded language, because an opportunistic usage of a library as an embedding is considered problematic for end users. When using a library that was not intended as an embedding, the resulting library likely does not give end users appropriate domain syntax and semantic guarantees [MHS05]. Therefore, a language developer needs to clearly intend to embed a language. Second, the illusion of the embedding can have weaknesses. There can be differences in the quality of the illusion dependent on the embedding style and host language. One can expect a slower performance for embedded programs than if they would be executed on a non-embedded implementation. Further, host tools may reveals unnecessary technical details, such as technical error messages, that cannot be understood by end users [Kam98]. If the illusion of the embedding has weaknesses, it still needs to be convincing for end users.

Although the definition of a virtual language helps to clarify the nature of embedded DSLs, this thesis does not use this term to replace the term "embedded DSL". Because this term has been established in today's literature, this thesis retains to the old terminology.

2.2.1. Open Issues in Existing Embedding Approaches

This section elaborates on open issues with embeddings addressed by this thesis.

2.2.1.1. Lack of Support for Concrete Syntax

When a developer implements a program on top of the library of a language embedding, the developer must encode the DSL program into the host language's syntax. Consequently, the program expressions do not have the exact *concrete syntax*, which a traditional DSL would have. For example, consider Haskell/DB one of the most advanced pure embedding in a functional language. Leijen et al. [LM00] use Hudak's technique and monads to homogeneously embed *SQL* [DD89] statements in Haskell. Haskell/DB provides interesting safety guarantee that are implicitly checked be the Haskell compiler. However, since pure embedded DSL programs must be encoded in Haskell, they have a *compromised syntax*, i.e. DSL programs are encoded to comply with the host syntax and therefore contain various characters that are irrelevant with respect to the problem domain. The literature often refers to those irrelevant characters as *syntactic noise*. In particular in functional languages used by pure embedding, the compromised syntax is significantly different from a desirable concrete DSL syntax. According to Mernik [MHS05] and Kosar [KLP+08], DSL programs with such a compromised syntax make them less efficient to encode and harder to understand.

For illustration, Figure 2.3 compares the concrete and compromised syntax of an example SQL query. While on the left hand side, with non-embedded SQL, the end user has a concrete syntax that is close to the problem domain, on the right hand side, the equivalent SQL query encoded in compromised syntax for Haskell/DB. Unfortunately, when using embedded DSLs that lack of support for concrete syntax, end users are confused by this syntactic noise, which leads to a significant lower end user productivity [KLP+08].

```
SELECT name, mark
FROM Students
WHERE name=[name] AND mark=[mark]
```

(a) Concrete syntax in non-embedded SQL

```
type Student =
  Row(name :: String, mark :: Char)

queryMark :: String −> Query Student
queryMark = \n −>
  do{ student <− table students
     ;  restrict  (student!name .==. Lift  n)
     ;  project
       (  name = student!name
       ,  mark = student!mark)
       )
     }
```

(b) Compromised syntax in the pure embedding of Haskell/DB (from [LM00])

Figure 2.3.: Concrete versus compromised syntax of an SQL query

2.2.1.2. Lack of Support for Crosscutting Composition for DSLs

The separation of concerns is one of the most fundamental principles in software engineering [Par72]. Still, with most mainstream languages, crosscutting concerns cannot be localized and remain scattered and tangled over several modules. To address this, for general-purpose languages, many aspect-oriented mechanisms have been provided. Such an aspect-oriented mechanism composes the crosscutting logic of aspects into all relevant modules. This process of crosscutting composition is often referred to as *weaving*.

Nonetheless, also DSLs may suffer from scattering and tangling issues, such as Charfi et al. have shown this for BPEL [CM04] and Rebernak et al. for grammars [RMHP06]. Despite this, for most DSLs, no aspect-oriented mechanism is available – even for mainstream DSLs. Without support for crosscutting composition, developers cannot localize the scattering and tangling in DSL programs, which makes them hard to maintain.

With current language implementation approaches—not only for embeddings—but also for non-embedded approaches, there is a lack of means for separating crosscutting concerns in DS base languages. The problem is that current language implementation approaches have no adequate means to integrate DS abstractions into the aspect-oriented mechanism. In particular, the lack of crosscutting composition is due to the fact that there are no special means to represent events in the execution of a DSL program and means to change the execution of these events.

2.2.1.3. Lack of Support for Fine-Grained Composition

There are several works in non-embedded approaches that discuss sophisticated compositions of DSLs [EVI05, KL07b, HBA08]. In particularly interesting are compositions of several *domain-specific aspect languages* (DSALs). DSALs are special DSLs that internally make use of aspect-oriented concepts. In addition to *general-purpose aspect languages* (GPALs), DSALs provide a domain syntax that allows implementing aspects in a more declarative way. Unfortunately, existing embedding approaches lack support for composing such DSALs.

Kojarski et al. [KL05, KL07b] compose *Cool* [LK97] and *AspectJ* [KHH$^+$01]. Cool is a domain-specific aspect language for implementing coordination in OO software. Kojarski et al. found *co-advising* and *foreign advising conflicts* between *Cool aspects* and *AspectJ aspects*. E.g. such conflicts can happen whenever an AspectJ aspect advices a method on which also advised by a Cool aspect. Specifically, when AspectJ and Cool co-advise a join point and the AspectJ advice is executed before the Cool advice, the inserted AspectJ advice may violate the coordination logic, which is inserted by the Cool advice, and which protects the join point activity from side effects by concurrently executed threads. Kojarski et al. designed a special weaver framework called *AweSome* that wraps Cool aspects around AspectJ aspects in order to solve such conflicts.

Further, Havinga et al. [HBA08] compose Java, Cool, and a DSL for caching return parameters of method calls (cf. Section 10.3.2.2, p. 228). Havinga extended their *JAMI* framework to allow composition of the two DSALs. They found that there are many co-advising conflicts between Cool and their caching DSL. They found that resolving the weaving interaction of crosscutting concerns from different domains cannot be correctly resolved at the level of a complete aspect module, but it needs an ordering mechanism at the pointcut-and-advice level. In particular, AspectJ can only order aspects but not advice, therefore it cannot resolve such domain-specific interactions. Generally, any weaver that does not have support for ordering at the level of pointcut-and-advice is not applicable for DSAL composition. To solve the problem, they show that they can resolve these interactions by specializing the aspect weaver for the composition.

```
 1 class Document { //A Java class
 2    List<String> content;
 3
 4    void addLine(String line)  {...}
 5    void setContent(List<String> content)  {...}
 6    List<String> getContent()  {...}
 7    long wordCount() {...}
 8 } ()
 9
10 coordinator Document { //A Cool aspect
11    selfex addLine, setContent;
12    mutex {addLine, setContent}; mutex {addLine, getContent}; mutex {addLine, wordCount};
13    mutex {setContent, getContent}; mutex {setContent, wordCount};
14 }
15
16 cache Document object { //A Caching aspect
17    memoize wordCount,
18    invalidated by assigning content
19       or calling addLine(java.lang.String);
20 }
```

Listing 2.1: A Java class and two DSAL aspects that have a conflict (from [HBA08])

For example, consider the following example application in Figure 2.1 taken from [HBA08]. There are two DSAL aspects that advise the class Document in lines 1–8.

The first aspect is a Cool aspect (Figure 2.1, lines 10–14), it defines several exclusive regions at the level of methods. The keyword selfex defines that all subsequent methods are self exclusive for a particular thread, e.g. addLine and setContent are not allowed to be invoked again, as long as the corresponding method remains on the stack of the current thread. The keyword mutex defines that two or more methods are exclusive for a particular thread, e.g. addLine and setContent may not be invoked in the same control flow.

The second aspect is a caching aspect (Figure 2.1, lines 16–20), it defines a *memoization* concern that caches the return value of the Document.wordCount() method. Instead of repetitively calculating the return value, memoization will looks up the wordCount() in the cache as long as it is not invalidated, which happens whenever assigning a new value to content or when calling the method addLine(String).

There is an interaction between these two DSAL aspects, because both aspects co-advise the same method addLine. In [HBA08], Havinga et al. elaborate that the interactions of these two aspects lead to composition conflicts. When the caching advice and the coordination advice are composed incorrectly, this leads to invalid cached values or violations of exclusive regions. Indeed, Havinga et al. show how developers can adapt the JAMI framework to prototype a DSAL composition that resolves the conflicts.

Currently, non-embedded approaches are the only option for implementing a fine-grained composition of DSALs. However, the non-embedded approaches are not as light-weight and flexible as embeddings. For every DSAL composition, first, language developers have to implement a new parser, second, they need to invasively manipulate the code of the weaver, and third, they need to recompile the language infrastructure. Because the non-embedding approaches can only adapt the language infrastructure once, the current way of composing aspect languages cannot deal with an open set of DSALs that are composed later on in the user domain. For an open and flexible composition, it would be interesting to have systematic support for composing arbitrary DSALs without the demand to adapt the code of the AO mechanism. Unfortunately, there is a lack of support for fine-grained composition that deals with such interactions.

2.3. Architectures for Language Implementation

Traditionally, DSLs are implemented as pre-processors, interpreters and compilers that are not extensible. The resulting DSL implementation is rather monolithic and developers cannot extend it without changing their internals. With such monolithic architectures, it is practicable not possible to compose interpreters and compilers. To address this, the literature has proposed various language architectures to avoid monolithic language implementations. There are: (1) *front--end/back-end architectures*, (2) *extensible language architectures*, (3) *meta-level architectures*, (4) *reflective architectures*, and (5) *meta-object protocols*. We briefly explain these architectures because this thesis is based on the terminology, structures, and techniques they use.

2.3.1. Front-End/Back-End Architectures

Modern languages organize their infrastructures into a *front-end* and a *back-end*. Often, front-ends are implemented as compilers that compile programs into an *intermediary representation*, such as *bytecode*, which is a machine-code like representation that is prepared for execution. Back-ends are often implemented as *virtual machines* (VM) [SN05] that are special software systems that model an abstract machine on which intermediary program representations are executable. Splitting into front- and back-ends allows a greater reuse of the back-end. For example, Java splits into the Java compiler, which compiles Java source code into Java bytecode, which then is executed by a Java VM (JVM). Because of using bytecode, whenever a new Java compiler front-end became available, often the old JVM back-end implementation could be reused.

2.3.2. Extensible Compilers

Extensible compilers of GPLs allow defining domain-specific extensions to GPLs. Examples of extensible compilers are: *Polyglot* [NCM03], *JastAdd* [EH07b], or the *Java syntactic extender* [BP01], *LISA* [MLAZ00], or *Silver* [VWBH06, WKBS07, VWBGK08]. Most extensible compiler are not language independent (e.g., restricted to Java). They have a fixed AST representation (e.g., an Java AST) that can be extended with domain-specific extensions (e.g., tuples). They allow rather little incremental extensions to the AST, but they are not adequate for replacing large parts of the AST. Most extensible compilers do not support composing ASTs of different languages. When implementing a new extensible DSL, therefore, a new extensible compiler must be implemented, which one can expect as a rather large initial cost for implementing a DSL. There are only few extensible compilers (e.g., *JastAdd* [EH07b]) that use special techniques to overcome this limitation, but they require that developers need to be experts.

2.3.3. Meta-Level Architectures

A *meta-system* [Mae88, Glu91] is a computational system whose domain is another system, i.e., it integrates, controls, and processes other systems as objects. Meta-level architectures are used to implement language themselves. In general there is a meta-language in which another language is specified. The execution of the meta-language then creates a language implementation in the target languages, which conforms to the specification. This approach is often used by parser generators.

There are different flavors of meta-system architectures, such as *macro systems*, *code generators*, and *meta-programming* Examples for systems that support meta-programming are parser generators, code generators, C pre-processor, tree rewriting systems (e.g., ADF+SDF), C++ templates. Many meta-programming systems only support code generation in particular target languages, such as *HiveMind* and *Javassist* for Java.

In general, when extending a language implementation, a problem with meta-level architectures is that even with a little change to the specification of the language under development, the developer has to regenerate and/or recompile the complete language implementation, which is disruptive. When developers are disallowed to make extensions in the generated code, a language cannot further be changed, e.g., for customization in the user domain, once the target code has been generated.

2.3.4. Reflective Architectures

A *reflective system* [Smi84, Mae87] is a meta-system whose domain integrates, controls, and processes itself as objects. Smith defines a reflective system as *"computer system able to reason about itself"* [Smi84] and that the system's self-representation needs to be *causally connected* to its domain. Maes explains that a *"system is said to be causally connected to its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect upon the other"* [Mae87]. Unfortunately, reflective architectures are only widely available for GPLs, but not for DSLs.

2.3.5. Meta-Object Protocols

The concept of a *meta-object protocol* (MOP) [KRB91] has evolved from reflective languages. In contrast to reflective architectures, which provide rather low-level means for adaptations of programs, a MOP provides a high-level means for adaptations of the OO language semantics. There are special *meta-objects* that are responsible for the interpretation of objects, and that developers can use for adapting the behavior of their objects. With meta-objects, developers can provide adaptations to objects in retrospect.

A MOP follows the *open implementation* [Kic96] design principle by exposing part of the implementation strategies of the OO language implementation. In other words, a meta-object protocol is an open implementation of an object-oriented language. A system built according to the open implementation principle provides two orthogonal interfaces: the so-called *primary interface*, exposes to applications the primary functionality of the system. The other interface, called the *meta-interface*, provides application-level access to the implementation strategy of the primary-interface. Using the meta-interface allows users to change the implementation strategy behind the primary interface that is hidden for applications. This principle allows system designers to design their system open for (unanticipated) adaptations later on.

In a MOP, the meta-objects provide high-level and reflective operations, called *meta-methods*, that interpret part of an object's semantics at the *meta level* (e.g., method dispatch). The MOP defines all workflows as sequences of calls that interconnect the meta-methods that participate in one extensible part of the language's semantics. Note that, instead of static OO language semantics, a MOP-based language implementation covers a range in the design space of language semantics rather than representing only a single point in the spectrum of possible OO semantics. Application programmers can use the MOP to create new *variant languages* that meet their application-specific requirements. Hence, the distinction between language designers and users is blurred.

In particular, the technology that this thesis proposes uses MOPs and it is inspired by the flexibility MOPs provide for coping with adaptations of object-oriented languages. Existing work on MOPs focuses on using the open implementation principle for OO languages and special system. However, there is more potential to use open implementations in language design and implementation.

What is not addressed by existing MOPs is to adapt language semantics beyond OO semantics. This thesis argues that it would be interesting to provide a similar flexibility that MOPs provide for OO languages to other languages, such as DSLs. With such flexibility, language end users could adapt the semantics of DSL constructs.

What is missing is a general approach that describes how to implement a meta-protocol that allows adapting the back-ends of a DSL implementation. It would be interesting to have executable DSLs with a meta-level that allows: (i) analyzing and (ii) adapting their domain models and constructs, or (iii) adapting the execution semantics. With such flexibility, language end users could adapt strategies in a DSL implementation, which would enable evolutions of DSL semantics in the user domain, like MOPs enables adapting OO semantics.

3. Desirable Properties for Language Embeddings

This chapter discusses desirable properties for language embeddings. The proposed set of desirable properties either has been identified by related work or identified as part of this thesis to address open problems of language embedding approaches. For each desirable property, we discuss one or more non-trivial scenarios that exemplify the need for the corresponding property. These scenarios have been found studying the history of an existing language—the *Logo* programming language. The author has selected *Logo* because it is a small language that encompasses both general-purpose and domain-specific abstractions. The central question we will ask for each scenario is why the corresponding desirable property should be supported by a language embedding approach, and how the property helps implementing new languages and extending existing ones in a modular and reusable manner.

Parts of this chapter have been published in the following paper:

- Tom Dinkelaker. **Review of the Support for Modular Language Implementation with Embedding Approaches.** TUD-CS-2010-2396, Technische Universität Darmstadt, Germany, November, 2010.

In this chapter, first, we give a general overview of the *Logo* programming language that will be used to exemplify the properties. Then, we discuss the properties and their corresponding scenarios.

The *Logo* Programming Language. The *Logic Oriented Graphic Oriented* language [FPB$^+$70] (*Logo* for short)[1] is a programming language that was designed for a *constructive*[2] teaching of programming to children and non-professional programmers through interaction with a graphical system. The *Logo* environment involves a robot-like turtle creature with a pen positioned on a canvas. The turtle can be directed to move and paint on the canvas through a set of simple commands (e.g., forward, left, right, and backward). *Logo* has also been used to develop simulations, physical experiments, and to create multimedia presentations.

The first version of *Logo* (BBN Logo) was developed by Papert et al. [FPB$^+$70] as a special dialect of the LISP programming language family. There is no standard syntax for *Logo*,

[1]Details about *Logo*'s history can be found under: `http://el.media.mit.edu/Logo-foundation/logo`.

[2]The *constructivist method* is inspired by the Swiss psychologist Jean Piaget

in contrast, there are more than 245 dialects[3], which differ mostly in the platforms or in the language features supported. During its evolution, the different dialects and their versions have been extended with new language features to improve the level of abstraction and to provide new functionality. Example extensions are *subroutines* and *functions* [New73], object-oriented programming features[4], support for concurrently drawing turtles[5], as well as support for 3D graphics[6].

Consider a dialect called SIMPLELOGO that defines the most important commands of *Logo*. SIMPLELOGO exposes characteristics of complex DSLs: (a) *domain-specific literals*, (b) *domain-specific operations*, and (c) *domain-specific abstraction operators*.

Domain-Specific Literals. The dialect defines domain-specific literals to refer to basic colors, namely: **black**, **blue**, **red**, **green**, **yellow**, **white**.

Domain-Specific Operations. The dialect defines the following operations for drawing:

- **forward** n**:** Moves the turtle by n units in the direction of the current orientation.
- **backward** n**:** Moves the turtle by n units in the opposite direction.
- **right** a**:** Turns the turtle by a degrees to the right changing its current orientation.
- **left** a**:** Turns the turtle by a degrees to the left.

Domain-Specific Abstraction Operators. The SIMPLELOGO dialect defines the domain-specific abstraction operator **turtle** that is used to compose a sequence of commands. The command **turtle(name:**s**,color:**c**) {** *code* **}** creates a new turtle on the canvas. The created turtle has the name s and draws the *code* sequence that is provided inside the curly brackets using the given color c.

The left side of Figure 3.1 shows a SIMPLELOGO program. It defines a turtle named "Square", initializes the turtle with the color red, and passes to it a sequence of commands enclosed in the curly brackets. The program draws a red square whose edges have a length of 50 units, as shown on the right side of the figure.

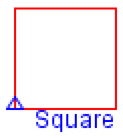
3.1. Extensibility

When there are new requirements for a language, i.e. the language evolves in time, a language developer needs to extend the language's implementation. To cope with changing requirements,

[3]An overview of various *Logo* dialects is given by the Logo Tree Project: `http://elica.net/download/papers/LogoTreeProject.pdf`, last visited (26. Nov 2010).
[4]For example, ObjectLogo is an OO-extension of *Logo*: `http://www.digitool.com/ol-specs.html`.
[5]For example, concurrency support is available in StarLogo: `http://education.mit.edu/starlogo/`.
[6]For example, three dimensional drawings are possible in Logo 3D: `http://sourceforge.net/projects/logo3d/`.

```
turtle{name:"Square",color:red) {
  forward 50
  right 90
  forward 50
  right 90
  forward 50
  right 90
  forward 50
  right 90
}
```

(a) The source code



(b) The output (line color is red)

Figure 3.1.: An example SIMPLELOGO program

language implementation approaches should support extensibility [Ste99]. In general, a language implementation approach is said to support extensibility if it allows the developers to extend their language implementations [Hud96, EH07a].

When looking at the history of *Logo*, new *Logo* infrastructures have been implemented from scratch again and again without reusing existing implementations, despite the broad similarity in their implementations of the basic language features in the various *Logo* dialects. In retrospect and w.r.t. reuse of implementation efforts, it would have been desirable if these dialects had been implemented in an extensible way reusing available language features—not implementing them from scratch.

Non-embedded language approaches have proposed various techniques for language extensibility [Hud96, Par93, NCM03, EH07a]. In these approaches, a *base language*[7] is extended with new language features that form a so-called *extension* to the base [NCM03]. The most important benefit is that the language features of the base language must not be re-implemented in the extended language [vDKV00, MHS05].

When extending a language, we can distinguish two kinds of extensions w.r.t. what facet of the language is extended: extensions that add new keywords to the language's syntax, and extensions that do not add new keywords but that extend the language's semantics. Each kind of these extensions is discussed in the following.

3.1.1. Adding New Keywords

An important form of *language evolution*[8] is to add new language keywords, or respectively language constructs, to an existing language [MHS05]. Looking at the evolution history of *Logo*, this language has started with a very limited set of keywords [FPB$^+$70] (similar to SIMPLELOGO). Since its early days, more keywords for richer forms of interactions have been proposed resulting in new versions of the *Logo* dialect.

[7]The term *base language* should not be confused with the term *host language*. A language extension is related to its base language, which is extended by the extension. In contrast, an embedded language is related to its host language, which hosts the library of the embedding.

[8]Language evolution is the process when one version of a language evolves into a new version, which involves adding, refining, and removing expression types of the language's syntax and semantics.

For such continuous language evolutions, an implementation approach needs to support *extensibility* with which developers can build incremental extensions to existing languages for adding new language constructs [KAR$^+$93, Ste99, Vis08]. Supporting this *incremental* extensibility is particularly important for domain-specific languages, because they evolve more frequently than general-purpose languages [MHS05]. When new keywords are frequently added to a language, it is beneficial if the language implementation approach supports incremental extensibility [MHS05, Vis08].

Example Scenario: In the following, we discuss three extensions to SIMPLELOGO that are motivated by existing *Logo* dialects. While from perspective of an end user, each extension has its own motivation, from the perspective of a language developer, extending SIMPLELOGO with each of those extensions is technically not very different. Although the three extensions are admittedly similar, we discuss them here, since they reflect the fact that there are often multiple extensions to a base language, and that such distinct paths of evolutions need to be consolidated later.

As the first example extension, consider a *Logo* dialect called EXTENDEDLOGO, which extends SIMPLELOGO with keywords found in many *Logo* dialects. Examples are the keyword **setpencolor** that allows selecting the current paint color for a turtle. The keyword **home** moves a turtle back to the initial position at the center of the canvas. The keyword **clearscreen** clears all painted pictures from the canvas. After using the keyword **penup**, the turtle does no longer paint on the canvas when moving until **pendown** is used. The keyword **hideturtle** allows hiding the physical body of the turtle (often displayed as a triangle), and **showturtle** shows the physical body again. This extended set of keywords enables more sophisticated interactions in the domain of turtle graphics.

As the second example extension, consider adding shortcut commands to SIMPLELOGO. Shortcut commands are aliases to the corresponding verbose commands. A dialect called CONCISELOGO extends SIMPLELOGO with various shortcut keywords, such that **forward** can be written as **fd**, **backward** as **bd**, **right** as **rt**, and **left** as **lt**. Shortcut commands allow writing programs in a more concise way. For example, although the program version with shortcut commands in Listing 3.1 paints the same square, its code is more concise than its verbose version in Figure 3.1a.

As the third example extension, consider adding new abstraction operators, such as the repeat-loop construct that has been proposed by the *UCBLogo*[9] dialect. The operator **repeat (**n**) {**$body$**}** takes two parameters: n is an integer that defines how often the code sequence in $body$ should be executed. Using an abstraction operator allows writing programs in a more abstract way by avoiding repetitive code. For example, the program in Listing 3.2 paints the square analogous to Figure 3.1 but it is written in fewer lines.

When language extensions are introduced as new versions of a language, continuous evolution of a language demands *incremental* extensions of the language. For example, a language developer can incrementally extend SIMPLELOGO by adding the new keywords from the EX-

[9]The Berkeley Logo (UCBLogo) Homepage: `http://www.eecs.berkeley.edu/~bh/logo.html`.

```
turtle{name:"Square",color:red) {
  fd 50
   rt 90
  fd 50
   rt 90
  fd 50
   rt 90
  fd 50
   rt 90
}
```

Listing 3.1: A program using the shortcut commands of the CONCISELOGO dialect

```
turtle(name:"Square", color:red) {
  repeat (4) {
    forward 50
    right 90
  }
}
```

Listing 3.2: A program using the repeat abstraction from the UCBLOGO dialect

TENDEDLOGO, UCBLOGO, and CONCISELOGO extensions. By chaining the three extensions to extend each other, they can be integrated into one dialect, called COMPLETELOGO, that encompasses all keywords.

Requirements: To cope with continuous language evolution, a language implementation approach should support incremental extensions of syntax and semantics. It is important that incremental extensions do not only support adding new keywords but also overriding existing ones. Only when there is an extension mechanism that enables language developers to precisely select what parts of existing language implementations they want to reuse, a language implementation approach can minimize the implementation efforts of the developers for extending languages.

3.1.2. Semantic Extensions

In contrast to adding new keywords, there is a need for semantic extensions that do not alter the syntax of a language, but only extend the language's semantics [KRB91, HORM08]. When extending a language's semantics, we can distinguish two classes of semantic extensions w.r.t. whether the existing semantics are preserved: *conservative* and *non-conservative* extensions. A *conservative semantic extension* does not change the meaning of existing language constructs in their base language. When the meaning of existing constructs is preserved, programs written in the base language still produce the same results when they are evaluated using the extended language. In other words, conservative extensions maintain backward compatibility. In contrast, *non-conservative extensions* can alter the meaning of existing constructs in their base language. Therefore, when programs written for the base language are evaluated using the non-conservative extension, the evaluation can produce different results. In that case, there is no *operational equivalence* [Fel90] between the extension and its base. The next two subsections discuss conservatives (Section 3.1.2.1) and non-conservative extensions (Section 3.1.2.2).

3.1.2.1. Conservative Semantic Extensions

To support backward compatibility for end user programs, it is desirable for a language implementation approach to support semantic extensions that do not change existing semantic invariants of their base languages.

Example Scenario: Consider a conservative extension that evolves *Logo* from sequential to parallel execution of commands. Since traditionally *Logo* commands are executed sequentially, COMPLETELOGO evaluates the code in Listing 3.3 in one sequence. The program draws a picture with 4 squares (with size 50 at the x/y-positions 0/0, 100/0, 0/100, and 100/100), whereby one turtle after the other paints the commands in its body in the sequence the commands are defined.

```
turtle (name:"Square1", color:red) {
  //stay at (x=0,y=0)
  repeat (4) { forward 50; right 90; }
}

turtle (name:"Square2", color:red) {
  penup(); right 90; forward 100; left 90; pendown(); //goto (x=100,y=0)
  repeat (4) { forward 50; right 90; }
}

turtle (name:"Square3", color:red) {
  penup(); forward 100; right 90; forward 100; left 90; pendown(); //goto (x=100,y=100)
  repeat (4) { forward 50; right 90; }
}

turtle (name:"Square4", color:red) {
  penup(); forward 100; pendown(); //goto (x=0,y=100)
  repeat (4) { forward 50; right 90; }
}
```

Listing 3.3: A logo program with four turtles

Today, concurrent versions of *Logo*[10,11] have been proposed [Res90]. Therefore, consider an evolution of SIMPLELOGO extending the turtle operator that executes the command sequences of all defined turtles in parallel. In this case, it is desirable that a language developer can implement a concurrent version CONCURRENTLOGO as an extension to SIMPLELOGO. While reusing the implementation of the color and drawing operations, the language developer only needs to refine the turtle abstraction operator of the SIMPLELOGO implementation, so that the enclosed command sequence is executed in a separate *thread*. Note that while CONCURRENT-

[10]The StarLogo Homepage: `http://education.mit.edu/starlogo/`.
[11]The NetLogo Homepage: `http://ccl.northwestern.edu/netlogo/`.

LOGO semantically extends SIMPLELOGO, this extension does not change the base language syntax as no additional commands are defined. For example, when executing the same code from Listing 3.3 with CONCURRENTLOGO, the turtles will paint the four squares concurrently.

Requirements: When providing conservative semantic extensions, language developers want to be sure that extensions preserve all semantic invariants. E.g., when executing a program in an extension that parallelizes, it must be ensured that according to invariants the outcome of the program is not changed according. In case turtles with different pen colors paint concurrently, a parallel execution must ensure that the program results in the same picture, as if it would paint sequentially. It is not trivial for CONCURRENTLOGO program to ensure the correct order between concurrent paint operations. Since the turtles act autonomously, it is hard to predict the actual path of a turtle, and paths of different turtles may cross each other. When paths cross, painting them in different orders can result in differences in the output. Such an incorrect parallelization violates the expected sequential painting semantics of *Logo*. To ensuring correct semantics, it is necessary to detect, prevent, or resolve violations of the semantic invariants.

3.1.2.2. Semantic Adaptations

There are the special requirements for language implementations to be open for extensions in the user domain [KRB91]. This is in particular interesting in two special cases. First, the user's requirements for a language are not exactly known before delivering the language implementation to the user's domain. Second, if the requirements for a language are expected to change *late*, that means after the language implementation has been delivered, e.g. at runtime of a program. Since a language designer cannot foresee all possible end user requirements for such a language, its language implementation should be designed according to the *open implementation* principle [Kic96], which allows semantically adapting a language's implementation in the user domain by exchanging parts of its implementation strategies.

To provide support for adapting languages, variability needs to be built into language implementations. We call this variability of languages in the user domain *late variability*. Having support for such late variability in a language has a similar motivation like having support for late variability in other software systems with changing requirements [vG00, VGBS01]. The customizability, enabled through late variability in languages, would enable a better reuse and extensibility of language implementations in different end user domains. Late variability enables user-specific extensions to be provided even at runtime.

Unfortunately, existing language embedding approaches do not address late variability in their embeddings. Existing language embedding approaches use black-box extension mechanisms of their host languages, such as *functions* [Hud96, HORM08], *types* [OSV07, HORM08, Gar08], *traits* [HORM08], and *macros* [Pes01], but these extension mechanisms do not adequately support late variability in language implementation. With them, language embeddings can be extended, but there are several problems. First, in [Hud96], often black-box extension mechanisms do not support adapting the internals of an embedding for a new requirement. Therefore, language developers may have to invasively change the source code of the language implementation, leading to *copy-and-paste solutions* [Par08] with repetitive development efforts. For example, when using generative programming for the embedding [Gar08] and the developer adds a new constraint to a language, the developers have to regenerate the entire source code of the language implementation. Second, in [OSV07, HORM08], it is a problem when the developers have to change the language implementation at multiple places. For example, the code that enforces a constraint in the language implementation must be repeated for the execution semantics of every syntactical abstraction (e.g., for every command), which leads to unwanted repetitive code that is scattered and tangled with the code implementing the default execution semantics. Third, there are approaches [Hud96, Gar08] that do not allow adapting the language after its implementation has been compiled. For example, it is not possible for a language developer to add or customize constraints after the language has been compiled and delivered to the end user domain.

For better flexibility, it would be desirable to support late variability in a systematic way.

Example Scenario: When a language is used in a special context, there can be the requirement that a special constraint must be enforced. For example, when using CONCURRENTLOGO in a physical domain in which there are plotting robots on a canvas, we need to detect and prevent collisions of plotting robots before damage can happen. If collisions are not detected by the default language implementation, the language end user cannot use this implementation out-of-the-box.

A language developer could extend a language implementation with semantics that enforce constraints. But extending a language for a single customer is problematic since there are additional maintenance costs with each extension delivered to some customer. When single users have a special requirement, it is better to deliver an open implementation [Kic96] of the language so that a developer can customize for specific needs.

Therefore, consider customizing the language implementation of CONCURRENTLOGO such that it detects and prevents collisions between turtles in concurrent programs. A collision is defined to happen whenever there are more than one turtle at the same position and the turtles have their pens down to paint on the canvas. Collision in CONCURRENTLOGO programs likely happen when lines of the figures cross each other.

Another problem with enforcing such a domain-specific constraint is that it cannot be easily implemented as an incremental extension reusing the existing implementation, since it has to be enforced at all over the language implementation. For example, before moving a turtle on the canvas, it has to be determined whether moving the turtle will result in a collision. If a potential collision is detected, the potential collision must be prevented, e.g., by deferring the move operation of the turtle until the turtle that is in the way has moved on. The problem with adding code that enforces such a constraint is that language developers must adapt all moving commands. When the logic for an extension crosscuts several parts in its base language implementation, incremental extensions cannot localize this logic, consequently the maintenance costs increase.

Requirements: To address these problems, a language implementation approach should provide an extension mechanism that systematically enables open implementations of languages. On the one hand, a language developer needs to design a language in an open way, e.g. by allowing strategies in the language implementation to be replaced. On the other hand, an expert who knows the end users' requirements needs to customize the language implementation for end users, e.g. by replacing a default strategy through a more specific one.

3.2. Composability of Languages

When there are diverse requirements by groups of end users in different application domains, this often motivates having a specialized language for each domain. For better maintainability, as motivated in the previous section, it should be possible for specialized languages to evolve independently from each other in a hierarchy of extensions. But, since application domains of independent languages often overlap, language developers want to reconcile two or more specialized languages into one [BV04, OSV07, Par08]. Unfortunately, incrementally extending one of the specialized languages with others languages is not adequate for composing them, because the multiple extended languages share similar constructs, which leads to code duplication in the extension implementation [Par08].

To address this limit of hierarchical extensibility, several language implementation approaches have been proposed that support composability of languages [EVI05, KL05, Cle07]. For overlapping domains, the language developer can decompose languages into smaller reusable sublanguages, from which one can compose these sublanguages into new a language. We refer to such a composed language as a *composite language*, and we refer to the sublanguages as the *constituent languages*. When composing languages, the advantage is that language developers must develop each constituent language only once and that the implementations of constituent languages can be shared among several composite languages. But, with sharing, there can be interactions between languages, which can lead to conflicts that must be prevented. Therefore, in the following, we discuss composing languages with and without interactions.

3.2.1. Composing Languages without Interactions

At times problem domains overlap, i.e. the same language constructs are used in several languages for different domains. In such scenarios, it is desirable to reuse those constructs of their language implementations. To make constructs reusable, a developer can partition language constructs into exclusive and shared sets, whereby the developer implement the semantics for each set of language constructs in a language component. In many cases, the language constructs do not have any syntactic and semantic interactions, hence developers can implement the components independently.

Example Scenario: Assume there is the stand-alone language for *Logo* with its domain of turtle graphics, and another stand-alone DSL BUSINESSRULESDSL for writing business rules. Further, there are two similar extensions to these languages (e.g., MATHLOGO and MATHBUSINESSRULESDSL) that use the same constructs for a mathematical domain. Both extensions provide pre-defined operations (e.g., **sin** α) and abstraction operators for defining custom functions (e.g., **define** and **apply**) .

The mathematical domain overlaps with the other two, since mathematical operations help drawing figures in *Logo* and mathematical operations help writing business rules. Because of this overlapping, it would be interesting to make the shared mathematical operations in MathLogo available in line with MathBusinessRulesDSL.

Technically, hierarchical extensibility allows extending the two languages, but even if the extensions are identical for both base languages, they need to be re-implemented twice which leads to code duplication.

Instead of developing multiple extensions (e.g., the two mathematical extensions for each base language), the language developers can implement a reusable stand-alone language (e.g., MathDSL that is a stand-alone maths language) and compose it with other stand-alone languages (e.g., *Logo* composed with the MathDSL, and BusinessRulesDSL composed with MathDSL).

Therefore, consider composing *Logo* with the mathematical operations from the independently developed language Functional – a subset of MathDSL. We first introduce Functional, and then we explain its composition with *Logo*, called FunctionalLogo.

The Functional language provides a keyword **define**[12] to define *functions* by associating a name to a *lambda expression*. Further, Functional defines a keyword **apply** to call such functional abstractions by their name. Listing 3.4 shows a program in Functional that defines a subroutine. The command **define(name:**$routineName$**) { paramList... –>** $body$ **}** in line 1 defines a subroutine "pow2" that takes a number x and returns its square value. E.g. in line 4, applying "pow2" with 10 returns 100.

```
1 define(name:"pow2") { x ->
2   x * x
3 }
4 apply("pow2",10)
```

Listing 3.4: A Functional program defining a function

A composition of the above languages enables functional abstractions in *Logo*. This composition is natural, as many existing *Logo* dialects allow defining subroutines. In the resulting composite language FunctionalLogo, the end users can use both set of abstractions—for drawing and for defining functions.

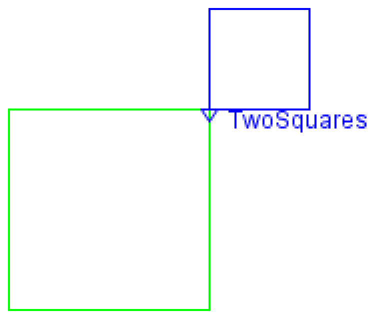
As an example program in FunctionalLogo, consider the code example in Figure 3.2a that uses abstractions from CompleteLogo and Functional. In line 1, the command define is used to define a function named "square" that paints a square figure with a variable length of its edges passed to the function as the parameter length. The function can be called by using the command apply that takes as the first parameter the function's name and a (variable-length) list of parameters passed on to the function. In the code in Figure 3.2a, the turtle paints a blue square (line 10) and a green square (line 13) using the function, as illustrated in Figure 3.2b.

[12]The define is inspired by Scheme [SS98].

```
define(name:"square") { length ->
  repeat (4) {
    forward length
    right 90
  }
}

turtle (name:"TwoSquares", color:red) {
  setpencolor blue
  apply("square")(50)
  right 180
  setpencolor green
  apply("square")(100)
}
```

(a) The source code



(b) The output (lower left square is green, upper right is blue)

Figure 3.2.: A FUNCTIONALLOGO program defining and using a function

Requirements: To compose stand-alone languages, such as COMPLETELOGO and FUNCTIONAL, a language developer needs to compose their syntax and semantics. For the sake of reusability, the language developer likely wants to reuse the existing implementations of the constituent languages for implementing the composite language.

3.2.2. Composing Languages with Interactions

When composing languages, the language developer has to compose their syntax and semantics in a correct way so that expressions have a well-defined meaning in the composed language. In case the languages' syntax or semantics are not orthogonal to each other, the languages cannot be composed straightforward. This is because there can be *syntactic* and *semantic interactions*, as we will discuss in the following.

3.2.2.1. Syntactic Interactions

For composing the syntax of constituent languages, the syntax of each embedded language must be integrated into the composite language in a consistent way. But when the syntax of one language is incompatible with the syntax of another language, there is a *syntactic conflict*. Such conflicting languages cannot be composed straight ahead.

Example Scenario: For example, consider composing two non-orthogonal languages: FUNCTIONAL with another language STRUCTURAL that allows defining data types. Say the two languages have been developed independently, but accidentally both languages use the keyword **define**. While FUNCTIONAL uses define to define new behavioral abstractions, STRUCTURAL uses define to define new structural abstractions.

Further, consider the program in Listing 3.5 that uses the ambiguous keyword define in 1 and 4 with two different intentions. There is a syntactic conflict, since those two languages' sets of keywords are not disjoint. Because of their conflicting syntax, it is not clear to which semantics the keyword define refers to—does the end user in line 1 (or resp. line 4) want to define a function or define a data type?

```
define(name:"tuple") { x = Integer; y = Integer; }
def t1 = init (name:"tuple",x:3,y:5);
...
define(name:"tupleSquare") { tuple ->
  return init (name:"tuple",x:(tuple.x * tuple.x),y:(tuple.y * tuple.y));
}
...
```

Listing 3.5: A program that uses the ambiguous keyword define

Requirements: To support composing languages, it is desirable that an implementation approach can detect, resolve, and prevent syntactic conflicts in language compositions. In case of a conflict, e.g. when two languages define the same keyword, it is not clear which language implementation is responsible for evaluating the keyword. There need to be a mechanism that helps developers to declare a resolution of such syntactic interactions.

3.2.2.2. Semantic Interactions

When composing sublanguages, their semantics need to be composed correctly, so that expressions in the composed language always have a well-defined meaning. When non-orthogonal semantics of sublanguages are composed, the evaluation of a language construct in one language may affect evaluating a language's construct in another language. In this case, we speak of a *semantic interaction* between the constituent languages. When composing non-orthogonal sublanguages, it is not trivial to create a composed language from existing implementations [KL05, KL07b], since interactions between the constituent languages can be unintended. If an interaction is unintended this can lead to unintended composition semantics, in that case, we speak of a *semantic conflict*.

Example Scenario: For example, consider composing the three languages: FUNCTIONAL, STRUCTURAL, and CONSOLE. FUNCTIONAL and STRUCTURAL both come with an environment. In the environment of FUNCTIONAL, names are bound to lambda expressions, and in the environment of STRUCTURAL, names are bound to data types. In CONSOLE, there is an expression **writeln** id that takes the identifier of an object and prints out a string representation of that object.

In a composition of the three languages, language end users can use functional and structural abstractions to define *abstract data types* (ADT), where a set of functions is defined on data types. When composing the language implementations, it must be possible that the implementation of one language in the composition can access the environment of other language implementations.

Note that, in such a composition, there are semantic interactions, e.g. in the implementation of writeln. E.g., consider the dependency that for implementing writeln, the CONSOLE language must access the environment of FUNCTIONAL and STRUCTURAL languages in order to read their objects. When a language developer identifies a dependency between constituent languages, the developer may need to extend the language implementations (e.g. FUNCTIONAL and STRUCTURAL in order to access their environments).

Further, there can be semantic conflicts, e.g. when a program defines a function and a type with the same name and an abstract data type definition uses that name, because it becomes unclear to what the name refers to. E.g., consider the following example conflict. There is an ADT list with elements and a function sort that takes a list and returns a list with sorted elements. In a different part of the application, there is a data type sort to model a partition of sets of elements into different classes. When the program uses the expression writeln "sort", it is unclear whether the identifier "sort" refers to the sort function or to the sort type definition.

When a language developer identifies such a semantic conflict between the two constituent languages FUNCTIONAL and STRUCTURAL in their composition specification, the developer cannot reuse stand-alone implementations of the two languages straight-ahead. The conflicting constituent languages FUNCTIONAL and STRUCTURAL can still be composed by tailoring a well-defined composition of their implementations. E.g., the composite language can have the special semantics that the identifiers of all composed language must be unique. Note that such a composition must enforce unique names in all constituent languages. The composition prevents the conflict, since it is no longer possible to define a function and a data type with the same name.

Requirements: For composing languages with semantic interactions, language embedding approaches should allow intended interactions and prevent unintended ones. For conflict cases, approaches should allow detecting, preventing or resolving semantic conflicts that can lead to incorrect compositions with unintended execution semantics. When a composition implementation needs to enforce constraints in the constituent languages, often the composition needs to adapt the constituent language implementations.

In general, one cannot assume that language implementations that have been designed and implemented in isolation are designed with compositions in mind. When two languages are composed, there always can be subtle side effects in parts of the semantics that lead to unintended interactions. Despite this, developers can extend the implementations of all independent languages by specializing them for such a composition. Each extension needs to provide an explicit interface that allows defining how existing implementations can interact with it. When extending the constituent language implementations for a composition, each constituent language should control what parts of its context it exposes to other languages and what parts of the context of other languages it takes into account for its own semantics [KL05].

3.3. Enabling Open Composition Mechanisms

For one particular composition scenario, often its constituent languages can be composed in a pre-defined way. To facilitate implementing certain scenarios, a language implementation approach should support a declarative composition of languages.

For each scenario and types of interactions, there are special requirements and assumptions by a mechanism about the kind of languages it can compose and interactions it can handle. Only when all requirements and assumptions by a mechanism are met in a composition scenario, language developers can use the mechanism for composing the languages. Since the set of possible scenarios is open, there can be special requirements and assumptions in new composition scenarios. Therefore, developers should be allowed to define new composition mechanisms. Since there is a cost associated with defining a new mechanism, the existing composition mechanisms should be extensible in order to foster reusing common parts in the composition logic.

The following sections motivate language-composition mechanisms that help controlling syntactic interactions in Section 3.3.1, and semantics interactions in Section 3.3.2.

3.3.1. Open Mechanisms for Handling Syntactic Interactions

Composition mechanisms that allow controlling syntactic interactions can be classified according to how conflicts are handled. There are those composition mechanisms that enforce that composition must be conflict free, discussed in Section 3.3.1.1. Further, there are the composition mechanisms that resolve conflicts in a certain way, namely by renaming conflicting keywords discussed in Section 3.3.1.2, and by prioritizing expression types of the languages in a composition discussed in Section 3.3.1.3.

3.3.1.1. Generic Mechanism for Conflict-Free Compositions

When independent languages are composed, a language implementation approach should prevent syntactic conflicts [BV04, KL07b].

Example Scenario: For example, let us compare two of the aforementioned composition scenarios. When composing SIMPLELOGO and FUNCTIONAL, the composition FUNCTIONALLOGO is syntactically correct because the two languages have disjoint sets of keywords. In contrast, when composing FUNCTIONAL and STRUCTURAL, there is the syntactic conflict because both languages use the keyword define.

Requirements: To support safe compositions of independent languages, there should be a composition mechanism that automatically detects syntactic conflicts and provides feedback to the language developer. E.g., when trying to compose FUNCTIONAL and STRUCTURAL, the mechanism should report the conflicting keyword define. To allow language developers composing various languages in conflict-free compositions, the composition mechanism should be *generic*, i.e. the composition logic to detect conflicts should not be specific to particular languages.

For cases in which there are special requirements on a language composition, the generic composition mechanism should be open for extensions. When a language developer composes several language, the developer may decide to restrict the lexical regions in which certain keywords can be used. E.g., when a developer composes FUNCTIONAL and COMPLETELOGO, the developer declares that it is forbidden to use the COMPLETELOGO keywords in a function body. Preventing using COMPLETELOGO keywords in functions can be interesting, because these keywords produce side effects and make functions impure. If a program uses a lexically restricted keyword in a wrong lexical region, we refer to this as a *context-sensitive syntax conflict*. To prevent context-sensitive syntax conflicts, the language developer must have the possibility to specialize generic composition logic for handle those conflicts by taking into account the keywords' contexts and the constituent languages.

3.3.1.2. Supporting Keyword Renaming

When languages with syntactic conflicts are composed, a language implementation approach can resolve such conflicts by adjusting parts of the syntax for a composition, e.g. by overriding one of the conflicting expression types [Cor06].

Example Scenario: For example, for a meaningful composition of FUNCTIONAL and STRUCTURAL, the language developer can rename the define keywords. The developer can declare to rename the conflicting keywords into defun for FUNCTIONAL and make for STRUCTURAL, so that using these renamed keywords in programs is unambiguous.

Requirements: To support composing languages with syntactic conflicts, there should be a composition mechanism that enables the language developer to declaratively resolve the interactions by changing the conflicting parts in the syntax. When adjusting parts of the syntax, one has to keep in mind that the composite language may not be backward compatible to the old syntax. Existing programs are liekly to be no more working, until their expression's keywords also have been renamed to the new syntax.

3.3.1.3. Supporting Priority-Based Conflict Resolution

Often, language extensions are implemented independently from each other, still the language developers can plan for composing extensions with the base. Composing several extensions to the same base language is easier than composing stand-alone languages, because the extensions can rely on the same base-language theorems. When extensions have a common base, even when there are interactions, conflicts are less frequent. Therefore in such compositions, often it is sufficient to resolve conflicts using a priority.

Example Scenario: For illustration, assume that UCBLOGO, CONCISELOGO, and EXTENDEDLOGO are developed as independent extensions. But in contrast to incrementally adding each extension into a chain to create COMPLETELOGO, as proposed in Section 3.1.1, assume that each extension directly extends SIMPLELOGO and we do not want to build additional incremental extensions to those extensions. In this case, it would be more economically to compose COMPLETELOGO from the three extensions CONCISELOGO, UCBLOGO, and EXTENDEDLOGO, so that we can use e.g. loop abstractions and shortcuts together in DSL programs, as shown in Listing 3.6. In such a composition, language developers likely want to reuse the existing implementations of these extensions.

Requirements: When there are multiple extensions to a shared base language, a language implementation approach should support composing the implementations of those extensions [KL07b, EH07a]. In contrast to composing stand-alone languages, interactions between extensions and their base language are easy to resolve. For example, the expression types of base language are available in all extensions of SIMPLELOGO, e.g. forward is available in UCBLOGO, CONCISELOGO, and EXTENDEDLOGO. In conflict-free stand-alone languages, such common

```
turtle (name:"Square", color:red) {
  repeat (4) {
    fd 50
    rt 90
  }
}
...
```

Listing 3.6: An excerpt of a COMPLETELOGO program that uses the shortcuts of CONCISELOGO and the repeat-loop construct of UCBLOGO

keywords are disallowed for a conflict-free composition, since keyword semantics would not be well-defined. In contrast, because the extensions have a common base, sharing the common keywords of the base language is not a conflict, since the interaction can be resolved.

A possible resolution to compose languages that have syntactic interactions is to prioritize the constituent languages, which always uses the keyword semantics of the language with the highest priority. Another approach is to declare an individual priority for each expression type. By ordering the expression types, they become unambiguous again.

3.3.2. Open Mechanisms for Handling Semantic Interactions

When semantics are non-orthogonal, it is not straight-forward for a language developer to compose constituent languages. Therefore, it is desirable to have composition mechanisms that support the developer in scenarios by providing common logic for handling particular kinds of semantic interactions.

Semantic interactions can be generic or specific. To illustrate these two, this section motivates two important examples for such semantic interactions. In Section 3.3.2.1, the first example – cross-cutting composition – is a kind of semantic interaction that is useful for combining various languages, and therefore this asks for a general mechanism to be provided by the DSL approach, which is independent of the particular language to be composed. The second example in Section 3.3.2.2 resolves a composition conflict between two particular languages, and therefore this asks for a specific solution, which works only for those particular two languages. For compositions of semantically interacting languages, a DSL approach should allow developers to provide both generic and specific composition mechanisms.

3.3.2.1. Generic Mechanism for Crosscutting Composition of DSLs

Existing language embedding approaches focus on composition scenarios where the use of abstractions from one domain does not affect the evaluation of abstractions from another domain. We refer to such non-interacting compositions as *black-box compositions*, since they compose languages using black-box abstractions. The problem with black-box compositions is that, when multiple DSLs with crosscutting concerns are composed, programs exhibit scattering and tangling symptoms, as elaborated below.

Example Scenario: Consider a black-box composition scenario in which language semantics to be composed are orthogonal to each other. We show that if only black-box composition is supported, programs that span over several domains might exhibit scattering and tangling symptoms if the domain semantics are crosscutting.

The *Logo* extension, called TRACINGLOGO, allows recording the trace of turtles, which enables end users to debug *Logo* programs. The extension provides an operation **trace()** that prints out the most relevant information about the current turtle, encompassing its name, its x/y position on the canvas, and its color. Further, the extension provides another operation **show** str to write a string to the console.

```
define(name:"polygon") { length, edges ->
# show "begin of evaluation of function 'polygon'"
# trace()

# i = 0;
  repeat (edges) {
#   show "begin of "+i+"-th iteration of repeat"
#   trace()

#   show "forward "+length
    forward length
#   trace()

#   show "right "+360/edges
    right (360/edges)
#   trace()

#   show "end of "+i+"-th iteration of repeat"
#   trace()
  }

# show "end of evaluation of function 'polygon'"
# trace()
}

turtle(name:"Octagon",color:green) {
# show "begin of evaluation of turtle 'Octagon'"
# trace()

  polygon 8, 25

# show "begin of evaluation of turtle 'Octagon'"
# trace()
}
```

Listing 3.7: A program with a tracing concern

The code presented in Listing 3.7 records the trace of the program that is shown in Listing 3.8. Although the tracing concern is implemented using the domain-specific keywords, the concern is not localized at a single region in the code. As indicated by the lines highlighted with "#", the tracing logic is scattered over the function and turtle abstraction, and tangled with the functional concern drawing the octagon.

```
begin of evaluation of turtle 'Octagon'
Octagon: x=0, y=0, orientation=0, color=green, ...
before applying function 'polygon'
begin of evaluation of function 'polygon'
Octagon: x=0, y=0, orientation=0, color=green, ...
begin of 1-th iteration of repeat
Octagon: x=0, y=0, orientation=0, color=green, ...
forward 25
Octagon: x=0, y=25, orientation=0, color=green, ...
right 45
Octagon: x=0, y=25, orientation=45, color=green, ...
end of 1-th iteration of repeat
...
begin of 8-th iteration of repeat
...
end of 8-th iteration of repeat
end of evaluation of function 'polygon'
Octagon: x=0, y=0, orientation=0, color=green, ...
Octagon: x=0, y=0, orientation=0, color=green, ...
after applying function 'polygon'
end of evaluation of turtle 'Octagon'
Octagon: x=0, y=0, orientation=0, color=green, ...
```

Listing 3.8: An excerpt of a program trace

Requirements:

The scattering and tangling symptoms are not restricted to the above DSL, but also other DSLs suffer from these problems. Examples are *workflow languages* [CM04] (e.g., BPEL [AAB$^+$07]), *query languages* [Alm] (e.g., SQL [DD89]), *grammar specifications* [RMHP06, RMH$^+$06] (e.g., BNF or SDF2 [Vis97b]), and languages for modeling *finite state machines* [Zha06]. Although scattering and tangling is a general problem in DSLs, there is little research on aspect-oriented programming for DSLs.

To modularize crosscutting concerns in DSL programs, a language implementation approach needs to support compositions of DSLs in which the use of abstractions from one domain interacts the evaluation of abstractions from another domain. In such a language composition, the evaluation of expressions in one language (i.e. its evaluation protocol) is not longer a black box for the other languages. Therefore, we refer to such a composition scenario as a *crosscutting composition*.

For example, consider a crosscutting composition of the above DSLs by adding aspect-oriented concepts to *Logo* in a dialect called AO4LOGO. Listing 3.9 shows an aspect-oriented version of the program. The AO program implements the same functional concern and tracing concern as in Listing 3.7. But, the AO program does not suffer from scattering and tangling, since the code that implements the tracing concern is localized between lines 1–13, and is not mixed with the function concern between lines 15–24. Such a DSL with aspect-oriented concepts is called a *domain-specific aspect language* (DSAL). We will elaborate on the details of the DSAL programs in Chapter 7 and Section 10.3.2.2.

To enable such crosscutting compositions of DSLs, this requires a composition mechanism that allows declaratively specifying where and how languages may interact. To allow language developers reusing composition logic that is common for a composition of various dependent languages, the composition mechanism should be *generic*. Still, for cases in which there are special requirements, the generic mechanism should be *configurable* for a particular scenario which involves several semantically non-orthogonal languages.

```
 1  aspect(name:"TracingAspect") {
 2    before  (papply()) {  show "begin of evaluation of function  '$name'"; trace();  }
 3    after   (papply()) {  show "end of evaluation of function  '$name'"; trace();  }
 4
 5    before  (prepeat()) {  show "begin of $iteration−th  iteration  of repeat";  trace();  }
 6    after   (prepeat()) {  show "end of $iteration−th  iteration  of  repeat";  trace();  }
 7
 8    before  (pmotion()) { show "$command $args"; }
 9    after   (pmotion()) { trace();  }
10
11    before  (pturtle ())  {  show "begin of evaluation of turtle  '$name'"; trace();  }
12    after   (pturtle ())  {  show "end of evaluation of turtle  '$name'"; trace();  }
13  }
14
15  define(name:"polygon") { length, edges −>
16    repeat (edges) {
17      forward length
18      right  (360/edges)
19    }
20  }
21
22  turtle (name:"Octagon",color:green) {
23    apply("polygon")(8,25)
24  }
```

Listing 3.9: A program with a tracing aspect

3.3.2.2. Supporting Composition Conflict Resolution

When composing multiple semantically interacting languages, there can be composition conflicts that are complicated to resolve [KL07b, HBA08]. In general, since such composition conflicts must take into account the semantics of the application context, the system cannot automatically resolve such conflicts [Kni07].

Example Scenario: Consider composing languages with aspects not only from one domain but from various domains. For example, there are aspects for tracing/logging, security, caching, and persistence. It is a well-known fact [DFS02, SP06, KL07b, HBA08] that realization of these crosscutting concerns has conflicts, such as security can interfere with logging, caching can interfere with persistence, and so forth.

Requirements: For composing semantically interacting languages, a language implementation approach needs to detect such composition conflicts. Since the system cannot resolve composition conflicts automatically, the provided composition mechanisms should be open and configurable by end users i.e. the application developers.

3.4. Support for Concrete Syntax

Since most language embedding approaches are restricted to comply with the concrete syntax of the host language, they often do not allow defining adequate syntactic abstractions [BV04, Tra08]. This missing support for arbitrary concrete syntax for language embeddings is one of the biggest obstacles for their adoption [KLP$^+$08, MHS05].

There are numerous problems that hinder concrete syntax in language embeddings. First, in embedding approaches that do not support concrete syntax, the end users are required to encode program expressions in abstract syntax themselves, which is tedious and which could be avoided if there is a mechanism that automatically maps programs in concrete syntax to abstract syntax. Second, an embedded language generally cannot define or refine a keyword that already is defined in the host language. Hence, embeddings cannot define a new domain-specific semantics for existing host syntax. Third, tokens that use special characters, such as brackets, operator symbols, and Unicode symbols, cannot be freely used in embedded expressions. Often the special characters cannot be used for defining identifiers. Finally, while most host languages have only support for overriding prefix/infix operators or defining new prefix/infix operators, overriding and defining complex operators is not supported by host languages. In particular *mixfix operators* [Mos80, DN09]—a mixed form of prefix, infix, and suffix—are mostly not supported.

In the following, we will elaborate scenarios that require a support for concrete syntax.

3.4.1. Converting Concrete to Abstract Syntax

When the concrete syntax of a language is not directly supported in the host language, for embedding expressions of an arbitrary language, an embedding approach should allow encoding the concrete syntax of this language in abstract syntax [Fow05].

According to [Kam98, ALSU07], the *abstract syntax* of a language consists of two sets: *abstract syntax types* and *abstract syntax operators*. The type set is defined as $T = \{\tau_1, ..., \tau_m\}$. The operator set Θ consists of operators $\Theta = \{\sigma_1, ..., \sigma_n\}$, whereby each operator σ_i is a mapping $\sigma_i : \tau_{i_1} \times ... \times \tau_{i_j} \to \tau_k$ with a unique signature.

To represent basic expressions, the language developer needs to support an abstract syntax encoding for each kind of expression. Each kind of expression $e : \tau_k$ begins with its operator followed by its sub-expressions $e_1 : \tau_{i_1}$, ..., $e_j : \tau_{i_j}$ whose types match the operator's signature: $e = \sigma_i(e_1, ..., e_j)$.

The next paragraphs present an example of a formal encoding in abstract syntax and discuss what embedding approaches must support for such an encoding. Since an encoding in abstract syntax is verbose, it is desirable that end users do not need to encode their program in an abstract syntax themselves.

Example Scenario: For example, consider the following specification for the abstract syntax of SIMPLELOGO, which requires that the embedding approach supports the encoding of: (1) every kind of expression in abstract syntax, (2) sequences of sub-expressions, and (3) an entire program.

First, to represent SIMPLELOGO expressions, the language developer needs to define its abstract syntax types and operators. SIMPLELOGO has the types Integer, Color, String, Command, Turtle $\in T_{SimpleLogo}$. SIMPLELOGO has the color operators {black, ..., red} $\in C \subseteq \Theta_{SimpleLogo}$, where $\forall c \in C.c : \varnothing \rightarrow$ Color. It has the move operators {forward, backward, right, left} $\in M \subseteq \Theta_{SimpleLogo}$, where $\forall m \in M.m$: Integer $\rightarrow$ Command. It has the abstraction operator $turtle \in \Theta_{SimpleLogo}$, where turtle : String $\times$ Integer $\times$ CodeBlock $\rightarrow$ Turtle. E.g, a basic expression is forward(50).

Second, for compositionality of expressions, the encoding needs to support composition operators on expressions. E.g., the developer can define a family Φ of *sequence operators* of which each $seq_{\tau_{e_p},\tau_{e_q}} \in \Phi$ composes two sub-expressions e_p and e_q, which will be evaluated one after the other, whereby $seq_{\tau_{e_p},\tau_{e_q}} : \tau_{e_p} \times \tau_{e_q} \rightarrow \tau_{e_q}$. For example, we can compose the sub-expressions e_1 and e_2 with $e = seq_{\tau_{e_1},\tau_{e_2}}(e_1, e_2)$, whereby $\tau_e = \tau_{e_2}$.

Third, for encoding an entire program, the encoding needs to support conveniently writing sequences of expressions.

For convenience, the encoding defines short form for composing expressions into sequences, namely the semi-colon infix operator, which allows omitting seq keyword and the type indices. With the semicolon operator, one can write the sequence $seq(e_1, e_2)$ as $e_1; e_2$. In addition, the encoding defines another short form for chaining using a *recursive* sequence of sequences, e.g. $e = seq(e_1, seq(e_2, seq(..., seq(e_{n-1}, e_n))))$ as $e = e_1; e_2; ...; e_{n-1}; e_n$. Such a chained sequence is again an expression that has the type of the last expression of its sequence.

To encode nested expressions, the encoding defines a *code block* as a recursive sequence of expressions plus a *context* that contains *static* information (e.g. links to the enclosing code block) and *dynamic information* (e.g. variable bindings). In the encoding, one can write a code block using curly brackets, e.g., {e_1;e_2;...;e_{n-1};e_n}. A code block again is an expression that has the type CodeBlock.

Finally, the encoding wraps all program expressions into a code block. For example, consider Listing 3.10 that encodes the SIMPLELOGO example program in abstract syntax that was presented in Figure 3.1 (p. 31).

Requirements: To support abstract syntax for arbitrary embedded languages whose concrete syntax is specified as a *context-free grammar* (CFG), there must be two generic mappings. The first mapping needs to map concrete syntax of the embedding to abstract syntax. The second mapping needs to map abstract syntax to the host language syntax to make it executable.

For the first mapping, it is desirable that the mapping from concrete to abstract syntax is generic. That means it must not be specific to a concrete language embedding, but the trans-

```
{
  turtle ("Square",red,{
      forward(50);
      right(90);
      forward(50);
      right(90);
      forward(50);
      right(90);
      forward(50);
      right(90);
    }
  )
}
```

Listing 3.10: The abstract syntax representation of the program from Figure 3.1a

formation should be customizable for any language embedding. For this, the generic mapping is parameterized by the concrete syntax for language embedding, which consists of grammar productions of its CFG. The parameterized mapping then allows converting end user programs given in concrete syntax to an equivalent representation in abstract syntax. Even when end user must not read code in an abstract syntax, the encoding in the abstract syntax should be close the concrete syntax, so that one can revert the mappings for expressions in executable form to their abstract syntax and then to their concrete syntax, e.g. when we need to debug the program.

3.4.2. Supporting Prefix, Infix, Suffix, and Mixfix Operators

Most host languages used for embeddings have only restricted support for defining domain-specific operators [BV04, MHS05, Tra08]. Often it is only possible to override prefix and infix operators that are pre-defined in the host language. Still, it is desirable to support operators at every position, whether it is *prefix*, *infix*, *suffix*, or *mixfix*.

Example Scenario: Consider providing prefix, infix, and suffix operators on functions in FUNCTIONALLOGO and a mixfix operator used in an existing DSL:

Prefix: For example, one can define a prefix operator "**not(**$pred$**)**" that negates a predicate function expression $pred$.

Infix: Another example is to provide an infix operator to compose two functions "f **compose** g" to build a composite function h, where $f : X \rightarrow Y$, $g : Y \rightarrow Z$, $h : X \rightarrow Z$, such that $h(x) = f(g(x)), \forall x \in X$.

Suffix: An example for a suffix operator is "**(**k**)prime**", which can be used to refer to the derivative of a function k.

Mixfix: *Mixfix operators* [Mos80, DN09] have multiple operands whose positions are mixed with the keywords of the operator. Note that mixfix operators are of particular interest, because they are often used to define domain-specific abstraction operators. An example

for such an operator is the SELECT-statement in SQL [DD89]:
"**SELECT** c **FROM** t **WHERE** p", where c is a selection of columns, t is a selection of tables, and p is a predicate on the rows of the tables.

Requirements: More and more languages, allow the programmer to override pre-defined operators that use special characters, such as, +, -, *, /, !, «, such as Groovy. Some languages even allow arbitrary user-defined operators that are prefix, suffix, and infix. But, most languages used for embedding do not support the definition of mixfix operators. Thus, it would be desirable to have also support for mixfix operators.

Another challenge in many host languages, such as Java, Groovy, Scala, Haskell, is that Unicode cannot be used freely, e.g. it is disallowed by the host language parser to use Unicode in mixfix operators. Consider encoding the above operators in Unicode, such that "not($pred$)" becomes "$\neg pred$", "f compose g" becomes "$f \circ g$", "(k)prime" becomes "k'". For mixfix, consider defining a series "$\sum\ [1 \leq k \leq N]\ a(k)$" that denotes the sum $\sum_{k=1}^{N} a(k)$ over the finite sequence a. For more concrete syntax, it is desirable to support Unicode for DSLs [Sim95, MJ98, DN09]—ideally, at any place of a context-free grammar.

3.4.3. Supporting Overriding Host Language Keywords

It is desirable to freely use arbitrary keywords, operators, and delimiters in DSLs [BV04, BdGV06, FC09].

Example Scenario: Consider an embedding with DSL expressions that use the reserved keywords of the host language:

Textual keywords: It is desirable to allow defining new conditional expression types, such as "if (...) then {...} else {...}", even in case "if" is a reserved keyword in the host language and the existing conditional does not support the keyword "then", like in Java, Scala, or Groovy.

Delimiter keywords: It is desirable to allow defining user-defined delimiters, such as using curly brackets to define sets "{1,3,5}", even in case curly brackets are used in the host language to delimit code blocks.

Requirements: In general, the host language's parser disallows using keywords in embedded DSL expressions that are also reserved keywords in the host language. The problem is that most host language parsers disallow the reserved keywords to be used as identifiers. To still allow using such keywords in DSL expression types, it is required to transform DSL expressions that use those keywords into legal host language expressions.

Another challenge in many host languages is that delimiter keywords (such as, brackets) cannot be overridden. Often defining new delimiters or Unicode delimiters is also disallowed by the host language parser.

3.4.4. Supporting Partial Definition of Concrete Syntax

There are special embedding approaches [BV04, Tra08, KM09, RGN10] that support concrete syntax for DSLs by using *meta-programming* (cf. Section 2.3.3). They use meta-programming to create *meta-programs* that rewrite DSL programs in concrete syntax to host syntax. For this, the meta-program parses the program's expressions in concrete syntax, it creates an AST, and it rewrites the AST to equivalent expressions in the host language's syntax.

The problem with using meta-programming for embedding is that parsing requires the language developer to specify the complete syntax of an embedding. Specifically, the developer has to specify the complete DSL syntax, the complete host language, and how expression types of the two languages are integrated. However, defining a full-fledged concrete syntax for a language using a formalisms can be very expensive [vdBvDK$^+$96, Moo01, MHS05]. Providing a complete syntax definition for a large language is a tedious and error-prone task that can take several person months [vdBvDK$^+$96, vdBSV97, Moo01]. The large costs for the concrete syntax are a competitive disadvantage of these embedding approaches compared to traditional embedding approaches [Hud96, HORM08] that do not need to specify concrete syntax.

Example Scenario: To illustrate the overhead of requiring the presence of a complete syntax definition, reconsider the syntax of FUNCTIONALLOGO. Because *Logo* is a simple language, in many host languages even without using meta-programming, *Logo* can be embedded with an abstract syntax that is very close to the domain. Compared to traditional language implementation approaches, the advantage of the embedding approach is that the language developer does not have to define a formal syntax at all.

Consider that we would like to provide more concrete syntax for an embedding of functional composition. Instead of using the abstract syntax compose(...) from Section 3.4.2, we would like to use the ∘ Unicode operator. Because, in many host languages, such Unicode infix operators are not supported, we need to rewrite the expression in concrete syntax to host syntax (e.g., we can rewrite expressions like "$f \circ g$" into a corresponding encoding in abstract syntax such as "compose(f,g)'). But in current approaches, as soon as the concrete syntax of just one expression type is incompatible with the host syntax, for using meta-programming, the developer has to define the complete syntax of the embedding. Requiring the complete syntax increases the initial costs for embedding the DSL. Thus, language developers must provide the complete syntax, they must pay most of the costs for having a concrete syntax for the formalization of expression types that do not have a compromised syntax.

Requirements: In particular, for language embedding approaches, it is desirable to minimize the costs for concrete syntax. To better control the costs of having a concrete syntax, it would be interesting if language developers must not define the complete concrete syntax, but only for expression types where the concrete syntax is incompatible with the host syntax and that need to be rewritten. Further, it would be interesting if the developers did not have to define a complete concrete syntax before developing the semantics, but if instead they could add concrete syntax incrementally at the granularity of single expression types. When there is support for partial syntax, then developers can start with implementing an embedding with abstract syntax. The developers can deliver a full-functional language prototype with abstract syntax in a shorter time, and they can add concrete syntax later on. The benefit is that end users can already use the embedding with abstract syntax.

Later on, the developers still can add concrete syntax, step-by-step, for each expression type. Note that when embedded languages, such as SIMPLELOGO, in many cases, some expression types encoded in abstract syntax just happen to have exactly their concrete syntax. Only for a few expression types, the concrete syntax cannot be met. For these expressions, it is not always necessary to formalize the concrete syntax, since the syntax is already adequate. Therefore, it is beneficial if the developers do not have to define the complete concrete syntax. It is cheaper, if they only need to formalize the concrete syntax in cases where the abstract syntax is not adequate and needs to be replaced by concrete syntax.

3.5. Enabling Pluggable Scoping

In programming languages, a *scope* is a region in a program in which identifier are defined and bound to a value. A *scoping scheme* defines how such a *binding* of identifier propagates through the program. There are several dimensions with respect to how bindings are propagated [Tan09]: (1) *call stack propagation*: whether a binding propagates through the call stack or not, (2) *delayed evaluation propagation*: whether it propagates into nested abstraction operators or not, and (3) *activation*: whether it is active in a certain time frame or not.

When embedding a language, in traditional embedding approaches, the host language controls the scoping of language constructs. However, there are several cases in which the language developer needs to control the scoping of the language constructs [LLMS00, Tan09]. Therefore, in the following, we discuss scenarios that require controlling the scope of embedded language constructs.

3.5.1. Supporting Dynamic Scoping

Language embedding approaches inherit the semantics of their host language. Thus, they inherit the scoping rules of their host to resolve identifiers in expressions, which is a fixed scoping strategy for most host languages.

The two major scoping schemes in programming languages are *lexical scoping* (also known as static scoping) and *dynamic scoping*. While host languages with lexical scoping allow lexically-scoped language embeddings, dynamic scoping is not available to them and vice versa. Often supporting lexical scoping is sufficient for an embedding approach, but this is only because most of today's programming languages use lexically scoping.

However, there is a need to freely select the scoping strategy for an embedded language. For example, there are languages that use dynamically-scoped variables in lexically-scoped functions [LLMS00]. When we want to embed such a language with special scoping requirements, host languages that only support one closed scoping scheme cannot be used for the embedding. For such situations, language developers need to embed scoping schemes into a host language that are different from its native scoping scheme.

Example Scenario: Consider embedding a dynamically-scoped *Logo* dialect into a lexically scope host language. In a lexically-scoped host language, local variables are only bound within the enclosing lexical scope. That means, a local variable is only *visible* inside the body of an abstraction operator, but it cannot be accessed from outside.

With lexical scoping, when we define a variable inside an abstraction operator, the variable binding is only available inside this abstraction operator, but it is undefined or bound to other values in abstraction operators that are not enclosed. In contrast, with dynamic scoping, as indicated in Listing 3.11, it is possible that one abstraction operator defines a variable (e.g. as length in line 2) that other abstraction operators can bind to another value, even if those abstraction operators are not lexically enclosed (e.g. as in the body of the other turtle in line 9).

For example, with dynamic scoping, the variable length (line 4) has value 100, since line 9 overrides the variable binding in line 2 before calling the function. In contrast, with lexical scoping, the variable length (line 4) would still have the value 50, since the variable binding is resolved over the lexically-enclosing context that binds length in line 2. For dynamic scoping, it is necessary to persist the corresponding binding (of variable to its value) defined in an abstraction operator into a global context, so that the binding is available after the control flow of such a nested structure has been exited.

```
 1  turtle (name:"Teacher", color:red) {
 2    length=50
 3    define(name:"complexShape") {
 4      fd length; rt 120; fd length; rt 120; fd length; rt 120;
 5    }
 6  }
 7
 8  turtle (name:"Pupil", color:red) {
 9    length=100
10    apply("complexShape")()
11  }
```

Listing 3.11: Using a dynamically-scoped variable

Requirements: When a language embedding requires dynamic scoping, normally language developers select a dynamically scoped host language to embed it. Restricting the selection of possible host languages based on their scoping rules is problematic, since this restriction would disallow most host languages that are lexically-scoped to be used for the embedding. Further, when the embedded language needs both dynamically and lexically scoping, such as required in [LLMS00], host languages and embedding approaches that only support one closed scoping scheme can no longer be used. Therefore, as motivated above for *Logo*, it would be interesting to allow embedding dynamic scoping into host languages with lexical scoping.

3.5.2. Supporting Implicit References

There are languages that have *implicit references* [KM06] which are special references that are not user-defined but that are implicitly available in a certain context.

Frequently, GPLs provide special keywords for such implicit references that point to values that depend on the context the keyword is used in. Most prominently in OO languages, there are implicit references that are used to refer to objects in the current lexical context. For example, Java defines the keywords this to refer to the enclosing object and super to refer to its super class. Similarly, aspect-oriented languages define implicit references, too, such as AspectJ [Asp] defines the keyword thisJoinPoint that can be used in advice to refer to the current join point. Those primitive identifiers are special in that the host language defines a closed scheme how to resolve them.

Example Scenario: Such implicit references can also be defined by domain-specific languages. For example, consider an extension to FUNCTIONALLOGO that provides a special keyword thisTurtle that defines an implicit reference that points to the enclosing turtle. The keyword thisTurtle allows access to the turtle properties name and color. In Listing 3.12 in line 10, the expression "thisTurtle.name" prints out its name by accessing the turtle's name property via the thisTurtle keyword. In contrast, in line 14, Turtle2 prints out its color blue. The code in line 14 relies on the fact that every non-locally defined identifier will implicitly be resolved via thisTurtle, i.e. color is equivalent to thisTurtle.color.

```
 1 define(name:"print") { str ->
 2   //code that prints the "str" parameter
 3 }
 4
 5 define(name:"colorToStr") { color ->
 6   // returns name of the color
 7 }
 8
 9 turtle (name:"Turtle1",color:red) {
10   apply("print ")( thisTurtle.name)
11 }
12
13 turtle (name:"Turtle2",color:blue) {
14   apply("print ")( apply("colorToStr")( color ))
15 }
```

Listing 3.12: A FUNCTIONALLOGO program using the special name thisTurtle

Requirements: To enable an implicit reference, the language first needs to establish a special binding from the keyword name to the corresponding value in the enclosing scope. Second, the language needs to resolve implicit references in a special way—different from explicitly defined identifiers, such as variables or functions. Therefore, it is desirable to support embedding a special resolution scheme for implicit references. When the embedding of a resolution scheme is flexible enough, it can resolve references by taking into account the lexical or even the dynamic context the reference is used in.

3.5.3. Supporting Activation of Language Constructs

There are examples in which a language explicitly wants to control the activation [Tan09] of a certain language construct, such as dynamic extensions to classes and objects with *aspects* [PGA01] and *layers* [CH05]. With activation, one can explicitly control whether a certain language construct propagates or not, such as, the developer can declare that the extensions in an aspect or a layer are only effective in a certain region in the program.

In particular, activation is used in *dynamic aspect-oriented programming* [PGA01] to control the activation scope in which an aspect is effective. In dynamic AOP, a program can dynamically *deploy/undeploy* aspects to/from the running system, e.g., this allows activating or deactivating features those code crosscut several modules. There is good support for dynamic AOP for general-purpose programming languages, such as Smalltalk [Hir01, Hir09] or Java [PGA01, Bon03, BHMO04], But, unfortunately, dynamic AOP is not available for most DSLs. Today, dynamic AOP support must be enabled by invasively changing the code of the DSL execution infrastructure [CM04]. What is needed is a systematic language implementation approach that helps developing a dynamic AOP solution for an arbitrary DSL.

Example Scenario: Consider an evolution scenario that extends AO4LOGO with support for dynamic AOP, called LOGODYNAMICAOP. Listing 3.13 shows a dynamic tracing aspect. It implements the tracing concern similar to Listing 3.9, but the dynamic tracing aspect (Listing 3.13, lines 3–5) is not immediately active after it has been defined. This is because the aspect DynamicTracingAspect is defined with the parameter deployed:false. Therefore, when the polygon function is applied the first time in line 12, the evaluation of the function is not traced. Next in line 13, calling the method deploy activates the dynamic tracing aspect[13]. In contrast when the function is applied the second time, in line 14, the activated aspect traces the evaluation of the polygon function.

Requirements:

Note that there is the special requirement to optimize dynamic AO systems [HM04]. There can be a large overhead when executing programs with dynamic aspects. To achieve an acceptable performance with dynamic aspects, there need to be special optimizations that remove the overhead of aspects.

3.6. Enabling Pluggable Analyses

Traditional language implementation approaches support analyses, such as syntax checks, detecting redundant expressions, domain analysis and so forth. When it is possible to inspect the code of a program, this often called *intensional analysis* [COST04, ALY09]. Intensional analysis allows automatic reasoning on the program syntax and semantics. Normally, a parser makes a

[13]The code calls the method deploy on the reference dta that points to the aspect that was assigned in Listing 3.13 line 3.

```
DynamicAspect dta;

dta = aspect(name:"DynamicTracingAspect",deployed:false) {
   ... //Same aspect definition as shown in Listing 3.9.
}

define(name:"polygon") { length, edges −> ...
   ... //Same function definition as shown in Listing 3.9.
}

turtle (name:"Octagon",color:green) {
  polygon 8, 25 //evaluated without tracing
  dta.deploy()
  polygon 8, 25 //evaluated with tracing
}
```

Listing 3.13: A program with a tracing aspect

syntactic representation of a program available in form of an abstract syntax tree (AST). Traversing the AST nodes allows performing syntactic analyses that analyze the syntactic structure of a program. By storing information in AST nodes or by rewriting the AST nodes into another intermediary representation, it is possible to perform semantic analyses. Further, non-embedded compiler and interpreter approaches allow combining several modular analyses.

Language embedding approaches are said to not support sophisticated analyses for DSLs [MHS05]. This is because traditional embedding approaches do neither have a special parser nor is the AST of the host language available, the traditional approaches do not support such syntactic analyses. Although there are special embedding approaches that use exceptional features of their corresponding host language (such as meta-programming) to gain access to an AST representation, implementing an analysis with these approaches is rather complex. Therefore, the literature does not recommend to use current embedding approaches when domain-specific analysis or verification is needed [MHS05].

This is unfortunate, because syntactic analyses are an added value for their languages. They support the end users to automate analyses that would be tedious to check for humans. Although there are some embedding approaches that allow small syntactic analyses, these approaches only support analysis of languages with a closed syntax. Once a language developer has defined such an analysis, end users can apply it to analyze their programs. But, there are several limitations with evolving languages and when composing analyses.

To allow language embeddings to be used for scenarios where analysis is needed, it would be desirable to support extensible and composable *syntactic* and *semantic analyses*.

3.6.1. Syntactic Analyses

We would like to support *syntactic analyses* that enable reasoning over programs by analyzing their syntactic representation.

Example Scenario: Consider an analysis that checks that DSL code follows *coding conventions* [PK08] to ensure a good readability of source code. Tools for analyzing source code for coding conventions are widely available for general-purpose languages. Such analysis would be also desirable for DSLs, but often not supported by embedding approaches.

As an example coding convention, consider a domain-specific naming-convention that requires function names to be lower case. In an embedding, violations of such rules cannot be detected by the (generic) host language parser or compiler that is not aware of domain-specific properties.

Another example is checking that a certain threshold in a source code metric is not exceeded. Consider a convention in FUNCTIONALLOGO that a function body should not contain more than say 10 commands. Detecting such a metric helps to improve code quality through *cleaner code* [Mar08], that is better understandable and maintainable.

Requirements: When the syntax of a language evolves, the language developers must likely update all analysis implementations. Therefore, a language developer should be able to extend existing implementations of analyses instead of re-implementing them. Often multiple analyses need to be used in concert. Hence there should be support for implementing syntactic analyses in a modular way and for building compositions of multiple analyses.

3.6.2. Semantic Analyses

Before executing a program, often it is interesting to determine important properties of the program that abstract over concrete executions of the program. For determining properties of programs, traditional language approaches use *semantic analyses* that evaluate a program under abstract semantics [MHS05, HORM08, Par10], without actually executing the program under its default execution semantics. Such an alternative program evaluation is interesting because it enables *abstract interpretation* [CC77, Cou96]. Abstract interpretation is a well-established technique that is available for general-purpose languages to check programs for certain semantic properties and to use the retrieved information to find errors in a program, to improve the program's compilation and evaluation. Yet, for most DSLs, no tools for abstract interpretation are available.

There are various use cases for abstract interpretation of domain-specific programs. To identify possible bottlenecks in DSL programs, it is interesting to calculate runtime costs of DSL programs [WGM09], in particular, to support later optimization decisions. To help end users identifying errors in their DSL programs, there are semantic analyses that perform type checking, and there are other semantic analyses that verify domain-specific constraints. A language embedding approach should enable such analyses.

Example Scenario: Often in a domain, there is a *semantic contract* for a language that defines what a meaningful program is. Such a contract consists of a set of *domain-specific constraints* [SK97, GBNT01] that every program must comply with. The violation of such a constraint is considered to be a *semantic error*.

Consider a semantic contract for FUNCTIONALLOGO with a domain-specific constraint that requires that parameters to the *Logo* commands must be positive integer values. The program in Listing 3.14 has a semantic error that needs to be detected. There is an error because the parameter to the function application in line 10 will have a negative value that will be passed to the forward command in line 4, which violates the above constraint.

```
 1  define(name:"hexagon") { length ->
 2    turtle (name:"hexagon",color:red) {
 3      repeat (6) {
 4        forward length
 5        right 60
 6      }
 7    }
 8  }
 9  i = 100
10  apply("hexagon")(50-i)
```

Listing 3.14: A program with a semantic error

Requirements: To identify semantics errors in DSL programs, a language implementation approach should support abstract interpretations of domain-specific programs. Abstract interpretations can check DSL programs for possible violations of domain-specific constraints to ensure their correctness [vDKV00, CHR+03, PP08]. For determining whether a DSL program is meaningful, a pluggable semantic analysis can be used to analyze the program.

Such an analysis is based on an *abstraction function* that evaluates a program under *abstract semantics*. This function needs to check whether the program meets all constraints defined in the semantic contract of its language. For the above example, to determine whether there is a semantic error in the program, instead of execution the program, an analysis only needs to take into account that the values of the input parameters to the commands are not negative. Such semantic checks are particularly desirable because semantic errors can be hard to find in DSL programs, as they often require precise knowledge of the domain.

Often not only one semantic analysis is needed to ensure that a DSL program is correct, therefore a language embedding approach should enable composing several analyses.

3.7. Enabling Pluggable Transformations

A *program transformation* is a process that allows constructing a program by mapping it from one program representation to another [PS83]. A transformation allows transforming a program written for a certain language into another version of the program that is written in the same or in another language.

We can distinguish two types of transformations: *static* and *dynamic transformations*. *Static transformations* transform programs by taking only into account information from the static structure of a program. In contrast, *dynamic transformations* also take into account the runtime information, and thus they can be context-specific. In the following, we elaborate on the need for such transformations.

3.7.1. Static Transformations

We can classify static transformations into two classes: *syntactic transformations* that transform only the syntactic representation of a program and *semantic transformations* that change the evaluation of a program. We discuss a scenario for both classes.

3.7.1.1. Syntactic Transformations

Often when language end users write down expressions, they find themselves in typing redundant information that is clear from the context. One solution is to define syntactic sugar that allow language end users to express their intents more concisely. When language developers do not want to include a syntactic abstraction into the language itself, there is the possibility to add support for syntactic sugar to the language by defining a transformation that *desugars* a syntactic abstraction.

Example Scenario: Consider adding *sugar* for function applications to FunctionalLogo. Most *Logo* dialects use explicit keywords for defining functional abstractions and explicit keywords for applying a function. E.g., FunctionalLogo uses the keyword apply. Once a user has defined a function with a unique symbolic name, it is clear that this symbolic name refers to a function. Despite this, when applying a function, the user always has to use the keyword apply to call the function. For the users, it would be interesting to apply a function with a syntax like for primitive commands, because this would enable calling functions like commands.

Consider the program shown in Listing 3.15, which uses such syntactic sugar for functional applications as an extension to FunctionalLogo. Instead of having to use the keyword apply when using the function "hexagon", in line 3.15, the program uses the function's symbolic name that was associated to its lambda expression to call the function directly.

```
define(name:"hexagon") { length ->
   ...
}

turtle (...)  {
  hexagon 50 //need to be transformed to  apply("hexagon")(50)
}
```

Listing 3.15: A program using a static transformation for function applications

Requirements: For enabling such syntactic sugar, it would be interesting if language developers could implement static transformations that transform expressions whereby enriching them with additional information that is available from the context.

For example, for the concise function application in FunctionalLogo, the developer can extend FunctionalLogo with a static transformation that desugars expressions. The transformation rewrites a command expression with a function name (e.g., hexagon 50) to an equivalent function application expression by adding the keyword apply to the expression (e.g., apply("hexagon")(50)). Since the name of the function has been associated with a lambda expression, it can be syntactically replaced by the corresponding value of the lambda expression that is then applied with the subsequent parameters (here 50).

In general, there can be two kinds of transformations, namely *exo-transformations* that have a different input and output language, and *endo-transformations* that have the same input and output language. For example, the above transformation is an exo-transformation, since a program with an extended syntax (e.g. with the keyword hexagon) is transformed to a program with a limited syntax (e.g. FUNCTIONALLOGO without hexagon). It depends on the purpose of the transformation, what the right domain (source) and co-domain (target) of a transformation is.

3.7.1.2. Semantic Transformations

There are static transformations that do not only change the syntax, but which can also change the execution semantics of the expressions of a program. While this potentially may give a program a different meaning, in most cases, it is a desirable characteristic of a transformation that the transformation preserves *semantic invariants* for the program after the transformation. Such a semantic invariant is a part of the semantic contract of a language, and programs of the language assume all invariants to be always be true. In case all transformation rules guarantee that the transformed version of the program will still satisfy its initial specification, we speak of a *correctness-preserving transformation* [PS83].

Example Scenario: An example for a correctness-preserving semantic transformation is a domain-specific optimization that transforms DSL programs to improve their execution time. Although most DSLs do not have hard performance requirements [EFDM03, CM07], in certain domains, the performance of DSL programs is important [EFDM03, MHS05], e.g. SQL [DD89]. To meet special performance requirements, a language embedding approach should support optimizations [Hud98, BKHV03, HORM08].

Consider providing an optimized *Logo* dialect that combines FUNCTIONAL with CONCURRENTLOGO and is called OPTIMIZINGLOGO. This dialect optimizes applications of functional abstractions by exploiting parallelization. Whenever a function that satisfies certain properties[14] is applied in a turtle abstraction, the function application can often be optimized by delegating the task to another turtle, which will evaluate the function in parallel and let the delegating turtle continue executing the rest of its commands. This optimization enables turtles to delegate parts of their work and let multiple turtles paint figures collaboratively.

When optimizing a DSL program, the meaning of the program should not change. Therefore, optimizations can be implemented as transformations that are conservative extensions of the language execution semantics. Such a transformation must guarantee that non-optimized programs are transformed to optimized programs that have a behavior that produce equivalent outputs in less time. For example, in case of the above optimization, FUNCTIONALLOGO programs are

[14]The function must not have any data flow dependencies, all it parameters must be known and there is no dependency on the function result. Note that this property holds for most FUNCTIONALLOGO functions.

transformed to parallel corresponding versions that draw the same figures. Consequently when implementing a transformation that optimizes DSL programs, no new syntax should be added to the language and only the internal execution semantics need to be refined to optimize them. From the perspective of language semantics, this optimization needs to evaluate a sequential FUNCTIONALLOGO program as if it was programmed concurrently.

Requirements: In particular, such a domain-specific optimization is interesting because a correctness-preserving transformation does not change the syntax or observable semantics of the result. Therefore, such a transformation is *transparent* to the end users, and it does not impose any additional programming efforts for them. In the above example, the optimizing transformation is particularly interesting, since children can use the language without explicitly dealing with special abstractions for parallelization (e.g, threads). Still, the optimization transforms the program for them such that it executes in parallel.

In general, also for semantic transformations, there can be *exo-transformations* and *endo-transformations*. For example, the above transformation is an endo-transformation, because the domain and co-domain (source and target) are equal.

3.7.2. Dynamic Transformations

A *dynamic transformation* allows the transformation of expressions in a program depending on their evaluation context. In other words, a subset of their transformation rules dependent on the program context i.e. the state of the program's objects. There are also context-dependent transformations that have rules that are either only applicable in a certain context, or where the transformation itself is parameterized by the context.

Example Scenario: An example of a dynamic transformation of a DSL is an extension that adds an adaptive optimization strategy, i.e. an optimization whose decisions depend on the execution state of the program to be optimized.

Consider optimizing FUNCTIONALLOGO programs for execution on a central controller that transmit drawing commands to a remote fleet of physical plotting robots (of a certain limited number). Unfortunately, straight-forward optimizations like OPTIMIZINGLOGO would not be adequate for this scenario. Recall that OPTIMIZINGLOGO always parallelizes a function application into a separate thread. It assumes that it is always worth optimizing a function application and that there are an unlimited number of resources available. Yet, likely there is only a limited amount of physical turtle robots available. The problem is that OPTIMIZINGLOGO is non-adaptive and its decisions do not take into account context information. Therefore, it makes too conservative assumptions about what should be optimized.

As a solution to this problem is the concept of *adaptive optimizations* [AFG+00] that optimizes programs depending on their runtime context.

Consider a version of *Logo* with support for adaptive optimizations, which we call ADAPTIVEOPTIMIZINGLOGO. For each function application, ADAPTIVEOPTIMIZINGLOGO decides whether it is worth to optimize the function call, e.g. to paint the function's commands in parallel. For this, it could take into account dynamic context information, such as the estimated amount of work to draw the function, or whether there is an idle robot currently available in the fleet. In contrast to the non-adaptive optimization in OPTIMIZINGLOGO, ADAPTIVEOPTIMIZINGLOGO only optimizes a functional application if it is considered worth doing so and if enough resources are available at that time.

Requirements: Because there are large initial costs when using existing language approaches to implement an adaptive optimization system for a language, these systems are often only available for languages that have high performance requirements. Most adaptive optimization systems are for general-purpose languages, where a *virtual machine* analyses programs for frequently called subroutines and lets a just-in-time compiler optimize those routines to speed up the execution. Only a limited set of high performance DSLs make use of adaptive optimizations, such as SQL [DD89], which uses adaptive optimization to optimize queries. Today, adaptive optimizing DSLs are implemented as individual solutions, but to the best of the author's knowledge there is no general approach that facilitates implementing adaptive optimizations for DSLs.

In general, also for dynamic transformations, there can be *exo-transformations* and *endo-transformations*. What is particularly interesting about the ADAPTIVEOPTIMIZINGLOGO transformation is that it is an endo-transformation that can be performed on a program while it is executing. For this it is necessary that the domain and co-domain (source and target) are equal, and that transformation results remain *causally connected* to the source program. That means it must transform the running program, which is only possible if there is the runtime that is uniform and causally connected with the code that is processed by the transformation. To the best of the author knowledge, there are only approaches that support such transformations for general-purpose languages, but not for DSLs. Still, such endo-transformations would be interesting for adaptive optimizations of DSLs, what is missing is a DSL approach that enables support for endo-transformations.

3.8. Review of the Support for the Desirable Properties in Related Work

In previous work [Din10], the author of this thesis has published an extensive review of the support for the above desirable properties by the related work on embedding approaches. The review explains details of the relevant mechanisms that target the properties in existing embedding approaches. It identifies open issues and limitations of current techniques. To overcome the current shortcoming of embedded approaches, the review proposes a road map for the research on embedded domain-specific languages, whereby it draws conclusions from studying the available support for the desirable properties in related work on DSLs and GPLs.

Table 3.1 presents the detailed result of validating the support for the desirable properties that related work proposes. For the top most desirable properties (gray lines), for each sub-property, the detailed results are aggregated, such that, the best support that is available for *all* scenarios is taken into account. Whereby, when one or more scenarios are not supported at all, the support is only partial.

Table 3.1.: Review of supported properties by related embedding approaches

Desirable Property (○: partial support, ◖: important limitations, ●: full support)	Pure Embed. [Hud96]	Tagless Embed. [CKS09]	Unembedding [ALY09]	Jargons [Pes01]	EDSL Groovy [KG07]	EDSL Ruby [TFH09]	TwisteR [AO10]	π Pattern Lang. [KM09]	Helvetia [RGN10]	Staged Interpr. [COST04]	Ext. Meta-Prog. [SCK04]	Converge [Tra08]	Fluent [Eva03, Fow05]	EMF2JDT [Gar08]	Scala EDSL [OSV07]	Polymorphic [HORM08]	Embed. Gen. [Kam98]	Embed. Compiler [EFDM03]	Ruby Gen. [CM07]	MetaBorg [BV04]	TXL [Cor06]
3.1. Extensibility	○	○	○	○	◖	◖	◖	◖	○	○	○	○	○	○	○	○	○	○	○	○	○
3.1.1. New Keywords	●	●	●	◖	●	●	●	●	◖	○	◖	○	○	○	●	●	◖	◖	◖	◖	◖
3.1.2. Semantic Extensions	○	○	○	○	◖	◖	◖	◖	○	○	○		○	○	○	○				○	○
3.1.2.1. Conservative Extensions	●	●	●	○	◖	◖	◖	◖	○	●	●		○	◖	◖	◖				○	○
3.1.2.2. Semantic Adaptations					◖	◖	◖	◖	○	○											
3.2. Composability of Languages	○		○		○	○	○	○	◖				○	○	○	○					
3.2.1. Languages without Interactions	●		●		●	●	●	●	●				◖	◖	●	●					
3.2.2. Languages with Interactions	○		○		○	○	○	○	◖												
3.2.2.1. Syntactic Interactions	◖		◖		○	○	○	◖	◖												
3.2.2.2. Semantic Interactions					○	○	○		◖												
3.3. Composition Mechanisms	○		○		○	○	○	○	○				○	○	○	○				○	○
3.3.1. Syntactic Interactions	◖		◖		○	○	○	○	○				○	○	◖	◖				○	○
3.3.1.1. Conflict-Free Compositions	◖		◖					◖	○				◖	◖	◖	◖				●	◖
3.3.1.2. Resolving with Renaming	◖		◖		◖	◖	◖	◖							◖	◖				●	◖
3.3.1.3. Resolving with Priorities					◖	◖	◖		●						◖	◖				●	◖
3.3.2. Semantic Interactions							○		○											○	○
3.3.2.1. Crosscutting Composition							◖		◖											○	
3.3.2.2. Resolving Composition Conflicts																				○	◖
3.4. Concrete Syntax	○	○		○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
3.4.1 Concrete-to-Abstract							○	●	●								○			●	◖
3.4.2 Mixfix Operators	○	○	○	○	○	○	○	●	●	●	○	○	○	○	◖	◖	○	○	○	●	●
3.4.3. Overriding Host Keywords					●			●	●	●	●	●									
3.4.4. Partial Syntax																					
3.5. Pluggable Scoping				○			○	○												○	○
3.5.1. Dynamic Scoping				○																○	○
3.5.2. Implicit References								●												●	●
3.5.3. Activation of Constructs							○	○												○	○
3.6. Pluggable Analyses		○	◖				◖		○	○	○	○				◖		○	○	◖	◖
3.6.1. Syntactic Analyses		○	◖				◖		○	○	○	○				◖		○	○	●	●
3.6.2. Semantic Analyses		○	◖				◖		○	○	○	○				◖		○	○	◖	◖
3.7. Pluggable Transformations		○	○				○		○							○		○	○	○	○
3.7.1. Static Transformations		○	○				○		◖	●	●	●				◖		○	○	◖	◖
3.7.1.1. Syntactic Transformations		○	○				○		◖	●	●	●				◖		○	○	◖	◖
3.7.1.2. Semantic Transformations		○	○				○		◖	●	●	●				◖			○	◖	◖
3.7.2. Dynamic Transformations							○														

The identified properties are currently only supported by non-embedding-based approaches, there is a need for a new language embedding approach that systematically supports these properties. Particularly, the following issues need to be addressed:

- For a better composability of languages, there should be support for composition of interacting embedded languages.
- While existing composition mechanisms in the host language can be used, there is a need for extensible composition mechanisms.
- When requiring arbitrary concrete syntax, it is desirable to reduce the costs for defining the concrete syntax.
- There should be support for pluggable scoping.
- There should be support for composing analyses and transformations.

3.9. Summary

In this chapter, a set of desirable properties for language embedding approaches have been proposed. The selection of these properties has been motivated by related work and concrete scenarios have been presented. The review results clearly indicate the current limitations of the language embedding approaches, which miss important support for properties that are often supported by non-embedded approaches. Unfortunately, because of the lack of this support, the adoption of embedding approaches is currently limited.

This thesis claims that a language embedding approach only facilitates the language developers with their tasks in an adequate way, when it supports all these properties for developing and evolving languages. The identified open issues and questions regarding language embedding approaches will be addressed in the remainder of this thesis.

Part II.

Core Architecture for Reflective Embedding

4. The Reflective Embedding Approach

As outlined in the previous chapters, since languages and their implementations are continuously evolving, language developers require an incremental and precise facility for language evolution. Therefore, this chapter presents the concept of *reflective embedding* that embeds languages with a meta-level into a reflective host language. The meta level helps language developers to keep language implementations open for later evolutions.

In the reflective embedding approach, a language developer follows the structure of a disciplined architecture — the *Reflective Embedding Architecture* (REA for short). The architecture prescribes how to structure the embedded library into four levels with well-defined responsibilities. When language developers embed languages, REA's structure gives them guidelines how to embed a language, which helps them to systematically design and implement their languages in an open way.

In contrast to traditional embedding approaches [Hud96] where embeddings are created using black-box abstraction mechanisms, REA builds upon the *reflective* language features the host provides. Because a language developer homogeneously embeds a language as a library, the reflection feature is inherited from the host to the embedded language. Hence, developers can use the reflective features inside the language implementation and programs of the embedded language.

Using reflection inside a language implementation enables embedding a language that can analyze and adapt its semantics, which is encoded in its embedded library. Because the architecture's levels decouple parts in the implementation of a language, it is possible to extend, to adapt, or to replace levels of a language implementation for evolution.

Note that the reflective embedding approach makes only few assumptions about the host language. The host language needs to be an object-oriented programming language that provides certain language features, namely *first-class code blocks*[1] and *meta-objects* [KRB91]. The architecture builds on objects with classes, interfaces, and single inheritance. The architecture uses code blocks for encoding abstraction operators that allow grouping compound expressions. The architecture uses meta-objects for reasoning about and adapting objects. This exact set of language features is necessary to allow a uniform description of the architecture, not for technical

[1]A code block was defined in Section 3.4.1 (p. 48) as a piece of code. A first-class code block is a code block of which one can obtain a reference that can be passed around as a value. These code blocks are like all objects associated with a meta-object that executes the code block, i.e. evaluating an expression in a code block creates a call that is reflected by its meta-object.

reasons. While this feature selection allows describing the architecture for a broad set of various host languages[2].

What is special in REA is that languages are embedded as a set of objects and that every language has a meta-level. Because REA embeds languages as objects in a *reflective* host language, one can reflect on languages and their constructs as first-class objects by using the meta level without changing their base levels.

Parts of this chapter have been published in the following papers:

- Tom Dinkelaker, Michael Eichberg, and Mira Mezini. **An Architecture for Composing Embedded Domain-Specific Languages.** In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD)*, 2010.

- Tom Dinkelaker, Martin Monperrus, and Mira Mezini. **Supporting Variability with Late Semantic Adaptations of Domain-Specific Modeling Languages.** In *Proceedings of the 1st International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines*, at the International Conference on Aspect-Oriented Software Development, 2010.

- Tom Dinkelaker, Christian Wende, and Henrik Lochmann. **Implementing and Composing MDSD-Typical DSLs.** Technical Report, TUD-CS-2009-0156, Technische Universität Darmstadt, Germany, October, 2009.

The remainder of the chapter elaborates the concept of the reflective embedding approach in Section 4.1, and its implementation in Section 5.

[2]Although the thesis commits to code blocks and meta-objects, these features can be easily replaced by alternative ones with an equivalent expressiveness. *Prototypes* [US87] can be used instead of classes. *Delegation* [Ste87] can replace inheritance. *Code blocks* can be replaced by *closures* [AS96]. *Closures* can replace objects [Dic92]. In most cases, meta-objects can be replaced by another reflection mechanisms, which sometimes support only a limited form of reflection, such as *interceptors*, *wrappers*, and *aspects* [KLM+97]. When replacing one of the required features in the architecture with another one, there would be a slightly different description of the architecture that the author expects to be more language specific and more low-level. Committing to the above selection of language features enables convenient and modular embeddings of languages that are well encapsulated.

4.1. The Reflective Embedding Architecture

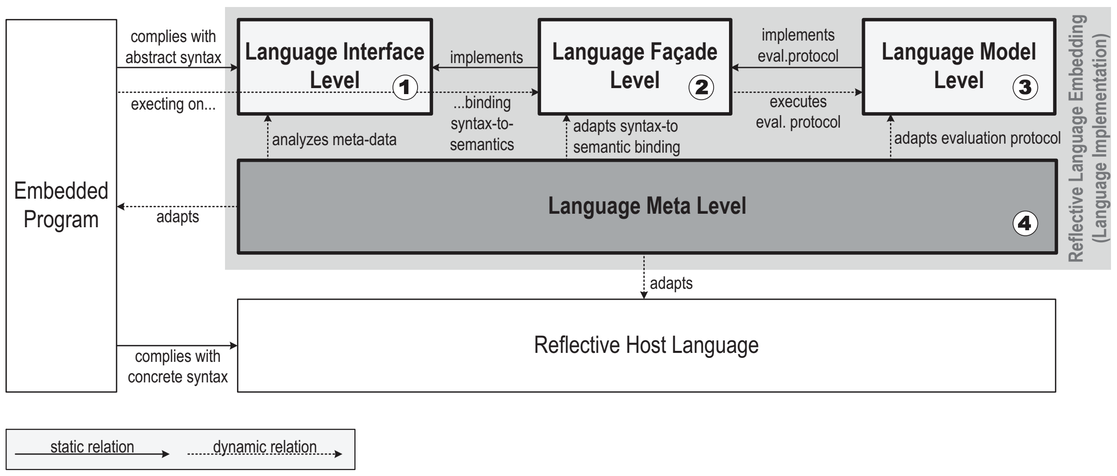


Figure 4.1.: The levels of the reflective embedding architecture

Figure 4.1 presents a high-level overview of the reflective embedding architecture (REA for short). As illustrated, the architecture is built on top of the reflective host language. A *language implementation* consists of four levels of which each has a particular responsibility:

(1) the *language interface level*: defines the *abstract syntax* of the language as a set of expression types, which constitutes the language's *syntax definition*. Note that the abstract syntax should not be confused with an *abstract syntax tree*[3], which is only one possible representation of abstract syntax. Although syntax can be concrete, since in the second part of this thesis every embedded language has an abstract syntax, we will refer to it as the *syntax* for short.

(2) the *language façade level*: binds expression types of the syntax to semantics, which constitutes the so-called *syntax-to-semantics binding* of the language. This binding describes how a syntactic construct creates and manipulates its corresponding semantic first-class representations that are encoded in the next level.

(3) the *language model level*: defines a first-class representation for each language construct of the embedded language, which defines the structure and behavior of the construct's

[3]We use the term *abstract syntax* in the sense of [Kam98, ALSU07, Fow05] to reflect that embeddings do not have a concrete syntax but that they reuse the host syntax to encode their expressions. Note that a language's abstract syntax is related to its *abstract syntax tree* representation, but it is not the same—in fact all possible representations of a program—whether the same program is represented by a concrete textual syntax, by a concrete syntax tree, or by an abstract syntax tree—they all can be mapped to the same abstract syntax representation.

default execution semantics[4]. Finally, there is a resulting set of inter-connected classes in the language model, which constitute the language's *evaluation protocol*.

(4) the *language meta level*: through which language developers can reify language constructs and provide alternative semantics for them. The meta level provides a meta-interface, which is called the language's *meta-protocol*. Through this interface, developers and users can reflect on the first-class representations in the language model in order to analyze and adapt them.

Since the architecture follows Hudak's language embedding principle [Hud96], the initial investment for implementing a language is low. Language developers do not have to implement a special parser and compiler for the embedded language because the embedded language's abstract syntax and its semantics are directly encoded in the reflective host language. The architecture organizes a language implementation into the four levels, whereby in each level the developer provides a part of the language implementation. Because developers can later extend and replace a level or a part of it, a language implementation remains extensible throughout its lifecycle.

The development process of a language is as follows. To implement an executable language, a *language developer* implements each of the four levels of the reflective embedding architecture. The instantiation of the architecture serves as a default evaluator for programs that are written by a language end user. Finally, the language developer can compile the evaluator's library using the host compiler and then ship its binary to the end user. Nonetheless, the shipped language remains extensible in the user domain.

4.1.1. Levels of the Architecture

The remainder of this section elaborates on the steps that are necessary to instantiate the levels of the architecture. Each level implements a part of a particular language. For each level, this section provides a general recipe how to instantiate this level. Further, this section exemplifies how each level is instantiated, by implementing an evaluator for the SIMPLELOGO language from Section 3 (p. 29). Figure 4.2 gives an overview of the levels for SIMPLELOGO. This section will explain this levels step by step in the following subsections.

4.1.1.1. Language Interface Level

The *language interface level* defines the abstract syntax in form of a so-called *language interface* that defines possible types of expressions of the embedded language. In the following,

[4] Note that while often there is a direct relation between an expression type and a language construct, these two concepts are not the same. While often there is a particular expression type at the syntax level to define or use its language construct at the semantic level, there is no requirement for an one-to-one relation. First, not every expression must related to one distinct language construct, e.g. delimiters and comments must not be represented as the semantic level. Second, one expression type can be mapped to many language constructs, e.g. a constraint expression. Third, a set of nested expression types may be needed to create only one language construct, e.g. a class definition is made of various sub-expressions. Despite there is a distinction in syntactical and semantic representations, generally, a keywords is causally connected to its language construct.

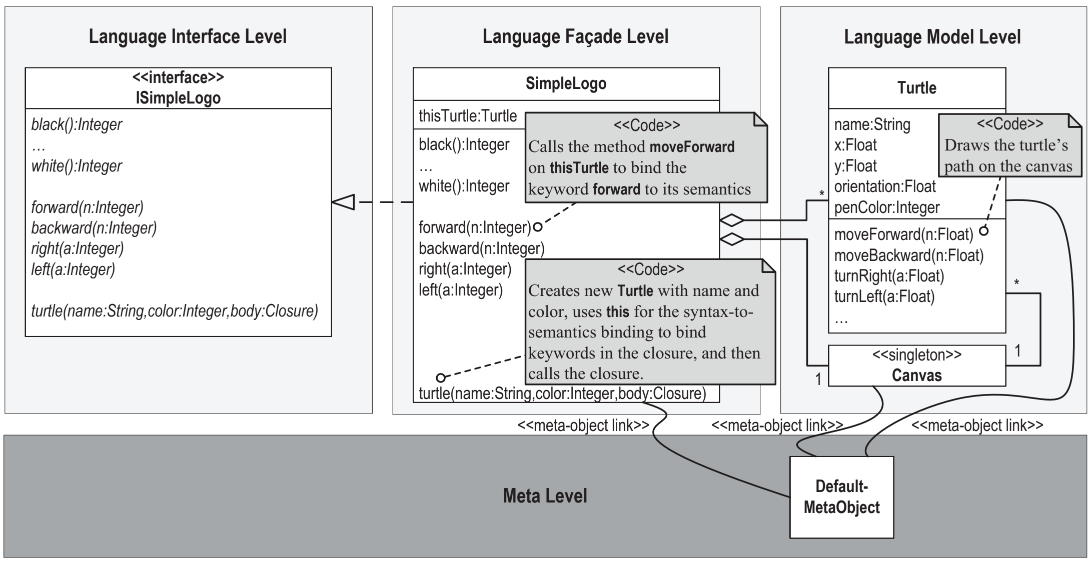


Figure 4.2.: Levels of the architecture for SIMPLELOGO

we discuss the details of the development process for defining the abstract syntax of an arbitrary language, whereby we exemplify the process for the *Logo* dialect SIMPLELOGO.

For each possible expression type in the embedded language, the developer defines a method with an unique signature in the language interface. The method name constitutes a *keyword* in the language, the types of parameters define the categories of the sub-expressions the expression is composed of, and the return type of the method defines the category of expression. Together, the methods define all possible expression types that are available in the embedded language.

The approach distinguishes three classes of expression types: (I) *literal expressions*, (II) *operations*, and (III) *abstraction operators*. The class of the expression type determines the format of the method to be defined, as described below:

(I) a method for each literal: Literals encode *terminal* expressions.

For each literal, there is an accessor method that has the literal's keyword name, it takes no parameters, and it returns a value of a certain type. Often the value returned by such a literal method is a constant, but there are also literals that have a value that depends on the context the expression is used in (such as keywords like this cf. Section 3.5.2, p. 55).

For example, Figure 4.2 shows the language interface of SIMPLELOGO, called ISimpleLogo. For every literal that represents a color in SIMPLELOGO, ISimpleLogo defines an accessor method. Each method is parameterless, it has the corresponding name of the color, and it returns an integer encoding its RBG representation. ISimpleLogo defines the methods black(), blue(), red(), green(), yellow(), and white(), that all return a value of type Integer.

(II) a method for each operation: Operations encode *non-terminal* expressions that are linked to their sub-expressions, i.e. the operands.

For each operation, there is a method that has the name of the operation's keyword and that takes a parameter for each operand. Each parameter type uses a type to hold the corresponding sub-expression. The number of the parameters corresponds to the arity of the operation. Note that the order and the types of parameters in the method' signature must match the operands of the expression type.

For example, as shown in Figure 4.2, the language interface ISimpleLogo defines the domain-specific operation methods forward and backward for moving the turtle. As both operations are unary, their methods take an integer parameter indicating the relative distance in units to move. Further, the interface defines the methods right and left with one integer parameter for turning the turtle in the corresponding direction for the number of degrees provided by the parameter.

(III) a special method for each abstraction operator: Abstraction operators[5] are special non-terminals for defining structural or behavioral abstractions, such as turtle.

Technically, abstraction operators seems similar to operations, since both a non-terminal expression types. But in contrast to an operation, an abstraction operator can establish a new lexical or dynamic scope for its nested sub-expressions.

For each abstraction operator, there is a method that has the keyword name of the operator and that takes a set of parameters. Like above, the method can have parameters of any types and order. What is special with such an abstraction operator is that at least one parameter has the type CodeBlock[6]. This code block parameter will contain sub-expressions that are nested inside the abstraction operator.

For example, as shown in Figure 4.2, for the domain-specific abstraction operator turtle in SIMPLELOGO, the language interface ISimpleLogo defines the method turtle, of which the first parameter is the turtle's name as a string, of which the second parameter is its initial pen color as an integer, and of which the last parameter is a sequence of commands in form of a code block.

Besides defining the abstract syntax of a language embedding, the language interface may define a contract in form of pre-conditions, post-conditions, and invariants. Language developers can derive this contract from an informal or formal language specification. It depends on the power of the host language, whether language designers can provide the contract only as comments, or if it is possible for them to express the contract in the interface definition, on whose implementations the language enforces this contract.

[5]At times, abstraction operators also called *nested elements* or *control structures*.
[6]The first-class value of a code block has the type CodeBlock.

4.1.1.2. Language Façade Level

The *language façade level* defines a façade class that binds the language's syntax to concrete semantics. This *syntax-to-semantics binding* allows preparing a DSL program for evaluation by transforming the program's textual representation into a first-class representation in the language model.

For implementing a concrete language façade, the language developer follows the specification of the language. The responsibility of façade implementation is to bind a language's syntax to semantics by implementing its language interface. For each expression type declared by a method in the language interface, the façade implements a method with the corresponding signature. The code in each method's implementation does not directly evaluate expressions, but it connects the syntactic representation of the corresponding expression type to the language model in which concrete semantics are encoded. In other words, each method implementation binds the expression type to its corresponding concrete semantics. After all methods of the façade class have been implemented, instances of this class can be used as an *evaluator* for programs of the corresponding language.

Besides its public interface, the façade can define private members for internal handling and context information. It defines private methods for internal handling, e.g., for resolving names and references. It defines private fields that refer to objects in the language model. These fields are used to store the first-class representation of programs under evaluation and their context. More specifically, with fields, developers can constitute an *evaluation context* with *static references* to the program structure, a *heap*, a *stack*, as well as special *scoping strategies* for variable look up and binding.

When it comes to implementing a keyword method, there are two fundamental modes for evaluation that define *when* to evaluate an expression: *immediate* or *delayed* evaluation. When evaluating in immediate mode, the method implementation has the immediate side effect to evaluate the expression and to return its result, or to change the program's runtime context. When evaluating in delayed mode, the method implementation prepares the expression for evaluation by appending its first-class representation to a language model instance from which the expression can be evaluated later on. Depending on the required language semantics, a language developer selects the right evaluation mode for each expression type, enabling opportunities for later resolutions and optimizations.

For example, Figure 4.2 shows the language façade class SimpleLogo that implements SIMPLELOGO. By implementing all methods declared in the language interface of ISimpleLogo, the façade class establishes a syntax-to-semantics binding for every expression type in the language. In SimpleLogo, all expression types use the immediate evaluation mode. As an instance for an abstraction operator, Figure 4.2 sketches the pseudo code for the turtle keyword method implementation. The turtle method defines a new turtle with the name and initial pen color from the parameters. Next, the method immediately evaluates the commands that are provided in the body code-block parameter, which moves the turtle on the canvas. The implementations of the literal methods for the colors immediately return the constant values of the RGB representation of the colors. The implementations of the operation methods immediately move the enclosing turtle on the canvas. As Figure 4.2 indicates by the pseudo code e.g. for the forward keyword method implementation, the method implementations use the language model classes, which binds the keywords to concrete execution semantics. The method implementations for forward and backward produce the side effect to move the turtle relative to its current position on the canvas for the distance given by the input parameter. The method implementations of the left and right operations change the orientation of the turtle relative to its current orientation for a given number of degrees. After implementing the SimpleLogo façade, instances of this class can be used as an evaluator for SIMPLELOGO programs.

The above SIMPLELOGO implementation only uses immediate evaluation, but delayed evaluation is also conceivable for this scenario. As an example for a delayed evaluation mode in SIMPLELOGO, consider two keywords plan and do. The plan keyword is a special abstraction operator that takes a sequence of other operations in a parameter. It does not evaluate the operations immediately, but places them in a queue. For example, the expressions "plan { right 90; forward 50 }" plans to turn the turtle 90 degrees to the right and then move it forward for 50 steps. The operation do executes all operations that are stored in the queue. For example, the program "plan {...}; plan {...}; do;" defines two groups of planned commands. Both groups of commands will not be evaluated until the program uses the do operation. The delayed evaluation mode enables delaying the evaluation of commands inside the plan abstraction operator.

4.1.1.3. Language Model Level

The *language model level* encodes the detailed semantics for evaluating the expressions of a language. This level enables language developers to decompose a language into a set of *domain types* of which each encodes a part of the language's domain and semantics.

At the beginning, the developer analyzes the *language domain* for its semantics, whereby identifying the domain's underlying *language constructs*. It is a strategic question how the language developer should model a particular language construct as a first-class citizen in the language model. The answer to this question can partially sketched depending on whether the language construct defines *structure* or *behavior*, as elaborate below.

A *structural* expression type describes a part of the program structure, e.g. the structural abstraction operator turtle. When there is a *structural* expression type at the syntax level, the language developer usually models a new language constructs in form of a domain type. To model a construct, a language designer or developer identifies necessary properties that store properties of the language construct and possible operations. To implement the construct in the host language, a language developer implements the language construct using a class defining appropriate members for the construct.

A *behavioral* expression type describes an action, e.g. commands such as forward and behavioral abstraction operator repeat. When there is a *behavioral* expression type at the syntax level, usually the language developer models it as a domain operation as part of some domain type.

After identifying all language constructs and implementing the complete language model, the latter serves as the evaluation protocol of the language. In this *evaluation protocol*, all domain types, their attributes and methods contribute to define the complete execution semantics for the language. Note that breaking down the evaluation of language constructs into little pieces and steps make them later extensible at the meta level.

For example, Figure 4.2 shows an excerpt the language model of SIMPLELOGO containing abstractions of the *Logo* domain. Note that for a domain-specific language, such as *Logo*, when speaking about its language model, in the following, we will refer to the language model as the *domain model*. Further, we refer to a classes in the *domain model* as a *domain type* with corresponding *domain properties* and *domain operations*. There is a singleton domain type Canvas to which one can add Turtle objects using the domain operation add, and on which one can paint using several primitive drawing operations, such as drawLine. According to the domain semantics, the Turtle domain type has the attributes for its name, its x and y position on the canvas, its orientation, and its current penColor. Further, the class provides domain operations for moving the turtle, such as moveForward, moveBackward, turnRight, turnLeft. The implementations of these operations encode the execution semantics for painting the path of the Turtle on the Canvas object. For example, as indicated by the pseudo code for method moveForward, moveForward will draw the turtle's path drawing a line on the Canvas object from the current position of the Turtle object and in the direction of its current orientation.

For understanding the default semantics of such a language implementation, it is important to understand how the language façade and its model are connected. Besides binding syntax to semantics, the language façade class hides the complexity of the detailed evaluation protocol in the language model. The bodies of the language façade methods contain constructor calls that create instances of domain types and calls to semantic operations on these types. For example, the keyword method turtle calls the constructor of Turtle to create an instance — a first-class representation for thisTurtle, and then, the keyword method forward calls the operation moveForward of this Turtle to move it.

In general, for any language that has computable semantics, a language developer can encode such a language model that respects its language's specification. For language embeddings, often the language specification is provided merely as an informal definition. For example, for SIMPLELOGO, only an informal definition is available. However, it is also possible to use a formal specification of a language's semantics. Examples for existing formalisms are *operational semantics* [Plo81], *denotational semantics* [SS71, Ten76], or *communicating sequential processes* (CSP) [Hoa78]. Independently of the selected specification, a language developer must provide appropriate semantics classes in the language model. When using a formalism, a language developer must provide appropriate classes whose semantics encode the theorems and proofs of the language in the corresponding formalism. Note that although it is possible to use such a formalism as the basis of the language specification, REA does not require the language developer to do so.

4.1.1.4. Language Meta Level

When instantiating a language with traditional language implementation approaches, the resulting language implementation has one fixed language syntax and semantics. In those approaches, whenever the requirements for the language syntax or semantics change, the language developers have to invasively change the code of the language implementation.

In contrast to traditional approaches, the reflective embedding architecture provides a *language meta level* that allows developer to adapt a particular language implementation or a set of language implementations, as indicated by the dashed arrows in Figure 4.1. The meta level enables modular gray-box extensions of language façades and their language model levels—without invasively requiring developers to change the code of these levels. For a given language implementation, a meta level can analyze the meta-data defined by language interfaces, it can adapt an existing syntax-to-semantic binding, and it can adapt parts of the evaluation protocol. Since the meta level opens up language implementations in the user domain, language developers and even educated end users can tailor a given language implementation using a custom meta level extension to their domains. Moreover, the meta level allows adapting parts of user programs and of the host language, e.g. if an embedding requires instrumentation.

For instance, in the context of SIMPLELOGO, consider adding enforcement logic for domain constraints to an existing implementation of the language. The benefit of supporting several evaluation strategies is that the language can be adapted without changing the code of the language implementation.

The elements of the interface, the façade, and the model levels are connected to *meta-objects* in the meta level. In the architecture, each object has a *meta-object link* to its meta-object that takes care of its execution. By default, every object is associated with the so-called *default meta-object*, which executes objects under standard OO semantics. For example, Figure 4.2 shows that initially every class of the language implementation is associated with the default meta-object MetaObject that is concerned with its execution. For adapting an object, one can associate the object with a specialized meta-object that executes the object under different semantics.

For example, Figure 4.3 illustrates that there is a special meta-object ConstraintCheckingMetaObject for the domain type Turtle, e.g. which checks that a turtle never moves out of its canvas. To enforce such a constraint on Turtle objects, the developer simply updates the corresponding meta-object link that is highlighted in bold. When evaluating a program using the language implementation with the special meta-object, the execution of the program is changed, because the meta-object adapts the evaluation protocol such that constraints are checked.

Most importantly, the meta-objects enable opening up the implementations of objects residing in the other levels of a language implementation in the reflective embedding architecture. The meta-object enable modular adaptations that enable cooperatively defining parts of the semantics. While one party of the language developers may define the reusable language semantics for the party of customers (i.e. end users), there may be a third-party of language developers who adapt those semantics using specialized meta-objects. This third-party can either provide semantic adaptations that are *transparent* for end users, or it can provide a *meta-interface* with adjustable parameters for the end user and use these parameters inside the implementation. With the meta-objects, an end user can deploy the final language semantics in their problem domain—either by installing a transparent adaptation, or by adjusting parameters through the meta-interface.

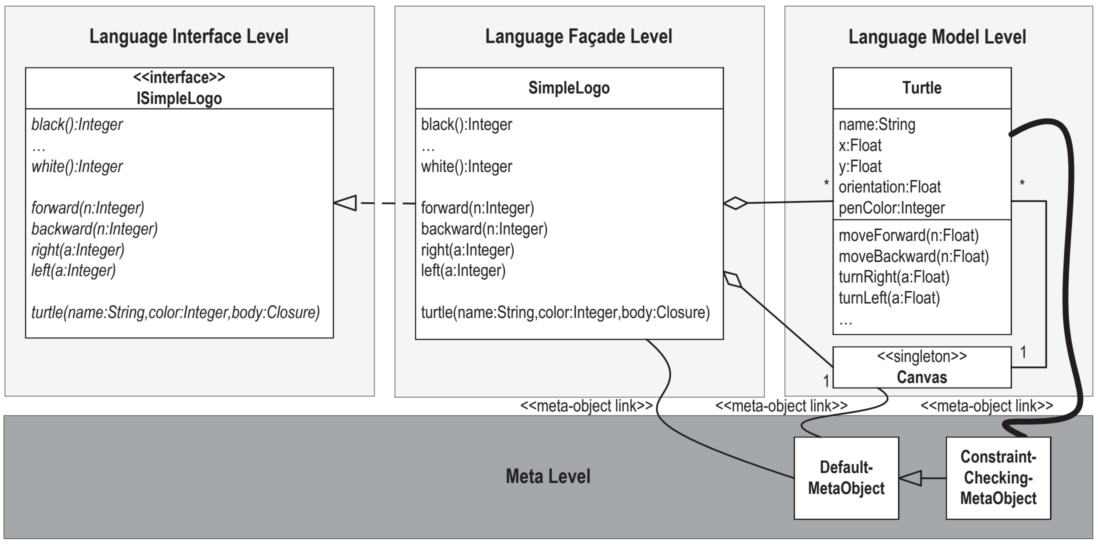


Figure 4.3.: The architecture for SIMPLELOGO with alternative semantics

It is central for the architecture that one can use meta-objects to extend language façade classes. When a façade has a special meta-object, it is actually a façade with a meta-level. A façade with a meta-level is called a *meta-façade*. Having support for meta-façades is interesting because meta-façades enable implementing *global strategies* and *context-dependent strategies* [BV04, Cor06] for languages. Concrete use cases of meta-façades will be elaborated later on. In particular, Section 4.2.3.3 (p. 109) presents how meta-façades can be used to implement global analyses and transformations for composing languages.

To recapitulate, another way to understand the role of meta-objects for adapting a language is that these objects implement a *meta-protocol* for those languages, i.e. an *meta-interface* for the end user to adapt the language. From the perspective of existing work on meta-object protocols, how REA uses meta-objects can be seen as proving specialized meta-objects that implement *sub-protocols* [KRB91] of a meta-object protocol. But by using meta-objects for adapting language implementation, a sub-protocol's contribution is the knowledge that is cast in its meta-interface and the guarantees it provides to apply specific adaptations in a certain domain. In the following, we will discuss the practical advantages of having meta-protocols for languages using various examples in order to evaluate the applicability of meta-protocols to various fields of language research.

4.1.2. Program Evaluation

To evaluate a program, a syntactic representation of the program's expressions is executed, which results in side effects and finally computes the output of the program.

In the basic architecture, end users encode programs in abstract syntax. For example, consider that an end user encodes the example SIMPLELOGO program shown in Listing 4.1 with abstract syntax (cf. Section 3.4.1 for the encoding). To write a new program in the embedded language, the end user embeds the program's expressions into a code block inside a host language program. In the reflective embedding architecture, a code block's context contains *static* and *dynamic information*, as demanded in Section 3.4.1 (p. 48). The static information encompasses lexically enclosing code blocks. The dynamic information encompasses a *façade-object link* and a *meta-object link* for this block as well as variable bindings. Because the expressions are part of a *code block* that is connected to a special meta-object, they become first-class and can be evaluated as discussed in the following.

```
{
  turtle ("Square",red, {
      forward(50);
      right(90);
      forward(50);
      right(90);
      forward(50);
      right(90);
      forward(50);
      right(90);
  })
}
```

Listing 4.1: The abstract syntax of the program from Figure 3.1a (cf. Section 3.4.1)

In the next paragraphs, we discuss the general steps for evaluating a program—these are: *compilation*, *setup*, and *expression evaluation*. For each step, we elaborate on its details and exemplify it by stepping through the above example SIMPLELOGO program.

Given a default façade, a default model, and the default meta level, these three parts together provide the *default execution semantics* (*default semantics* for short) that can be used to evaluate programs. The default semantics of a language is based on: (a) a default syntax-to-semantic binding, which is given by the default language façade, (b) a default evaluation protocol, which is given by the default language model, and (c) the default language meta level, which is the host language default meta-object. Unless special semantics are selected by language developers, the default meta level implicitly executes with the default semantics. That means it executes the objects in all other levels using its default meta-objects.

In the *compilation step*, there are two phases: compilation of the language and compilation of the program. First, the language developer compiles the embedded language into a binary, whereby the host compiler validates that language façade implements all methods that the language interface declares. Second, the end user compiles the embedded program to a binary. When compiling, the host compiler inserts indirections that take care of the reification of first-class code blocks, field accesses, and method calls. In a strongly-typed language, the compiler checks that the program only uses expressions that are defined in the language interface. Note that this compilation step is optional. The step can be completely omitted in case the host language is an interpreted language.

The *setup step* creates an instance of the language façade. The newly created instance of the language façade is initialized with an empty interpreter environment, i.e., no expression of the program has been loaded yet. Then, the setup step prepares the context of the program code block. First, the setup step associates the façade as the evaluator of the code block, such that the created façade instance serves as an evaluator for executing the program. Second, the setup step uses the default meta-object, or optionally it associates a special meta-object with the code block. These two entities provide a dynamic context for evaluating the code block. To uniformly bind syntax-to-semantics for all expressions in the program, it is the responsibility of the façade to propagate itself to the context of all nested code blocks in the program. Further, for uniform execution semantics, the façade propagates the selected meta-object to all nested code blocks and to all domain objects it creates.

The *expression-evaluation step* evaluates the program code block with all contained expressions. The most important participants in this step are the program code block, the (default) meta-object of code blocks, the language façade, and its model. In a nutshell, when evaluating a program's code block, for each expression encountered in the block, the code block's meta-object produces a method call and dispatches this call to the code block's language façade. This façade then passes the call through the different levels in the architecture, which yields a computation of the expression.

By default, expressions are evaluated in the sequential order in which they are defined. Before evaluating an expression itself, all of its sub-expressions are evaluated from left to right, whereby the evaluation of each sub-expression produces another method call.

When evaluating an expression or sub-expression (e.g., `forward 50`) and producing a method call via the block's meta-object, the call uses the name of the expression's abstract syntax operator (e.g., `forward`), and the evaluated sub-expressions are passed to the call as parameters (e.g., `50`). Next, the block's meta-object dispatches the method call to the most-specific method implementation of the language façade, which binds concrete semantics to the expression. Executing the method implementation evaluates the expression, which produces possible side effects and possibly returns a computation, as we will elaborate in an example below. For each expression, the method call can return nothing (when it has only a side effect on the language model), it can return a primitive value with the evaluation result, or it can return a reference to an object of the language model.

The aggregated method calls that result from evaluating all expressions, their side effects and computations accomplish the evaluation of the complete program.

For example, evaluating the SIMPLELOGO program in Listing 4.1 paints a red square on the canvas. In the figure, there are three method calls with three different semantics. First, the figure illustrates ordinary method calls using solid lines. Second, when evaluating a code block, there is a special method call «`evaluate`» to its first class value, which the figure illustrates using a dotted line. Third, when encountering an expression inside the code block, the block's meta-object produces a method call and dispatches this, which the figure illustrates using a dashed line. Step-by-step, we will explain the evaluation of the program in the following.

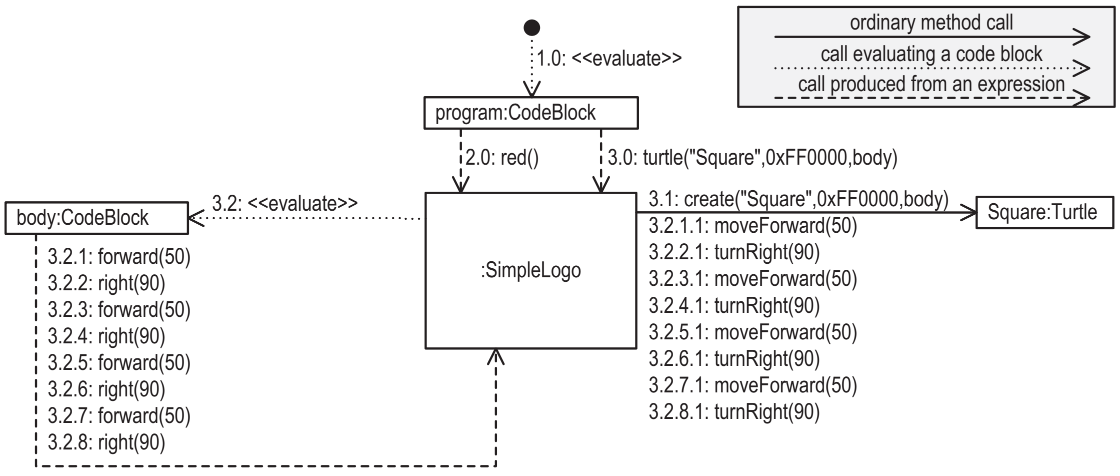


Figure 4.4.: Evaluation of the SIMPLELOGO program from Listing 4.1

First, the step 1.0 in Figure 4.4 evaluates the program code block in Listing 4.1 (p. 80), lines 1–12. In this case, the program uses the concrete semantics provided by an instance of the language façade SimpleLogo and the default meta-object. Because of using the default meta-object for evaluating the code block that wraps the program, it uniformly delegates all method calls that it produces to the SimpleLogo façade.

In Listing 4.1, line 2 creates a new turtle abstraction (turtle(n,c) {$body$}) with three sub-expressions (n, c, $body$) that are evaluated before the enclosing expression.

The first sub-expression (n) evaluates to a primitive string value, namely "Square".

The second sub-expression (c) uses the domain-specific literal red, which produces a polymorphic method call (step 2.0 in Figure 4.4) to method "red" with no parameters. To evaluate the second sub-expression, the program code block's meta-object produces a call to method red and dispatches this call to the SimpleLogo façade, which returns as its result the RBG representation of the color *red* as an integer, namely "0xFF0000".

The third sub-expression ($body$) defines the nested code block from lines 2 to 11. It creates a code block containing the command sequence that is not yet evaluated, and it returns the first-class value of the created code block.

Then for the turtle expression, the block's meta-object produces a call (step 3.0) to method "turtle" with the three evaluated sub-expressions as the parameters. The block's method object dispatches the method call to the language façade instance of SimpleLogo executing the turtle abstraction operator implementation. First, the abstraction operator creates a new Turtle instance (step 3.1) from the language model at position (x=0,y=0) with name "Square" and the pen color red. Second, it propagates the language façade and the meta-object to the code block. Third, it evaluates the code block (step 3.2) containing the sequence of commands. Because the nested code block is associated with the meta-object and the façade instance, it uses them to evaluate its nested expressions.

When evaluating the nested code block, the expression forward 50 in line 3 has the sub-expression 50 that evaluates to the primitive value 50. When encountering the forward expression, the nested block's meta-object produces a call (step 3.2.1) to method "forward" passing to it the value 50 as a parameter. Next, the meta-object dispatches the call to the façade instance of which the forward operation implementation moves the Turtle instance through a method call Turtle.moveForward(50) (step 3.2.1.1). Similarly, line 4 uses the expression right 90 that produces a call (step 3.2.2) to method "right" with the parameter 50. The nested block's meta-object dispatches the call to the implementation of the right operation that manipulates the orientation of the Turtle instance using another method call Turtle.turnRight(90) (step 3.2.2.1). The other forward and right expressions in the nested code block are evaluated analogously to the above expressions (steps from 3.2.3/3.2.3.1 up to 3.2.8/3.2.8.1).

4.2. Key Mechanisms for Language Evolution

To support language evolution, the reflective embedding architecture provides three key mechanisms: (1) *language polymorphism*, which enables defining different versions of semantics for a given language, (2) *semantic adaptations*, which enables partially tailoring a given language semantics, and (3) *language combiners*, which enables composing languages. Language developers can use these mechanisms to augment language embeddings for upcoming evolutions.

In the following, we discuss the conceptual details of the three key mechanisms.

4.2.1. Language Polymorphism

When developing a language for several stack holders, there are situations in which programs that comply with a given syntax need to be executed under different semantics. Before it is possible to evaluate a program in a given syntax, the syntax must be bound to alternative semantics. When evaluating a program under such different semantics, the evaluation process treats a possible concrete semantics as a *black-box*. This black-box must comply with the language interface, but it is free to realize various execution semantics internally.

A key property of the reflective embedding approach is the strict separation between a language specification and its implementation. For a given language, the specification prescribes the syntax and semantics of its keywords, but often it does not prescribe the concrete implementation of the keywords and the possible evaluation strategies of which different are conceivable. For a given language, a program is always developed with respect to the language specification, but it are not tied to any particular language implementation. This is because the architecture has a language interface that abstracts from possible façades that implement it. The interface's abstraction establishes a loose coupling between a program and possible evaluators of its language, and therefore, it enables support for exchanging the evaluator (i.e. language polymorphism).

To achieve this loose coupling, it is central that code blocks have a special meta-object and a link to a façade instance that realizes a *dynamic syntax-to-semantic binding*. Because a code block's meta-object evaluates every expression—inside the block, given in abstract syntax—into a method call—with a signature corresponding to the abstract syntax—and because the meta-object delegates this call to the currently selected façade instance, the dynamic method dispatch takes care of retrieving the right semantics—by executing the façade's method with the corresponding signature. Because a code block's meta-object dispatches the method calls on a possible inheritance hierarchy of language façades, always the most specific keyword method implementation in one of the façade classes in the hierarchy is executed. By inheriting keyword methods, language polymorphism enables building *hierarchical extensions*.

To allow program execution under different semantics, language polymorphism allows *abstracting over semantics* of a given language interface that has various implementations. To provide alternative language semantics for evaluating programs, a language developer reimplements the default execution semantics with alternative execution semantics by providing an alternative language implementation of the language interface. To set up new alternative semantics for evaluating a program, it is sufficient to instantiate a different façade and to register this façade with its code block's meta-object. Through the dynamic method dispatch of the language polymorphism mechanism, the so established *dynamic syntax-to-semantic binding*—a key property of the architecture—enables multiple interpretations with the semantics executing on the corresponding façade instance.

The following two section elaborate on the detailed aspects of hierarchical extensions and abstracting over semantics that are enabled through language polymorphism.

4.2.1.1. Hierarchical Extensions

In the reflective embedding architecture, the syntax and semantics of a language embedding is statically extensible. Language polymorphism allows inheriting a base language's syntax, which is inspired by *grammar inheritance* [AMH90, KRV08, Par08], but in addition, it allows inheriting the base language's semantics.

Given a base language to be extended, language developers can provide new version of the language with syntax and semantics that extend the base language, whereby reusing parts of the base implementation. They incrementally extend the language syntax and semantics whereby inheriting the base's language interface, its language façade, and its language model.

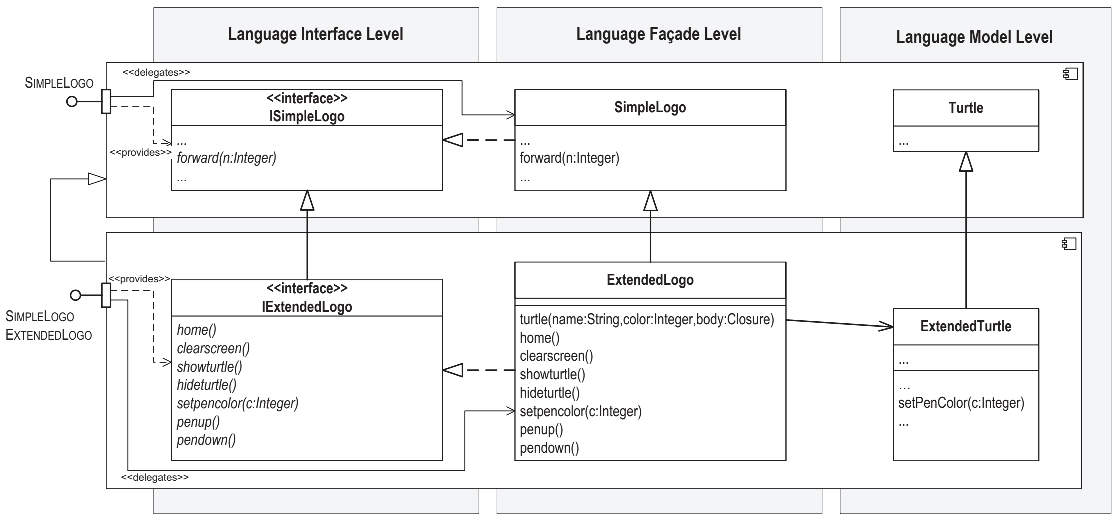


Figure 4.5.: A language extension to SIMPLELOGO

As an example, reconsider the scenario from Section 3.1.1 (p. 31) that adds the EXTENDED-LOGO keywords to SIMPLELOGO. To add the new keywords, the language developer must define only the new parts for their syntax and their semantics, as shown in Figure 4.5. To define new syntax, the language developer needs to extend the base language interface with IExtended-Logo that declares new keyword methods for the additional expression types, namely setpen-color, home, clearscreen, penup, pendown, hideturtle, and showturtle. To define new semantics, the developer implements the methods of this interface in the class ExtendedLogo—a subclass of SimpleLogo. For instance, the class ExtendedLogo implements the setpencolor method. To extend the evaluation protocol, when new language constructs are introduced, the language model needs to be extended. For instance, to implement the side effect of evaluating the keyword setpencolor, we need access to the pen color of the Turtle objects. Therefore, the developer creates a subclass ExtendedTurtle that extends the base class Turtle and that provides a setter setPenColor for adjusting the pen color. A consequence of extending the Turtle class is that we also have to override the turtle keyword in ExtendedLogo, so that the overridden keyword method instantiates this subclass instead of base class Turtle.

Note that, because of the indirection that the code block's meta-object separates a program from the façade that evaluates it, the developer can select a façade instance with the right syntax and semantics for his or her requirements. This façade can be one of several possible extensions from an inheritance chain of language extensions.

While subclasses of an extension are allowed to extend and override existing syntax and semantics, there is an important guarantee. As long as an extension is conservative (cf. Section 3.1.2.1, p. 34), object-oriented polymorphism enables a controlled extension of the execution semantics for existing programs. When using such a conservative extension to evaluate a program that has been written for the base language, the extension evaluates the program whereby it obtains equivalent results as if the base language would have been used for evaluation. That means, in this case, the code block's meta-object implementing a default OO dispatch preserves *backward compatibility* for base programs.

4.2.1.2. Abstracting over Semantics

In the reflective embedding architecture, one can customize the evaluation of programs using object-oriented composition to enable a new syntax-to-semantics binding for a language at runtime. Given a language interface with its default semantics, a language developer can provide alternative semantics by implementing the language interface with an alternative language façade and model.

To bind the syntax to its execution semantics for a program, one sets up an instance of the façades as the code block's meta-object of this particular program. Abstracting over semantics of a program is inspired by the idea of recursively folding over HOAS proposed by Carette et al. [CKS09], and pluggable semantics proposed by Hofer et al.[HORM08]. Nonetheless, REA extends these two existing concepts with *runtime pluggability*.

Support for abstracting over semantics and pluggability are crucial properties in REA, since REA's other mechanisms are based on top of these properties. In particular, these two properties enable analyses and transformations in REA.

4.2.1.2.1. Analyses

Abstracting over semantics supports implementing analyses. For example, Figure 4.6 presents two different façades for EXTENDEDLOGO that both implement its language interface IExtendedLogo. The first façade is implementing the default execution semantics of EXTENDEDLOGO.

The second façade reimplements the complete language interface IExtendedLogo in the façade class ExpressionCounterExtendedLogo, i.e., it binds the syntax to a totally different semantics—a so-called *analyzer façade* that implements an analysis as an abstract interpretation that counts

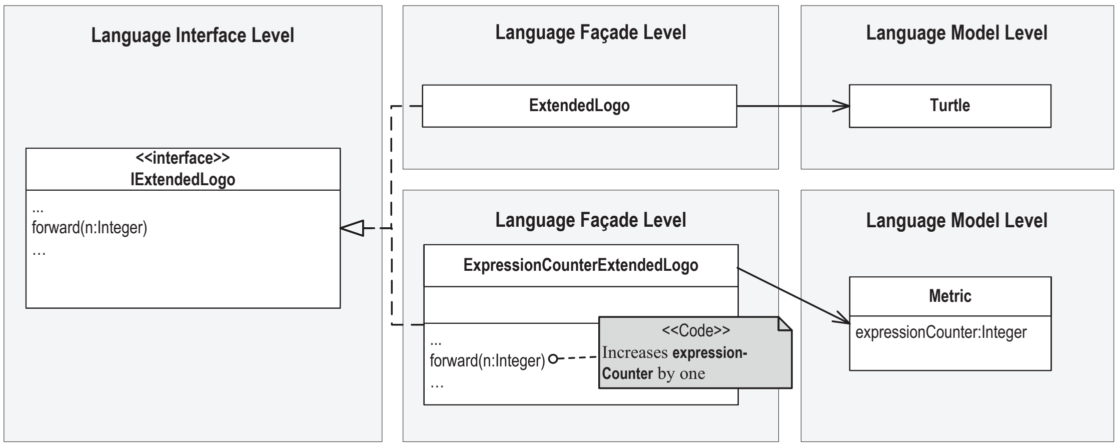


Figure 4.6.: Two different semantics for EXTENDEDLOGO

expressions. Instead of drawing the turtle on a canvas, ExpressionCounterExtendedLogo implements the alternative semantics for every keyword method declared in the IExtendedLogo interface. In the abstract-interpretation implementation, every method implementation increases an expressionCounter value. Note that, as Figure 4.6 indicates, for its abstract interpretation, the ExpressionCounterExtendedLogo uses a different language model—the domain of the metric for expression counting, which is modeled as the class Metric that holds the expressionCounter as an Integer. Note that Section 10.6.1 (p. 247) presents the implementation details of the above analyzer façade.

Abstracting over semantics support is not limited to local analyses that analyze expressions in isolation to each other, like expression counting. It is also possible to implement analyses with global strategies that abstract over the concrete expression types, similar to global strategies and that can take into account context information in source-transformation systems [BV04, Cor06]. To implement a global analysis, the developer needs to implement an analyzer façade with a meta-level—a meta-façade (cf. Section 4.1.1.4). Such an analyzer is therefore called a *meta-analyzer*. such a meta-analyzer comes with a special meta-object that implement a pattern matching on method calls and an a generic strategy how to analyze these calls.

There are several example use cases of meta-analyzers. In particular, meta-analyzers enables to analyze method calls from DSL programs from different languages, e.g. to detect ambiguous expressions in a language composition, as it will be discussed later on in Section 4.2.3.2

4.2.1.2.2. Transformations

Default execution and analyzes are not the only use cases for abstracting over semantics, language polymorphisms also enables transformations. Technically, transformations are similar to analyses, except that the domain is not an abstract domain model but a concrete target language model. The target language model is subsequently used to execute the transformed program.

For a given transformation, embedded DSL programs are encoded in the source language. The transformations have at the following levels in REA. It has a special façade that implements the language interface of the source language and they have a special language model—the target language model. When using the transformation to evaluate the program under the transformation semantics, the evaluation creates another representation of the evaluated program in the target model. The transformed target model can then be pretty printed into a string representation or itself be executed.

In REA, transformations support both *exo-transformations* and *endo-transformations*. REA support implementing exo-transformations as embedded compilers. In particular, Section 10.3.2.2.2 (p. 229) implements an embedded compiler for standard SQL.

Further, REA supports implementing endo-transformations with special transformer façade that defines a meta-level. Such a transformer is called a meta-transformer. A meta-transformer has a special meta-object that adapts method calls on the transformer façade, and it dispatches the adapted method calls back to the façade. What is special about endo-transformations is that they enable the transforming programs that are running.

A meta-transformer allows implementing strategies for a transformation. Similar to global strategies in analyses, meta-transformers enable global transformation strategies to be implemented that abstract over concrete expression types to be transformed. Further, a meta-transformer can take into account context information for the transformation results. There are several example uses cases of meta-transformers in REA. Specifically, meta-transformers enables to adapting method calls from DSL programs, e.g. to define a resolution strategy that disambiguates expressions in a language composition, as it will be discussed later on in Section 4.2.3.2.

4.2.1.3. Program Evaluation under Alternative Semantics

To calculate the average time to draw commands in a program, we can evaluate the same program twice on both façades. When using the ExpressionCounterExtendedLogo façade to evaluate the program presented in Listing 4.2, the façade performs an abstract interpretation that counts 25 sub-expressions[7] for the program.

With OO composition, language polymorphism enables a dynamic syntax-to-semantic binding because of the three important consequences explained below.

First, a façade does not have to implement a fixed syntax-to-semantics binding itself. For example, a façade can hold references to other façades and delegate method calls to these façades to obtain a syntax-to-semantic binding from them. This allows implementing modular façades using a set of collaborating classes.

Second, it is even possible to make the choice of the concrete façade for a language at runtime, e.g. when the right façade depends on the available runtime infrastructure.

Third, multiple sophisticated alternative semantics are conceivable for one particular language. With such multiple semantics, we process the same program several times, each time

[7]The program in Listing 4.2 has 25 sub-expressions: $1 \times$ turtle, $1 \times$ red, $9 \times$ forward, $9 \times$ right, $1 \times$ penup, $1 \times$ left, $1 \times$ pendown, $1 \times$ black, and $1 \times$ setpencolor.

```
{
  turtle ("A_red_and_a_black_square",red,{
    forward(50); right(90); forward(50); right(90); forward(50); right(90); forward(50); right(90);
    penup();
    right(90); forward(100); left(90); //goto (x=100,y=0)
    pendown();
    setpencolor(black)
    forward(50); right(90); forward(50); right(90); forward(50); right(90); forward(50); right(90);
  })
}
```

Listing 4.2: A program using the EXTENDEDLOGO language extension

using another façade. In such a processing chain, it is possible to first analyze a program (e.g. testing a property), and then, to only execute it depending on the result of the analysis (e.g. only if a required property is fulfilled by the program).

In the context of this thesis, the resulting dynamic syntax-to-semantic binding is used to implement versatile semantics for embedded languages. The architecture's support for abstracting over semantics is used to enable language compositions as well as various analyses and transformations that allow analyzing, or respectively transforming, the representation of a program under evaluation. To illustrate support for program analyses, Section 10.6.1 (p. 247) shows how to check coding conventions in DSL code, and Section 10.6.2 (p. 254) demonstrates how to check semantic invariants for DSL programs. To illustrate how abstracting over semantics enables transformations, Section 10.7 (p. 258) demonstrates optimizing embedded programs.

4.2.2. Meta-Level Extensions: Semantic Adaptations using the Meta Level

The architecture supports *meta-level extensions*, which enables gray-box extensions of existing language implementations – without changing their code. A meta-level extension is a set of meta-objects that extends a language implementation. A meta-level extension is necessary, when a language end user has a special requirement on an existing language, but it is not possible to adapt the code of the language implementation. In these situations, *gray-box extensions* can adapt internals without changing the code, and therefore, they can better address scenarios, in which there are multiple conflicting requirements on one language implementation, than extensions with language polymorphism. Gray-box extensions are necessary when a new requirement invalidates parts of the language specification and its implementation. If only one user has this requirement, it may be too much effort to maintain a separate language extension. To cope with language evolutions of single users, in the reflective embedding architecture, a language developer can adapt parts of a language's semantics in the user-domain. We refer to such adaptation as a *semantic adaptation*. To adapt the semantics of a language embedding, the reflective embedding architecture allows adapting the internals of a language implementation by using the host language's meta-object protocol. Note that if there is an educated user that knows how to use the host language's meta-object protocol, even this user can semantically adapt a language.

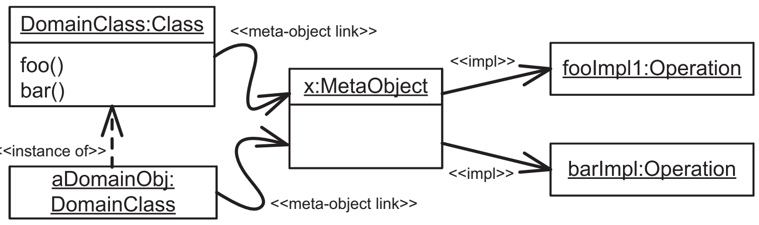


Figure 4.7.: The meta level for language implementations

For adaptations, the developer or the user often do not need to know the implementation details of the adapted language but only its interfaces.

The presence of the meta level in the architecture is fundamental to allow semantic adaptations. The language semantics are first-class because the semantics of the language constructs are reified in meta-objects as indicated by the abstract example in Figure 4.7. In a language implementation, every class (such as DomainClass) is associated with such a meta-object (such as x), which initially is the default meta-object of any new instance of this domain class. A meta-object dispatches the method calls received by a domain object to a concrete implementation of its corresponding operation. The links meta-object and/or impl can be changed at runtime, which is the key to allow dynamic semantic adaptations. The setup of the links to the meta-object and its links to the operations determine the final semantics of a language construct, because the meta-objects are responsible for handling the evaluation of each language construct. For each language construct, its final semantics is defined by dispatching each method call in the evaluation process via meta-object links and over the operation link to the right implementation of the domain class operation. Initially the links of the default meta-object refer to the default implementation of the corresponding domain operation as defined e.g. in the language model. E.g., to adapt a language's semantics, a language designer can exchange the links to alternative meta-objects or to alternative operation implementations.

To semantically adapt a given language with its default semantics defined in an existing implementation, a language developer who knows the user's domain provides special meta-objects that adapt the execution of its façade and model. Specifically, the language developer provides specialized method implementations that encode an alternative semantics for the façade and model. The new meta-objects dispatch to specialized method implementations that provide an alternative semantics for the language implementation. E.g., by using reflection to change the classes of the language implementation, one can replace parts of the evaluation protocol, which effectively adapts the language semantics.

For example, Section 3.1.2.2 (p. 35) has mentioned that when growing *Logo* with support for concurrency, the language implementation additionally has to guarantee that there must be no collisions between physical turtles controlled by a *Logo* program. A meta level extension enables a modular enforcement of such constraints using a specialized meta-object, similarly to the class ConstraintCheckingMetaObject as indicated in Figure 4.2. In this scenario, the special meta-object changes the method dispatch of the language implementation to realize an alternative semantics that guarantees absence of collisions. Instead of directly dispatching a method call to a method that moves the turtle, the special meta-object dispatches every call to an operation that moves the turtle to an alternative method implementation that first checks for possible collisions and prevents them before actually moving the turtle. Later, Section 10.1.2.2 (p. 215) elaborates the implementation of this semantic adaptation scenario in detail.

Semantic adaptations allow reusing the existing language implementation while adapting the evaluation of programs with alternative meta-objects. When encountering keywords in the program, the keyword-method calls and calls inside the language model are then handled differently. With a meta-level extension, the alternative meta-object can dispatch calls inside the language implementation to alternative method implementations that may encode different semantics. Therefore, existing language implementations can be extended without changing existing classes, even though no façade or model is exchanged, this still results in an adapted evaluation of the program.

It can be complicated to extend a language's semantics at the meta level, since the developer needs to select what parts of the language to extend. Therefore, language developers require means to express where meta-level extensions should be effective. To meet this demand, in Section 4.2.2.1, the architecture provides semantic adaptation scopes to let a language developer selectively control adaptations on any object in the architecture. Section 4.2.2.2 discusses what extensions points in the architecture a language developer can extend with these scopes.

4.2.2.1. Semantic Adaptation Scopes

When adapting objects' fields and methods at the meta level, the language developer needs to control what parts of object are adapted by a meta-level extension. To control adaptations, the architecture proposes to precisely scope semantic adaptations along three kinds of dimensions:

1. *locality of adaptations* (Section 4.2.2.1.1),
2. *atomicity of adaptations* (Section 4.2.2.1.2), and
3. *time of adaptations* (Section 4.2.2.1.3).

These dimensions help the different stake holders of a language to collaboratively provide parts of the semantics. First, they guide the language developer who designs a language where it is possible to plan for extensions. Second, they guide the language developer who implements the language how to build an implementation that is open for extensions. Third, they guide the language developer who customizes the language for the end users, or the end user himself, how parts of the language implementation can be extended.

4.2.2.1.1. Dimension: Locality of Adaptations The first dimension along which language semantic adaptations are classified is the *locality of adaptations*. For example, the semantics of method dispatch can be extended in certain scopes. This dimension is discrete and has 2 values:

D1.a) *global semantic adaptation* (class wide) and

D1.b) *local semantic adaptation* (instance local).

A global semantic adaptation affects the semantics of a certain language construct in general. For example, consider embedding an OO language by modeling OO language constructs such a classes, objects, methods, and fields. In such an OO language embedding, in the language model, there would be a type to model methods, say Method, and this type would have an operation, say dispatchCall(...), that encodes method dispatch semantics. Semantic adaptations that override that method allow adapting the method dispatch semantics for every method call from a *single dispatch semantics* to a *multi-dispatch semantics*. For a multi-dispatch semantics, it is necessary to takes into account the types of all method parameters for finding the most specific method implementation. Thus, for every call to method dispatchCall(...), the adaptation can execute another method instead, say dispatchCallWithMultiDispatch(...). Because, such a global semantic adaptation changes the semantics of all instances of a type, it scopes the adaptation of a language construct's semantics over all programs.

On the contrary, a local semantic adaptation affects only one particular usage of a language construct. For the previous example, since multi-dispatch semantics do not only take into account the receiver type but also the parameter types to find the right method implementation at runtime, it poses an overhead on every method call. Therefore, it can be advantageous to apply the semantics adaptation of the method dispatch mechanism only to the methods of those classes for which multi-dispatch is advantageous (e.g., classes that implement the *visitor pattern* [GHJV95]). Local semantic adaptations allow precisely controlling the scope of the effects of a semantics adaptation on parts of the language model.

Scoping adaptations can be used to precisely control the semantic adaptations in other programming paradigms. In aspect-oriented programming, either semantics of all aspects can be adapted or the semantics of only one specific aspect instance can be adapted. In a similar way for a domain-specific language, a language construct that represents a domain concept can be adapted.

To recapitulate, with global and local semantic adaptations, either the semantics of the abstraction is adapted in general or it is only adapted for one use of this abstraction.

To scope the locality of an adaptation, the architecture either adjusts the meta-objects of types or objects. Consider the abstract example in Figure 4.8 that shows how the two kinds of variability with regard to the locality dimension – domain type versus domain object – are supported in the architecture, as elaborated for each case below.

The upper part shows the effects of semantic adaptations whose scope is an entire domain type; the lower part corresponds to an adaptation that is specifically scoped for a particular domain object, thus, only domain objects are shown there. Both parts show the relation between domain types and domain objects to their corresponding semantics (encapsulated in a meta-object) before and after the adaptation.

In the upper left quadrant, the domain type A is bound to the default semantics represented by the meta-object x. Every new domain object that is created, e.g., a, runs under the default semantics. The semantic adaptation defines new semantics for domain type A. In the upper right quadrant, a new meta-object y is defined to represent the new semantics and A is associated with

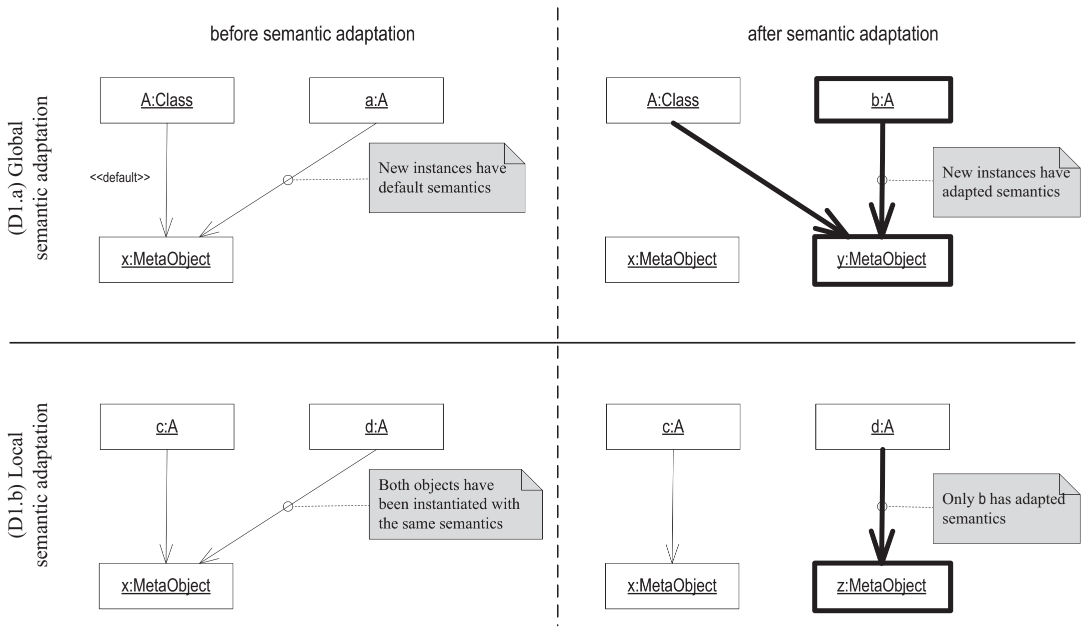

before semantic adaptation
after semantic adaptation
(D1.a) Global semantic adaptation
A:Class
a:A
<<default>>
New instances have default semantics
x:MetaObject
A:Class
b:A
New instances have adapted semantics
x:MetaObject
y:MetaObject
(D1.b) Local semantic adaptation
c:A
d:A
Both objects have been instantiated with the same semantics
x:MetaObject
c:A
d:A
Only b has adapted semantics
x:MetaObject
z:MetaObject

Figure 4.8.: Dimension 1 – locality of adaptations

it. Any domain object created subsequently runs under the new semantics: the domain object b is created after the adaptation, hence, it is linked to the new meta-object y. Objects that were created before the semantic adaptation continue to run under the previous semantics, e.g., a is still linked to the meta-object x.

In the lower left quadrant, the domain objects c and d are created with the same semantics. The object-level semantic adaptation depicted here modifies the semantics of d **only**. The lower right quadrant shows the domain objects, meta-objects, and their relations after the semantic adaptation has taken place. While c keeps the former semantics, d uses the new semantics represented by meta-object z.

As a concrete example for using the scope of variability in the *Logo* domain, consider adapting the execution semantics for an optimization in the same way for every turtle in a program, in contrast to, adapting only the execution semantics of one particular turtle in an individual way. With a global semantic adaptation, it is possible to adapt the execution semantics of all turtles in a program for using a standard optimization. Conversely with a local semantic adaptation, we can restrict an adaptation to a single instance. This allows a selective optimization for an individual turtle. A selective optimization only affects the drawing operations of a particular turtle, while all other turtles execute with the standard optimization semantics.

4.2.2.1.2. Dimension: Atomicity of Adaptations The second dimension of semantic adaptations is the atomicity of adaptations that are made. The size of these adaptations ranges from parts of the semantics of a language construct to its complete semantics:

D2.a) *domain-operation level*: exchanges only one operation or field of a certain type

...

D2.b) *module level*: exchanges all operations or fields of a certain type

Indeed changing one part of a language's semantics often requires also changing another part of the semantics, When several parts need to be adapted, these multiple "elementary" semantic adaptations must be packed as one *unit of adaptation*. For example, when adapting the method dispatch in an OO language, often the field dispatch must be adapted as well. In the case of adapting the language model of the OO language, we must not only adapt the implementation of Method.dispatchCall, but also the dispatch logic for fields, say by overriding the methods Field.dispatchRead and Field.dispatchWrite. To gain altered semantics that are consistent, it is important that multiple of such adaptations are applied at once.

To enable adaptations with a different atomicity, the architecture applies one meta-object that contains one or several adaptations as one unit. Consider the abstract example in Figure 4.9, which depicts how adaptations at different levels of atomicity are supported in the architecture. The upper part shows the most fine-grained semantic adaptation at the level of an atomic domain operation. Unlike Figure 4.8, the meta-objects are represented along with the domain operations. Doing so, we can highlight that the adaptation can separately impact a particular operation. In the upper left quadrant, the object a is attached to meta-object x; in the upper right quadrant, the same object is attached to a new meta-object y, which is the result of cloning x and binding foo to a new implementation, called fooImpl2. This way, the whole default semantics gets reused except the re-bound domain operation(s).

In general, it is likely that a semantic adaptation affects several places in the default implementation of the semantics. Obviously, it is preferable to apply the adaptations together as a unit of semantic adaptation. In contrast to the atomic adaptation at the level of a single domain operation, in this case the changes have to be packed into a bigger variability unit. This abstract unit is depicted as the gray rectangle in the lower left part of Figure 4.9. When an adaptation happens, all changes of this adaptation unit are performed in concert. The impacted domain objects are then bound to a new meta-object, which is the result of mixing the previous meta-object and the semantic adaptation unit. In Figure 4.9, after the adaptation, the domain object b is attached to the meta-object z, which binds both foo and bar to new implementations.

As a concrete example for using the atomicity of adaptations in the *Logo* domain, consider adapting the execution semantics for several keywords at once. When we want to prevent that the pen color can be changed, we must adapt the semantics of only one operation—the setpencolor

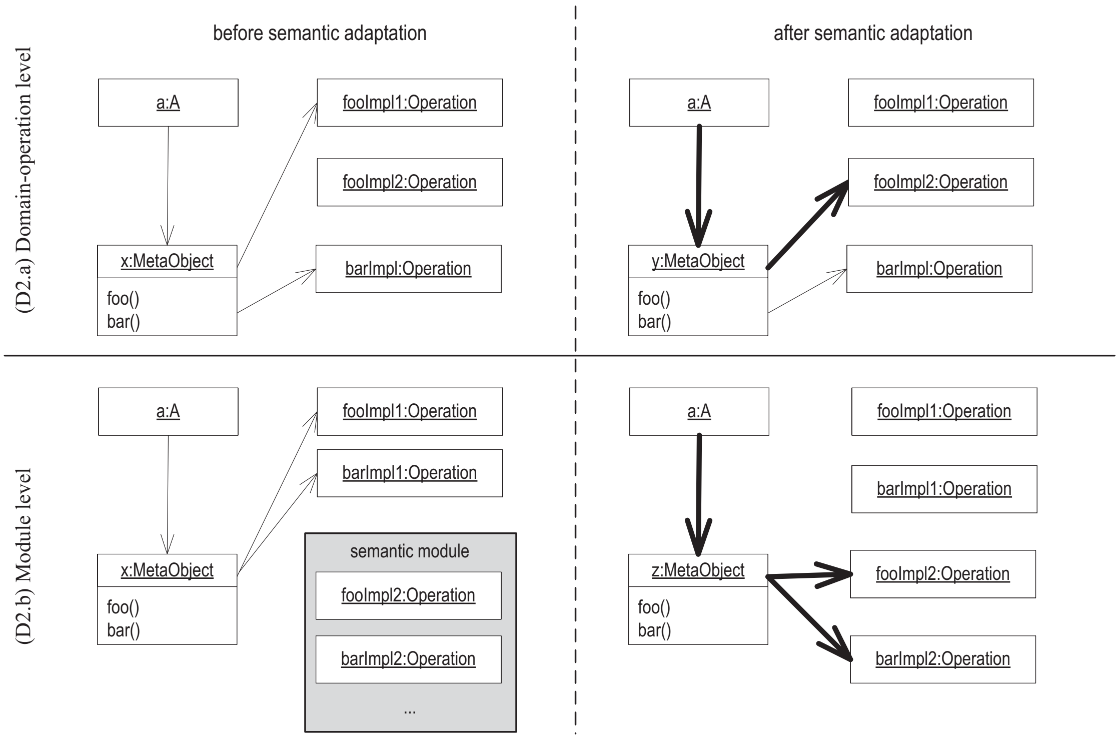


Figure 4.9.: Dimension 2 – atomicity of adaptations

operation. We can replace the operation that implements the semantics of this keyword with an empty implementation. Conversely, when we want to prevent collisions between turtles, we need to adapt the semantics of all drawing operation. By applying adaptations to multiple domain operations in one atomic step, we can ensure that a pre-condition is checked for all drawing operations before these operations are actually executed.

4.2.2.1.3. Dimension: Time of Adaptations The third dimension characterizes the relation between the point in time in which the semantic adaptations happen and the point in time when DSL programs are executed:

D3.a) *pre-execution semantic adaptation*, and

D3.b) *context-dependent semantic adaptation*.

In most cases, the right semantics for a program execution can be determined beforehand and stays fixed for a complete program run. But sometimes, the selection of the right semantics depends on the execution state of a DSL program, i.e., a change in the DSL program context triggers a semantic adaptation.

To enable different times of adaptation, the architecture distinguishes when meta-objects are replaced. Consider the following two abstract examples of execution traces that illustrate the

(D3.a) Pre-execution semantic adaptation

```
--- Loading Interpreter
+++ Adaptation
+++ Adaptation
%%% Program Execution
%%%
%%%
%%%
%%% End of Execution
```

(D3.b) Execution context-dependent semantic adaptation

```
--- Loading Interpreter
%%% Program Execution
%%%
+++ Adaptation
%%%
+++ Adaptation
%%%
%%% End of Execution
```

Figure 4.10.: Dimension 3 – time of adaptations

two possible points in time when adaptations may take place. Figure 4.10 shows the difference in the execution traces of a pre-execution adaptation and a context-dependent adaptation. In both traces, the first step is to load the DSL program of which the corresponding trace is prefixed by "−−−". Then, two kinds of execution steps can occur: semantic adaptation ("+++") and normal DSL execution ("% % %").

The left-hand side of Figure 4.10 schematically depicts a pre-execution semantic adaptation: the semantics of the DSL changes before any domain object is created, or any call to a domain operation has occurred. Note that multiple adaptations can be applied as indicated. Then, the DSL program is evaluated until completion. In this case, the adaptation is independent of the DSL execution.

The right-hand side of Figure 4.10 depicts an execution of a context-dependent semantic adaptation. Unlike the previous trace, the adaptation happens during DSL execution, depending on the concrete values of domain objects. This is symbolized by the interlacing of several regular domain operation execution and semantic adaptation steps. Such context-dependent adaptations enable semantic self-adaptation of DSL programs.

As a concrete example for using the context-dependent semantic adaptation in the *Logo* domain, consider adapting the execution semantics for the domain operations at different times. When we know that using a static optimization strategy is always beneficial, we can apply a semantic adaptation for the optimization before executing a *Logo* program. Conversely, there are optimizations for which it cannot be decided whether the optimization is beneficial, because it depends on the application context whether there will be an improvement when using this optimization. In such cases, the architecture allows using a context-dependent semantic adaptation to apply the optimization at runtime when we know that this condition is met.

Context-dependent semantic adaptations deserve a closer discussion. Therefore, Section 6 (p. 135) motivates using context-dependent adaptations to enable *self-adaptive* [Hor01] in DSLs, and demonstrates more concrete use cases and that elaborates their implementation. Furthermore, other applications of context-dependent adaptations are conceivable, such as for implementing *runtime checking/verification* [CR06] and *dynamic activation* of language constructs [PGA01, CH05, Tan09].

4.2.2.2. Extensions Points of the Architecture

After we have discussed how dimensions allow controlling adaptations by restricting their effect to particular regions, this section gives an overview of the flexibility of the meta level that enables late adaptations in the reflective embedding architecture. The meta level enables flexible semantic adaptations, however the line of reasoning is not that every possible semantic extension of a language should be implemented using late adaptation. Often, it is sufficient to extend a language with polymorphism, which is preferred. Rather, the argumentation line is that late adaptation should be used whenever polymorphism does not facilitate a certain extension scenario. To understand how late adaptation are applied, we first discuss the general process, and second we discuss what parts of a language late adaptations can change.

There is a general process for how developers collaborate for adapting language implementations in the user domain. One the one hand, to control what is adaptable, the language developer who provides a language implementation exposes various extension points that can be adapted. On the other hand, to control what to adapt, the language developer who wants to customize a language implementation for the end user can specialize the language at these extension points.

There are various kinds of adaptations and possible applications that are supported by the meta level. Central to the meta level is that the structure and execution behavior of a language implementations that resides inside the architectural levels are reflected as first-class entities via the meta-objects, and therefore these first-class entities can be adapted. Since the meta-objects expose the structure of classes in the language implementation, one can add structural elements to classes or change existing elements, i.e., one can add fields and methods to those classes. Since those meta-objects also expose the behavior of the classes in the language implementation, developers can adapt the execution of these classes, and therefore the language implementation.

By intercepting method calls and field accesses on those classes, developers can manipulate calls to methods and accesses to fields. For every method call, the meta-object can manipulate the receiver, the method name, and its parameters, to provide alternative execution semantics. For every field access, the meta-object can manipulate the receiver, the field name and value to be stored or to returned. Changing a method call or field access results in different computations and therefore alternative language semantics.

For instance, to adapt SIMPLELOGO to enforce a domain-specific constraint, a special meta-object of the `Turtle` class can reify every call to the move operations, such as the `moveForward` operation; and before executing such a call, it can intercept the call and check whether there will be a collision and prevent it.

To understand what adaptations are possible in general, it is important to understand the consequence of meta-objects adapting a particular level in the architecture. At each level of the architecture, the extension points allow adapting different parts of a language implementation, which is determined by what extension points are exposed by the corresponding level. As we will elaborate in the following paragraphs, (1) at the interface, one can adapt the abstract syntax of a language, (2) at the façade, one can adapt its syntax-to-semantic binding, and (3) at the model level, one can adapt its evaluation protocol.

(1) Language Interface Level: Each language interface defines an extension point for a language's abstract syntax and its expression types. With a special meta level for the interface level, a language developer can adapt groups of languages. The meta-objects can quantify over calls to several façades, and they can selectively adapt those façades that implement a certain interface or that define certain expression types. Semantic adaptations at the interface level are similar to extending the language interface using language polymorphism, but such adaptations are more flexible, since the meta-level extension must not be part of one inheritance chain of the adapted languages.

While language polymorphism extends only one language, adaptation with such a meta level makes it possible to add/remove a particular expression types to a group of languages, e.g., by adding new keyword methods to all façades that implement a given interface, or respectively by removing existing keyword methods from them.

For example, we can use this to adapt the syntax of the extensions MATHLOGO and MATHBUSINESSRULESDSL from Section 3.2.1 (p. 37) that both define common mathematical keywords of a language interface IMathLanguage (e.g., the domain operations sum, sin, define, and apply). To simplify the syntax of these two language extensions, the developer can remove keywords from them that are too complex for, or not required by, the end user (such as, the keywords sum and sin) . To do so, a language developer defines a new meta-object that intercepts keyword method calls and does not proceed a call if a certain condition is not met, such as the condition that an object implements a certain interface (such as IMathLanguage) or such as that the call has a certain signature (such as, in sum(List<int>) or int sin(int)). For realizing the above example, the developer defines the meta-object with the condition that the call is not proceeded if the call's signature has the method name sin or sum. Thus, effectively the meta-object hides the corresponding methods (e.g., sin and sum) from all façades, but allows all other keywords (e.g., define and apply). With respect to the adaptation scope, this example adaptation has a *global* locality (D1.a) since it changes all façade instances of type IMathLanguage, its atomicity is at the *module-level* (D2.b) since it removes two methods, and its time of adaptation is *pre-execution* (D3.a).

While language polymorphism allows extending a language statically, a semantic adaptation makes it even possible to change the abstract syntax at runtime, similar to *extensible grammars* (cf. Section 2.3.2, p. 25). Developers can define strategies that extend a language's abstract syntax depending on the keywords the language defines. This is interesting, if developers want to apply an adaptation but they do not know the specific syntax of the language to be adapted. For example, consider adding a new expression type depending on the availability of another expression type. The developer can implement an adaptation that adds a new expression type, say cos, but only if the language defines the expression type sin. Note that this adaptation can be independent from the concrete semantics of the sin expression type. Yet, the semantics of the cos expression type can be derived from the concrete semantics of sin, e.g. $cos(\alpha) = sin(90° - \alpha)$. This way, the adaptation can be applied to different languages, no matter how they implement sin. To adapt the available abstract syntax of a particular language implementation at runtime, a language developer adjusts the *meta-object link* of the façade classes to point to a special meta-object that adds a corresponding method to the façade for the expression type. With respect to the scope of this adaptation, the locality is *global* (D1.a) since it changes all façade instances of type IMathLanguage, the locality is *operation-level* (D2.a) since add only one method, and the time is *context-dependent* (D3.b) since cos is only available when sin is present.

While a hierarchical language extension is specific for a base language, a semantic adaptation can be generic for a group of languages implemented on top of the reflective embedding architecture. For example, we adapt a language's abstract syntax with a generic strategy that enables case-insensitive syntax in an otherwise case-sensitive embedding in the host language. To make the abstract syntax case insensitive, a language developer can use a special meta-object that intercepts every method call to some façade and changes the method dispatch semantics to ignore the case in method names, when looking up the most concrete method implementation that binds the syntax to semantics. With respect to the adaptation scope, the locality is *global* (D1.a), the locality is *module-level* (D2.b) since all expression types become case insensitive, and the time is *context-dependent* (D3.b) since the strategy depends on what expression types are provided by the adapted façades.

The use of the meta level enables a greater flexibility than what is available in embeddings without a meta-level. In traditional embeddings, the abstract syntax of an embedded language is fixed and cannot be changed. In contrast, in the reflective embedding architecture, the abstract syntax can be extended at any time.

(2) Language Façade Level: Each language façade defines an extension point for the syntax-to-semantic binding of each expression type. Therefore, at the meta level, a language developer can adapt the syntax-to-semantics binding of particular façade instances. Semantic adaptations at the façade level are similar to façade extensions using language polymorphism, but such adaptations are more fine-granular (i.e., per expression type) and can be more generic (i.e., for several base languages).

While inheriting from a façade establishes a new complete syntax-to-semantic binding, adapting a façade allows more fine-granular changes to the syntax-to-semantic binding at the granularity of expression types. Developers can use this flexibility to adapt the syntax-to-semantic binding of an expression type to use another evaluation mode, e.g. changing it from *immediate* to *delayed*. For example, we can redefine the syntax-to-semantic binding of the turtle expression to delay the keyword's evaluation such that it executes drawing commands in a separate thread, as motivated in Section 3.1.2.1 (p. 34). The meta-object of the façade can intercept calls to the abstraction operator method turtle and proceeds the call in a new thread. With respect to the scope of this adaptation, the locality is *global* (D1.a), the locality is *operation-level* (D2.a), and the time is *pre-execution* (D3.a).

While inheriting from a façade establishes a complete syntax-to-semantic binding for all expression types of one particular language, a semantic adaptation to a façade allows selectively changing the syntax-to-semantic binding of only one expression type, or defining a generic strategy that changes the syntax-to-semantic binding of expression types that do not need to be from the same language. For example, a generic strategy can extend the syntax-to-semantic binding of FUNCTIONAL to allow short function applications, as motivated in Section 3.7.1.1 (p. 60). When a program calls a function directly (e.g., via the expression “square(50)” in abstract syntax) without using the apply keyword, then in FUNCTIONAL, by default, there is no syntax-to-semantic binding defined for this expression type in the façade class (e.g. there is no keyword method square(Integer)). However, the meta-object can intercept the method call to the façade (e.g., square(50)), and next, the meta-object can reuse the syntax-to-semantic binding of the apply keyword to evaluate the expression (e.g., apply("square")(50)). Since the meta-object makes only the assumption that the apply keyword needs to be present, the same meta-object can be used to adapt not only the stand-alone language FUNCTIONAL, but also any other language that implement this keyword method, such as the FUNCTIONALLOGO extension. With respect to the scope of this adaptation, the locality is *global* (D1.a), the locality is rather *operation-level* (D2.a) since only the syntax-to-semantic binding of the apply keyword is extended, and the time is *pre-execution* (D3.a).

(3) Language Model Level: When extending its model at the meta level, this adapts the language’s evaluation protocol. Semantic adaptations at the model level are similar to model extensions using language polymorphism, but such adaptations are more modular and flexible.

While a model extension enables extending single domain types, a semantic adaptation in a model supports a modular adaptation to multiple parts of the evaluation protocol. For example, in the context of SIMPLELOGO, developers can adapt its language model to enforce domain constraints. Such adaptation dynamically analyzes the sequence of calls to the operations on the first-class representation of a Turtle in the language model, and prevents the execution of the operation if a constraint would be violated, as motivated in Section 3.1.2.2 (p. 35). To enforce the aforementioned constraint, a language developer defines a special meta-object that adapts the Turtle class by overriding its methods, such that a constraint is validated to prevent collisions. The benefit of the adaptation is that the meta-object encapsulates the logic to enforce the constraint, which is therefore not scattered across several methods and classes.

With respect to the scope of this adaptation, the locality is *global* (D1.a), since the constraint is checked on all Turtle objects, the locality is *module-level* (D2.b), since all move operations of the turtle are checked, and the time is *pre-execution* (D3.a).

While extending a model established only a new closed version of the evaluation protocol, a semantic adaptation in a model can adapt the evaluation of expressions at runtime. For example, we can use this to adapt the evaluation protocol to realize dynamic aspects, e.g. as motivated for SIMPLELOGO in Section 3.3.2.1 (p. 44). A language developer installs a special meta-object that reifies join points when executing AO4LOGO programs. Only when aspects are deployed and a certain pointcut matches at the current join point, this meta-object executes the corresponding advice, e.g. it executes an aspect that implements the tracing concern. With respect to the scope of this adaptation, the locality is *global* (D1.a), since aspects are deployed globally for all types, the locality is *module-level* (D2.b), since all domain operations are traced, and the time is *context-dependent* (D3.b), since undeployed aspects do affect the execution of method in the model.

4.2.2.3. Program Evaluation under Adapted Semantics

When evaluating a program under the default semantics, the default meta-objects execute the language implementation, still they do not play a significant role, since they execute the objects in the language implementation with standard OO dispatch. In contrast, when evaluating a program with special meta-objects, these meta-objects change the execution of the language implementation, resulting in an execution with adapted semantics.

At a first glance, semantic adaptations with method objects seems similar to abstracting over semantics with language polymorphism, since both result in alternative program executions. But, semantic adaptations are not restricted to black-box extensions, since they can extend language implementations with new keywords, adapt keywords, and remove existing keywords. Semantic adaptations use meta-objects for gray-box extensions. Semantic adaptations are not restricted to hierarchical extensions. They can extend existing language interfaces, and override existing fields and methods. For the above reasons, semantic adaptations allow a more flexible late binding of syntax-to-semantics than language polymorphism.

To set up a program for execution under alternative semantics, therefore, there are only minor adaptations necessary for the three steps of the standard evaluation process from Section 4.1.2 (p. 79).

In the compilation step, nothing needs to be changed. Nonetheless note that after compiling the default language implementation, the language developers can compile the special meta-objects separately from the code of the default implementation.

In the setup step, there is a simple modification: not only the code block's façade is set up, but in addition, the alternative semantics is enabled by associating special meta-objects with the classes of the language façade and its model.

In the expression-evaluation step, there are no changes necessary and the semantic adaptations are applied completely transparent from the viewpoint of the user. The special meta-objects are in place and adapt the evaluation of the program running on top the unchanged language implementation. The special meta-objects can adapt the semantics of every language construct by dispatching method calls to alternative method implementations or fields.

For example, we can use a meta-level extension to adapt the execution semantics of SIMPLELOGO. Consider evaluating SIMPLELOGO with the ConstraintCheckingMetaObject from Figure 4.2. When a program uses the keyword forward, the corresponding keyword method calls the operation moveForward on the domain type Turtle. Instead of directly executing the operation, the ConstraintCheckingMetaObject intercepts the call and possibly executes an alternative semantics instead. Its alternative semantics first checks a set of pre-conditions, and if all pre-conditions are met, it proceeds with executing the original semantics, otherwise it reports an error. This way, the meta-object allows extending the evaluation protocol of SIMPLELOGO without changing the language's implementation.

From the perspective of the execution semantics, when executing a program under an adapted semantics using a special meta-level, the meta-objects partially contribute to the evaluation of programs, i.e. they determine the final semantics. When embedding a language into a reflective host language with a MOP, its meta-objects enable a more flexible embeddings, since programs become first class, they can be inspected and manipulated.

4.2.3. Extensible Languages Combiners

The reflective embedding architecture enables language composition with so-called *language combiners*. Composing languages enables evaluating programs that mix expressions of overlapping domains. The language developer selects an appropriate language combiner based on a *composition specification*[8] for these languages. Given a set of language implementations, language developers can use a combiner to compose the syntax and semantics defined in the languages. For example, reconsider composing FUNCTIONAL and COMPLETELOGO to execute FUNCTIONALLOGO programs, such as the one presented in Figure 3.2 (p. 39), which mixes expressions of the FUNCTIONAL and COMPLETELOGO domains (e.g., the keywords apply of FUNCTIONAL and forward of COMPLETELOGO).

Language combiners are a mechanism that enables language developers to conveniently compose languages that have been developed independently. By using an combiner, they can reuse several modular language implementation, thus they save time for implementing the composition by relying on the composition logic that the combiner provides. When language developers want to realize a language composition of several stand-alone languages, such as FUNCTIONAL and COMPLETELOGO, they likely want to reuse existing implementations of the constituent languages for the composite language. When using an appropriate combiner to compose several languages, in many cases, the implementations of the constituent languages do not need to be changed at all. The developers do not need to manipulate the code of their interfaces, façades, models, but they can extend them when necessary or desirable.

The reflective embedding architecture comes with a set of pre-defined generic language combiners for composing languages in certain *composition scenarios*. A *composition scenario* is defined for a particular set of assumptions on the languages to compose (e.g. all languages must be conflict-free). The language combiners are generic in the sense that they can compose any set of language embeddings as long as these languages meet their assumptions. For composing particular languages, the language developer is responsible for selecting an appropriate language combiner that provides the right composition semantics to meet the requirements of the composition specification.

Figure 4.11 presents the pre-defined language combiners of the reflective embedding architecture:

Independent combiners: Section 4.2.3.1 discusses the BlackBoxCombiner for composing *independent languages* that do not have syntactic or semantic conflicts

[8]A composition specification can be an informal or formal definition that defines how to compose the syntax and semantics of several languages. This thesis assumes an informal specification. In contrast, Kojarski et al. propose to use formal specification in form of denotational semantics [KL05, KL07b].

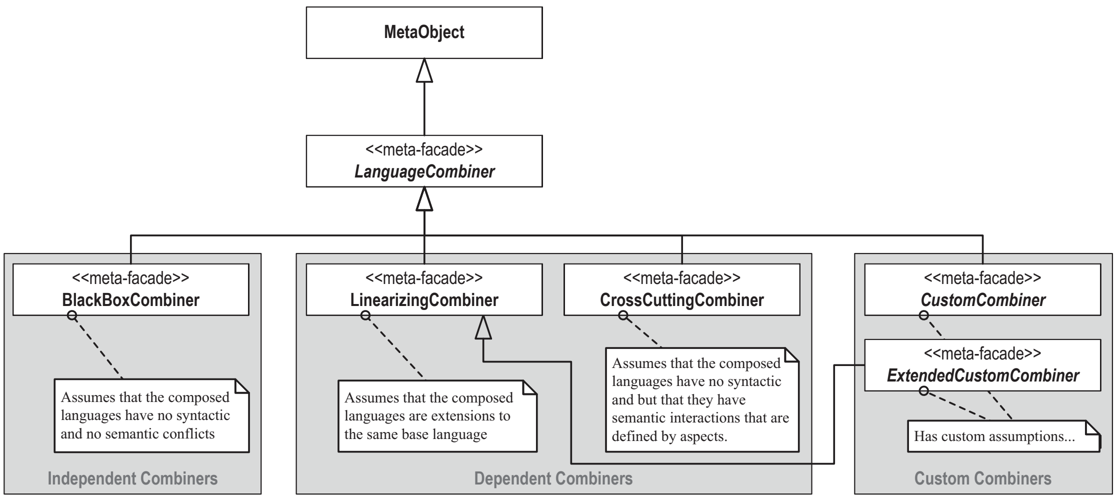

MetaObject
<<meta-facade>>
LanguageCombiner
<<meta-facade>>
BlackBoxCombiner
Assumes that the composed languages have no syntactic and no semantic conflicts
Independent Combiners
<<meta-facade>>
LinearizingCombiner
Assumes that the composed languages are extensions to the same base language
<<meta-facade>>
CrossCuttingCombiner
Assumes that the composed languages have no syntactic and but that they have semantic interactions that are defined by aspects.
Dependent Combiners
<<meta-facade>>
CustomCombiner
<<meta-facade>>
ExtendedCustomCombiner
Has custom assumptions...
Custom Combiners

Figure 4.11.: Overview of the language combiners

Dependent combiners: Section 4.2.3.2 discusses the LinearizingCombiner for composing *dependent languages* with interacting syntax and the CrossCuttingCombiner for composing languages with interacting semantics.

Custom combiners: In general, the set of possible scenarios for language compositions is not limited. Therefore, when no appropriate combiner is available, a language developer needs to implement new custom combiners, as discussed in Section 4.2.3.3.

4.2.3.1. Composing Independent Languages

The architecture provides the generic BlackBoxCombiner for composing stand-alone languages that have been implemented independently and that do not conflict. We call compositions of conflict-free languages *black-box compositions* because the combiner only takes into account the interfaces of the language to be composed, it treats the language implementations as black-boxes, and it composes the languages without intervening with the internals of the individual evaluation protocols. The BlackBoxCombiner composes several language implementations so that if one of the constituent languages defines an expression type, this expression type will be available in the language composition. The BlackBoxCombiner does not define a syntax and semantics itself. Instead, the combiner exploits the power of the meta-object protocol for composing the constituent languages' syntax-to-semantic bindings defined in their façades. For a meaningful composition, the BlackBoxCombiner requires that the constituent languages must not have syntactic or semantic conflicts. While the combiner detects presence of syntactic conflicts, and it enforces the absence of such conflicts in compositions, it is the responsibility of language developers to consult the composition specification of the languages to eliminate all doubts about possible semantics conflicts.

For example, reconsider composing the languages FUNCTIONAL and COMPLETELOGO. Since the two languages do not have syntactic and semantic conflicts, a language developer can use the BlackBoxCombiner to combine their constituent language implementations without changing their code.

Composition Protocol: Internally, a combiner implements a *language composition protocol*. In the reflective embedding architecture, this protocol uses OO composition and the meta-object protocol of the host language as the means for language composition. Language combiners take advantage of the levels in the reflective embedding architecture that implicitly expose points where to compose languages. At the language interface level, the combiner analyses the languages' interfaces to determine the syntax of the constituent languages to be composed. At the language façade level, it extracts the syntax-to-semantics bindings from the languages' façades. At the language model level, it accesses the evaluation protocols that are defined in the languages' models. At the language meta level, the combiner integrates the languages' syntax, syntax-to-semantics bindings, and evaluation protocols with its well-defined composition semantics.

Technically, a language combiner is a *meta-façade*. Further, because a combiner has access to the meta-level, its meta-façade can gain control over the meta-objects of the languages to be composed. Having access to the languages' meta levels enables the combiner to process meta-data of the constituent languages and to adapt their evaluation protocol to a composition. A combiner is a meta-analyzer that use the meta level to analyze the meta data defined in the language implementations of the constituent languages.

Having access to the meta level, a combiner can use meta-objects to integrate the syntax and the semantics for the language composition by combining the syntax-to-semantic bindings defined in their façades. Specifically, a combiner is a meta-transformer that intercepts, delegates, and adapts keyword method calls to integrate the evaluation protocols of the languages.

For integrating the evaluation protocols of several languages, when evaluating a program that mixes expression of several languages resulting in keyword method calls, the combiner needs to handle all possible keyword method calls. Each call represents the evaluation of one expression, which may result in further method calls inside one of the language models. Therefore, the combiner needs to handle all method calls inside the language models.

To adjust the evaluation of a keyword-method call in a composition, the combiner uses *reflection* to intercept, delegate, and adapt the method call for this keyword. For example, the BlackBoxCombiner intercepts every keyword method call received by the combiner. It analyzes the intercepted call to find the corresponding façade. The BlackBoxCombiner does not transform the call, but it only delegates the call the corresponding façade, which will evaluate the keyword and return a result, which is passed through the BlackBoxCombiner, which returns then finally returns the computation of the language composition.

Specifically, the BlackBoxCombiner uses OO composition and the meta-object protocol to compose the languages. Instead of defining keyword methods itself, the BlackBoxCombiner façade references the language façades of the constituent languages to be composed. Being a meta-object allows the BlackBoxCombiner to intercept method calls to keyword methods and delegate these calls to other façades—the façades of the constituent language. The combiner defines a composition protocol that derives a combined syntax-to-semantic binding for the language composition. This combined syntax-to-semantic binding is then used to bind expressions in the composite language to a well-defined evaluation semantics.

Note that when composing several languages, we always must ensure that the constituent languages are syntactically and semantically independent as explained below.

When composing several languages, the BlackBoxCombiner validates that the constituent languages are syntactically independent[9]. By validating, the BlackBoxCombiner can assume that it can evaluate each expression type in isolation Knowing that expressions of the composed languages are both syntactically and syntactically independent, for each expression type in a black-box composition, there is always one particular well-defined method that binds the expression's syntax to its semantic.

Before using the BlackBoxCombiner to compose two syntactically independent languages, the language developer is required to validate that the constituent languages are also semantically independent[10]; hence, the BlackBoxCombiner can assume that it can evaluate each expression type in isolation.Knowing that expressions of the composed languages are both syntactically and semantically independent, for each expression type in a black-box composition, there is always one particular well-defined evaluation semantics.

In detail, to determine a composed semantics for an expression type, the BlackBoxCombiner uses the introspective capabilities of the meta level to analyze the language façades of the languages to be composed in order to determine what expression types they define. For each expression type found in one of the façades, the combiner reuses its syntax-to-semantic binding from the façade of the corresponding constituent language for the composite language. Because the combiner enforces that expression types of constituent languages are disjoint, the combiner can derive a well-defined evaluation semantics for each expression type. Otherwise, if more than one façade define the same expression type, this is a syntax conflict and the combiner raises an exception. The fact that each expression type belongs only to exactly one language allows the combiner to always find the right semantics for an expression type. This is always defined by the corresponding façade that defines the method with a matching signature for that expression type.

[9]Conceptually, the constituent languages are said to be syntactically independent if the set of their grammars' productions are disjoint (cf. Section 3.2, p. 37). In the implementation, the constituent languages' implementations are said to be independent if their language interfaces and façade classes do not have methods with the same signature.

[10]Conceptually, the constituent languages are said to be semantically independent if the language composition specification does not specify a semantic dependency (cf. Section 3.2, p. 37). In the implementation, the constituent languages' implementations are said to be independent if their classes do not have dependencies in their interfaces (and semantic contracts).

Program Evaluation with Composed Syntax and Semantics: The steps to evaluate a program that mixes keywords of several languages are slightly different than the steps for a program with only one language.

In the compilation step, it is worth mentioning that the host compiler allows *separate compilation* of the program, every constituent language, and the combiner.

In the setup step, to compose the languages, a developer creates a new instance of the BlackBoxCombiner passing to it instances of the all language façades. Figure 4.12 indicates how the BlackBoxCombiner instance is connected to the façades of the constituent languages for the example composition of FUNCTIONAL and COMPLETELOGO. Then, the BlackBoxCombiner instance is set up as the façade for the program code block.

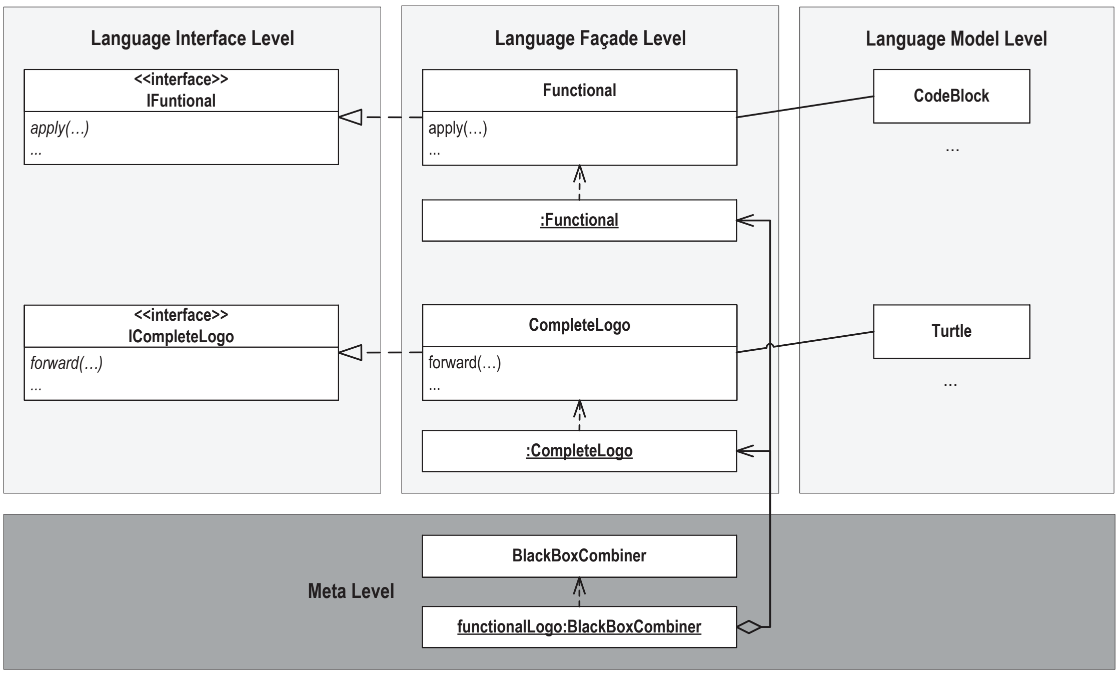


Figure 4.12.: Composing two languages with the BlackBoxCombiner

In the evaluation step, when encountering expressions in a program that originate from different languages, the BlackBoxCombiner determines the right evaluation semantics for each expression type. Figure 4.13 illustrates keyword dispatch in the composition protocol of the BlackBoxCombiner. Depending on the type of an expression, the combiner selects the keyword method from one of the language façades with the corresponding signature for that expression type, and uses this binding to evaluate the expression.

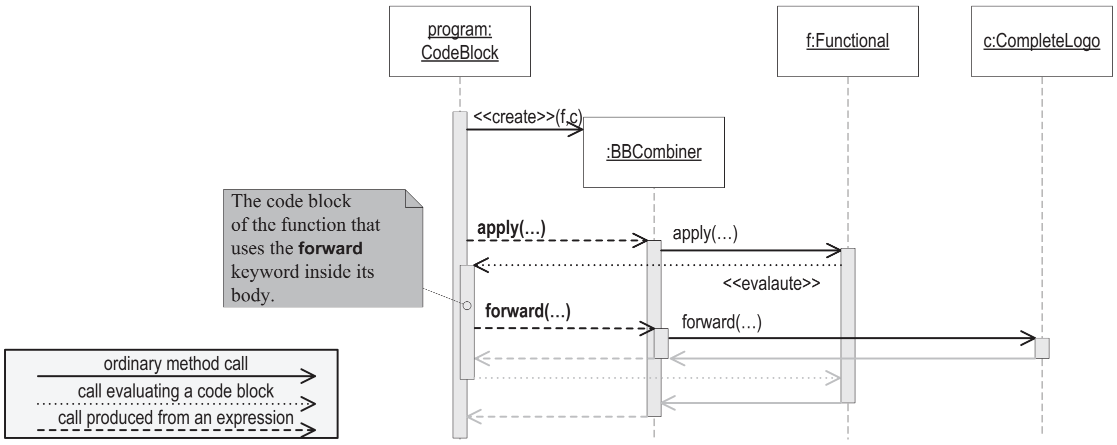


Figure 4.13.: Keyword dispatch with the BlackBoxCombiner

4.2.3.2. Composing Dependent Languages

The architecture comes with pre-defined combiners for composing dependent languages. In the following, we discuss compositions of dependent languages and how combiners can handle syntactic and semantics interactions between them.

To achieve compositions of the semantics of dependent languages, dependent combiners adapt language components using their meta-level in order to adapt their syntax in language façades and evaluation protocols in their models. It is crucial that dependent combiners have the possibility to adapt every method call inside the evaluation protocol of each language, instead of evaluating a computation directly. On the one hand, the dependent combiner can intercept the evaluation protocol of the influencing language, e.g. to extract information from it. On the other hand, it can adapt the evaluation protocol of the influenced language to provide alternative semantics, e.g. to inject information into it. This way the dependent combiner can integrate the first-class objects of the dependent languages by manipulating their structure and behavior.

4.2.3.2.1. Syntactic Interactions There is a LinearizingCombiner that automatically resolves syntactic interactions between languages. It assumes that there is a partial order defined on all languages to be composed. The composition protocol of the LinearizingCombiner works similarly to the BlackBoxCombiner, but does not enforce disjoint sets of expression types between the languages.

For example, reconsider Section 3.3.1.3 (p. 43) that motivated composing COMPLETELOGO from the independent extensions UCBLOGO, CONCISELOGO, and EXTENDEDLOGO. Recall that there are syntactic conflicts. When mixin composing the language's keywords, the combiner uses façades instances of the extensions. Because all extensions inherit the base language keywords, there is a *common ancestor problem* (also known as the diamond problem). Because of this problem, the BlackBoxCombiner cannot decide on which of the extensions to execute the inherited keywords of the base language. To compose such independent extensions, the developer needs to use the LinearizingCombiner instead.

When an expression is encountered with a certain expression type, and when this expression type is declared by several languages (i.e. they declare the same expression signature for the corresponding keyword and sub-expressions), the LinearizingCombiner can resolve the syntactic interaction between the languages in case of such independent extensions. The reason is that this is a special case: all languages extend the same base class, thus there is no syntactic difference between the inherited methods that conflict each other. To resolve the syntactic ambiguity for such an expression, the LinearizingCombiner always uses the syntax-to-semantic binding that is defined by the language façade with the highest order. Consequently, the combiner can disambiguate the inherited expressions, by defining a linearized *mixin*-like composition for the languages.

In general, when programs use expression types from different dependent languages, a dependent combiner needs to compose the languages' syntax such that interactions are possible, whereby preventing, detecting, and resolving conflicts. To detect and resolve syntactic interactions, language combiners can intercept and adapt keyword method calls from the program using the meta level. Instead of dispatching the keyword-method call directly, the language combiner can forward a keyword-method call to an alternative method implementation.

Note that the concrete dispatch logic for preventing, detecting and resolving syntactic interactions of expressions depends on the concrete syntactic composition scheme. Several schemes are conceivable and can be implemented on top of the architecture.

4.2.3.2.2. Semantic Interactions The reflective embedding architecture supports composition of dependent languages with semantics interactions, called *dependent compositions*. When composing dependent languages, their evaluation protocols have dependencies. That means, there are expressions for which the evaluation process cannot be computed in isolation. That means, for such an expression belonging to one of the constituent languages, the evaluation can depend on the evaluation of an expression in another language.

For example, Section 3.3.2.1 (p. 44) motivates composing COMPLETELOGO with a DSL to modularly implement tracing in the AO *Logo* dialect, called AO4LOGO. Recall that there is a semantic interaction, because the *pointcut-and-advice* in AO4LOGO must intercept join points in COMPLETELOGO, evaluate pointcuts, and may need to execute tracing logic at those points. In REA, the CrossCuttingCombiner enables such compositions of dependent languages.

In a nutshell, the CrossCuttingCombiner implements an invasive composition protocol for dependent languages with crosscutting semantic interactions. To compose several dependent language, a language developer instantiates the combiner passing to it instances of the dependent language façades. Further, the developer needs to declare the points where the dependent languages have interactions in their evaluation protocols, which exposes a first-class representation of the evaluation context of the language to other participants. For the composition, internally, the combiner exploits the flexibility of the meta level for analyzing, intercepting, and interacting with the evaluation protocol of each language that participate in a crosscutting composition.

While CrossCuttingCombiner can compose the evaluation protocols at any point in the execution of a participating language, each language embedding remains well encapsulated. Adaptation are only effective for one particular composition scenario, the language implementations remain intact. Because the crosscutting composition protocol only temporarily adapts the evaluation protocols at those points that explicitly have been declared for interactions.

In [DMB09, DEM10], the author of this thesis has published the implementation details of the CCCombiner. The CCCombiner is an integral part of the *Pluggable and OPen Aspect Run-Time* (POPART for short). Since the CrossCuttingCombiner is a central contribution of this thesis, Section 7 (p. 157) will summarize how to use POPART for crosscutting composition and its relevant implementation details. More interested readers are referred to [DMB09, DEM10].

4.2.3.3. Composing Languages with Custom Combiners

Since the pre-defined combiners only support concrete language composition scenarios, alternative combiners can be implemented as custom combiners. In the following, we give an overview of possible schemes to implement in a custom combiner.

When none of the pre-defined language combiners provides an appropriate composition protocol, a language developer can provide a new language combiner. Even when existing combiners are inadequate, a language developer may refine an existing combiner to enable new schemes for language compositions in the architecture. Alternatively, combiners can be implemented from scratch.

4.2.3.3.1. Extending the Open Crosscutting Combiner In [DMB09], the author of this thesis proposes a concept for an aspect-oriented (AO) language with an integrated meta-level, which is called a *meta-aspect protocol* (MAP).

The definition of the MAP is leant against the definition of the meta-object protocols [KRB91], a meta-aspect protocol is *an interface to the aspect language that gives users the ability to incrementally modify behavior and implementation of aspect-oriented abstractions*.

To allow adaptations, the MAP makes use of the open implementation principle for AO languages. At the implementation level, an open implementation is a special module that developers design with a *primary interface* and a *meta-interface*. While end users access the primary interface in order to use the module as an ordinary module, the meta-interface provides special primitives to adapt the modules internal implementation strategies.

A MAP opens up the aspect language semantics for dynamic user adaptations. When using the meta-aspect protocol, the same set of aspects can be interpreted under different semantics. Analogous to MOPs, this flexibility is enabled by reifying important parts of the AO language semantics as first-class entities available.

Figure 4.14 shows the overall architecture of an aspect runtime with a meta-aspect protocol. The MAP is built on top of the MOP. Such meta-protocols provide two kinds of interfaces to programs, a *primary interface* and a *meta-interface*. In a MOP, OO programs use the primary interface to define objects, and OO programs the meta-interface to adapt the semantics of objects, e.g. the method dispatch. Similarly, the MAP provides such a primary interface and a meta-interface for the aspect language.

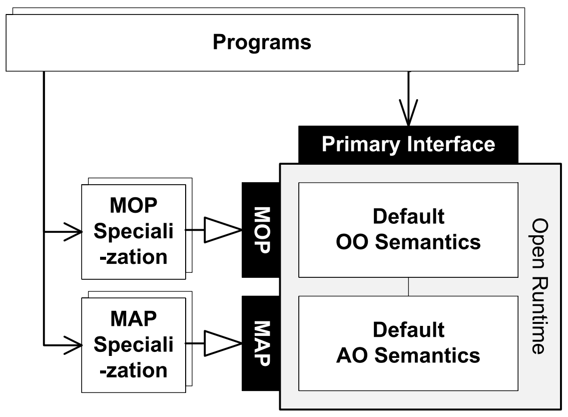


Figure 4.14.: Architecture of the meta-aspect protocol

The CCCombiner defines the primary interface. It embeds the default semantics of AO concepts, by reflective embedding, which will be elaborated in Section 7.3 (p. 160). AO programs can use the default semantics, nonetheless programs can extend the provided AO concepts.

For enabling semantic adaptations, there is a second interface to the CCCombiner and its language model. This meta-interface allows access to the embedded AO concepts of the CCCombiner. Language developers can use the meta-interface for semantic adaptations to customize the embedding of AO concepts for specific domains. Because AO concepts are embedded in REA, their first-class nature allows extension of the embedded types and their operations that reflect the semantics of aspects.

To enable well-defined adaptations of AO semantics, the CCCombiner defines variation points in the evaluation protocol for aspects. There are three main *variation points* that have been selected to allow semantic variations found in the literature. To enable variation of AO semantics, language developers can extend the AO concepts by refining the AO embedding. First, the developers can extend language façade of the CCCombiner e.g. to define new pointcut designators, as it will be elaborated in Section 7.4. Second, they can specialize the AO evaluation protocol by extending the types and overriding their methods in AO language model, which allows e.g. to enable debugging and optimizations for AO concepts, as it was elaborated in [DMB09]. Third, they can refine the management concern in the CCCombiner e.g. to adapt the composition semantics for aspect to resolve semantic composition conflicts between aspects, as it will be elaborated in Section 10.3.2.3 (p. 235).

By extending these variation points, language developer can implement *domain-specific aspect languages* (DSALs). A domain-specific aspect language is a special DSL that makes use of AO concepts for separation of crosscutting concerns in (DSL) programs. For implementing DSALs, a developer can reuse and specialize the AO concepts for particular a domain, whereby the developer can reuse the common AO semantics. As it will be elaborated in Section 10.3.2 (p. 227) and evaluated in Section 11.1.2 (p. 271), this significantly reduces the implementation effort.

4.2.3.3.2. Implementing a new Custom Combiner

For example, recall from Section 3.3.2.1 (p. 44) that when composing FUNCTIONAL and STRUCTURAL, there can be syntactic and semantic conflicts that need to be detected and resolved for a meaningful composition.

There is a syntactic conflict, because both languages define keyword define. Still a composition of the two syntactically conflicting languages is possible by defining a custom combiner that renames keywords extending the syntax-to-semantic binding in the composition. The combiner rebinds the define keyword method implementation in FUNCTIONAL to bind to the new keyword defun, and it rebinds the define keyword method implementation in STRUCTURAL to bind to a new keyword make, such that programs that use the renamed keywords can be executed without conflicts.

There is also a semantic interaction because both languages define a direct application for functions or respectively for data types. For functions, when using a function name, such as funName(...), this will call the function funName. For abstract data types, when using a data type name, such as typeName(...), this will instantiate a new value of typeName. But, when there is a function and a data type with the same name, it is not clear what semantics for the direct application should be used. For example, when there is a function myName as well as a type myName, it is not clear what myName(...) means: a function application or a type instantiation? Still, a meaningful composition of the two semantically conflicting languages is possible by defining a custom combiner that enforces that function names and names of abstract data types must be unique.

To compose several languages with semantic interactions or conflicts, a language developer only has to implement a new combiner class that implements a custom *composition protocol* that composes the syntax and semantics of all constituent languages.

This custom composition protocol needs to integrate the evaluation of expressions of the composed languages to work in concert, by following the corresponding defined composition semantics. To combine the languages' syntax-to-semantics bindings, its composition protocol composes the languages' keyword methods defined in the language façades. To combine the languages' computations, its composition protocol composes the languages' evaluation protocols defined in the language-model classes. The detailed logic that need to be provided for the composition protocol of a custom combiner can depend on a concrete composition scenario, its assumptions, its domain semantics and on possible generic composition schemes of which several schemes are conceivable and can be implemented on top of the architecture.

4.3. Summary

In this chapter, we have discussed a new architecture for embedding languages. The architecture enables open implementations of languages for which a new syntax and semantics can be provided. The architecture organizes language implementations into four levels: the language interface level, the language façade level, the language model level, and the language meta level. These levels enable decoupling parts of the language implementation and allow those parts to be adapted.

To recapitulate, the key mechanisms that facilitate language evolution are:

Language Polymorphism: Language polymorphism enables hierarchical extensions. Abstracting over several possible evaluations is possible since the reflective embedding architecture strictly separates a language specification from possible implementations.

While language polymorphism enables safe extensions to existing language implementation and programs, polymorphism on languages provides subtyping guarantees for the languages, as long as implementations respect inherited contracts.

Semantic Adaptation Scopes: The reflective embedding architecture uses meta-objects to partly adapt the semantics of a language implementation along different dimensions. Semantic adaptation provides more flexibility for reusing existing language implementations for end users with special requirements, but the downside of the increased flexibility is the absence of special guarantees.

So far, the principle support has been presented, but particularly semantic adaptation of languages deserves a more detailed discussion. Therefore, this thesis dedicates Chapter 6 elaborating on how the reflective embedding architecture supports non-trivial semantic adaptations of languages.

Language Combiner: Language combiners support composing languages in different concrete scenarios. While the combiners build on the flexibility of the architecture to compose languages, the combiners are abstractions for language composition. Combiners provide special guarantees for the composition scenarios in which they are employed. For the combiners, the presence of the meta level is fundamental to compose constituent languages independent from their concrete implementation. Analyzing part of the syntax and semantics in language implementations is only possible because all languages are available as first-class objects. Composing languages by integrating their abstract syntax, their syntax-to-semantic bindings, and their evaluation protocols is only possible because the combiners are meta-objects that can adapt every method call between objects within a language implementation.

So far, the principle support for composition has been presented, but particularly the composition of dependent languages deserves a more detailed discussion. Therefore, this thesis dedicates

Chapter 7 for elaborating how the reflective embedding architecture supports non-trivial compositions of dependent languages.

Further, it remains to be shown how the key mechanisms can be implemented. Therefore, the next section presents an implementation of the reflective embedding architecture and the key mechanisms for a concrete host language.

However, it remains to be shown that the language embedding approach indeed allows implementing complex languages and that the approach enables language evolution. Particularly, the following issues need to be addressed:

- What language features in host languages help language developers to grow embeddings with new language constructs?
- Is it possible to embed more sophisticated kinds of language constructs?
- How good does the reflective embedding architecture support the desirable properties for language evolution that have been identified in Section 3? In particular, how can we extend and compose several languages?

The following chapters address these issues.

5. Implementation in Groovy

This chapter elaborates how to instantiate the reflective embedding architecture in Groovy. The author has selected the Groovy programming language as the host language because it provides a good realization of host language features that the reflective embedding architecture requires. The first section introduces the required features from Groovy. The second section elaborates on how one can instantiate the architecture's levels in Groovy. The third section elaborates on the techniques to enables Groovy for the execution of embedded programs. The last section describes the implementation details of the key mechanisms of the architecture.

5.1. Groovy in a Nutshell

Groovy [Gro, KG07] is a pure object-oriented *scripting language* that seamlessly integrates with Java [GJSB00] and that compiles to Java bytecode. The syntax is close to Java and one can call Groovy code from Java and vice versa without converting passed objects. Besides a *meta-object protocol*, Groovy provides attractive language features, such as *class reloading* and *closures* as first-class entities. Groovy is particularly well suited for language embedding due to its support for its *flexible syntax*, its support for *closures*, and its *meta-object protocol*. In the following, we present these features.

Flexible Syntax. Groovy syntax is flexible, because Groovy often allows encoding the same expression in various syntactic forms.

Groovy supports so-called *command expressions*[1] that enable omitting unnecessary delimiters in programs like brackets clear from the context. When calling a method, one can omit the brackets. For example, to call method "def println(String str)", one can write "println "test"" instead of "println("test")". Command expressions support named parameters that enable the definition of methods with optional parameters or that make it possible to call the same method with a different set of actual parameters. E.g. "turtle(name:"John", color:blue)" is a call to a method named turtle with the two parameters name and color and the values "John" and the number that represent the color blue. When such method calls are evaluated, Groovy stores the parameters in a Map and passes the map to the method. The corresponding method definition needs to be "def turtle(Map params){...}".

Groovy supports overriding infix operators for arbitrary classes. E.g., to override the logical operator "&", a class must define the method and, or respectively for operator "|" the method or.

[1]Codehaus.org Homepage: Command Expression based DSL: `http://docs.codehaus.org/display/GroovyJSR/GEP+3+-+Command+Expression+based+DSL`

In Groovy, there is *syntactic sugar* for accessing a *property* of a class via its getter and setter. E.g., when a class C defines the accessors getX and setX but does not declare a *public* field, one can directly access x on an instance c. Using the expression "c.x" will call the getter returning its value. Using the expression "c.x = a" will call the setter changing its value.

The syntactic flexibility in Groovy, often facilitates defining a language embedding with an abstract syntax that is close to corresponding established domain syntax, because the expressions can omit unnecessary delimiters and other parts in their encoding.

Closures. To realize first-class code blocks, the instantiation of the reflective embedding architecture uses Groovy closures.

In Groovy, a closure is an object that represents a piece of code. Closures can be used as methods that take parameters and return values. E.g. "{x -> x*x}" defines a closure that takes a parameter x and returns its square value. Since closures are objects, one can assign them to a corresponding variable—e.g. "Closure sq = {x -> x*x}"—and pass around the reference to the object. To execute a closure, one calls it as if it were a method, e.g., sq(2).

If the last parameter of a method is a closure—e.g., "def turtle(Map map,Closure c){...}"—then one can define the closure inline and do not need to pass it as a parameter inside the round bracket of the method call; e.g. to call turtle one can write "turtle(name:"John",color:blue){ fd 50; ... }" (which is equivalent to "turtle([name:"John",color:blue],{ fd 50; ... })".

In Groovy, a closure does not completely close over its lexical context, but its binding can be changed. Within a closure's body the variable this refers to the enclosing object—the so-called owner (e.g., to access fields of the enclosing object). Additionally, each closure has a *delegate* object to which the closure delegates all calls that it cannot answer itself. By default the delegate object is the owner. In the previous example the call to println would be delegated to the closure's enclosing object; i.e. its owner.

The possibility to define and call closures in multiple ways and the support for named parameters facilitates a syntax for abstraction operators that is close to corresponding established domain syntax.

Meta-Object Protocol. To enable more flexibility for language embeddings, the instantiation of the architecture uses Groovy's meta-object protocol.

As shown in the lower half of Figure 5.1, every value is an instance of the class Object which implements the interface GroovyObject. The meta-level is shown in the upper half of Figure 5.1. Method calls and field accesses on an object are handled by a corresponding meta-object, which is an instance of MetaClassImpl that implements the interface MetaObjectProtocol. The latter declares *meta-methods* used for interpreting an object's behavior, e.g., invokeMethod, getProperty, setProperty, etc.

When a base-level method is called on an object, the *meta-method* invokeMethod is called on the object's meta-object. The default implementation of invokeMethod in MetaClassImpl dispatches the base-level method call to the most specific implementation defined in the object's

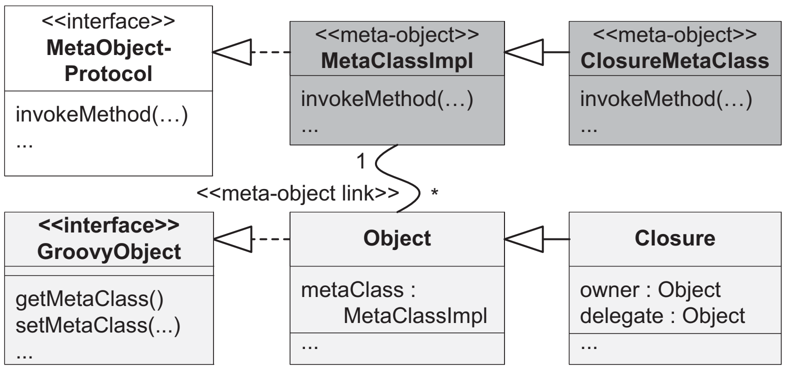


Figure 5.1.: The meta-object protocol of groovy

class or in one of its super classes. In a similar way, the meta-methods getProperty and setProperty dispatch field accesses to the most specific instance field[2].

The user can specialize MetaClassImpl and override specific meta-methods. Every class has a link to its default meta-object. When a class is instantiated, the default meta-object is used for the created instance. One can change the default meta-object defined for a class in the registry, thus changing the semantics of all objects of that class. Alternatively, one can change the meta-object of a single object by calling the method setMetaClass on it.

For convenience, the Groovy MOP allows defining application-level meta-objects. In Groovy, every object can be a meta-object—the meta-object of itself. To allow an object acting as the meta-object of itself, its class can implement the special methods setProperty, getProperty, invokeMethod, missingMethod and missingProperty. The first three have the same semantics as the methods with the same signature in the default meta-object. In case the class implements those methods, the default meta-object will use these more specific implementations instead of its default implementation, but only for instances of this corresponding class. The last two methods missingMethod and missingProperty have special semantics. The default meta-object calls these methods, whenever an undefined method is called, or respective an undefined field is accessed.

In Groovy, closures require specific meta-objects. For instance, Groovy provides a special meta-class ClosureMetaClass for closures, which are instances of the class Closure. Closures encapsulate their own evaluation context, which consists of bindings from the lexical context, i.e. instance variables of their creating object (owner) and local variables. Hence, closures cannot use the default MetaClassImpl, which would dispatch method calls and field/variable accesses to this—the closure itself that does not define them. The special ClosureMetaClass dispatches method calls as well as field/variable accesses inside the closure to its context. Moreover, every closure can have a delegate object. If the latter is not null, method calls and field/variable accesses are dispatched to it. This allows manipulating the evaluation context of a closure after the closure is created. In Groovy, every closure has a field resolveStrategy that allows adapting

[2]If an accessed field is not present, by convention, the groovy MOP tries to find a corresponding getter or setter method.

how the MOP resolves variable inside the closure. There are four relevant strategies: (1) the default OWNER_FIRST: prefers binding variables from the closure's owner context and only looks up unbound variables in the delegate context, (2) DELEGATE_FIRST: prefers the delegate context over the owner, (3) OWNER_ONLY: only looks up variable bindings in the owner context, and (4) DELEGATE_ONLY: looks up only in delegate context. The delegates and those strategies are crucial for enabling closures that do not completely close over the lexical context. The instantiation of the architecture uses in particular delegates as well as strategies to realize the dynamic syntax-to-semantics binding mechanism.

The ClosureMetaClass implements the closure-delegate mechanism and will be used for realizing the keyword method dispatch.

5.2. Reflective Embedding in Groovy

This section describes the instantiation of the levels of the reflective embedding architecture in Groovy. When instantiating the reflective embedding architecture in a concrete host language, one must select available host language constructs to implement the conceptual elements of the abstract architecture. Moreover, it is important to fine-tune the abstract development process for the host language. To exemplify the concrete process, we discuss embedding *Logo* in Groovy. Along the discussion, we highlight what parts of the process have been fine-tuned for Groovy.

5.2.1. Language Interface Level

In Groovy, the language interface can be defined as an interface class with one keyword method for expression type, like described in the abstract process from Section 4.1.1.1 (p. 72), but there are minor deviations compared to the abstract process. Those derivations help to improve the syntax of the resulting embedding in Groovy, as elaborated below:

(I) a getter method for each literal: In Groovy, literals are defined slightly different to the abstract process. Literal keyword methods are defined using a getter method that uses the literal's keyword name (in camel case) prefixed by "get".

For example, Listing 5.1 shows the Groovy code that defines the language interface ISimpleLogo of SIMPLELOGO. In lines 3–8, ISimpleLogo defines a getter method for each color literal with the corresponding color name. All getters return the integer representation of their color.

With the help of Groovy's syntactic flexibility, a program can directly use a literal keyword to read/write from the property (e.g., red) and the MOP will call the corresponding accessor.

(II) a method for each operation: In Groovy, operations are defined exactly the same way as in the abstract process. Recall that these methods use the operations' keywords and take as many parameters as their operation's arity.

For example, in lines 11–14, ISimpleLogo declares the operations, e.g. forward.

Note that, in Groovy, we also can fine tune for defining operations with two options. There is the possibility to define operation methods with a variable length of parameters and methods with optional parameters. This can help to reduce the number of methods that developers need to declare.

(III) a special method for each abstraction operator: In Groovy, abstraction operators define their parameters slightly different. For a better syntax, it is recommended that an abstraction operator takes two parameters, a HashMap as its first parameter, and a Closure as the last parameter. The map stores the abstraction operators' named parameters, and the Closure represents the abstraction operator's body.

For example, in line 17, ISimpleLogo declares the abstraction operator turtle that takes the two HashMap and Closure parameters and in this case returns nothing.

```
 1 interface ISimpleLogo {
 2   /* Literals */
 3   int getBlack();
 4   int getBlue();
 5   int getRed();
 6   int getGreen();
 7   int getYellow();
 8   int getWhite();
 9
10   /* Operation */
11   void forward(int n);
12   void backward(int n);
13   void right(int n);
14   void left(int n);
15
16   /* Abstraction Operators */
17   void turtle(HashMap params, Closure turtleBody);
18 }
```

Listing 5.1: Language interface of SIMPLELOGO

It is a strategic decision how to encode a concrete method signature for an abstraction operator. In the most cases, one can obtain a good abstract syntax with the HashMap/Closure combination. This is because Groovy's command expressions allow us to fine-tune the resulting abstract syntax to be close to concrete syntax. Thanks to named parameters, developers can encode mixfix operators (cf. Section 3.4.2, p. 50), in a more or less acceptable notation. Thanks to passing closures outside bracket, developers can nicely encode abstraction operators.

Note that, the presence of the language interface level in the architecture is desirable from a design perspective, but there is no obligation for such an interface to be present in Groovy. In case the language interface is present, the Groovy compiler enforces that the language implementation conforms to this syntax definition. Still, the language developer can decide to completely omit the interface, thanks to Groovy's weak typing facilities.

5.2.2. Language Façade Level

At the language façade level, the language developer[3] implements the language interface as a façade class[4] that also extends the Interpreter class. An excerpt of the implementation of the Interpreter class is shown in Listing 5.2. By default, the body delegate is the façade itself (line 2). This ensures that the closure-delegate mechanism uniformly dispatches occurrences of domain-specific keywords in the bodies of abstraction operators to the same façade instance.

```
1 public abstract class Interpreter {
2   protected Object bodyDelegate = this;
3     ...
4 }
```

Listing 5.2: Excerpt of the Interpreter class

For example, to provide a language façade for SIMPLELOGO, the class SimpleLogoInterpreter extends Interpreter and implements the ISimpleLogo interface. The class is responsible for storing an evaluation context and implementing all keyword methods, as elaborated in the next two paragraphs.

For the evaluation context, the class holds references to a Canvas object (line 6) and a Turtle object (line 3), which are instances of the language model classes. To provide execution semantics, the SimpleLogoInterpreter follows an immediate evaluation mode, which immediately computes expressions. In lines 5–9, its constructor creates the canvas as an instance of class TurtleGraphicsWindow from the language model and stores this instance as one of the attributes of the interpreter, which constitutes the global context.

For keywords, the class provides method implementations, as exemplified below for the three types. For each literal method that retrieves a color shown in lines 12–17, the method returns the integer value of the corresponding RBG-representation given by its Color class from the language model. In case of the operation methods shown in lines 20–23, they simply call the corresponding method for that command on the turtle instance passing to it the parameters[5]. In case of the abstraction operator method shown in lines 26–31, the turtle abstraction operator instantiates a new instance of the class Turtle from the language model, sets the bodyDelegate as the delegate of the closure parameter that contains a sequence of commands, and calls the closure to evaluate the command sequence. Setting the bodyDelegate is important so that keywords used inside the closure are uniformly handled by the same façade instance.

[3]Note that, the language designer who designs the language interface, and the language developer who implements it, they must not be the same person.

[4]Normally, the facade class is a Groovy class, but it also can be a Java class. When using a Java class, there are some restrictions w.r.t. Groovy MOP. To leverage the maximum flexibility of the reflective embedding architecture, it is recommended to implement every class in Groovy.

[5]Since the language model classes expect Double parameters to their domain operations, Groovy auto-boxes and converts the values from int to Double

```
public class SimpleLogoInterpreter extends Interpreter implements ISimpleLogo {
  protected TurtleGraphicsWindow canvas;
  protected Turtle turtle ;

  public SimpleLogoInterpreter() {
    canvas = new TurtleGraphicsWindow();
    canvas.setTitle ("Logo EDSL (based on JavaLogo)");
    canvas.show(); //Display the window
  }

  /* Literals */
  public int getBlack() { return Color.BLACK.value; }
  public int getBlue() { return Color.BLUE.value; }
  public int getRed() { return Color.RED.value; }
  public int getGreen() { return Color.GREEN.value; }
  public int getYellow() { return Color.YELLOW.value; }
  public int getWhite() { return Color.WHITE.value; }

  /* Operations */
  public void forward(int n) { turtle .moveForward(n); }
  public void backward(int n) { turtle .moveBackward(n); }
  public void right (int a) { turtle .turnRight(a); }
  public void left (int a) { turtle . turnLeft (a); }

  /* Abstraction Operators */
  void turtle (HashMap params, Closure turtleBody) {
     turtle = new Turtle(params.name,java.awt.Color.BLACK);
    canvas.add(turtle );
    turtleBody.delegate = super.bodyDelegate;
    turtleBody. call ();
  }
}
```

Listing 5.3: Language façade of SIMPLELOGO

5.2.3. Language Model Level

At the language model level, in Groovy, the language designer implements the internals of the language semantics using ordinary Groovy classes[6]. The classes define fields and methods, and their implementations encode the details of the language semantics, i.e. the language's theorems encoded in Groovy code. Note that often, language developers do not have to implement a complete model from scratch, since the model can use existing code from libraries that are in Groovy or Java bytecode.

For SIMPLELOGO, Figure 5.2 shows an excerpt of the language model of SIMPLELOGO The classes in the language model implement the semantics of the *Logo* domain. We have already discussed the general semantics of the model in Section 4.1.1.3 (p. 76). What is special in Groovy is that the implementation of the language model uses more specific host language types and that it can reuse existing code from Groovy or Java libraries. On the one hand, to encode

[6]Alternatively, Java classes can be used, but when using Java classes these cannot be adapted by the Groovy MOP without restrictions. Therefore, for language implementation in Java, REA's meta-level is not available.

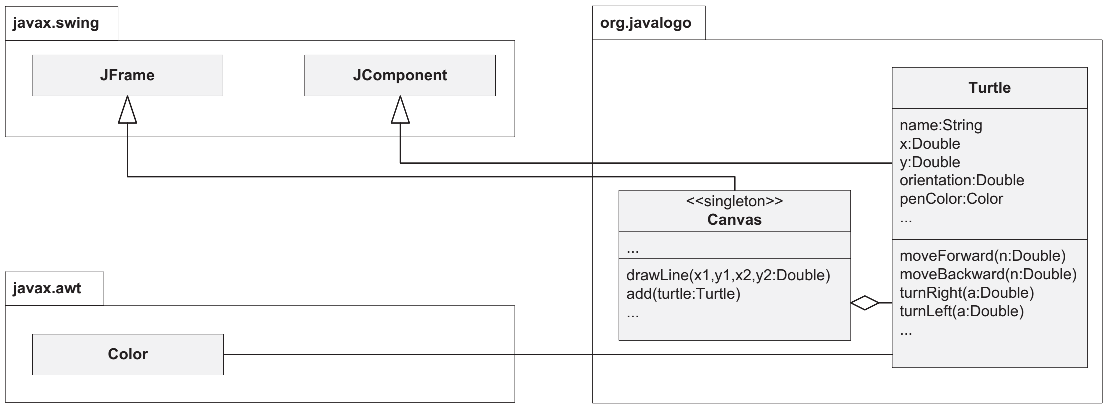


Figure 5.2.: The language model of SIMPLELOGO

parts of the internals of the language, the model implementation uses the available libraries of the host language, such as the primitive Java type int, the Groovy Closure class, and the Java AWT class Color. On the other hand, for drawing turtles, the model implementation reuses the classes from an existing Java-based implementation of *Logo*, e.g. the *JavaLogo* project [Jav02] that provides a *Swing*-based [ELW98] graphical user interface (GUI).

5.2.4. Language Meta Level

At the language meta-level, Groovy allows using default or adapted semantics for a language, as elaborated below.

For default semantics, the default Groovy meta-object MetaClassImpl is used to execute the language implementation. In this case, the meta-level does not change the language implementation, hence technically this makes the presence of a special meta-level optional. Since there are no special meta-objects, instances of the MetaClassImpl meta-object executes the language implementation with the standard OOP semantics. Consequently, neither the abstract syntax in language interface, nor the syntax-to-semantic binding in the façade, nor the evaluation protocol in the model are executing under changed semantics.

For adapted semantics, a language developer can extend the Groovy MOP to open up a language implementation at the language meta-level. For every class in the language implementation, the Groovy MOP can add new keyword methods or override existing ones by adapting the method dispatch for these classes, or one can introduce new fields for literals. Adapting the fields and methods of the language implementation classes is possible, because the Groovy MOP intercepts every method call, or respectively field access, as mentioned in Section 5.1.

To adapt a given language, a language developer needs to implement a special meta-object by subclassing the MetaClassImpl class[7]. A subclass overrides the invokeMethod of the super class to adapt the method dispatch, or respectively the getProperty and setProperty method to adapt field

[7]Since the MetaClassImpl class in Groovy is actually a Java class, the developer has the choice to implement the meta-object in Java or Groovy.

accesses. To apply an adaptation, the developer select the meta-object to execute the classes of the language implementation to adapt.

There are several strategic decisions to make for the developer what to adapt with a meta-object. The meta-objects can be rather generic for several classes in the language implementation, or they can be specialized for single classes or objects, such as only the façade or a single language model class. Discussing the means that help developers make their decisions deserves a more detailed discussion, therefore, Section 5.4.2 addresses this issue.

5.3. Program Evaluation in Groovy

This section discusses how programs are represented and evaluated in the reflective embedding architecture.

To encode program expressions in abstract syntax, each kind of expression type follows a specific Groovy encoding, which will call methods of the façade, such as shown for the program in Listing 5.4. The encoding represents a literal expression as a field access, e.g. for the literal red, it reads the corresponding field in line 2, which will call the getter getRed(). Further, it represents an operation expression as a method call in prefix notation with its sub-expressions as the parameters, e.g. for forward, in line 3, its call the method forward(). Finally, it represents abstraction operator as a method call also in prefix notation, where sub-expressions are named parameters in a map and where nested sub-expressions are wrapped in a closure (e.g. cf. turtle in lines 2–6). Note that the Groovy encoding is close to the domain syntax, since the encoding allows omitting brackets and nicely passing closure parameters.

To associate a program code block with a concrete façade, the encoding wraps the whole program in a closure (logoProgram in Listing 5.4 line 1), and it sets the closure's delegate to an instance of façade class SimpleLogoInterpreter of the language (line 8). Recall that setting the delegate field of a closure dynamically establishes a syntax-to-semantic binding in Groovy. Next, we call the closure, i.e., we execute the SIMPLELOGO program (line 9). This execution runs in two steps: (a) create and initialize a turtle object, which is an instance of the class Turtle, and (b) execute the commands in the turtle body.

```
1 Closure logoProgram = {
2   turtle (name:"Square", color:red) {
3     forward 50
4     right  90
5     ... /* Implementation is continued as in Figure 3.1 (a) lines 4–10 */
6   }
7 }
8 logoProgram.delegate = new SimpleLogoInterpreter();
9 logoProgram();
```

Listing 5.4: Associating a DSL program to its interpreter

Now, if during the execution of the logoProgram closure a call to a DSL keyword is encountered (here turtle) the call is dispatched to the DSL program closure's delegate; i.e. the corresponding façade instance of SimpleLogoInterpreter. The declaration at line 2 in Listing 5.4 is actually a call to turtle(HashMap params, Closure turtleBody) on the logoProgram closure[8] and hence forwarded to its delegate—the façade instance. Thereby, paramHashMap contains the turtle's name, and turtleBody is a closure containing the body of the turtle declaration (lines 2–6, Listing 5.4).

The execution of that call creates a Turtle object (line 27, Listing 5.3) and calls the closure turtleBody (line 30, Listing 5.3) after setting its delegate to bodyDelegate (line 29 in Listing 5.3) of the super class (cf. Figure 5.2, line 2), which refers to the façade instance itself by default.

The execution of turtleBody immediately evaluates the sequence of commands in the body. Thereby, every DSL keyword within the body gets dispatched to bodyDelegate, i.e., to the SimpleLogoInterpreter instance. For instance, the forward declaration in line 3 in Listing 5.4 yields a call to method forward(...) on the façade instance. It is fundamental that in Groovy the façade sets the delegate of the closures of its abstraction operators (Listing 5.3, lines 29) to the designated bodyDelegate of its super class Interpreter. Recall that in the Interpreter class (cf. Listing 5.2) the body delegate refers to the façade itself (line 2). Because abstraction operators use the bodyDelegate for their enclosing closures as the closure delegate, this ensures that occurrences of domain-specific keywords are uniformly dispatched to the same façade instance.

5.4. Key Mechanisms in Groovy

For a particular host language, the architecture's key mechanisms can be distinguished whether they can be reused from existing mechanisms in host language or whether a mechanism must be implemented on top of existing host language mechanisms. Depending on the quality of the language constructs and internal mechanisms in a host language, only some of the language mechanisms from the host language can be used for realizing the key mechanism for language embeddings. When none of the existing mechanisms in the host language are adequate for a certain evolution, a language developer can eventually embed a new mechanism or extend an existing mechanism.

In Groovy, we could reuse language mechanisms for implementing language polymorphism and the meta-object protocol for evolutions of embeddings. This is because languages are embedded into the host language as a library, thus embeddings are homogeneous and do not break existing host-language constructs and tools. For enabling language polymorphism, class inheritance and OO method dispatch in the host language can be leveraged for the language embedding. Section 4.2.1 (p. 84) shows that inheritance can be used to enable extensibility for embeddings. For enabling the dynamic syntax-to-semantics binding, Section 5.4.1 shows that object composition allows customizing language semantics by replacing a façade with another one. Further, reflection provides a systematic support for non-invasive adaptations of language implementations, discussed in Section 4.2.2 (p. 89).

In Groovy, existing language mechanisms, such as *runtime mixins* and *categories* [KG07], do not provide adequate support for language composition, since these do not detect or prevent conflicts and do not provide any guarantees. Therefore, the reflective embedding architecture

[8]Due to Groovy's command expressions and closures.

allows embedding a new language-composition mechanism on top of Groovy's meta-object protocol. Fundamental for embedding new mechanisms is that the Groovy MOP enables analysis and adaptation to define a meta-level for language embeddings. Because in REA, languages are first-class, their syntax and semantics can be analyzed via introspection and adapted via intercession. Because program expressions are represented as first-class objects and because every step in the evaluation of an expression constitutes a method call, the structural representation and the evaluation protocol of expressions can be adapted. For composability, the meta-level enables defining generic logic for composing languages, discussed in Section 4.2.3 (p. 102). Because in REA experts can provide sophisticated mechanisms on top of the host, users can reuse and customize these mechanisms as first-class objects, which opens up language development to non-expert programmers.

5.4.1. Language Polymorphism

The language polymorphism mechanism leverages the Groovy inheritance for *black-box extensions* of languages.

To add new keywords to an existing language embedding, a language developer can grow a language by extending the base language's interface class, its façade class, and its model classes. By inheriting from these classes, existing functionality is reused. For an extension, the developer must only specify and implement the domain semantics of new keywords.

Consider implementing EXTENDEDLOGO that adds more drawing commands to SIMPLELOGO, such as, setpencolor and clearscreen. Listing 5.5 shows the language interface and implementation of EXTENDEDLOGO. In line 1, the language interface of the new dialect IExtendedLogo extends the existing base language's interface ISimpleLogo, hence it inherits all declared methods from its super interface (e.g., forward). In a similar way, the language façade ExtendedLogoInterpreter extends the base language's façade class SimpleLogoInterpreter. On the one hand, the language façade inherits all implemented methods from SimpleLogoInterpreter façade. On the other hand, the ExtendedLogoInterpreter add implementations for the methods declared in IExtendedLogo. At this point, we only show the implementation of two of these methods, namely clearscreen (line 12) and setPencolor (line 14). Note that in this scenario, we did not have to extend the language model classes, since the façade class can reuse the model classes as they are provided.

Figure 5.6 demonstrates how to use an instance of ExtendedLogoInterpreter (line 1) for evaluating a program that still assumes an old version of the language (lines 3–8) and another program in the extended language (lines 12–19).

5.4.2. Meta-Level Extensions

Semantic adaptations leverages the Groovy meta-objects for adapting a given language semantics. In contrast to language polymorphism, the meta-objects enable *white-box extensions* that are extensions that interfere with the internal implementation strategies of the language.

Consider implementing the support for collision detection for CONCURRENTLOGO, as discussed in Section 3.1.2.2 (p. 35). With white-box extensibility, developers can elegantly encode interpretation concerns those code crosscut throughout the embedding by using the Groovy

```
interface IExtendedLogo extends ISimpleLogo {
  void home();
  void clearscreen();
  void showturtle();
  void hideturtle ();
  void setpencolor(int c);
  void penup();
  void pendown();
}

class ExtendedLogoInterpreter extends SimpleLogo implements IExtendedLogo {
  void clearscreen() { canvas.clearScreen(); }
   ...
  void setpencolor(int c) { turtle .setPenColor(c); }
}
```

Listing 5.5: Implementation of the EXTENDEDLOGO dialect

```
ISimpleLogo evaluator = new ExtendedLogoInterpreter();

Closure programInOldLanguageVersions = {
  turtle{name:"Square",color:red) {
    forward 50; right 90;
     ...
  }
}
programInOldLanguageVersions.delegate = evaluator;
programInOldLanguageVersions();

Closure programWithShortcutNames = {
  turtle{name:"Square",color:red) {
    clearscreen();
    forward 50; right 90;
    setpencolor green;
     ...
  }
}
programWithShortcutNames.delegate = evaluator;
programWithShortcutNames();
```

Listing 5.6: Using the extended set of commands from EXTENDEDLOGO

MOP, such as collision detection. Specifically, for collusion detection, a developer needs to implement a special meta-object that prevents collisions by intercepting façade and model-level method calls when Turtle objects painting concurrently. In the remainder, this section sketches the collision detection implementation.

Figure 5.7 shows a special meta-object class for Turtle objects. This meta-object class adapts the Turtle class's move operations (e.g., forward) to detect and prevent collision between multiple turtles. To intercept calls to all move operations, the class implements the method invokeMethod that the Groovy MOP will execute for each call on a Turtle object. As indicated in line 7, the meta-object only adapts method calls to the move operations moveForward and moveBackward. Before executing these move operations, the invokeMethod detects and prevents the collision (lines 7–9). Finally, when there is no (more a) collision, the call is proceeded by the super class of the meta-object (line11).

```
 1 class CollusionPreventingTurtleMetaClass extends MetaClassImpl {
 2   CollusionPreventingTurtleMetaClass(Class theClass, ...) {  super(theClass);...  }
 3 
 4   Object invokeMethod(sender, object, methodName, arguments, ...) {
 5     assert (object  instanceof  Turtle );
 6 
 7     if  (methodName.equals("moveForward") || methodName.equals("moveBackward")) {
 8         // code that  detects  and prevents  collisions   ...
 9     }
10 
11     return super.invokeMethod(sender, object, methodName, arguments, ...);
12   }
13 }
```

Listing 5.7: Excerpt of a specialized meta-object class that prevents collisions

To use the semantic adaptation, a developer or user simply needs to sets up all specialized meta-objects as the meta-objects of relevant classes in the language model. As shown Listing 5.8 in line 3, by changing the metaClass field of the Turtle class, the program associates the new CollisionPreventingTurtleMetaClass meta-object with the Turtle class.

When the program closure is evaluated in Listing 5.8 line 12, the evaluation of the program closure will create new Turtle objects that are all associated with new instances of Collision-PreventingTurtleMetaClass. Therefore, all calls to this domain object will be intercepted by the meta-object and collisions can be prevented.

```
 1 ICompleteLogo evaluator = new ConcurrentLogo();
 2 
 3 Turtle .metaClass = new CollusionPreventingTurtleMetaClass(Turtle.class, ...);
 4 
 5 Closure multiTurtleProgram  = {
 6    turtle{name:"T1",color:red) {    ...  }
 7    turtle{name:"T2",color:green) {   ...  }
 8    turtle{name:"T3",color:blue) {    ...  }
 9     ...
10 }
11 multiTurtleProgram.delegate = evaluator;
12 multiTurtleProgram();
```

Listing 5.8: Semantic adaptation of SIMPLELOGO with the Groovy MOP

With respect to the adaptation scope, this example adaptation has a *global* locality (D1.a) since it changes the behaviors of all Turtle instances, its atomicity is at the *module-level* (D2.b) since it intercepts all methods that move a Turtle, and its time of adaptation is *pre-execution* (D3.a). Later on, the collision detection implementation will be further elaborated in Section 10.1.2.2 (p. 215).

5.4.3. Extensible Languages Combiners

To compose languages, language developers can use one of the language combiners from Section 4.2.3 (p. 102). The following paragraphs show how these combiners are implemented in Groovy.

5.4.3.1. Composing Independent Languages

The architecture supports composing independent language embeddings by composing their dynamic syntax-to-semantic bindings using the Groovy MOP.

In Groovy, the composition logic for independent languages with black-box composition is implemented in the meta-façade BlackBoxCombiner shown in Listing 5.9. The BlackBoxCombiner extends the LanguageCombiner class (cf. Figure 4.11, p. 103) and is a meta-object because it implements MetaClassImpl. It references independent façades to which it dispatches uses of the expressions (i.e., domain-specific keyword method calls). The constructor checks that there is no syntactic conflict between the composed façades, using *Java reflection* [AGH05, FF04]. Further, to make the combiner responsible for evaluating the expressions that are nested in abstraction operators, the constructor sets the combiner as the bodyDelegate of all façades. The keyword dispatch is implemented by overriding GroovyObject's invokeMethod (lines 10–17) that the Groovy MOP will invoke for every keyword method call. The invokeMethod searches for the corresponding façades that implements the keyword method by using reflection (lines 11–13), it enforces the call to be unambiguous[9] (line 15), and it delegates the method call to the façade that implements the keyword method (line 16).

For example, consider composing COMPLETELOGO and FUNCTIONAL into the composite language FUNCTIONALLOGO, as it has been proposed in Section 3.2.1 (p. 37). We have already discussed the conceptual aspects of how to compose the two independent language. Therefore, this section only presents the Groovy implementation of FUNCTIONAL. Then, this section demonstrates how the developer can use the BlackBoxCombiner to compose the latter with the embedding of COMPLETELOGO.

Listing 5.10 presents the embedding of FUNCTIONAL with the language interface IFunctional (lines 1–4) and its façade (lines 6–19). In line 7, there is the definedFunctions map from Strings to Closures. The define abstraction operator method (lines 9–12) creates a new function by binding the function name (from the named parameter params.name) to its body closure via the map. The apply operation method (lines 14–17) retrieves the closure from a defined function from the map. Note that this façade does not require a special language model, since functions can be

[9]Although the BlackBoxCombiner's constructor checks that signatures are unique over all façades, it must enforce the condition again, since new keyword methods can be added at runtime using the meta-level.

```
class BlackBoxCombiner extends LanguageCombiner
                        implement MetaClassImpl {
  Set<Interpreter> facades;
   ...
  BlackBoxCombiner(Interpreter... facades) {
    // 1) uses Java introspection to validate that each signature in the facades is unique, and
    // 2) sets itself as the bodyDelegate of all facades
  }
   ...
  Object invokeMethod(sender, receiver, mName, args, ...) {
    Set<Interpreter> respondingFacades = facades.findAll { interp ->
      interp.metaClass.respondsTo(interp,mName)
    };
    if (respondingFacades.empty()) throw new MissingMethodException("Keyword $mName undefined.");
    if (respondingFacades.size() > 1) throw new SyntaxConflictException("Keyword conflict $mName.");
    return respondingFacades.first().invokeMethod(sender, receiver, mName, args, ...);
  }
}
```

Listing 5.9: The BlackBoxCombiner class

readily mapped to Groovy closures. With an instance of this façade, developers can evaluate FUNCTIONAL programs.

```
interface implements IFunctional {
  void define(HashMap params, Closure body);
  Closure apply(String name);
}

class Functional extends Interpreter implements IFunctional {
  HashMap<String,Closure> definedFunctions = new HashMap<String,Closure>();

  void define(HashMap params, Closure body) {
    definedFunctions.put(params.name,body);
    body.delegate = bodyDelegate;
  }

  Closure apply(String name) {
    if (!definedFunctions.containsKey(name)) throw new UndefinedFunctionException("'$name' undefined");
    return definedFunctions.get(name)
  }
   ...
}
```

Listing 5.10: Excerpt of the implementation of FUNCTIONAL

Listing 5.11 demonstrates using the use of the BlackBoxCombiner. First, in line 17, the setup step instantiates the façades for all language embeddings used in the program. Then, in the same line, the setup step creates a BlackBoxCombiner and pass to it the façades. In the last step, we set the BlackBoxCombiner as the program closure's delegate (line 17) before executing the program (i.e., before calling the respective closure in line 18).

```
 1 Closure program = {
 2   define(name:"square") { length ->
 3     repeat (4) {
 4       forward length
 5       right 90
 6     }
 7   }
 8 
 9   turtle (name:"TwoSquares", color:red) {
10     setpencolor blue
11     apply("square")(50)
12     right 180
13     setpencolor green
14     apply("square")(100)
15   }
16 }
17 program.delegate = new BlackBoxCombiner(new CompleteLogo(),new Functional());
18 program.call ();
```

Listing 5.11: Evaluating the FUNCTIONALLOGO example from Figure 5.10 with a composed interpreter

5.4.3.2. Composing Dependent Languages

Developers can compose dependent language embeddings using dependent combiners, as discussed in Section 4.2.3.2 (p. 107). At this point, this section only sketches the Groovy implementation of the simplest dependent combiner—the LinearizingCombiner. Implementations of other dependent combiners and custom combiners are presented later on.

Listing 5.12 presents the implementation of the LinearizingCombiner. Its implementation is similar to the BlackBoxCombiner, but there are important differences. First, in this combiner, the constructor (lines 4–7) does not enforce the signatures of keywords method to be unique. Second, its implementation invokeMethod (lines 9–16) does not throw an exception when more than one keyword method is found by for the same signature. Instead, in line 15, the combiner simply invokes the first method found (with the highest priority) and ignores the rest. These two derivations are necessary to allow composing dependent languages resolving syntactic conflicts taking into account the order in which the language are composed.

5.5. Summary

This chapter has presented a proof-of-concept for the reflective embedding architecture by providing a reference implementation for the core architecture in the Groovy programming language. The same architecture can be implemented in other languages.

In the context of this thesis, the core architecture was also implemented in the Ruby language. The are minor differences between the two instantiations. Since Ruby does not use interfaces, the

```
class LinearizingCombiner extends LanguageCombiner {
  List<Interpreter> facades;
  ...
  LinearizingCombiner(Interpreter... facades) {
    // 1) does not enforce unambiguous keywords
    // 2) sets itself as the bodyDelegate of all facades
  }
  ...
  Object invokeMethod(sender, receiver, mName, args, ...) {
    Set<Interpreter> respondingFacades = facades.findAll { interp ->
      interp.metaClass.respondsTo(interp,mName)
    };
    if (respondingFacades.empty()) throw new MissingMethodException("Keyword $mName undefined.");

    return respondingFacades.first().invokeMethod(sender, receiver, mName, args, ...);
  }
}
```

Listing 5.12: The LinearizingCombiner class

language interface levels is not present in Ruby. Further, since Ruby does not provide an explicit meta-object protocol and a closure-delegate mechanism, for Ruby, slightly different techniques needed to be used. Still, the central architecture's key mechanisms could be implemented in Ruby. Describing the details of the Ruby implementation is out of the scope of this thesis. Therefore, the interested reader is referred to the following report:

- Tom Dinkelaker, Christian Wende, and Henrik Lochmann. **Implementing and Composing MDSD-Typical DSLs.** TUD-CS-2009-0156, Technische Universität Darmstadt, Germany, October, 2009.

However, it remains to be shown that the language-embedding approach indeed allows implementing sophisticated languages and that the approach enables language evolution. In particular, the following issues need to be addressed:

- What particular host language features help language developers to grow embeddings with new specific language constructs? In particular, it is interesting to explore the opportunities that the flexibility provides through using the reflective features of the host language.
- Is it possible to embed more sophisticated kinds of language constructs, such as, data types or aspects?
- How can the core architecture be extended to support composition?
- In the core architecture, only embedding with an abstract syntax could be provided. How can concrete syntax be enabled for embedded languages?

To address these issues, in the following sections, we will present several extensions to the core architecture that provide solutions for the mentioned problems.

Part III.

Extensions to the Core Architecture

6. Reflective Domain-Specific Languages

The previous chapters have presented the reflective embedding architecture's core with its systematic support for late semantic adaptations. So far, the architecture's flexibility is only available for language developers, but not for end users. Unfortunately, language developers cannot always anticipate the requirements in the end users. Therefore, this section motivates the need for *reflective DSLs* that are special DSL that have open semantics. It elaborates this by means of an example that needs to adapt the language semantics in a user domain. To enable semantic adaptations in the user domain, this chapter describes how language developers can implement a reflective DSL on top of a reflective embedding in REA. It uses evaluations the resulting implementation by discussing required qualities for reflective languages, as Maes proposes them in [Mae87]. Further, it validates the use of reflective DSL by means of a case study.

Let analyze what variability is present in today's language and why its important. Recently, variability in language implementations is handled by extensible language approaches [MHS05], such as extensible interpreters and compilers, like *JastAdd* [HM03, EH07b]. With extensible interpreters/compilers, an expert can handle variation at compile-time of the language infrastructure, but these approaches fail to allow language end users to adapt the languages they want to use.

The requirement of supporting language adaptation by end users is addressed by applying the *open implementation principle* [Kic96, KLL$^+$97] to programming languages. An open implementation is a special module that developers design with a *primary interface* and a *meta-interface* to the module. The primary interface is simply an ordinary interface to the module that end user programs access to interact with the module. Further, the meta-interface allows end user programs to control the module's implementation strategy. When programs can control a module's implementation strategy, clients themselves can adapt modules to a specific context, which gains advantage over traditional closed modules.

From the beginning, the open implementation principle has been used in language implementations. There are many existing open language implementations for object-oriented programming languages—the meta-object protocols (MOPs) [KRB91]. However, the open implementation principle is interesting for other programming languages, e.g. that are not general-purpose or are not object-oriented.

Domain-specific languages are another class of languages that could benefit from having support for adaptations. The semantics of a given DSL is subject to: (1) *evolutions* of the language, e.g. [ZGL05], when a new version of the language semantics becomes available; (2) *language optimizations* without changing the semantics, e.g. [She04, SCK04]; and (3) *dis-consensus* on the right language semantics, e.g. UML state charts [OMG04], for which various semantics need to be supported in one language implementation. So far, this issue has only been little explored by means of extensible DSL compilers and interpreters [vDKV00]. Mernik stated [MHS05] that *"building DSLs [...] in an extensible way"* is an open problem.

Established extensibility mechanisms from general-purpose languages are frequently requested for DSLs [MHS05], examples are *aspects* [RMWG09] and *reflection* [CHBB09]. Still, there are only a few DSLs that use aspects to improve their extensibility, such as workflow languages like BPEL [CM04, Cha08], or grammar-specification languages [SLS03, RMWG09]. Reflection is more general than aspects and would be also interesting for DSLs. Counsellet al. state that *"a reflective DSL for analyzing code is long overdue"* [CHBB09]. Moreover, it would be interesting if a DSL program would be allowed to reason about its own domain. What is missing is a general concept for deriving DSLs with reflective features.

This chapter argues that by following the open implementation principle in a DSL implementation, a language developer can use to design an adaptable DSL that supports *late* semantic adaptations. Late semantic adaptations means, by order of importance:

1. There are different people that define (part of) the semantics. The DSL architect defines *early* the default semantics of the DSL. Later on, the DSL user may *lately* adapt the default semantics in an application-specific manner.

2. The semantic adaptation occurs after the DSL architect has released the default DSL implementation.

3. There are semantic adaptations that cannot be done by configuring a DSL interpreter or compiler. For example, they cannot be implemented by static adaptation of the semantics, when these adaptations depend on the execution context of a particular DSL program. Such late adaptations are similar to *late binding* of methods in OO programming, where *late* also expresses a dependency to the execution context.

To qualitatively validate the flexibility the reflective embedding architecture provides for DSL development, this chapter presents a study that discusses how to enable a wide range of late semantic adaptations of DSLs. This chapter shows indeed that DSLs can be implemented with a plan for growth by using two important methods from programming language research. First, inspired by reflective languages, the first idea is to grow a DSL into a reflective DSL, so that users can implement programs that reason about their semantics. Second, inspired by meta-object protocols, the concept embeds a DSL as an *open implementation* in order to provide a *meta-interface* to the user for adapting the DSL semantics. What is an particularly interesting property of the resulting DSL embeddings is that a DSL embedded into a reflective host language easily becomes itself an reflective language—a *reflective DSL*. With reflection for DSLs, the distinction between the role of the language developer and the end user is blurred.

The task to implement an open reflective DSL is not fundamentally different from open implementations and reflective languages. For a reflective DSL, a language developer needs to expose the internal structure of DSL to the user. For an open DSL, on the one hand, a language

developer includes *extensions points* into an open DSL implementation, at which later on a user can provide *extensions*. On the other hand, the user can apply such extensions the open DSL implementation to customize it for special needs. By implementing DSL as open implementations, both—the language developer and the end user—collaboratively develop the final semantics of the language. While extension points are very powerful tool for users, language developers can easily implement extension points on top of REA.
Note that parts of this chapter have been published in the following paper:

- Tom Dinkelaker, Martin Monperrus, and Mira Mezini. **Supporting Variability with Late Semantic Adaptations of Domain-Specific Modeling Languages.** In *Proceedings of the 1st International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines*, at the International Conference on Aspect-Oriented Software Development, 2010.

The remainder of this chapter is structured as follows. Section 6.1 discusses the need for supporting late variability in language semantics by means of an example, which is used as the running example throughout the chapter. Section 6.2 discusses the implementation of a DSL with default semantics. Section 6.3 discusses how to grow any DSL into a reflective DSL. Finally, Section 6.4 demonstrates the support for semantic adaptations in the architecture.

6.1. Late Variability in Domain-Specific Language Semantics

This section elaborates the need for open implementations of DSLs.

6.1.1. The FsmDSL Running Example

As a running example, this chapter introduce FSMDSL—a DSL for defining *finite state machines* (FSM). State machines can be defined with the following keywords:

- **fsm** $name$ **{**$fsmDefinitionBody$**}**: defines a new *state machine*. The keyword is followed by a name for the state machine and a block with one or more *state definitions* (in $fsmDefinitionBody$).

- **state** $name$ **{**$stateDefinitionBody$**}**: defines a new *state*. After the keyword, there is a corresponding name and a block that defines *actions* and *transitions* to other states (in $stateDefinitionBody$). Inside the body of the definition of a state, the following elements can be defined: (a) a set of variables using the keyword def. (b) a set of unconditional actions that will be executed either once when the automaton enters the state (keyword on entry) or once when the automaton leaves the state (keyword on exit); and (c) a set of transitions is defined to other state using the keyword when.

- **on entry {**$action$**}**: defines an action with a block of code that will be executed when entering into this state. Each state may have zero or one *on entry* action.

- **on exit {**$action$**}**: This keyword pair defines an action with a block of code that will be executed when leaving this state. Each state may have zero or one *on exit* action.

- **when** $eventName$ **{**$transitionAction$**}**: defines a *transition* from this state to another state. For each transition, there is an *event name* that guards the *transition action*. In the transition action, the state normally enters the *next state*, e.g. by executing **enter** $nextState$. When the state machine receives an event with the corresponding name defined by a transition, the state machines leaves the current state and enters the next state of the transition.

For illustration, consider the FSMDSL program in Listing 6.1, which implements a travel package booking process that combines booking several positions like a hotel, a flight, and a rental car. The goal of this listing is to give the intuition of how a FSMDSL program looks like, and to support the definitions and arguments that help us to state the problem. In Listing 6.1, the keyword fsm (line 1 defines a new state machine with the name TravelPackage that models two steps in a workflow. The first step is modeled as the state BookingFlight that is defined using the keyword state (line 2), and the second step of the workflow is model as the state BookingHotel (line 28).

In Listing 6.1, the action to be performed on entrance of the BookingFlight state is a call to a flight booking Web service. In the default semantics of this FSMDSL, events are handled synchronously. When receiving the acknowledgment from the Web service some_flight_webservice modeled by the event done in state BookingFlight, the FSM enters the BookingHotel state via the corresponding transition for this event (lines 10–12). Only when an error event is received, the corresponding transition action replaces the targetService through another synchronous service another_flight_webservice that has the same service interface, and then, the action repeats the failed step by reentering it.

6.1.2. Semantic Variability in FsmDSL

A DSL implementation is said to support *application-level semantic adaptations*, if it allows the semantics of domain types and objects to be modified from within a DSL program in an application-specific way.

For illustration, similarly to [HB04], consider that the default Web service for booking flights can fail. Let us also assume that the only other partner available is another_flight_webservice[1], which works asynchronously. However, the default semantics of the FSMDSL only handles synchronous Web services. In this case, this alternative partner cannot be integrated into the synchronous state machine, because the default semantics does not support mapping the event of an asynchronous message reception to a synchronous input event of a state.

The problem is solved if the semantics of the FSMDSL could be changed within the FSM program, more precisely by the logic responsible for the recovery transition of the BookingFlight state.

When Listing 6.1 uses an open implementation of FSMDSL that supports semantic adaptations, the TravelPackage program is able (cf. Listing 6.1, lines 15-25): (1) to handle the error

[1]For the sake of brevity, the code that selects the alternative partner from a Web service registry is omitted.

```
fsm 'TravelPackage' {
  state 'BookingFlight' {
    def targetService = "some_flight_webservice"

    on entry {
       ... // call to "book" operation on targetService
    }

    // nominal mode
    when 'done' {
      enter 'BookingHotel'
    }

    // error recovery mode
    when 'error' {
       // change the target Web service another_flight_webservice works asynchronously
       targetService = "another_flight_webservice"

       // redefine here event handling semantics to handle asynchronous events
       // and adopt the new semantics for this state
        [...]

       // ... and stay in the current state
       enter 'BookingFlight'
    } // end when
  } // end state

  state 'BookingHotel',{
    on entry { ... }
  }
}
```

Listing 6.1: Adaptive business process

event in the BookingFlight state, (2) to replace the failing partner[2], and (3) to adapt itself to the new Web service by changing the event reception semantics of the state BookingFlight in order to listen to asynchronous events (i.e., the asynchronous reception of the response message).

Such *self-adaptive* programs enable autonomous computing [Hor01] where the program configures, heals, optimizes, or protects itself to a well-defined adaptation goal without human interference into the running system. Meta-level architectures and reflective systems have been proposed since a long time to be employed for self-adaptation [MC02, HB04]. Still, there is little research on enabling self-adaptive programming for DSLs that lead to proposing solutions for individual DSLs. To leverage the same flexibility for other DSLs, it requires a systematic method for DSL implementation with support for self-adaptive programming.

Note that the adaptation of the FSMDSL semantics happens dynamically while the DSL program is running and depends on the execution context of the DSL program – in this case, the

[2]It is assumed that the alternative partner Web service has the same interface, i.e., the same message types are exchanged. The format of a message is independent from the Web service binding [CNW01] that selects a synchronous or asynchronous style for message interaction.

occurrence of the event error. Furthermore, the adaptation is fine-grained: only the semantics of one program part is affected; it is now possible to recover from the error in state BookingFlight, but the default synchronous semantics of other states is not modified.

Existing approaches to implementing DSLs do not support application-level semantic adaptations, in the way illustrated above, where the application basically redefines specific parts of the language semantics. In existing approaches changing the semantics at runtime is only possible if the semantic adaptation had already been envisioned at design time in the original DSL implementation.

However, it is not possible to envision every possible semantic adaptation a priori at design-time. Even if it was possible, the resulting explicit support for adaptations would bloat the implementation of the default semantics with additional attributes and conditional logic. Such an *one-size-fit-all* solution hampers the design of the default semantics which is used by most of the DSL programs. Last but not least, the DSL semantics could not be causally connected to the application state (i.e., dependent on the application state).

6.2. Implementing the Primary Interface of the FsmDSL

To create an open language implementation, a language developer simply embeds the language on top of REA, as it was described in Section 5. The default language implementation acts as the *primary interface* of the open language implementation through which programs interact with default syntax and semantics (cf. explanation of the primary interface and meta-interface in Section 4.2.3.3, p. 109). In the following, this section elaborates the embedding the primary interface of FSMDSL in REA, as a running example for demonstrating the concept throughout this chapter.

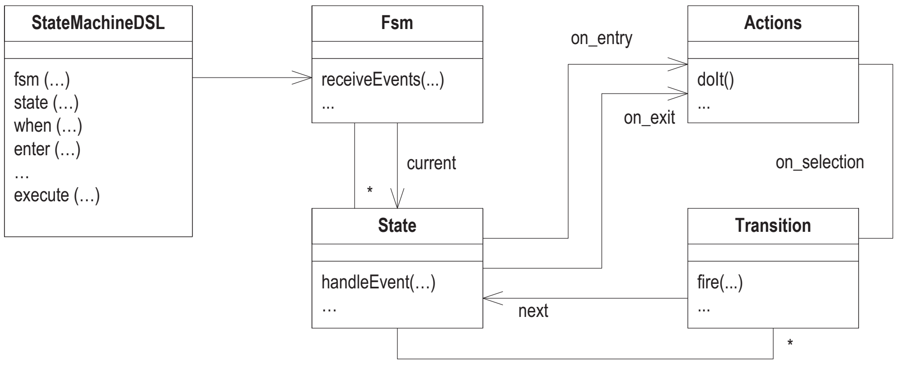


Figure 6.1.: Language model of the state machine DSL

Consider embedding and the implementation of the default semantics for the FSMDSL for state machines.

Figure 6.1 depicts the design of this DSL. The domain model consists of classes Fsm, State, Transition, Actions. The Fsm class maintains a set of states, refers to a current state and implements some default semantics of state machines, e.g., the method receiveEvents(...) defines the dispatch mechanism of events received by a state machine. State instances maintain a set of outgoing transitions and may have an on_entry action and implement the state semantics. For instance, the method handleEvent(...) defines the state event handling mechanism. A Transition points to the next state. The fire(...) method is called whenever a transition is selected. The class Action encapsulates a set of domain-specific actions; the semantics of an action execution is encoded in the doIt method. Finally, the language façade is implemented in class StateMachineDSL. There is a method for each keyword in the DSL. When called, these methods instantiate domain objects.

Listing 6.2 shows an excerpt of an embedded DSL program for state machines. The DSL program is contained in the closure dslPackage (line 2–11) which is configured in line 12 to have an instance of StateMachineDSL as its evaluator façade and whose evaluation starts at line 13.

Line 13 triggers the evaluation of the DSL program. This will load the state machine definition (lines 3–7), which creates the domain objects in the language model that represent the textual definition, and then the program executes a sequence of events (line 10). Next, the program sends a sequence of events in a method call to a domain object (resp. myFsm and execute), given in a specific execution context ({'ok','error',...}). The reception of each event triggers a cascade of method calls on the domain objects that were created when loading the state machine, which internally selects a transition and updates the current state of the machine.

```
 1  // this closure contains the DSL program
 2  def dslPackage = { // the DSL program
 3    fsm 'myFsm', {
 4      state 'S1', { ... }
 5      ...
 6      state 'SX', { ... }
 7    }
 8
 9    // executes the FsmDSL program when by sending the event list
10    myFsm.execute({'ok','error ',...})
11  }
12  dslPackage.delegate = new de.tud.statemachine.StateMachineDSL() // default facade for state machines
13  dslPackage(); // evaluates the closure
```

Listing 6.2: State machine embedded in Groovy

6.3. Growing a DSL into a Reflective DSL

This section explains how to grow an embedded DSL embedding into a reflective DSL. It explains the language developers perspective. First, in Section 6.3.1, we discuss the concept of reflective DSLs. After this, in Section 6.3.2, we discuss the implementation of reflective DSLs.

6.3.1. Reflective DSLs

Inspired by reflective general-purpose languages [Smi82, FW84, Mae87], this thesis defines a *reflective domain-specific language* as a DSL that allows a DSL program to reason about itself. In a reflective DSL, a DSL program can change its internal structure and behavior, whereby its first-class representations stays causally connected to its domain.

After the language developer has embedded a DSL in REA, REA implicitly makes a meta-level available for semantic adaptation by the language developer. The meta-level builds upon the general-purpose abstractions of the general-purpose host language. Therefore, end-user programs cannot conveniently access the meta-level using domain-specific abstractions, but only with general-purpose abstractions.

To bring semantic adaptations closer to the domain, this section presents how a language developer can grow any embedded DSL in REA into a reflective DSL. The development process for a reflective DSL starts from a non-reflective embedded DSL and is as follows. On the one hand, the developer adds support for *reification* [FW84] of domain-specific concepts into the DSL. On the other hand, the developer adds support for *reflection* [FW84]—both introspection *and* intercession—into the DSL. While this section presents the language developer's perspective of designing a reflection DSL, the next section will elaborate on how end users can use reflective DSLs.

To grow a DSL into a reflective DSL, the language developer needs to enable *structural* and *behavioral reflection* for selected language constructs of the DSL. As illustrated in Figure 6.2, to enable reflection, the developer needs to extend the DSL with *reflective extension*. The extension provides reflective language constructs for the domain, which are defined in an optional *reflective interface*, which acts as a *meta interface* to the language implementation (cf. explanation of the meta-interface in Section 4.2.3.3, p. 109). The end user of the reflective DSL can use this meta-interface for implementing strategies into their program.

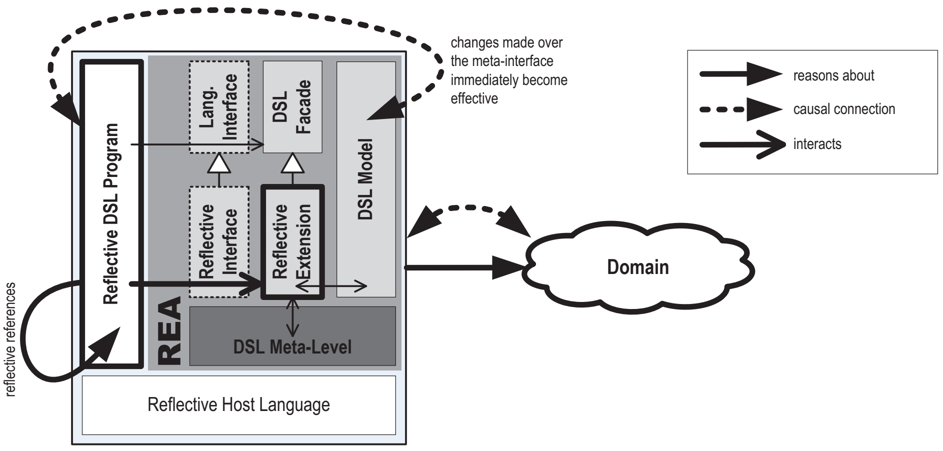


Figure 6.2.: Reflective DSL architecture

A *reflective program* uses reflective constructs in order to reason about its own structure and behavior. To enable structural reflection, the developer needs to turn a DSL implementation into a system that is *about its domain structure*. The system needs to allow answering questions about its structure, i.e. it can inspect and manipulate the structure and state of its domain objects. For making semantic adaptation effective, the system's domain objects need to be *causally connected* [Mae87] to their domain. To enable behavioral reflection, the developer needs to enable a DSL implementation that allows reasoning *about its domain behavior*. The system needs to allow answering questions about its behavior, i.e. it can inspect the events of its domain objects. The system needs to allow manipulating the behavior that results from reactions to the states and events in the domain objects.

The remainder of this section elaborates how a language developer can grow an arbitrary DSL into an reflective DSL. Section 6.3.2.1 and Section 6.3.2.2 demonstrates with a concrete example that, when embedding a language into a reflective host language, it is easy to obtain a reflective version of the language with only little changes made to its implementation. Section 6.3.3 argues that this is easy, because the *reflective embedding architecture* fulfills the requirements of the *reflective system architecture* by Maes [Mae87]. Moreover, Section 6.3.4 argues that once having obtained an reflective DSL after Maes's architecture, a language developer can further specialize by evolving into a more declarative one.

6.3.2. Implementation of Reflective DSLs

To extend a DSL with reflection, the language developer provides access to the language internals. They define special language constructs called *reflective references* that are implicit references (cf. Section 3.5.2, p. 55) to meta-level entities. A program can use a reflective reference to access the meta-level for structural and behavioral reflection.

6.3.2.1. Structural Reflection

To enable structural reflection, the language developer needs to extend the DSL to allow accessing the structural program representation.

Consider extending the FSMDSL to expose the most important language constructs inside a state machine, such as its states and the state machine itself. To expose the enclosing state, the language developer only needs to sightly extend the language façade with an reflective reference thisState that retrieves the instance of the enclosing state's domain-object representation from the language model. Listing 6.3 presents the ReflectiveStateMachineDSL that extends StateMachineDSL. The sub-class exposes the enclosing state of the state machine by providing the getter getThisState() (line 14) that returns the first-class representation from the DSL language models. To expose the state machine with the reflective reference thisFsm, a corresponding getter getThisFsm() (line 16) returns the representation of the whole state machine. To access the states and the machine via their names, it is possible to expose the first-class representation via their name identifiers by overriding the meta-method propertyMissing (19–22) that the Groovy MOP will invoke, whenever using an identifier that is not explicitly defined as a variable. The overridden meta-method either returns (1) the state machine if the pretended property's name identifier is equal to the name of the state machine, in line 20, (2) it tries to look up the state with the corresponding name and returns this if defined, in line 22, or (3) it returns null otherwise.

```
 1 class StateMachineDSL extends Interpreter {
 2   Fsm currentStateMachine;
 3   Fsm fsm(...) { ... }
 4   State state (...) { ... }
 5   void when(...) { ... }
 6   void enter (...) { ... }
 7   void on_entry (...) { ... }
 8   void on_exit (...) { ... }
 9    ...
10 }
11
12 //the reflective extension for state machines
13 class ReflectiveStateMachineDSL extends StateMachineDSL {
14   State getThisState() { return currentStateMachine.getCurrentState(); } //exposes the enclosing state
15
16   Fsm getThisFsm() { return currentStateMachine; } //exposes the enclosing state machine
17
18   // resolves reflective references when using fsm/state names outside the DSL program
19   public Object propertyMissing(String name) {
20     if (name.equals(currentStateMachine.getName())) { return currentStateMachine; }
21     else { return currentStateMachine.getState(name); }
22   }
23 }
```

Listing 6.3: Reflective extension of the default interpreter for structural reflection

To evaluate a reflective program, such as the one in Listing 6.4, the user only has to set up the extended façade ReflectiveStateMachineDSL as the program's closure delegate line 24. The program can use different forms of structural reflection, as explained by the following examples.

First, a reflective program can access the DSL programs structures inside entry and exit actions. For example, inside an action, the program can use the reflective reference thisState (line 5 in Listing 6.4) to obtain a reference to the enclosing state's State object and prints out its string representation. Note that, such obtained State objects and Fsm objects depend on the enclosing lexical context in which the reflective reference occurs. Similarly, the reflective reference thisFsm (line 5) returns a first-class representation of the enclosing state machine (e.g. "myReflectiveMachine").

Second, a reflective program can also access the structural representation of a state machine from outside the machine's definition block. Specifically, the program can directly access parts of the machine's structural representation via the identifiers defined inside the machine. For example, line 12 uses a state machine's identifier ReflectiveMachine to obtain a reference to the first-class representation of the state machine and prints out a string representation of it. Similarly, line 13 uses a state's identifier S1 to obtain a reference to a first-class representation

```
 1 def dslPackage = {
 2   // the DSL program
 3   fsm 'myReflectiveMachine', {
 4     state 'S1', {
 5       on_entry { println "entering "+thisState+" in "+thisFsm; } // introspection via reflective reference
 6     }
 7     ...
 8     state 'Sx', { ... }
 9   }
10
11   // Introspection by identifier name (resolved via propertyMissing)
12   println "My state machine: "+myReflectiveMachine.toString(); // introspects a state machine by name
13   println " It 's state 'S1' has the following transitions : "+S1.transitions ; // introspects a state by name
14   ...
15   ...
16   // Intercession
17   State errState = new State("ErrorState"); // creates a new first −class DSL construct
18   myReflectiveMachine.addState(errState); //intercedes a state machine by name
19   myReflectiveMachine.states.values().each { State s −>
20     s.addTransition(new Transition(s,errState ," error "));
21   }
22   ...
23 }
24 dslPackage.delegate = new ReflectiveStateMachineDSL();
25 dslPackage();
```

Listing 6.4: A program using structural reflection

of the corresponding state. The program can access a domain object's attributes, such as line 13 accesses the transitions of S1 and print them out as a list.

Third, a reflective program can manipulate the structure of the program using intercession. For example, lines 17–18 create a new state ErrorState and add it to the above state machine. Further, lines 19–21 add a transition from every defined state to the ErrorState, when receiving an error event. Note that in contrast to a non-reflective version of this program, the benefit of the reflective version is that the error state can be added in a modular way.

With the Groovy MOP, structural reflection can also change the structure of existing classes and objects. Specifically, the user can add fields or methods. To adapt a domain type or object, the user retrieves the meta-object of it. To get the meta-object of the domain type, the developer uses Groovy's metaClass property, e.g. State.metaClass. To get the meta-object of a domain object, the developer accesses metaClass property of the domain object, e.g. "S1.metaClass". For example, the program can add a new field to an individual State object, e.g. to S1, with the following code "S1.metaClass.counter = 0;". After such a *structural intercession*, the program can access the introduced field like an ordinary field, e.g., "S1.counter++" increases the introduced counter field.

6.3.2.2. Behavioral Reflection

To enable behavioral reflection, the developer uses the Groovy MOP features to intercede with embedded DSL semantics. The support for behavioral reflection relies on the available support for structural reflection to conveniently access the first-class representations of DSL constructs. Since in Groovy closures are first-class objects, indeed, there is no difference between manipulating fields and methods.

To adapt a program with behavioral reflection, the end users access the meta-object of a DSL construct to manipulate its domain operations. With closures, users can provide an alternative implementations for a domain operation, and with the Groovy MOP, they can replace this domain operation by the alternative one. The reuse of functionality for behavioral reflection from the host language is interesting, in particular, because implementing behavioral reflection for a new language from scratch is expected to be difficult [Mae87].

To adapt a domain method, the user retrieves the meta-object of its domain type or domain object. In the meta-object of a type or an object, for each domain operation (e.g., handleEvent), there is property with the name of the domain operation that holds a link to a closure, which is initially is its default implementation. Next, the user can adapt the operation by assigning the link to another closure (e.g., "State.metaClass.handleEvent = {...}").

Consider adapting a FSMDSL program to count how often an event was received by an individual state, such as in the program in Listing 6.5. Line 9 adds a counter to the State domain object with name S1. Lines 12–16 replace the domain-operation link to handleEvent of this state to refer to a closure that prints out a logging statement and increases the counter. Note that, inside the closure of an alternative implementation of a domain operation, to invoke a method on the enclosing object, in Groovy, the developer needs to use the keyword delegate, which has a similar meaning as the this[3] keyword has in a method body of the class.

6.3.3. On the Design Properties of Reflective DSLs

To review the concept of reflective DSLs, the following paragraphs compare the resulting design properties of the obtained DSL to the properties that Maes proposes in her seminal paper. In [Mae87], Maes discusses several important design properties of reflective architectures. The following discussion reviews how a reflective DSL implemented on top of the reflective embedding architecture meets these design properties. In particular, the discussion emphasizes how these properties for the embedded language are inherited from its reflective host language.

(1) disciplined split between object-level and meta-level: Maes argues that the language design should connect every object to a meta-object, but objects and meta-objects should not be confused.

In reflective embedding, thanks to the MOP, every class in the language implementation has a meta-object, including the language façade and every class in its model. Because REA uses different types at the meta-level and the other levels, there is a clear split between the *object*

[3]The developer cannot use the keyword this in the body of the closure of an alternative implementation, since this refers to the enclosing object, which in this case, the enclosing script that adapts the domain operation, not the class or the object to be adapted.

```
def dslPackage = {
  // the DSL program
  fsm 'myReflectiveMachine', {
    state 'S1', { ... }
     ...
  }
   ...
  // intercede structure
  S1.metaClass.counter = 0;

  // intercede behavior
  S1.metaClass.handleEvent = { String event ->
    Transition t = delegate.transitions.get(event);
    println "HANDLE EVENT: receives "+event+" no.${++(S1.counter)} and takes transition "+t;
    return t;
  }
   ...
}
dslPackage.delegate = new ReflectiveStateMachineDSL();
dslPackage();
```

Listing 6.5: A program using behavioral reflection

level and the *meta level* in the implementation between a classes that extend Object and classes that extend MetaObjectImpl.

Note that the reflective embedding architecture is generalizes the proposal of Meas to whole languages. In REA, there is a first-class object of the language itself in form of the language façade, the language itself as a whole can be manipulated not only objects. But, there are also limits, since Groovy is subject to conflating levels. Recall, in Groovy, every class may override one of its meta-methods, such as methodMissing. When an object overrides a meta-method, it actually implements meta-level activities at the base level, in fact this conflates the levels. Although conflating levels can be very convenient, in case the user/developer does not carefully implement a meta-method, there is the danger of *infinite regression*—an endless loop between the base and the meta-level logic.

(2) uniform representation: Maes argues that the design needs to be represented in a *uniform* way.

In the reflective embedding architecture, all reflective concepts are represented as objects.

On the one hand, every GPL construct is represented as an object in Groovy, such as classes, closures.

In the other hand, every domain-specific language construct is represented through an object in REA. In the above example, these are Fsm objects, State objects, and Transition objects. Since the architecture follows a homogeneous approach to embed a reflective language, the resulting reflective system, i.e. the homogeneous embedding has no technological disruptions with the host language, its reflective architecture, and its tools, which makes the resulting language implementation uniform. For supporting reflective DSLs, it is crucial that REA does not commit to a special set of language constructs.

Additionally, REA provides also uniform high-level reflective abstractions for language components, which are not present in most other reflective languages. In particular, the architecture also represents languages as a set of objects, such as the language façade instance and the objects of the language model. REA mechanisms introspect and intercede these objects, such as language combiners compose language components, and there is support for analyses and transformations (cf. Section 10).

(3) complete representation: Maes argues that the meta level should contain the *complete* information about the base level.

In the reflective embedding architecture, this is not completely the case, but there are only minor limitations due to restrictions of the Groovy MOP. From the language model of a reflective embedding, everything is accessible as an object. Still, there are a few minor restrictions, since the current version of the Groovy MOP does not provide all conceivable features, i.e. Groovy does not reify basic blocks.

Specifically, REA currently does not support *reflective methods* [DDLM07]. Despite every Groovy closure is represented as an object with a corresponding meta-object, it is not possible to access the AST inside a closure at runtime.

Despite this, note that later on Section 10.6 will demonstrate how to obtain access to the program's AST. The language developer can implement an analysis to obtain an AST-like representation of DSL constructs used inside a closure, that are reified as a sequence of method calls using an analyzer façade. Moreover, in a reflective embedded DSL, it depends on the studiousness of the language developer to expose all information about the domain in an adequate way. For example, the FSMDSL reflective DSL favors simplicity over *complete* reflection, specifically it does not completely reify events as first-class objects, but it maps them to String objects.

(4) consistent self-representation: Maes further demands the design to use a *consistent self-representation*, i.e. the representation whose elements represents themselves. In other words, she argues that a system should implement itself with its own means, which is why reflective systems are often implemented as *meta-circular interpreters* [SS78]. For example, meta-level objects encode the semantics of base-level objects, meta-level procedures encode base-level procedures, and meta-level rules encode base-level rules.

In contrast to this, in the REA everything is represented with OO abstractions. More specifically, DSL constructs are not implemented in the DSL itself. For example, the above example represents a state machine using objects, but it does not represent a state machine through a kind of *meta state machine*. Although this is not true self-representation, the representation is still consistent. Since everything can be represented as an object and those object are represented with objects, everything is indirectly represented via self-represented objects.

Another perspective on that is that homogeneous embedded DSLs can be thought of as being *quasi meta-circular*, because of two reasons. First, when homogeneously embedding a DSL into a host language, by default, the DSL inherits the computational power of the host language. Second, when embedding a DSL into a meta-circular host language as a library, the embedded DSL does not destroy the meta-circular property of the host language. Because the host language is still a subset of a homogeneous embedded DSL, a meta-circular implementation of the host language together with the embedded library is a meta-circular implementation for the embedding. Consequently, the embedding inherits the meta-circular property from its host language. Despite this property of the reflective embedding architecture, unfortunately its current instantiation in Groovy is not meta-circular, since Groovy is not implemented as a meta-circular compiler.

However, one can argue that self-representation make little sense for most DSLs. A problem with self-representation of a DSL is that when one needs a Turing complete language to implement the DSL, this requires the DSL itself to be *Turing complete*. With Turing complete DSLs, one could realize self-representations through a meta-circular implementation. For instance, *tex* is Turing complete [Tay92], therefore one can use *tex* to implement *tex*. But, there are two reasons why self-representation is often not possible for many DSLs. First, declarative DSLs that are not Turing complete disallow implementing themselves, e.g. *HTML* cannot be used to implement *HTML* (without using *JavaScript* [Fla02]). Second, even if it would be possible to implement, still it can be expected to cost too much effort to do so, e.g. one can implement the *BPEL* workflow language [AAB+07] in itself, but BPEL abstractions are only adequate for *programming-in-the-large*. Instead, low-level implementations would be needed to implement BPEL's execution semantics in an efficient way. Consequently, self-representation cannot be considered as a good design goal for reflective DSLs in general.

(5) run-time modification: Maes argues that the representation should allow run-time modification.

In the reflective embedding architecture, thanks to the fact that the Groovy MOP supports runtime modification, reflective programs can modify every class or individual object during the execution of programs. Developers can add both fields and methods, and they can adapt existing fields and methods.

6.3.4. Evolving from Procedural to Declarative Reflection

Maes distinguishes reflective systems with respect to the abstraction level of the reflective language that is used to reason about the system and to manipulate it. There is *procedural* and *declarative reflection*.

A system supports *procedural reflection*, when the system itself exposes its own *interpretation-process*. In other words, there is a procedural representation of the system that is visible to the user and that can be changed by its own means. For procedural reflection, often the reflective language is implemented as a *meta-circular interpreter*, thus the system uses the same language to implement its reflective language, which is also available to the user for implementing reflective programs. An advantage of this approach is that, for language developers, implementing a *meta-circular interpreter* is an easy way to fulfill causal connection [Mae87]. But, often the resulting reflective language is a rather low-level language with disturbing details [KRB91]. A problem with such a low-level language is that it may not provide an efficient notation for users.

In contrast, if a system supports *declarative reflection*, then the users do not have direct access to the complete procedural representation of the system. They can only use declarative statements to change the system internals. In such systems, the user rather declares constraints to describe *what* the system must fulfill but not *how*. The declarative notation tends to be more safe and efficient for users, since the declarative notation restricts possible user inputs in a controlled way. Further, the system's restricted reflection capabilities often allow to implement its execution in a more efficient way; because of the restrictions optimizations become conceivable. However, for the language developer, it is more complicated to implement such a language.

In sum, there are advantages and disadvantages of both styles of reflection, and Maes mentions examples of reflective systems that can be classified between the two extremes, because they incorporate both procedural and declarative parts. According to Maes this suggests that the distinction is rather a continuum.

What is special in the reflective embedding architecture is that language embeddings are allowed to slide on this continuum, as illustrated in Figure 6.3. For example, initially, a reflective language embedding starts as a procedural reflection (index 1), but later on, a language developer can specialize a language interface and its implementation by adding more declarative language constructs on top of the low-level reflection mechanisms. As indicated in the figure, they can provide more safety guarantees by restricting the available power of the reflective system (index 2). Further later on, those declarative language constructs may allow optimizing the language implementation (index 3).

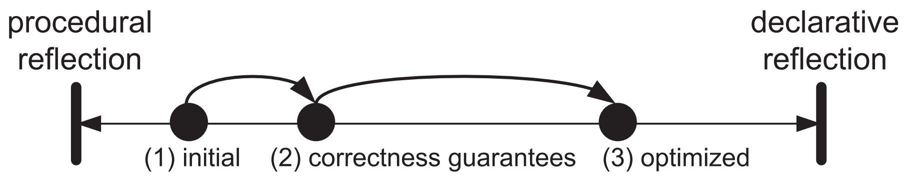


Figure 6.3.: Continuum of procedural and declarative reflection

As an example for evolving a reflective DSL to be more declarative, consider introducing a declarative abstraction that consistently adds transitions to all existing states of a machine. Specifically, the language developer can implement such a declarative abstraction by defining a new keyword method drain. With drain, users can consistently add transitions from all states to one particular state. For example, with the snippet "drain(fsm:thisFsm,to:ErrorState,when:"error")", the user adds a transition from every state in thisFsm to the ErrorState, whereby each transition is guarded by a label that requires the error event. Using the declarative drain keyword is safer than using the rather low-level reflection mechanism. When the user adds each transition using the low-level structural reflection mechanisms, the user could accidentally forget to change one of the states. With a more declarative reflection mechanism like in the above example, the provided abstractions help end users to apply reflective adaptations.

They can use reflection without taking care of adaptation details, thus their reflective program likely become less prone to errors.

From the above discussion, one can conclude that the reflective embedding architecture supports continuous evolution of reflection from a general-purpose reflective DSL to more a specialized and declarative DSL that abstracts from details of reflective programming.

6.4. Case Study: Using Semantic Adaptation Scopes to Adapt an Open DSL Implementation

This section presents how an end user can customize an open DSL implementation in their programs. On the one hand, the user can reason on programs. By using the DSL's reflection mechanism, the user can introspect with program. On the other hand, the user can reason on the language semantics. By using the (generic) meta-interface, the user can intercede with the semantics of the open DSL implementation. To demonstrate applications of introspection and intercession, this section first presents semantic adaptations for the FSMDSL, in Section 6.4.1. Then, after this, Section 6.4.2 to Section 6.4.4 demonstrate how to apply these semantic adaptations to the open DSL implementation, in a controlled way, by using with the three dimensions of the reflective embedding architecture (cf. Section 4.2.2, p. 89).

6.4.1. Possible Semantic Adaptations for the FsmDSL

The UML specification [OMG04] discusses several semantic adaptations for state machines. This section considers two of them.

Synchronous vs. asynchronous event handling: Listing 6.6 shows two possible implementations of State's domain operation handleEvent. The first implementation is the default one and encodes synchronous event handling; the second implementation supports asynchronous event handling.

```
// default: synchronous event handling
def handleEvent(Event event) { delegate.fsm.currentState = delegate.transitionSelection(event). fire (); }

// alternative : asynchronous event handling
def handleEvent(Event event) {
  if (delegate.queue.isEmpty) { delegate.fsm.currentState = delegate.transitionSelection(event).fire (); }
  else { delegate.fsm.queue.add(event); }
}
```

Listing 6.6: Two implementations of handleEvent

Deterministic vs. random transition selections: A given state of a state machine can have several transitions matching a given event, i.e. the FSMDSL is non-deterministic. In this case, a state machine implementation has to provide a transition selection policy. In REA, the semantics of transition selection is encoded in the transitionSelection method of the domain class State.

Listing 6.7 shows two possible implementations of transitionSelection. The default implementation of the transition selection policy is a deterministic one. It returns the first element of the collection of matched transitions. The alternative implementation selects a random item in the collection of matched transitions for a fairer load-balancing.

```
// default semantics: deterministic transition selection
def transitionSelection (Event event) { return delegate.transitions. findAll (event). first ; }

// alternative semantics: random transition selection
def transitionSelection (Event event) { return delegate.transitions. findAll (event).getRandom(); }
```

Listing 6.7: Two implementations of transitionSelection

The following sub-sections will discuss scenarios of using REA to apply alternative semantics for event handling and transition selection thereby varying the kind of adaptation along the three dimensions discussed previously.

6.4.2. Dimension: Locality of Adaptations

Listing 6.8 and Listing 6.9 show the implementation of the *locality of adaptations* dimension (D1) presented in Section 4.2.2.1.1 (p. 91).

```
 1 // this closure contains the DSL package (DSL program + adaptations)
 2 def dslPackage = {
 3   // the DSL program
 4   fsm 'myFsm', {
 5     state 'S1', { ... }
 6     ...
 7     state 'SX', { ... }
 8   }
 9
10   // adaptation of the default semantics of the domain class State by replacing the implementation of
11   // the transitionSelection method of the default meta−object associated with the State class
12   State.metaClass.transitionSelection = { event −>
13     return delegate.transitions. findAll (event). last
14   }
15
16   // executing the DSL program
17   myFsm.execute({'ok','error ',...})
18 }
19 dslPackage.delegate = new ReflectiveStateMachineDSL()
20 dslPackage()
```

Listing 6.8: Domain type semantic adaptation

Listing 6.8 illustrates domain type semantic adaptation (D1.a). The module dslPackage (lines 2–18)—a Groovy closure—consists of three parts: (1) The declaration of a DSL program (lines 4–8), (2) a piece of meta-program that tailors the semantics of the DSL (lines 12–14), and (3) a piece of code that starts the execution of the DSL program (line 20). Parts (1) and (3) have been explained in Section 6.2. Most importantly, the adaptation part (2) changes the transitionSelection method of the default state meta-object in line 12 to an alternative implementation—the closure in lines 12–14. As explained earlier, all instances of class State are affected by this kind of adaptation and will execute with the tailored transition selection semantics.

Listing 6.9 shows an example of semantic adaptation performed at the level of a single domain object (D1.b). The only difference to Listing 6.8 is in Listing 6.9, lines 11–13. While for domain type adaptation the program changes the meta-object of State, for domain object adaptation the program changes the meta-object of the domain object that represents the state *S1*. Only state *S1* has the new implementation of the transitionSelection method.

```
 1 def dslPackage = {
 2   // the DSL program
 3   fsm 'myFsm', {
 4     state 'S1', { ... }
 5     ...
 6     state 'Sx', { ... }
 7   }
 8
 9   // adaptation the default semantics of **only instance S1** by replacing the implementation
10   // of the transitionSelection method S1 and Sx do not have the same transitionSelection semantics
11   myFsm.S1.metaClass.transitionSelection = { event ->
12     return delegate.transitions. findAll (event). last
13   }
14
15   myFsm.execute({'ok','error ',...}) // executing the DSL program
16 }
17 dslPackage.delegate = new ReflectiveStateMachineDSL()
18 dslPackage()
```

Listing 6.9: Domain object semantic adaptation

6.4.3. Dimension: Atomicity of Adaptations

Listing 6.10 and Listing 6.11 illustrate the *atomicity of adaptation* semantic dimension (D2), presented in section Section 4.2.2.1.2 (p. 94). On the one extreme in this dimension, semantic adaptation affects a single domain method (D2.a); on the other extreme, the semantic adaptation may imply the construction of a completely new meta-object (D2.b).

Listing 6.10 is an excerpt of Listing 6.8. It focuses on the adaptation in line 6. The only element that is changed is a domain method of a domain class.

On the contrary, Listing 6.11 shows the creation of a semantic module and its use for tailoring the semantics of a domain class. Similarly to using classes to represent domain types, one can use a new subclass for modularizing alternative semantics for a domain type. In the example, lines 4–7 define such a subclass called TailoredState. Subclassing a domain type to create a new meta-object allows leveraging two key Groovy features used in Listing 6.11:

```
fsm 'myFsm', {
  ...
}

// The developer tailors only transition selection part of the semantics of States
State.metaClass.transitionSelection = { event -> return delegate.transitions.findAll (event). last ; }

myFsm.execute({'ok','error ',...})      // executing the DSL program
```

Listing 6.10: Method-level adaptation

1. The possibility to attach new semantics in form of an alternative meta-object[4] to an existing domain class, using Groovy's *meta-class registry* mechanism (line 11).

2. Groovy automatically creates a meta-object for each new class that is instantiated (line 11).

A meta-object for the new subclass is automatically created and stored in the class variable TailoredState.metaClass. In Listing 6.11 line 11, the adaptation part registers this meta-object also for the State class. As a consequence, the domain operation implementations of TailoredState will be used for State objects.

```
 1  ...
 2  // To package atomic adaptations into one unit,
 3  // the developer uses a sub-class that modularizes the semantic adaptation for its super class
 4  class TailoredState extends State {
 5    def transitionSelection (event) { /* cf. variation point transitionSelection */ }
 6    def handleEvent(event) { /* cf. variation point handleEvent */ }
 7  }
 8
 9  // Next, the developer can tailor the semantics of State in one unit
10  // for both event handling and transition selection
11  InvokerHelper.metaRegistry.setMetaClass(State, TailoredState.metaClass)
12
13  myFsm.execute({'ok','error' ,...})   // executing the DSL program
```

Listing 6.11: Semantic module adaptation

6.4.4. Dimension: Time of Adaptations

Listing 6.12 demonstrates using scoping adaptation with the *time of adaptation* dimension (D3), presented in Section 4.2.2.1.3 (p. 95).

Pre-execution adaptation (D3.a) is used to set up the desired semantics for a DSL program before running it. Pre-execution adaptation allows us to reuse a DSL program; in the running

[4]To preserve compatibility with existing clients, the new meta-object must obey the same object interface as the obeyed by the old meta-object.

example, a developer can reuse the program that has been implemented for synchronous Web services to interact with an asynchronous Web services. Assume that in the running example the language developer knows in advance that the synchronous some_flight_webservice has become unavailable (e.g. the service has been permanently *retired*) and we would like to replace it with another_flight_webservice which employs asynchronous interaction. In order to adapt the semantics with pre-execution adaptation, first, the DSL program is declared in lines 3–9. Next, specific semantics are selected in lines 11–13 for the program that was declared above. Finally, line 15 executes the program under the selected semantics; the semantics does not change while the program is running.

```
 1 def dslPackage = {
 2   // declaration of the DSL program
 3   fsm 'TravelPackage', {
 4     state 'BookingFlight', {
 5       on_entry { ... /* synchronous call to some_flight_webservice */ }
 6       when 'done' { enter 'BookingHotel' }
 7     }
 8     state 'BookingHotel', { on_entry { ... } }
 9   }
10
11   // pre−execution adaptation
12   TravelPackage.BookingFlight.targetService = "another_webservice"
13   TravelPackage.BookingFlight.metaClass.handleEvent = { event −> /* asynchronous implementation */ }
14
15   TravelPackage.execute({'ok','error',...})  // executing the DSL program
16 }
17 dslPackage.delegate = new ReflectiveStateMachineDSL()
18 dslPackage()
```

Listing 6.12: Pre-execution adaptation

In contrast, context-dependent adaptation (D3.b) allows programs to reflectively adapt their semantics at runtime. In this case, the semantic adaptation code becomes part of the code of the DSL program. Consider the DSL program in Listing 6.13 that implements the running example from Listing 6.1. The adaptation logic is situated in lines 11–17. At runtime, only when DSL execution reaches the lines 11–17, the program executes the adaptation logic, which adapts the language semantics to asynchronous event handling.

6.5. Summary

This chapter has presented a study on how to implement reflective DSLs that support semantic adaptations that may be application-specific and may occur as late as during the execution of DSL programs. The proposal of a concept of reflective DSL leverages two important concepts from general-purpose programming languages in the context of domain-specific languages—*reflection* [Smi82, FW84, Mae87] and meta-objects [KRB91]. Also, the study elaborated on an

```
def dslPackage = {
  // declaration of the DSL program
  fsm 'TravelPackage', {
    state 'BookingFlight', {
      on_entry { /* synchronous call to flight_webservice */ }

      // nominal mode
      when 'done', { enter 'BookingHotel' }

      // error recovery mode
      when 'error', {
        targetService = "another_flight_webservice"; // changes to an asynchronous Web service
        delegate.metaClass.handleEvent = { event -> // change the FsmDSL's internal event handling too
          /* asynchronous implementation */
        }
        enter 'BookingFlight' // re-enter current state
      } // end when
    } // end state

    state 'BookingHotel', { on_entry { ... } }
  }

  TravelPackage.execute({'ok','error','done',...}) // executing the DSL program
}
dslPackage.delegate = new ReflectiveStateMachineDSL()
dslPackage()
```

Listing 6.13: Context-dependent adaptation

instantiation of an open and reflective DSL implementation in the Groovy programming language.

The results of the study allow concluding that the presented techniques in this chapter leverage a similar flexibility for DSLs as provided by reflection mechanisms and MOPs for GPLs. An interesting point raised in the chapter's discussion about continuously evolution from procedural to declarative reflection is *how can a language developer build special reflective abstractions on top of existing low-level reflective mechanisms*. To address this point, the following chapter will elaborate how to enables more declarative means for adaptation, namely aspects as a partial reflective mechanism for a safe adaptation of DSL programs.

7. Crosscutting Composition of Embedded Domain-Specific Languages

Various DSALs have been proposed for different domains, such as coordination [Lop97], security [FS99], transactions [FD06], or grammars [SLS03, RMWG09]. These DSALs are implemented using pre-processors and hence inherit their limitations especially with respect to composability. Generic approaches to implementing (domain-specific) aspect languages [MKD03, LN04, UMT04, TN05, ACH$^+$06, HJZ07, GV07, KL07a, AET08, HBA08, HHJ$^+$08, HHJZ09] do not support the composition of aspects from one domain into a DSL from another domain [DMM09].

As a result, (domain-specific) aspect-oriented concepts for base DSLs are implemented from scratch for every new domain and combinations thereof. E.g., AO4BPEL [CM06] provides aspect-oriented (AO) support for composing crosscutting concerns (e.g., security) into BPEL [AAB$^+$07], and SQXML AOP [Alm] implements support for modular composition of crosscutting concerns into SQL.

To address this problem, this chapter presents how to compose several DSLs with interacting semantics. As Section 3.3.2.1 (p. 44) argues, if only black-box composition of EDSLs is supported, programs that use several EDSLs may exhibit scattering and tangling symptoms if the domain semantics are crosscutting. To address this issue, in REA, it is possible to embedding DSLs together with aspect-oriented concepts on top of the host's MOP. When aspects are embedded together with DSLs, the AO concepts enable crosscutting parts of the programs of one or several domains.

With aspects, end users can write more modular programs that have a better maintainability and understandability. In addition to that introducing AO concepts tames the MOP. Using aspects instead of the MOP to compose DSL programs takes advantage of the fact that restricting the power of the MOP provides more guarantees [LK03, KL04]. To enables such invasive but safe composition, this section shows that with such embedded aspect-oriented concepts, REA supports *crosscutting composition* of EDSLs.

Parts of this chapter have been published in the following chapters:

- Tom Dinkelaker, Michael Eichberg, and Mira Mezini. **An Architecture for Composing Embedded Domain-Specific Languages.** In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD)*, 2010.

The remainder of this chapter is organized as follows. Sec. 7.1 presents the example scenario that we use throughout the chapter to discuss the implementation of EDSLs. Sec. 7.2 discusses the scattering and tangling program in EDSLs. Sec. 7.3 discusses crosscutting composition.

7.1. Running Example: ProcessDSL—A Workflow Language

This section uses a simple *workflow language* [WfM] – called ProcessDSL – for illustration purposes. ProcessDSL is inspired by real-world DSLs for orchestrating Web Services [AAB^{+}07, DJKN07]. ProcessDSL provides the following domain-specific abstractions to define processes:

process: to define workflows.
task: to define a task as part of a workflow.
registry: to access the registry of Web services.
notify: to notify the process stakeholders.

Listing 7.1 demonstrates the use of ProcessDSL to select a credit among a set of offers. The EasyCreditProcess consists of two sequential tasks: (1) getOffers and (2) selectOffer. The first task (lines 4–11) starts with searching for all banking services in the registry. After that, each banking service is inquired to get the current interest rate. The second task (lines 13–18) selects the cheapest offer and requests the credit. At the end, notify (line 17) sends a message with the credit details to all stakeholders.

```
 1  process(name:"EasyCreditProcess") {
 2    def offers = [:];
 3
 4    task (name:"getOffers") {
 5      def services = registry.find ("Banking");
 6      services.each { bank ->
 7        def request = bank.createRateRequest(10000,"USD");
 8        def response = bank.call("getRate", request);
 9        offers [bank] = response;
10      }
11    }
12
13    task (name:"selectOffer") {
14      def selectedBank = ... //bank with the cheapest offer
15      def request = selectedBank.createBorrowRequest(10000,"USD");
16      def response = selectedBank.call("borrow", request);
17      notify "Credit from ${selectedBank.name}: $response";
18    }
19  }
```

Listing 7.1: An example ProcessDSL program

7.1.1. Example Scenario for Crosscutting in DSLs

To motivate the need for cross-cutting composition, let us assume that we also want to support the confidential exchange of messages. One approach is to extend PROCESSDSL and to define a new DSL - SECUREPROCESSDSL which adds the following domain-specific abstractions:

encrypt: for encrypting messages.

decrypt: for decrypting encrypted messages.

me: to access the identity that started the process.

SECUREPROCESSDSLcan easily be implemented by inheriting and refining PROCESSDSL. A class SecureProcessDSL implements the additional methods encrypt, decrypt, and getMe. The methods encrypt and decrypt are implemented using a cryptographic library and the getMe will return an object implementing the interface Identity that declares two getter methods getPublicKey and getPrivateKey. In a secure interaction, the concepts of process and service act on behalf of identities from the security domain. Process acts on behalf of the identity that started the process, and each service will hold the identity of its service provider. To connect the concepts of the two domains, the domain classes from the workflow domain Process and ServiceProxy implement the interface Identity that represent the corresponding identity.

However, inheriting the base language and only adding security primitives to it—this is naïve approach because the security related code written using the extended SECUREPROCESSDSL is scattered and tangled. As shown in Listing 7.1 in lines 8–10 and 18–20 (highlighted using "#"), the code for the security concern is not localized.

The remainder of this section discusses how tangling and scattering of DSL code can be addressed by supporting more invasive compositions of DSLs. The semantics of the composition of the process and security DSLs can be expressed by the following rule: "When the process DSL is composed with the security DSL, the interpretation of the former changes to ensure that external service calls are secured." Rather than having the programmer specify the interleaving of the abstractions from different DSLs as part of the program, the reflective embedding architecture provides dedicated composition operators that specify the interleaving at the level of DSL interpretations. These operators make use of aspect-oriented concepts, therefore they are called crosscutting combiners.

7.2. Embedding ProcessDSL

This section presents the implementation of an example embedding, in which programs suffer from scattering and tangling issues. While the discussion uses the domain of workflow languages in this section, these problems are not exclusive this domain, but the same problems also have been found in various domain, as mentioned in the previous section.

Consider Listing 7.2 that presents the implementation of PROCESSDSL in the façade class ProcessInterpreter. The ProcessInterpreter extends the Interpreter class and defines a getter method

```
 1    process(name:"SecureEasyCreditProcess") {
 2      def offers = [:];
 3
 4      task (name:"getOffers") {
 5        def services = registry.find ("Banking");
 6        services.each { bank ->
 7          def request = bank.createRateRequest(10000,"USD");
 8  #       def enc_request = encrypt(request,RSA,bank.publicKey);
 9  #       def enc_response = bank.call("getRate", [enc_request]);
10  #       def response = decrypt(enc_response,RSA,me.privateKey);
11          offers [bank] = response;
12        }
13      }
14
15      task (name:"selectOffer") {
16        def selectedBank = ... //get cheapest bank from offers
17        def request = selectedBank.createBorrowRequest(10000,"USD");
18  #     def enc_request = encrypt(request,RSA,selectedBank.publicKey);
19  #     def enc_response = selectedBank.call("borrow", [enc_request]);
20  #     def response = decrypt(enc_response,RSA,me.privateKey);
21        notify "Credit from ${selectedBank.name}: $response";
22      }
23    }
```

Figure 7.1.: Example program with tangled security concern

Registry getRegistry(){...} for the literal registry, the method void notify(String){...} for the corresponding domain operation notify, the method void process(HashMap, Closure){...} for the domain abstraction operator process, and void task(HashMap, Closure){...} for the operator task. To execute a PROCESSDSL program, as with every embedded DSL in REA, the program's closure must use ProcessInterpreter as its delegate.

7.3. Crosscutting Composition

Reconsider the problem of crosscutting composition in DSLs, which was discussed in Section 3.3.2.1 (p. 44), let's review the SECUREPROCESSDSL program in Listing 7.1. One can observe that the code for the security concern in lines 8–10 and 18–20 (highlighted using "#") is not localized. While SECUREPROCESSDSL was defined as an extension of PROCESSDSL, the same scattering problem also arises when using black-box composition to compose PROCESSDSL with an independently defined security DSL.

The problem is that the composition of the process and security DSLs is not black-box. The composition semantics can be expressed by the following rule: "When the process DSL is composed with the security DSL, the interpretation of the former changes to ensure that external service calls are secured." In other words, the interpretation of both languages crosscut each other.

To address the problem, in REA, the developer can use the CCCombiner that makes use aspect-oriented concepts to support more invasive compositions of EDSL interpretations. For crosscut-

```
public class ProcessInterpreter extends Interpreter {
  protected Process thisProcess = null;

  public Registry getRegistry() {
    return Registry.getInstance();
  }

  public void notify(String str) {
    String msg = "Notification (PID=$thisProcess.id)"+str;
    // ... send out an email to stakeholders
  }

  public void process(HashMap params, Closure processBody) {
    thisProcess = new Process(params);
    processBody.delegate = super.bodyDelegate;
    processBody.call();
    thisProcess.execute();
  }

  public void task(HashMap params, Closure taskBody) {
    taskBody.delegate = super.bodyDelegate;
    thisProcess.addTask(params, taskBody);
  }
}
```

Listing 7.2: Implementation of PROCESSDSL

ting composition, domain-specific join point models and domain-specific pointcut languages play an important role.

Domain-specific join points denote *points in the interpretation of an EDSL* exposed for invasive composition; the set of these join points constitutes a *domain-specific join point model* (DS-JPM). Domain-specific pointcuts select points in the interpretation of the EDSLs (i.e., points in DS-JPM) where invasive composition should take place; they are expressions of a domain-specific pointcut language (DS-PCL). Finally, advice operators before, around, after, and proceed are used to determine the order in which the invasive composition of the EDSLs should happen.

For illustration, consider a DS-JPM for PROCESSDSL consisting of four kinds of join points: (1) *process execution join points* occur when a process is executed; their context includes the process name and the task list. (2) *task execution join points* occur when a task is executed; their context includes the task name and its activities. (3) *service selection join points* occur when the registry is consulted to select services; their context includes the selection pattern and the set of selected services. (4) *service call join points* occur when a remote call to a Web service is done as part of process; their context includes the service name and the call's SOAP document. The DS-PCL for PROCESSDSL provides – among others – the pointcut designators: process_execution, task_execution, service_call, and service_selection.

Listing 7.3 shows the SecureEasyCreditProcess program (cf. Listing 7.1) rewritten using crosscutting composition of PROCESSDSL and security DSL. The program as a whole implements the same semantics as before, but this time the security concern (lines 3–9) and the workflow specification (line 12) are no longer tangled. Instead, AO abstractions are used to compose the two DSL programs. The aspect in lines 3–9 defines how PROCESSDSL programs are made secure. Basically, calls to external services are encrypted before being sent and are decrypted after receiving the response. The pointcut (line 4) matches against corresponding domain-specific join points and is written in PROCESSDSL's domain-specific pointcut language (DS-PCL) and selects all *external* service_calls in PROCESSDSL programs. The security related functionality is expressed using the security DSL.

```
 1  // Definition of EDSL programs (any order)
 2  Closure ca = {
 3    aspect(name:"Confidentiality") {
 4      around(service_call(".*") & if_pcd { external }) {
 5        request = encrypt(request,RSA,thisJoinPoint.service.publicKey);
 6        response = proceed();
 7        return decrypt(response,RSA,thisJoinPoint.process.privateKey);
 8      }
 9    }
10  }
11  Closure ecp = {
12    process(name:"EasyCreditProcess") { /* code from Listing 7.1 */ }
13  }
14
15  // Runtime setup
16  def pI = new ProcessInterpreter();
17  def secI = new SecurityInterpreter();
18  def ppI = new ProcessPointcutInterpreter();
19  def ccc = new CCCombiner([pI, secI, ppI],[ca,ecp]); | cf. Figure 7.3
20
21  // Execution of EDSL programs
22  ca(); // deploying the aspect | cf. Figure 7.3
23  ecp(); // executing the process with aspects | cf. Figure 7.5/7.6
```

Listing 7.3: Using crosscutting composition

In lines 16 to 18, the EDSL interpreters – including the interpreter for process pointcuts – are instantiated. After that, all interpreters and the DSAL programs are passed to the newly created CCCombiner. As in case of the BlackBoxCombiner, the CCCombiner also sets itself as the bodyDelegate of the interpreters (pI,secI,ppI). Further, the CCCombiner sets itself as the delegate of the closures (ca,ecp) wrapping the DSL programs. Next, the closure ca is called which deploys the aspect; i.e., external service calls are now intercepted and the around advice is executed. This finishes the setup of the runtime environment. Finally, the ecp closure is called to executed the (secured) workflow.

In general, the order in which EDSL programs are executed is important. For example, if the ecp closure is called before the ca closure, then no en-/decryption takes place. Though the aspect's closure is registered with the CCCombiner, the aspect is not yet deployed. Hence, the

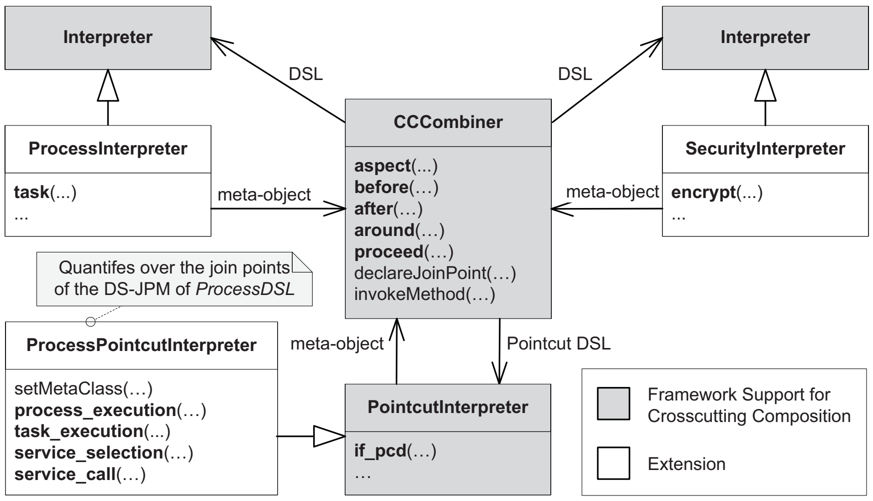


Figure 7.2.: Crosscutting composition of EDSLs

program first has to call the closures of the aspects, before the program executes the "main" program – unless developer wants to deploy aspects dynamically.

7.4. Homogeneous Embedding of Aspect-Oriented Concepts

The reflective embedding architecture embeds AO abstractions in the same way as any DSL abstractions. As all concepts are homogeneously embedded, semantic interactions between embedded DSL compositions can be defined, whereby reusing the existing embedded DSL components. For this, developers use POPART/REA's CCCombiner that implements crosscutting composition and which has been presented in Section 4.2.3.3 (p. 109). For a crosscutting composition of EDSL, instances of this class are the *object delegates* of all closures that use abstractions from invasively composed EDSLs.

The CCCombiner (center of Figure 7.2) refers to interpreters of EDSLs to be invasively composed (ProcessInterpreter and SecurityInterpreter in Figure 7.2), and to a DS-PCL interpreter[1] (ProcessPointcutInterpreter in Figure 7.2). These interpreters are passed to it at instantiation time (cf. line 19 in Listing 7.3).

PointcutInterpreter implements a general-purpose pointcut language supporting composition operators "&" and "|"; and higher-order pointcuts if_pcd, not, cflow, similar to AspectJ [Asp] pointcut designators. DS-PCLs are implemented in subclasses of PointcutInterpreter. For instance, ProcessPointcutInterpreter referred to by the CCCombiner in Figure 7.2, an excerpt of which is shown in Listing 7.4, extends PointcutInterpreter with methods for PROCESSDSL-specific pointcut expressions (lines 24 to 27 in Listing 7.4).

[1]The combiner in Figure 7.2 references a single pointcut language interpreter, but in general it may reference a BlackBoxCombiner that composes more than one pointcut language.

```
class ProcessPointcutInterpreter extends PointcutInterpreter {

  void setMetaClass(CCCombiner ccc) {
    super.setMetaClass(ccc);
    ccc.declareJoinPoint("ServiceProxy","call ",
      { appContext, jpBody ->
          HashMap jpContext = [:];
          jpContext["service"] = appContext.receiver;
          jpContext["operation"] = appContext.args[0];
          jpContext["request"] = appContext.args[1];
          jpContext["external"] = appContext.receiver.isExternal ();
          jpContext["process"] = ccc.thisProcess; // pret.−call
          jpContext["proceedClosure"] = jpBody;
          def jp = new ServiceCallJP(jpContext);
          jpContext["thisJoinPoint"] = jp;
          return jp;
      }
    );
    ccc.declareJoinPoint("Process","execute") {...}
    ccc.declareJoinPoint("Task","execute") {...}
    ccc.declareJoinPoint("Registry"," find ") {...}
  }

  Pointcut process_execution(String nameRegExp) {...}
  Pointcut task_execution(String nameRegExp) {...}
  Pointcut service_selection(String category) {...}
  Pointcut service_call(String operationRegExp) {...}
}
```

Listing 7.4: Excerpt of ProcessPointcutInterpreter

Behind a PointcutInterpreter there are classes that model join points and pointcuts. Join points are modeled as objects that keep information about the type and context of an execution point. For example, the JPM for PROCESSDSL is modeled by classes ProcessExecJP, TaskExecJP, ServiceSelectJP, and ServiceCallJP. Pointcuts are objects that implement a filter method, which given a join point object as a parameter returns true, respectively false, when the join point matches, respectively does not match the pointcut. The result of evaluating the pointcut expression “service_call(".*") & if_pcd{external}” by a ProcessPointcutInterpreter is a graph of Pointcut objects.

As already mentioned, a CCCombiner is attached as the delegate to closures enclosing code that use invasively composed EDSLs. For instance, ccc is set to be the delegate of all closures in Listing 7.3 (line 2, 11). A CCCombiner-delegate dispatches all process-specific and security-specific abstractions for interpretation to the respective interpreters. Yet, the latter pass the control back to the CCCombiner every time the body of a domain-specific abstraction operator is entered, since the CCCombiner is uniformly set to be the delegate for all such bodies. Pointcut expressions are dispatched to the PointcutInterpreter, which returns pointcut object graphs. Finally, keywords such as aspect, around, proceed, etc. are interpreted by the CCCombiner itself.

Aspects are modeled as containers of pointcut-to-advice pairs; advice bodies are modeled by closures.

```
class CCombiner extends Interpreter {

  List<Aspect> aspects = [];
  ...
  void aspect(HashMap params, Closure aspectBody) {...}
  void before(Pointcut pc, Closure advice)  {...}
  void around(Pointcut pc, Closure advice)  {...}
  void after(Pointcut  pc,  Closure advice)  {...}
  Object proceed()  {...}

  Map<Pattern,Closure> patternToReifier =  [:];

  void declareJoinPoint(String  className,
                         String  methodName, Closure jpReifier) {
    //  Associate className and methodName
    //  pattern with jpReifier.
    //  Association is stored in patternToReifier.
  }

  Object invokeMethod(sender, receiver, mName, args, ...) {
    //  Basically, the join point is reified and all
    //  pointcuts are evaluated. If a pointcut matches,
    //  then the associated advice is executed.
    //  (cf. Figure 7.6 for details.)
  }
}
```

Listing 7.5: The CCombiner class

```
public class  SomeInterpreter extends Interpreter  {
  ...
  public void setMetaClass(MetaClass mc){
    super.setMetaClass(mc);
    /*   Concrete interpreters are responsible for
         setting the meta-object of all their domain classes:
         for(domainClass : Class) {
             domainClass.setMetaClass(mc);
         }
    */
  }
}
```

Listing 7.6: Implementation of setMetaClass

Now, the question remains to be answered how join point objects come to being. Reflective embedding plays a key role in reifying DS-JPMs. To enable the CCombiner to reify join points it is no longer sufficient to just set it as the object delegate as just outlined. Additionally, it is imperative that the CCCombiner is also set as the meta-object of all domain classes of all composed interpreters. To set itself as the meta-object of all domain classes, the CCCombiner calls the method setMetaClass on the facade class of each EDSL; the class which implements the Interpreter interface. This method has then the responsibility to call setMetaClass on all domain classes (cf. Listing 7.6).

After that the CCCombiner is the meta-object of the EDSL interpreters it composes and can intercepts every method call to them. This enables the CCCombiner to reify join points, which are made known to it by calling declareJoinPoint. This method takes three parameters and returns a join point object. The first two parameters define a pattern for the signature of an interpreter method whose calls are join points. The third parameter is a so-called *reification closure*[2] – a closure responsible for reifying the corresponding join point (creating the corresponding join point object and filling in its context). The association of join point patterns to their reification closure is stored in a map (Listing 7.5, line 11). Since a pointcut language interpreter is implemented w.r.t. a specific DS-JPM, it is its responsibility to call declareJoinPoint with appropriate parameters. CCCombiner is oblivious of any DS-JPM.

For illustration, consider how a DS-JPM is defined for PROCESSDSL. The inherited method setMetaClass is overridden in ProcessPointcutInterpreter in Listing 7.4 (lines 3 to 22), to make the process-specific JPM known to ccc (lines 5–21). For instance, the code in lines 5–17 declares an invocation of ServiceProxy.call during a process interpretation as a join point and shows its reification closure[3]. This closure will be executed whenever a join point of this type occurs (i.e., the corresponding call is intercepted by ccc). In lines 7–13, the closure stores the application context, which it receives as its first parameter appContext, into the jpContext HashMap[4]. The "proceedClosure" entry in that map stores the second parameter of the reification closure (jpBody), itself a closure encapsulating the intercepted method call. Finally, the reification closure creates a join point instance of type ServiceCallJP (line 14) and returns it.

So far, the section presented how to declare a DS-JPM. In the following, to execute programs under impact crosscutting concern, in REA, the embedding of AO concepts creates join points at runtime. When the interpreters to compose execute under the control of a CCCombiner, whenever a point in their execution is reached it is dispatched to the enclosing CCCombiner by calling invokeMethod(sender, receiver, methodName, args, ...) on the latter. As a result, the matching patterns are looked up, the reifier closure is retrieved and called. Next, the reified join point is matched against all pointcuts of all defined aspects and advice is eventually executed, hence dynamically composing aspects. We will elaborate on this composition in the next sub-section.

[2]For brevity in source code, the subsequent text uses the acronym *reifier*.

[3]Last parameter to the declareJoinPoint call in line 5 is the closure that follows the parenthesis in lines 6–17).

[4]Note that, the thisProcess property (line 12) is a *pretended property* of the CCCombiner that is actually provided by ProcessInterpreter, and the access to this property is delegated to the latter via the MOP.

7.4.1. Crosscutting Composition at Work

To illustrate crosscutting composition support Figure 7.3 "emulates" the execution of the code of Listing 7.3, starting with line 16.

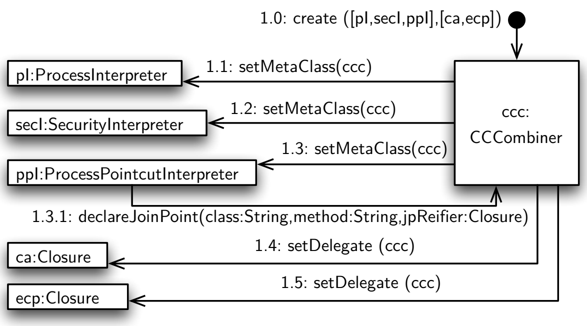


Figure 7.3.: Setup of the CCCombiner

To set up the CCCombiner, the interpreters for the PROCESSDSL, the security EDSL, and the DS-PCL are passed to the CCCombiner when it is created. Additionally, the EDSL programs' closures (ca,ecp) are also passed to the CCCombiner (line 19 in Listing 7.3, step 1.0 in Figure 7.3). As the result, ccc becomes the meta-object of all three interpreters (steps 1.1, 1.2, and 1.3 in Figure 7.3). As a sub-step of setting ccc to be the meta-object of the process pointcut interpreter (ppI), the corresponding DS-JPM is declared by calls to ccc.declareJoinPoint (line 5 in Listing 7.4; step 1.3.1 in Figure 7.3). Moreover, ccc is set as the delegate of the closures ca and ecp (steps 1.4 and 1.5 in Figure 7.3)[5].

[5]After creation of the CCCombiner, new closures can be added using its addClosure method.

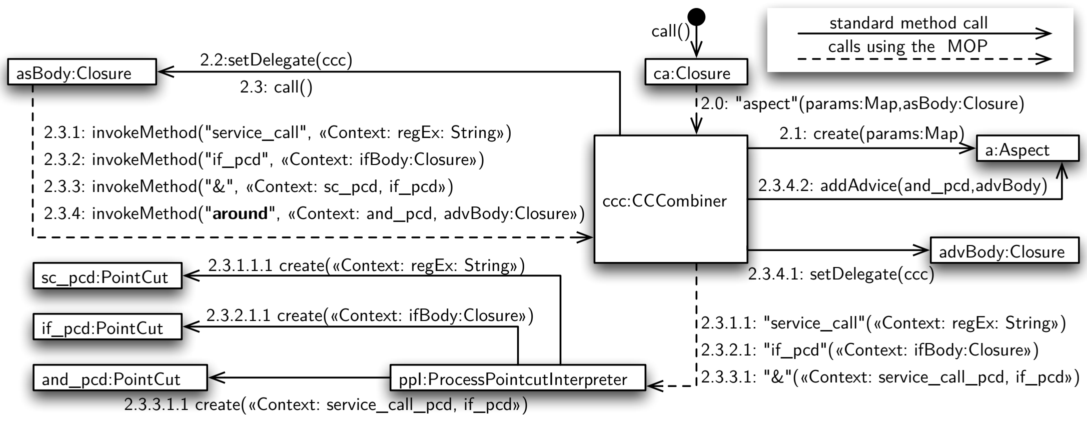

standard method call
calls using the MOP
call()
ca:Closure
asBody:Closure
2.2:setDelegate(ccc)
2.3: call()
2.0: "aspect"(params:Map,asBody:Closure)
2.1: create(params:Map)
a:Aspect
2.3.1: invokeMethod("service_call", «Context: regEx: String»)
2.3.2: invokeMethod("if_pcd", «Context: ifBody:Closure»)
2.3.3: invokeMethod("&", «Context: sc_pcd, if_pcd»)
2.3.4: invokeMethod("around", «Context: and_pcd, advBody:Closure»)
ccc:CCCombiner
2.3.4.2: addAdvice(and_pcd,advBody)
advBody:Closure
2.3.4.1: setDelegate(ccc)
sc_pcd:PointCut
2.3.1.1.1 create(«Context: regEx: String»)
if_pcd:PointCut
2.3.2.1.1 create(«Context: ifBody:Closure»)
and_pcd:PointCut
ppI:ProcessPointcutInterpreter
2.3.3.1.1 create(«Context: service_call_pcd, if_pcd»)
2.3.1.1: "service_call"(«Context: regEx: String»)
2.3.2.1: "if_pcd"(«Context: ifBody:Closure»)
2.3.3.1: "&"(«Context: service_call_pcd, if_pcd»)

Figure 7.4.: Evaluation of the ca closure and aspect registration

Next, in line 22 Listing 7.3, the aspect is initialized by calling the closure containing its definition. When encountering the keyword aspect in line 3 of Listing 7.3, the MOP maps it to a call to the enclosing delegate, ccc. This triggers the creation of aspect a (step 2.1 in Figure 7.4) and the evaluation of its body – a Groovy closure whose execution is triggered in step 2.3 after its delegate is previously set to ccc in step 2.2.

In the process of performing step 2.3, pointcut expressions encountered during the evaluation of the pointcut in line 4 of Listing 7.3 are mapped to corresponding denotations (Pointcut objects) in steps 2.3.1, 2.3.2, and 2.3.3 as shown in Figure 7.4. This happens in two stages. First, for each domain-specific pointcut expression in the program code the Groovy MOP triggers invokeMethod on ccc, the delegate of the aspect closure (steps 2.3.1, 2.3.2, 2.3.3). The latter dispatches the pointcut expressions at hand to corresponding methods of the pointcut language interpreter (steps 2.3.1.1, 2.3.2.1, 2.3.3.1). At the end of step 2.3.3.1.1, a graph of pointcut objects, and_pcd, is created; together with the advice body created as the result of steps 2.3.4 and 2.3.4.1[6]. In Figure 7.3 it is added to the list of pointcut-advice pairs of the aspect a. This completes the set up of the interpretation infrastructure for the Confidentiality aspect.

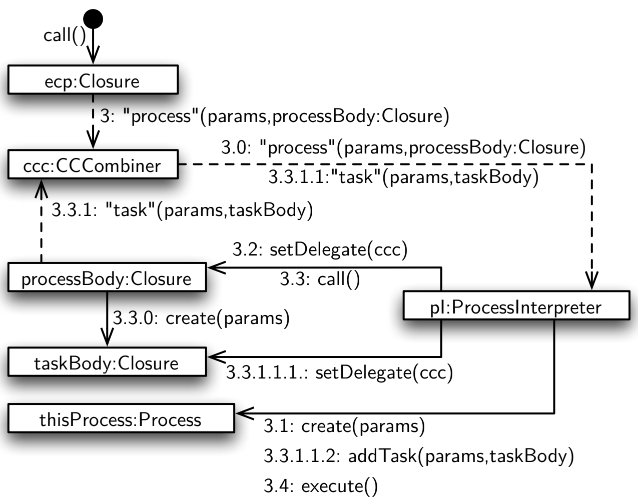

call()
ecp:Closure
3: "process"(params,processBody:Closure)
ccc:CCCombiner
3.0: "process"(params,processBody:Closure)
3.3.1.1:"task"(params,taskBody)
3.3.1: "task"(params,taskBody)
3.2: setDelegate(ccc)
processBody:Closure
3.3: call()
pI:ProcessInterpreter
3.3.0: create(params)
taskBody:Closure
3.3.1.1.1.: setDelegate(ccc)
thisProcess:Process
3.1: create(params)
3.3.1.1.2: addTask(params,taskBody)
3.4: execute()

Figure 7.5.: Evaluation of the ecp closure

Setting up the runtime for interpreting the process definition part of the program is similar. This is triggered when the keyword process is encountered in line 12 of Listing 7.3. This set up is represented by the steps 3.0 and 3.1 that are elaborated in Figure 7.5. When the process is initialized denotations for all process-specific abstractions in program code, such as process (Figure 7.5, steps 3, 3.2, 3.3) and task (step 3.3.0, steps 3.3.1, 3.3.1.1, 3.3.1.1.1, 3.3.1.1.2), are created and related in an object graph. Again, ccc is set to be the delegate of both the process

[6]These steps are triggered when interpreting the keyword around.

(step 3.2) and task bodies (step 3.3.1.1.1). This is important to ensure that the execution of the process semantics is intercepted by ccc such that join points are reified as they occur and matching aspects are composed at runtime.

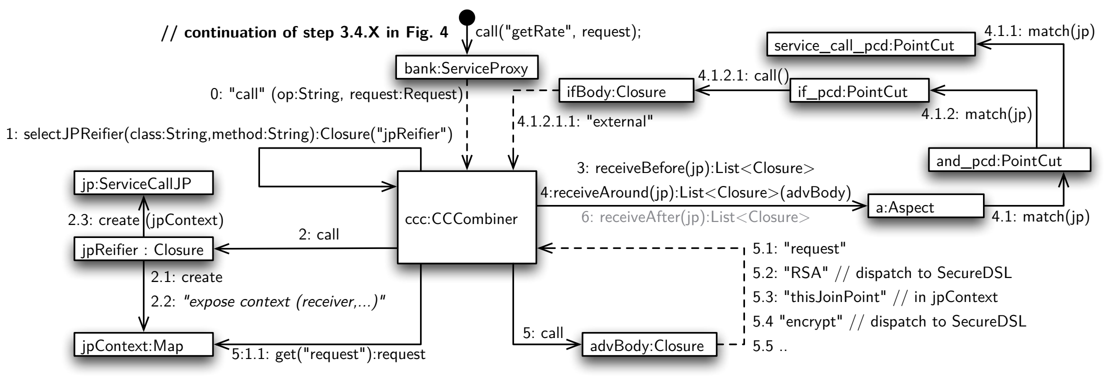

// continuation of step 3.4.X in Fig. 4
call("getRate", request);
bank:ServiceProxy
0: "call" (op:String, request:Request)
1: selectJPReifier(class:String,method:String):Closure("jpReifier")
jp:ServiceCallJP
2.3: create (jpContext)
jpReifier : Closure
2: call
2.1: create
2.2: "expose context (receiver,...)"
jpContext:Map
5:1.1: get("request"):request
ccc:CCCombiner
ifBody:Closure
4.1.2.1.1: "external"
4.1.2.1: call()
service_call_pcd:PointCut
4.1.1: match(jp)
if_pcd:PointCut
4.1.2: match(jp)
and_pcd:PointCut
3: receiveBefore(jp):List<Closure>
4:receiveAround(jp):List<Closure>(advBody)
6: receiveAfter(jp):List<Closure>
a:Aspect
4.1: match(jp)
5.1: "request"
5.2: "RSA" // dispatch to SecureDSL
5.3: "thisJoinPoint" // in jpContext
5.4 "encrypt" // dispatch to SecureDSL
5.5 ..
5: call
advBody:Closure

Figure 7.6.: Dynamic crosscutting composition of EDSL semantics

Runtime crosscutting composition of EDSL semantics is illustrated in Figure 7.6. The figure shows a snapshot of executing the EasyCreditProcess taken at the point of executing a getRate call on the bank object, which is of the type ServiceProxy. The call keyword is delegated for interpretation to the process interpreter and hence intercepted by ccc, the meta-object of the interpreter. This is where the interactions shown in Figure 7.6 start.

In step 1 (Figure 7.6), ccc checks whether the intercepted call to method ServiceProxy.call is a declared join point and retrieves the corresponding reification closure which it subsequently calls (step 2). Consequently, a join point object, jp, is created and its context is set in steps 2.1, 2.2, and 2.3. In steps 3, 4, and 6 all defined aspects, in this case aspect a, are “asked” whether they match the join point at hand with before, around, and after semantics respectively. In the given scenario - interpretation of bank.call - the current join point is an external service call that is matched by the pointcut of confidentiality aspect’s around advice (line 4 in Listing 7.3). Matching the pointcut is depicted by step 4.1 and its subordinate steps, the pointcut objects in the pointcut-advice associations of a are matched against the intercepted join point. When a pointcut matches the body of the corresponding advice (advBody) is returned and called subsequently (step 5)[7]. In the process of executing the advice body closure, abstractions from both EDSLs are dispatched for evaluation to ccc which delegates to the appropriate interpreters.

7.5. Summary

This chapter presented how the reflective embedding architecture supports crosscutting composition of DSLs. The approach was to embed domain-specific aspect language on top of meta-object protocols and aspect-oriented concepts as the underlying technologies. Crosscutting com-

[7]In general, it is possible that multiple pointcuts match and, hence, a list of advice bodies, which are closures, is returned.

position enables better modularity of programs that use several EDSLs whose domain semantics are crosscutting.

To recapitulate, the key mechanisms that facilitate crosscutting compositions are:

Crosscutting combiner: The reflective architecture can build abstraction on top of the meta-object protocol to realize *aspects*. Compared to using a MOP for adaptation, the aspect-oriented abstractions provide additional guarantees for the composition of embedded objects.

Domain-specific join points: The embedding of aspect-oriented concepts provides special means for domain-specific join points, as these were requested by [RMH+06, CNF+08, RMWG09]. Domain-specific join point models and domain-specific pointcut languages define an abstraction layer over the interpretation of embedded DSLs to compose.

With respect to crosscutting composition, there are several issues that remains to be be shown:

- This section has presented a crosscutting composition of two DSL, but how can multiple languages be integrated? Such as, a DSL and a GPL, or a more than two DSL. Therefore, Section 10.3.2.2 (p. 228) presents how to use this chapter's techniques to compose several language with different domains-specific join point models.
- When multiple languages are integrated, there can be composition conflicts. Therefore, Section 10.3.2.3 (p. 235) present a general mechanism to detect and resolve static composition conflicts between several aspect-oriented languages.
- Performance measurements are discussed in Section 11.3.2 (p. 276).

Note that this chapter has summarized how to use POPART for crosscutting composition and its most important implementation details. However, POPART support not only crosscutting composition of domain-specific languages, its open implementation of the POPART aspect language can be used as a language workbench for general-purpose aspect languages as well. Due to space restrictions, the description of the AO language workbench is out of the scope of this thesis. Therefore, the more interested readers are referred to [DMB09, DEM10].

8. Incremental Concrete Syntax for Embedded Languages

The previous chapters have presented various applications of reflective embedding for open DSL, aspect languages, and compositions thereof. However, since in those language embeddings, program expressions have an abstract syntax encoding in host language, they all suffer from the missing support for concrete syntax. Therefore, to address the missing support for concrete syntax, this chapter presents an extension to the reflective embedding architecture that enables support for concrete syntax in EDSLs implemented on top of the architecture.

The problem of missing support for concrete syntax is common in homogeneous embedding approaches [Hud96, Fow05, Gar08, HORM08]. While on the one hand, embedded DSLs enable an easy integration into their host language, there is the downside that each homogeneous EDSL commits to one particular host language. Therefore, it is not possible to integrate embedded DSL programs into other languages than the host language. Thus, when the same DSL is desired in several GPLs, language developers have to repetitively embed the DSL as a homogeneous EDSL in each GPL, resulting in duplicated investment costs. These issues are the major obstacles for the adoption of embedded DSL approaches [MHS05, Tra08].

At the cost of dropping homogeneity, heterogeneous embedding approaches address the problems of how to enable the DSL to use the domain's established notation / syntax and how to integrate DSL code with existing code and enable interactions. Concrete syntax is enabled by a specialized tool—in general some kind of *pre-processor*. It has to first rewrites code in concrete DSL syntax to an executable code in some GPL. After that, the compiler of the GPL compiles the code as usual. A well-known example of a *heterogeneous embedded DSL* is *SQLJ* [ME00] which enables the embedding of SQL code into Java code. While the example of embedded SQL uses a special pre-processor, there are generic heterogeneous embedding approaches that help language developers to implement pre-processors and code generators [Kam98, EFDM03, BV04, SCK04, COST04].

When we need both concrete syntax and integrability into different languages, there are tradeoffs of homogeneous and heterogeneous approaches. The main advantage of heterogeneous embedded DSLs is that they support the domain's established syntax (called *concrete syntax* in the following). This is particularly important as comparative studies [KLP$^+$08] have revealed that missing support for concrete syntax significantly lowers the end user's productivity in writing DSL code. The main obstacles of heterogeneous embedded DSLs are threefold. First, to support another host-language, the pre-processor typically has to be re-implemented from scratch. Second, pre-processors are known to be hard to compose. This makes it practically impossible to use multiple embedded DSLs in parallel. Third, the implementation of a DSL requires a formalization of its complete syntax, which is a significant effort. The problems of heterogeneous embedded DSLs are partially solved by homogeneous embedded DSLs since they are developed in the host language. In the latter case the EDSL's implementation is just a library. Hence, a specification of the syntax and corresponding tools is not required. This helps to significantly reduce the necessary effort. Additionally, using multiple DSLs in the same program is as simple as using multiple libraries. But, by having to reuse the syntax of the host language, the DSL's concrete syntax is typically not close to the concrete syntax of the domain [BV04, Tra08]. This severely hampers the comprehensibility of such DSLs as previously discussed.

To provide concrete syntax for DSL programs—without suffering from some of the main problems of traditional heterogeneous embedded DSLs—several approaches were proposed that use *meta-programming* [BV04, Tra08, KM09, RGN10]. These approaches generally suffer from the following issues. First, language developers still have to provide the complete formal syntax for each embedded DSL [BV04, Tra08, RGN10]. Second, if more than one host language is supported [BV04], the grammar of the complete syntax must be available. This specification has to be in the format used by the approach, which is generally not the case. Third, most approaches [Tra08, KM09, RGN10] support only one particular host language, which hinders the reusability of EDSL implementations. Overall, the effort necessary to develop an EDSL using these approaches is significantly higher than the effort to develop a homogeneous embedded DSL.

To address these issues, this chapter presents a generic pre-processor approach to implement embedded DSLs (EDSLs) that combine the advantages of heterogeneous embedded DSLs with homogeneous embedded DSLs and which does avoid the respective disadvantages. I.e., the pre-processor approach facilitates the development of EDSLs that provide concrete syntax, but avoids that the developer always has to specify the complete grammar of the EDSL and / or the host language. Using the pre-processor approach the effort to implement an EDSL with concrete syntax is practically identical to the effort necessary to develop a homogeneous embedded DSL for the respective domain. Furthermore, the pre-processor approach is *generic* in the sense that it enables concrete syntax for any reflectively embedded DSL, it is host language independent, and it can be used to embed DSLs in various host languages.

The fundamental idea is that a language developer starts with the implementation of a homogeneous embedded DSL and then—step by step—adds meta-information regarding the concrete syntax of the domain to the DSL implementation. By using so-called *island grammars* the developer only needs to specify those parts of the grammar of the host language that are relevant w.r.t. the EDSL implementation. Furthermore, only those parts of the EDSL's syntax need to be specified that are not compatible with the syntax of the host language.

Island grammars [Moo01] support parsing programs even if only a subset of the productions of the complete formal syntax of the language is specified. In general, an island grammar only completely formalizes the language constructs of interest (the islands). To match the remainder (the water), liberal productions are specified. These liberal productions, called *water rules*, recognize characters from expression types that are not recognized by the islands. When composing island grammars of multiple languages, programs can be parsed that mix expressions from those languages in unexpected ways [SCD03]. Traditionally, island grammars are used in reverse-engineering, because for legacy languages often no formal grammar is available. In contrast, the motivation to use island grammars in this thesis is to avoid that language developers always have to formalize the complete grammar of the embedded DSL and the host languages. Compared to the traditional use of island grammars in the context of reverse-engineering—where the goal is to avoid the formalization of an out-dated language—we employ them to reduce the development costs for a new language.

Parts of this chapter have been published in the following chapters:

- Tom Dinkelaker, Michael Eichberg, and Mira Mezini. **Incremental Concrete Syntax for Embedded Languages.** In *Proceedings of the 26th ACM Symposium on Applied Computing—Technical Track on Programming Languages (PL at SAC)*, 2011 (to appear).

The remainder of this chapter is organized as follows. Section 8.1 discusses the concept and Section 8.2 the implementation.

8.1. Concrete Syntax for Homogeneous Embedded DSLs

This section first discusses an example implementation of a homogeneous embedded DSL. Then, it will elaborate how to provide concrete syntax. After that, the inner workings of the approach are presented before the section concludes the presentation of the approach by discussing how to support further host languages.

The running example to demonstrate the approach is the embedding of SQL in the Groovy GPL. The goal is to enable writing programs like the one shown in Listing 8.1. The select statement (line 2 in Listing 8.1) is defined using SQL while the rest of the program is written in Groovy; both languages are seamlessly integrated. The program uses SQL's concrete syntax (cf. Listing 8.2) and prints out a list of peoples' names and ages from the result of an embedded SQL statement.

```
1 println "Registered people:"
2 List<Map> result = SELECT Name,Age FROM People
3 result .each{ Map row −> println "Name: ${row.Name} Age:${row.Age}"; }
```

Listing 8.1: An embedded SQL program in Groovy

8.1.1. Defining a Homogeneous EDSL

Language Interface: As in other approaches, a new homogeneous EDSL is implemented by a set of classes in the host language. These classes have to follow a small set of conventions to enable the integration with the approach.

$Statement \to Query$
$Query \to$ "SELECT" ␣ $Columns$ ␣ "FROM" ␣ $Tables$
$Columns \to Id$ ␣ "," ␣ $Columns$
$Columns \to Id$
$Tables \to Id$ ␣ "," ␣ $Tables$
$Tables \to Id$

Listing 8.2: The ideal grammar of TINYSQL

To embed a DSL the following developer implements an EDSL as usual in REA. To allow finding the interface in the classpath, the only additionally task is that the *language interface* must extend the DSL marker interface.

Listing 8.3 shows the interface for the homogeneous EDSL TINYSQL. The interface only defines a single method to execute select statements.

```
interface ITinySQL extends DSL {
  List<Map> selectFrom(String[] columns, String[] tables);
}
```

Listing 8.3: The language interface of TINYSQL with compromised syntax

At this step, the developer cannot take the concrete syntax into account. Given the constraints of the host language, the syntax of the homogeneous EDSL cannot be optimal in most cases. In the following, the term *contaminated syntax* refers to the syntax that is provided by the homogeneous EDSL.

Defining the Semantics: To implement the execution semantics of an EDSL, a language developer implements the EDSL's language interface. Since the focus of this chapter is to discuss support for concrete syntax, it only sketches the implementation of the execution semantics for EDSLs.

Listing 8.4 presents an excerpt of the class TinySQLInterpreter that implements the semantics of TINYSQL. An EDSL class extends the Interpreter class and implements the EDSL's language interface. The Interpreter class primarily defines the method to evaluate EDSL programs. Additionally, methods are defined to extract productions from EDSLs and to compose the EDSL with other EDSLs. For TINYSQL, the class implements ITinySQL. It has an attribute to hold an SQL connection and implements the selectForm method. The method first checks that all arguments are correct then it constructs an SQL statement and uses *Groovy SQL*[1] to execute the statement using a JDBC connection. The method rows in line 7 finally returns a result set as a List that contains Map objects as rows.

8.1.2. Enabling Concrete Syntax for EDSLs

This section elaborates on how language developers can add support for concrete syntax for their homogeneous embedded DSLs.

[1]Groovy SQL: `http://docs.codehaus.org/display/GROOVY/Tutorial+6+-+Groovy+SQL`.

```
class TinySQLInterpreter extends Interpreter implements ITinySQL {
  private groovy.sql.Sql connection = newInstance("jdbc:...");
   ...
   List<Map> selectForm(String[] columns, String[] tables) {
     ... //check arguments to be correct
     String query = "SELECT " + Util.implode(",", columns) + " FROM " + Util.implode(",", tables);
     return sqlConnection.rows(query);
   }
}
```

Listing 8.4: The implementation of TINYSQL (semantics)

Incremental Definition of Concrete Syntax: To incrementally define concrete syntax for embedded DSLs, the language developer adds annotations to their EDSL implementation classes. The annotation @DSLMethod is used to define the concrete syntax for every method that defines an expression type. The annotation has the element production that defines the right-hand side of the production rule for this method. The production rule is obtained as a string and defines the categories in the production rule.

In the string, the categories are delimited by the *whitespace escape character*, which is the underscore. Whereby, one underscore denotes a required whitespace ($\overset{!}{\textvisiblespace}$), and two underscores denote an optional whitespace ($\overset{?}{\textvisiblespace}$). Terminal categories simply use their ASCII representation. Non-terminal categories are defined by argument placeholders, which are prefixed by the *parameter escape character* (by default, "p") followed by the index of the argument in which the non-terminal will be passed to the method.

```
1 interface ITinySQL extends DSL {
2   @DSLMethod(production="SELECT__p0__FROM__p1")
3   List<Map> selectFrom(String[] columns, String[] tables);
4 }
```

Listing 8.5: The language interface of TINYSQL with concrete syntax

For example, Figure 8.5 shows the definition of the TINYSQL interface that defines concrete syntax. The annotation in line 2 defines a production ($Statement \rightarrow$ "SELECT" $\overset{?}{\textvisiblespace}$ $String[]$ $\overset{?}{\textvisiblespace}$ "FROM" $\overset{?}{\textvisiblespace}$ $String[]$) with four relevant categories on the right-hand side: "SELECT", $String[]$”, "FROM", and $String[]$. By convention, the left-hand side category of the production is $Statement$ which is expected to be one of the host language's categories and which identifies the places where the DSL's expression can be embedded (cf. line 2 and 3 in Listing 8.8).

Quotation: Given that the approach builds upon homogeneous EDSLs, the user generally does not have to quote DSL expressions. However, there are cases where a language developer may require controlling the integration between between an EDSL and a host language.

One common reason is to precisely define the scope of an EDSL expression. For example, in Listing 8.6, in line 2, the EDSL expression is quoted using #sql{...}[2].

```
1 println "Registered people:"
2 Map[] result = #sql{SELECT Name,Age FROM People}
3 result .each{ Map row −> println "Name: ${row.Name} Age:${row.Age}"; }
```

Listing 8.6: An embedded SQL program in Groovy

In TIGERSEYE, quotations are only necessary in case of EDSL keywords that could be misinterpret or in case a language developer wants to restrict the scope of identifiers. Quotations are directly supported and do not require special support. It is sufficient to define another @DSLMethod annotation that defines a production regarding the quotation (cf. Listing 8.7 line 3).

```
1 interface ITinySQLWithQuotation extends ITinySQL {
2
3    @DSLMethod(production="#sql{__p0__}")
4    Map[] quoteSqlStmt(List<Map> map);
5 }
```

Listing 8.7: The interface of TINYSQL with quotation

8.1.3. Compiling DSL Programs

Figure 8.1 gives an overview of the compilation process that transforms programs in concrete syntax into host syntax. This process is explained in the subsequent paragraphs.

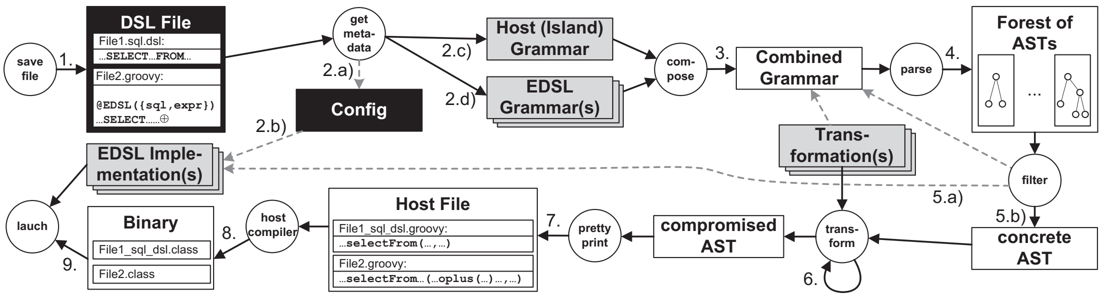


Figure 8.1.: Overview of the compilation process

Starting the Compilation Process: The compilation process starts (Step 1), when the DSL end user saves a file in the IDE that contains expressions in concrete DSL syntax. To specify the EDSLs used in a source file, there are two options for end users. First, users can use a special

[2]#sql{...} is not a valid statement in the host language. When a language end user uses a quotation, the end user explicitly escapes from the host language and opens up a scope in which SQL expressions can be used.

file extension, where the last suffix of the file is dsl. E.g., the file extension of file1.sql.dsl uses sql to specify that the TinySQL DSL is used. Second, users can add an annotation @EDSL to their code to specify the languages used in the respective code. The elements of the @EDSL annotation determine the concrete EDSLs. E.g., the annotation @EDSL(sql,expr) specifies that the sql and expr DSLs are used.

Extracting Grammars using Reflection: For every EDSL, the pre-processor looks up the EDSL interface (Step 2 a) from the configuration. Next (Step 2 b), the pre-processor uses *introspection* [Mae87] to look up all declared methods in the EDSL's interface. From the interface annotation, the pre-processor looks up the host language of the EDSL and loads the host grammar (Step 2 c). From meta-data provided in the method signatures and their annotations, the pre-processor extracts the EDSL grammar (Step 2 d).

Creating a Combined Grammar: Next (Step 3), the pre-processor tries to create the composite grammar by combining the host grammar with the grammars of all specified EDSLs. Before combining the grammars, the grammars are checked for possible conflicts. Currently, the concept makes use of *union grammars* [Cor06] to build the union of all productions defined for the host language and all EDSLs. For example, the combined grammar for parsing TinySQL embedded in Groovy contains all productions defined in Listing 8.2 and Listing 8.8. As long as two EDSLs do not define two productions that have exactly the same terminals and non-terminals the pre-processor can distinguish them and will call the correct method of the homogeneous embedded DSL.

Parsing into Forest of ASTs: When parsing (Step 4) the source code, the Earley parser creates a forest of all possible ASTs of the DSL program using the combined grammar. In the forest, every AST node is associated with the list of characters from the input stream and the corresponding production that has recognized the characters. Note that this forest contains only valid ASTs, but due to ambiguities – in particular in the presence of island grammars – there are several possible ASTs under which the DSL file can be recognized by the parser.

Selecting one AST in Concrete Syntax: The pre-processor uses *disambiguation filtering* [vdBSVV02] to select the AST with the well-defined execution semantics. Therefore, the pre-processor filters the forest of ASTs (Step 5) and select only one from it (Step 5 b). To find the right AST, pre-processor takes both the syntactic and the semantics information into account that is defined in the EDSL interface.

The filtering always prefers AST nodes associated with detailed productions over AST node with water productions. Further, the pre-processor performs semantic checks to eliminate ASTs from the forest that are invalid. For every sub-expression passed to another expression, the filter determines the corresponding production and method in the EDSL interface via introspection (Step 5 a). This allows the filter to take into account the return type of the method that defines the semantic type of the computation returned by this sub-expression.

Transforming the AST to Host Syntax: In the transformation step (Step 6), the pre-processor uses all transformations in the order they have been selected by the end user. The transformation rewrites every AST expression in concrete syntax into its contaminated syntax.

Pretty Printing, Host Compilation, and Launching: After the AST has been transformed to the host language syntax (Step 7), the AST is pretty printed into a file that only contains valid host language expressions. The pre-processor uses the unchanged host compiler to compile this file to binary form (Step 8) and launches the binary (Step 9).

8.1.4. Enabling a new Host Language

To support a new host language, a language developer has to specify the parts of the host language's syntax that are relevant when embedding DSLs. This set contains the most important productions that are needed to syntactically recognize boundaries of basic program structures in the host language. This will include: its syntax layout (white spaces and delimiters), identifiers, type definitions, brackets, variable declarations, method calls, object instantiations, block structures, and so forth. These productions are required to support embedding DSL expressions into programs written in the host language.

To ease the definition of production rules, there are special non-terminals. As discussed previously, there are special categories: $\sqcup$, $\overset{?}{\sqcup}$ and $\overset{!}{\sqcup}$. Additionally, Id is for identifiers that match the regular expression [A-Za-z0-9_+]. Further, for partial grammar definitions, developers can use the water rule $water$ that matches arbitrary characters but only when no other production matches.

For defining BNF-like custom productions, there is an API. Language developers use this API to add new production rules.

For example, in case of Groovy, the developer uses the above Java API to define the productions listed in Listing 8.8. The grammar uses the *water rule* in line 19 to avoid having to define the complete grammar of Groovy. This allows parsing statements of the host language and of the EDSL that have not been formalized.

Compared to approaches with a complete syntax definition, the integration of the host language provides less guarantees, because many expression will be parsed using the water rule. However, the language developer can always specify more productions or the complete host grammar if desired. Conversely, the partial grammar requires far less productions to be formalized and is sufficiently precise for most practicable purposes. When developing a new embedded DSL in a host language, the developer imports the partial grammar for the EDSL.

In addition to specifying parts of the host grammar the developer needs to provide three transformations to process ASTs of arbitrary EDSLs: The first transformation rewrites every AST node with concrete syntax to an equivalent AST node in contaminated syntax. The second transformation adds bootstrap logic into the AST that will evaluate the DSL statements. The third transformation pretty prints the AST into host language code.

All transformations are Java classes that implement the interface ASTTransformation, they implement a special *visitor* pattern [GHJV95] in which the method for visiting AST nodes has access to a context object that allows access to context information. Each transformation has a set of *assurances* and *requirements* on the AST under transformation. Before a transformation is applied, all its requirements are checked for by reading the meta-data that is attached to the AST. A transformation can add, change, and remove AST nodes and the meta-data attached to these

$S_{Groovy} \rightarrow Program$
$Program \rightarrow Statements$
$Statements \rightarrow Statement \overset{?}{\sqcup} Statements$
$Statements \rightarrow Statement$
$Name \rightarrow Id \overset{?}{\sqcup} \text{"."} \overset{?}{\sqcup} Name$
$Name \rightarrow Id$
$Type \rightarrow rType$
$Type \rightarrow pType$
$pType \rightarrow Name$
$Arguments \rightarrow Arguments \overset{?}{\sqcup} \text{","} \overset{?}{\sqcup} Type$
$Arguments \rightarrow Type$
$Application \rightarrow \text{"("} \overset{?}{\sqcup} Arguments \overset{?}{\sqcup} \text{")"}$
$Application \rightarrow \text{"("} \overset{?}{\sqcup} \text{")"}$
$MethodCall \rightarrow Name \overset{?}{\sqcup} Application$
$pType \rightarrow MethodCall$
$ConstructorCall \rightarrow \text{"new"}\ Name \overset{?}{\sqcup} Application$
$pType \rightarrow ConstructorCall$
$Statement \rightarrow \{ \overset{?}{\sqcup} Statements \overset{?}{\sqcup} Statements \}$
$Statement \rightarrow \text{\$water\$}$

Listing 8.8: Island grammar for Groovy

nodes. After a transformation is complete, the transformation stores its assurances by adding meta-data to the AST. Requirements and assurances allow the plug-in to validate the chain of transformations as defined in its configuration.

8.2. Implementation of the Generic Pre-Processor for Incremental Concrete Syntax

The proposed approach was implemented as an Eclipse plug-in called TIGERSEYE[3]. The generic pre-processor is a project builder that can be integrated with every project's build process. In the following, we give a brief overview of TIGERSEYE.

The developed pre-processor is host language independent. As a proof of concept, TIGERSEYE supports two host languages: Groovy and Java. The only requirement of TIGERSEYE is that the host language compiles to the Java Virtual Machine and that it is possible to extract meta-data from an EDSL implementation, e.g., provided by means of annotations. These requirements are met by most modern languages, such as C#, Scala, Smalltalk.

Using the current version of TIGERSEYE, language developers can develop embedded DSLs (EDSLs) using Groovy or Java. Using *Java reflection* [AGH05, FF04], TIGERSEYE introspects EDSLs to extract the EDSL's meta-data, such as information about the production rules and the transformations. The extracted partial grammar is made accessible as a set of first-class objects. When a program uses several EDSLs, the pre-processor first composes the EDSLs' grammars and meta-data.

[3]A tiger's eye is a gemstone in which one mineral is embedded into another mineral. The plug-in's source code will be made available after the review process.

Currently, if multiple EDSLs are used, TIGERSEYE always creates the union of grammars of the composed languages. This produces a conflict if two or more of the EDSLs define the same production. Therefore, the current pre-processor implementation only supports conflict-free compositions. However, given that the composed languages can generally share the same keywords—without generating a conflict—the current implementation suffices for most implementations. During the evaluation of TIGERSEYE, conflicts were a non-issue.

The parser implementation uses the Earley algorithm [Ear70] which can handle all types of CFGs. Instead of generating a parser from a BNF, the pre-processor's parser directly uses the grammar supplied by the language developer for the EDSL and the host language. An important advantage is that the generic parser makes the grammar available as a first-class object during the transformation process. This enables the pre-processor to extend and compose grammars of the host language and of the embedded DSLs.

The implementation of the Earley algorithm adds support for island grammars [Moo01] as outlined next. The standard Earley algorithm has three steps that are continuously repeated for every character, namely 1) scan, 2) predict, and 3) complete. To support island grammars, the scan step was extended. The new scan step first tries to accept the next character using the explicitly defined productions. Only if no production accepts the next character, the water rule recognizes this character as *water*. Each parse step is stored as an item in a so-called Earley set in the parser's chart, in which each item is associated with the character recognized in the step. After parsing, the chart provides all information for the full *forest of ASTs*.

To represent ASTs and to implement term rewriting, TIGERSEYE uses ATerms [vdBK07]. ATerms unify several term formats, optimize storage through *maximal subterm sharing*, and allow language-neutral exchange. To implement custom transformations – as extensions of the pre-processor – the language developer also has to use ATerms. The transformation steps are: 1) the pre-processor translates the Earley chart to ATerms, 2) then the pre-processor uses the ATerms' pattern matching facilities to perform transformations on the ASTs, after that 3) the pre-processor uses the ATerms ASTs to generate the host language code.

8.3. Summary

This chapter has presented an extension to the core architecture that by using island grammars significantly reduces the effort necessary to enable concrete syntax for embedded DSLs. Compared to related work, the main advantage is that only parts of a DSL's concrete syntax need to be specified.

This has three main implications: First, it is only necessary to specify a DSL's concrete syntax for those parts where the host language's concrete syntax does match the domain syntax. As the evaluation will show in Section 10.4.4 (p. 240), this reduces the effort necessary for specifying the DSL's grammar. Second, the approach supports to incrementally define a DSL's concrete syntax which helps to reduce the initial effort necessary when developing an EDSL. Third, the implementation of an embedded DSL can be – up to a very large degree – host language independent. This facilitates reuse of EDSL implementations; in particular, if the host language compiles to a bytecode representation such as Java or CIL bytecode.

Part IV.

Evaluation

9. Related Work

This section compares related embedding approaches. Because REA is a homogeneous embedding approach, Section 9.1 discusses homogeneous approaches first. In general, heterogeneous ones use different techniques, which are more related to non-embedded DSL approaches than to homogeneous EDSL approaches. Still, since TIGERSEYE/REA makes use of heterogeneous concepts, similarities to these approaches are discussed in Section 9.2.

9.1. Homogeneous Embedding

Homogeneous embedding approaches inherit most of their host language's features, since the embedding and its programs are seamlessly integrated with the host language, and it can use the host features freely. As a consequence, when there is a special interest in reusing certain host language features for the embedding, this strongly influences the choice of the embedding approach.

The literature proposes to use various host languages that make use of different features, namely (a) *pure functional languages*, (b) *dynamic languages*, (c) *multi-stage languages*, and (d) *strongly-typed object-oriented languages*. If there are no hard requirements for certain language features to be available in an embedded DSL, language developers have the free choice and likely prefer to use an embedding approach that integrated best with their application libraries they already use. Still, in many cases, the choice of the host language features is prevalent and can even outweigh complexity and the drawbacks of a special embedding approach. Therefore, this section first highlights special host features and particular problems addressed by them, and finally compares the approaches to REA.

9.1.1. Functional Languages

In pure-functional host languages, there are several approaches that use different techniques and encoding styles. Still, the differences of those functional approaches are not important for a comparison with REA. Therefore, first, each approach is explained, only the individual issues of a particular approach are discussed in place, and then, all pure-functional approaches are compared together with REA.

Pure Embedding: Hudak [Hud96, Hud98] uses the *pure* functional host language Haskell for embedding DSLs, which he calls *pure embedding*. A language developer defines domain-specific constructs of domain types using algebraic data types and domain operations using higher-order functions on these types. The major advantage of pure embedding is that language developers can rely on functional composition. To execute embedded programs, DSL programs recursively evaluate their embedded expressions returning a typed computation. For each computation, its value is tagged with the corresponding domain type. Polymorphic functions use pattern matching on these types, therefore, functions only accept well-defined expressions. To evolve languages, developers can compose languages from modularly implemented constructs by composing *monads* [Mog89, Wad90, Ste94, LHJ95], if the used algebraic types are compatible. Hudak demonstrates that with his approach, developers can implement small languages, such as a mathematical language for calculations on regions. Further, Hudak implements common features found in mainstream programming languages, such as *state* and *error handling*.

Tagless Embedding: Carette et al. [CKS09] propose a pure embedding approach that addresses several problems of Hudak's pure embeddings, which they call *tagless embedding*. They use functional composition to build *typed, embedded* DSLs in the *OCaml* language. In contrast to using data types to tag expressions, they encode expression in *higher-order abstract syntax* (HOAS) [PE88] that uses only lambda expressions to encode expressions. In contrast to Hudak, HOAS expressions are not tagged with type constructors. HOAS enables a homogeneous embedding of tagless program expression. To encode execution semantics, they implement recursive fold over the HOAS. Further, the authors discuss the transferability of their results in other typed functional host languages, such as Haskell and MetaOCaml.

The main advantage of this approach is that they represent typed expressions in the embedded language as typed expressions in the host languages, i.e. using the same types makes the type system uniform. Because of having a uniform type-system between the embedding and program, the host compiler can type-check and better optimize embedded programs than the other pure embedding approaches. A key property is that the resulting type-preserving interpretations have the guarantee that compiled programs execute without type failures. Different to Hudak, they use *functors* to bind expressions in a program to their semantics. Using functors allows them to parametrize a program with several evaluators, this allows programs to *abstract over semantics*, i.e. they can use different evaluators to interpret one and the same program representation under various semantics. In tagless embedding, abstracting over semantics enables analysis for program, which they demonstrate by implementing an expression counter analysis.

Unembedding: Atkey et al. [ALY09] address the problem that it is awkward to analyze and manipulate embedded DSL expressions encoded in HOAS. To solve this problem, they transform the HOAS of embedded expressions to *de Bruijn terms*—a special encoding. They call this transformation *unembedding*. In previous work, Atkey proofed that HOAS encodings can be isomorphically mapped to de Bruijn encodings and back [Atk09]. The advantage of de Bruijn encoding over HOAS is that language developers can implement analyses more conveniently. Domain types in embedding are defined using *generalized abstract data types* (GADTs) in Haskell. They demonstrate the applicability of their approach by presenting several small embeddings, such as untyped and typed lambda calculus, pattern matching, Boolean values, and numbers. They demonstrate a simple analysis that counts expressions in a program, and transformations between HOAS and de Bruijn terms, but domain-specific analyses and transformations are out of scope. Further, they show that their approach can enable *mobile code* and *nested relational calculus*, which permits nested queries in query languages. The downside of using GADTs is that embeddings may suffer from exhaustive pattern matching [CKS09]. Further, they identify several problems and limitations with the current Haskell type system, which cannot proof type soundness in certain situations that lead to so-called *exotic types*. They address some of these problems, e.g. with *type casts*, which unfortunately pose an additional effort on language developers.

Comparison with REA: In general, the pure embedding approaches provide a good support for for black-box extensibility. Unfortunately, pure embedding mostly demonstrates extension by re-implementing general-purpose language constructs (e.g. lambda abstractions) that are already available in their host, from which it is hard to draw conclusions about their applicability for DSLs. Still, pure embeddings have the benefit that they support type-safe embeddings, and therefore, they provide implicit safety guarantees. In contrast, in REA, since the Groovy compiler does not require the developer to use types and the signatures of method calls are not type checked, there is no implicit guarantee. For every additional guarantee, a language developer needs to implement a special analysis, e.g. like the keyword analyzer that checks that a program is syntactically correct. Because with pure embedding developers can create languages from closed black-box components, these components cannot be opened up and adapted by other language components. Since pure embeddings have no meta-level, pure embeddings do not support semantic adaptations like REA.

Further, they support composability of independent languages, but they do not address composing languages that have syntactic or semantic interactions. They use functional abstractions, therefore the language components are strongly encapsulated, which disallows composing embeddings with interactions.

None of the pure embedding supports concrete syntax or adapting their host language's scoping strategies. Atkey implements a more complicated general-purpose analysis, but in contrast to REA, domain-specific analyses are also out of scope. Only Atkey supports transformation, such as transforming HOAS to de Bruijn encodings and back, but none of the pure embedding approaches addresses domain-specific transformations like REA.

A rather practicable limitation is that these pure embedding approaches assume the developer to be a domain expert *as well as* an expert in functional languages, requiring knowledge e.g. in advanced type systems, monads, and higher-order functions. Their assumptions heavily restrict the pool of people available as language developers, since only few developers in industry have both skills. In sum, because of these reasons and combinations of thereof, so far, there are a few applications outside the academic community. In contrast, REA only requires the language developer to know OO programming.

To allow the same flexibility in functional host languages like REA supports, it is likely that a pure functional host languages would need special extensions to open up functions, e.g. to realize aspects. But opening up black-box functions can be expected to destroy the *pureness* of the host language [HO07], which puts into perspective the advantages of the pure embedding approach. In pure functional languages, a key limitation is that polymorphic functions are restricted to one *compilation unit*, e.g. a developer can only overload a function signature within one compilation unit. It is not possible to adapt abstractions that are encapsulated in functions. All possible adaptations of a function have to be anticipated in the function's parameters, when the function's signature is defined. If an adaptation has not been anticipated, a function signature or its case for pattern matching cannot be extended later on. To adapt the syntax of a pure embedding encoded in a module signature (or type class), it is required to open up the module's signature. To adapt the semantics encoded in module (or type class instance), it is required to exchange the method bodies in module implementations. To adapt the models, with GADTs, the developer would need to extend GADTs to allow constructors to have new return types, or the developer needs to inspect and manipulate universal types. There are several promising techniques in functional languages that could be interesting to use to enable *adaptable* pure embeddings, such as *first-class modules* [MS02], adapting *pattern matching* on function signatures [LH06] e.g. from *first-fit* to *best-fit*, and *open type classes* [LH06]. Incorporating reflective features into pure functional languages has been studied, but it has been reported that it is unclear whether these feature can preserve *pureness* and *type-safety* [WKD04].

9.1.2. Dynamic Languages

There is a long tradition to embed domain-specific languages in dynamic languages. This section gives an overview of embedding with dynamic host languages.

Jargons: Embedding domain-specific languages has been a well-known technique in languages of untyped functional languages, such as Scheme from the LISP family. In [Pes01], Peschanski refers to such an embedded language as a *jargon*. A jargon is implemented in Scheme using its macro system. Embedded programs are represented as S-expression in abstract syntax. To define the abstract syntax, the language developer uses a meta-language, which is itself implemented as a jargon—a meta-jargon. For each defined expression type, the jargon defines a macro of which the name defines a keyword and the macro parameters its sub-expressions. To define semantics, the macro body produces Scheme code. A code sequence is implemented using a so-called *schemedef*, which plays a similar role as a code block. To evolve jargons, hierarchical composition of jargons is supported by one jargon explicitly importing other jargons. Other forms of compositions are not discussed. The advantage of jargons is that there is no requirement for type annotations and no restriction by a type system. The disadvantage is that program executions can result in runtime errors.

In comparison to REA, using a macro system in jargons is technically quite different from using the MOP in REA. Schemedefs code blocks play a similar role for encoding abstraction operators in jargons like closures in REA. Jargons have an import mechanism that allows adding keywords to individual languages, similar to single inheritance in REA. However, jargons do not consider conservative extensions. Despite this, similarly to REA, jargons have a meta-level—the meta-jargon. But their meta-level is only for defining a meta-circular syntax, thus developers cannot use it for semantic adaptations. Unlike REA, jargons have no special mechanisms for evolution, such as REA's support for late binding of syntax to semantics, abstract over semantics, and semantic adaptations. In sum, unlike REA, jargons do not support sophisticated forms of language evolution.

Ad-hoc Embedding in Ruby: *Ruby* [Rub, TFH09] is a fully object-oriented scripting language that frequently uses embedded DSLs in its frameworks. Ruby allows modularly defining embedded DSLs in classes. Embedded programs are Ruby scripts in abstract syntax. Language developers define the abstract syntax for expression types in a class's method signatures and the corresponding method implementations define the semantics. There are numerous demonstrations of practicable embedded DSLs. For example, Ruby uses a family of embedded DSLs in its popular *Ruby on Rails* Web framework. To evolve embedded languages in Ruby, Ruby can re-open the class definition of a embedded language implementation. When re-opening classes, developers can add methods for new abstract syntax and semantics even at runtime, or they can rename individual existing methods to override them with a new method. There is little research for composing independently developed languages in Ruby, although it has special features for composition, such as runtime mixins, and features for invasive adaptations via reflection.

In comparison to REA, for Ruby, there are only example embeddings to guide developers, but there is no disciplined architecture like in REA. In Ruby, DSLs are rather implemented as ad-hoc embedding, which makes embedding in Ruby rather a craft than a discipline. Still, embedded DSLs in Ruby support a similar extensibility as REA. Similarly to REA, in Ruby, there is no restriction by a type system, and developers can re-open classes at runtime, which allows unanticipated extensions like in REA. With re-opening classes, Ruby allows to dynamically add keywords similar to REA that uses the MOP for this. However, when using Ruby's reflective features, this internally manipulates the program's AST at runtime, which allows individual syntactic and semantic adaptations of embedded DSLs in Ruby. When adding or renaming a method, this adds or renames the method in the AST and adaptation's changes are immediately visible to others. Unfortunately, Ruby's reflective features do no adequately support multiple adaptations, since new adaptations always override previous adaptations. In case, there are multiple adaptation to the same AST elements, then this constitutes an interaction, or it may even be a semantics conflict. Ruby allows isolated reflective adaptation, but Ruby has no mechanism to handle interactions and conflicts between several adaptations, therefore overriding adaptations can lead to erroneously adapted embeddings. In contrast to REA, in Ruby, there are no concepts like language combiners, no support for concrete syntax, scoping strategies, analyses or transformations.

TwisteR: Achenbach et al. [AO10] present an embedding approach for embedding aspect languages into Ruby, called *TwisteR*. TwisteR focuses only on implementing dynamic analyses using dynamic aspects, but unlike REA, TwisteR is no embedding approach for DSLs in general. They use this thesis's concept of a *meta-aspect protocol* that was presented in Section 4.2.3.3 (p. 109) and [DMB09]. They embed special scoping strategies for aspects that have been proposed by Tanter [Tan08, Tan09] to control the binding and activation of aspect for dynamic analyses. Further, they apply a special technique to intercept execution at the basic block level, which is similar to the concept of *sub-method reflection* [DDLM07], but has been developed independently. On top of these techniques they implement special abstractions for dynamic analysis. The advantage is that end users can easily embed dynamic analysis for debugging aspects, similar to [DMB09], and it enables explorative testing with non-deterministic input data. Although TwisteR aspects enable individual analyses and transformations of embedded programs, composing analyses and transformations is not supported, since TwisteR has no mechanism to handle interactions and conflicts. In contrast to REA, in Groovy, there are no concepts like language combiners, no support for concrete syntax, or scoping strategies.

Ad-hoc Embedding in Groovy: *Groovy* [Gro, KG07] supports embedding through two special concepts for building heterogeneous and homogeneous embeddings.

Groovy supports implementing extensible EDSLs using so-called *builders*. Builders do support creating objects as well as implementing heterogeneous embedded compilers similar to Kamin's approach [Kam98]. A Groovy *builder* must extend a special library class and add methods to it in order to define DSL syntax and semantics. Builders can be extended with single inheritance. There are no special means to compose several builders.

Further, Groovy supports another concept, called a *category*, that are special classes that defines static methods that are dynamically mixed into existing classes. Categories mixed in methods are often used to enable little DSLs. Mixing several categories at a time allows simple compositions of DSLs. However, the language compositions resulting from mixed categories provide no guarantees for correctness.

Similar to Ruby, embedding in Groovy is rather ad-hoc and it does not follow a well-defined discipline as REA. Due to using the same host, ad-hoc Groovy embedding could be as powerful as REA, but Groovy's ad-hoc embeddings are only limited because they do not exhibit an architecture that can exploit the Groovy features completely like REA. Unlike in REA, ad-hoc embeddings in Groovy provides no guarantees for language developers. They can compose only embedded libraries in a dynamic language without special analysis or transformation. The Groovy compiler only validates a part of the embedding or embedded programs, in particular the Groovy compiler does not validate correctness of method bodies. Therefore, it is not possible to check whether an EDSL program invokes only well-defined methods of an embedding and that the method body does not contain calls to missing methods. With REA's support for abstracting over semantics, there is support for analyses that validate the correctness of a program or a composition, whereby the analyses can analyze the keywords used in closures and method bodies. Further, with categories, it is not possible to compose several categories that define the same keywords, whereby detecting and handling the resulting interactions and conflicts. Since categories are based on static methods, they cannot be inherited. Thus, there is no support for language polymorphism. Similar to REA's BlackBoxCombiner that analyzes EDSLs for conflicts. Groovy lacks concepts like language combiners, support for concrete syntax, and scoping strategies. In Groovy, developers can use meta-programming to write AST analyses or transformations for compile-time. However, runtime analysis and transformation are not addressed.

Embedding in π: The π language [KM09] is a special host language that provides dedicated features to change the syntax and semantics at runtime. In particularly, interesting is that π programs can have any syntax of a CFG. The language developers defines DSL expression types in special modules called *patterns*. Each pattern recognizes a piece of the concrete syntax and gives it a *meaning*—an interpretation in the π language. The π interpreter processes DSL program line by line. When encountering expressions in a line, there must be always exactly one matching pattern for that expression type. Patterns can be redefined and they are lexically scoped, thus π always uses the closest enclosing pattern definition to interpret an encountered expression.

In comparison, π is more syntactically more declarative than REA in that it provides a special syntactic constructs to define concrete syntax. Similar to using annotations in TIGERSEYE/REA, the syntactic and semantic extensibility built into π makes it particularly natural to evolve embeddings. But in contrast to TIGERSEYE/REA, developers do not have to use verbose standard Java/Groovy annotations to define a concrete syntax. In π, developers can compose expression types at the level of patterns. But, there is no special module concept at the level of languages, therefore several embedded DSLs can only be composed by copying their sources into one π file. Programs are also not modularly defined. Unfortunately, π's exceptional language features do not allow adopting the approach to other host languages. The dynamic features of π also required that programs are executed with an interpreter, π cannot be compiled to bytecode like REA. Further, in contrast to REA, in π, there are no concepts for scoping strategies, analyses or transformations.

Helvetia: Renggli et al. embedded DSLs into Smalltalk [RGN10]. Their approach, called *Helvetia*, addresses the problem of providing support for concrete syntax and improving tool support. End users can encode DSL programs either in Smalltalk syntax. To embed a language without special syntax, the language developer directly defines a set of Smalltalk classes those methods and fields define expression types in abstract syntax. To embed a language with a special syntax, the language developer implements a parser in Smalltalk using a parser combinator library. To define execution semantics for special syntax, the developer uses an embedded DSL to implement transformation rules on AST nodes. After parsing a DSL program, its expressions represented in AST nodes are transformed to ordinary Smalltalk code and then compiled by the host compiler. The advantage of Helvetia is that it supports certain interesting evolution scenarios. Developers can extend a language by attaching additional parser components to an existing parser using combinators. They can define several parsers that can be used in parallel. Another benefit of choosing the Smalltalk platform is that Helvetia integrates well with the Smalltalk tools that developers can extend for syntax highlighting of their DSLs. Helvetia's homogeneous integration with Smalltalk allows the debugger to trace transformed code back to its textual representation in concrete DSL syntax. Unfortunately, Helvetia relies on the exceptional features of Smalltalk, e.g. that a compiler component is accessible at runtime. As a consequence, the approach cannot be adopted for other host languages that do not provide these features.

In comparison, Helvetia has a similar extensibility like REA, but Helvetia has a different architecture and it uses different technologies. Similar to REA, Helvetia supports implementing language components, called *language boxes* [RDN10]. Helvetia supports concrete syntax as parsing expression grammars that are composed using a parser combinator library. Similarly, as parser combinators are first-class objects, in REA, BNF and the Earley parser are both homogeneously embedded in Java, new parsers can be created and manipulated as object at runtime.

But, Helvetia does not support the full subset of CFGs like REA. Syntactic ambiguities are resolved by implicit priorities, which can leads to erroneous compositions that remain undetected by the developers. However, Helvetia uses transformation rules similar to source-transformation languages, but it transforms a DSL program at load time and not at compile-time. Helvetia transforms every program once. After a program is transformed, it cannot be transformed again. In contrast, TIGERSEYE/REA's pre-processing transforms program in concrete to abstract syntax before compile-time. However, still, when a program is transformed to abstract syntax, the abstract syntax of the program can be adapted using the meta-level, which enables runtime evolution e.g. for dynamic aspects. Helvetia supports scoping transformation rules e.g. on particular AST nodes or a particular context, which restricts the rule's application. However, the scoping mechanism does not allow assigning one and the same program to different semantics, therefore, it is not possible to abstract over semantics, like REA's first-class code blocks enable a dynamic syntax-to-semantic binding that can be changed at runtime. Similar to REA, Helvetia allows adapting programs of embedded languages, in particularly Helvetia uses reflection to adapt host language expressions. However, reflection is currently only used to adapt end user programs. Reflection does not adapt language implementations, like REA. Further, in contrast to REA, Helvetia does not support scoping strategies.

9.1.3. Staging/Meta Programming Languages

(Multi-)Stage Languages: A (multi-)stage host language [SBP99, COST04] has a small set of special language constructs for constructing AST nodes within programs, combining them, and generating executable code from ASTs, whereby often a static type system guarantees that all programs they generate are correct. In (multi-)stage host languages, developers can implement languages embeddings using meta-programming in a homogeneous way, i.e. programs that generate other programs in the same language. Staging-based embedding approaches address the problem of the interpretative overhead for embedded languages that is removed by generating code. There are several embedding approaches that use different host languages. Czarnecki et al. [COST04] compare staging in *MetaOCaml*, *TemplateHaskell*, and *template meta-programming* in *C++*. The difference between these host languages and the approaches are not relevant for a comparison to REA.

To embed a DSL, a language developer embeds AST nodes for each DSL expression as a data type for the DSL. Embeddings are structured in one or several stages. Each stage can rewrite, optimize, or compile expressions by introspecting AST nodes of the previous stage and reflecting changes to the next stage. For constructing part of the AST of a DSL program, (multi-)stage languages use a quasi-quotation mechanism. Quasi-quotation provides a *quotation operator* with which developers can embed expressions of the object language into the meta-language. For example, in *TemplateHaskell*, one can quote a Haskell expression in Oxford brackets [|..|] that *reifies* a corresponding AST representation of it. For combining expressions of different stages, often there is a special *anti-quotation operator* to escape inside a quoted expression. Finally, for execution, there is a *splicing operator* that *reflects* an AST back to code, i.e. it generates executable code. When using the splicing operator for program with quoted DSL expressions, from the quoted DSL expressions the stages generate plain host language code. Because staging allows compiling programs from the object language to the meta-language, there is no interpretative overhead.

In comparison, staging has a different focus than REA. Staging approaches target expert programmers and end users with special skills. It uses special host language features, and cannot be integrated with a host language that does not support quasi-quotation. Staging-based embedding have a very different architecture and use very different techniques than REA. The biggest advantage of staging is that, after compilation, there are no library calls to an embedded library, since the stages generate code (inline) at compile-time [COST04, Tra08]. In contrast, in REA, because Groovy autoboxes every primitive value and every method call is an reflective method call, there is a significant overhead on the execution, which only can partially be remove by the standard optimizations of the Java VM. Another advantage of typed multi-staged embeddings is that the host's type system can guarantee that (more or less) all generated code is well-typed [COST04]. In contrast, in REA, the Groovy host compiler does not perform enough type checks to ensure meaningful code. For each additional guarantee, REA needs an implementation of an new embedded analysis. Further, the quoting mechanism eases to mix expressions in the meta-language and the object language, which makes it relatively simple for language developer to switch stages. However, the stages have a fixed order, which cannot be changed.

However, staging supports only a subset of the evolution features. So far, only very simple forms of language constructs have been embedded that do not allow drawing conclusions about the applicability of staging approaches to DSLs. Staging-based DSLs are confined to their host languages syntax and semantics. The embedded AST of a DSL is hard to extend, since one cannot add a new AST node without redefining the AST's data type. In contrast to most homogeneous EDSL approaches, developers need to implement a front-end or respective an AST for the DSL, which puts the saving of this embedding approach into perspective. In contrast to REA, the AST of different DSLs cannot be combined, because the data types in the host language they use are not composable. Concrete DSL syntax is not supported, the abstract syntax in staging-based DSL has a large syntactic noise for end users. For template C++, there is the largest syntactic noise. In contrast to REA, staging does not address embedding scoping strategies. Domain-specific analyses are only supported as transformations, but implementing analyses as transformations is rather awkward. It is not possible to add new information to AST nodes, unlike to TIGERSEYE/REA where ATerms allow adding semantic information to the AST nodes. TemplateHaskell supports intentional analysis that allows to inspect code of other stages, but they do not use this to extend a stage by another stage. Transformation is natural for staging-based approaches, but they only support transformations to their host languages not to other target languages. Endo-transformation are not supported, since every stage creates a new representation of the program.

Extensional Meta-Programming: Seefried et al. [SCK04] address problems of both staging-based and heterogeneous embedding approaches. For homogeneous staging-based embeddings, they address the problem that with staging the language developer has to implement a *compiler* *front-end* for the embedded language, i.e. the developer has to encode the AST nodes for an embedded language (cf. [COST04, SBP99]). For heterogeneous embeddings, which are discussed later on in this chapter, they address the problem that the developer has to embed a new compiler back-end (cf. [Kam98, EFDM03]). To address these problems, they propose to use compile-time meta-programming for implementing embedded compilers in *TemplateHaskell*, which they call *extensional meta-programming*. An important detail is that they do not need to implement an AST in an abstract data type. Meta-programming enables them to embed optimizations in a homogeneous way, such as *unboxing* arithmetic expressions, *aggressive inlining*, and *algebraic transformations*. Because DSL programs are directly encoded as host expressions, no special front- or back-end need to be implemented. To validate their approach, they have re-implemented Elliott et al.'s *Pan* language [EFDM03], as a homogeneous embedding. They compare the performance of their implementation with and without optimizations, and to Elliott's heterogeneous *Pan* language embedding. Their optimization is implemented with extension meta-programming perform significant better than the embedding without optimizations. But, their measurements show that the original heterogeneous implementation of *Pan* still outperforms their optimized implementation.

In comparison, extensional meta-programming does not need to implement an AST, since expressions are represented in abstract syntax, similar to REA. New expression can be easily added. Extensional meta-programming can exploit TemplateHaskell for more type checks, similar to REA exploiting type checks in Groovy. Unfortunately, TemplateHaskell has no complete guarantee that every generated program is type safe, since type checking of template code is restricted for technical reasons. With respect to its support for evolution, extensional meta-programming has similar problems like staging-based approaches. Composition of different meta-programs is currently not addressed, but functional composition would enables non-interacting compositions. Because extensional meta-programming uses functions to encode domain abstractions, for the same reasons like with pure embedding approaches, extensional meta-programming does not support interacting or conflicting compositions. There is no support for concrete syntax or pluggable scoping, since the approach is completely confined to its host language's syntax and semantics. Analysis is only possible through transformation. While they address transformations at the expression level for implementing optimization of individual expression, global analyses and transformations are out of their scope.

Converge: Tratt [Tra08] also uses a compile-time meta-programming approach that has support for concrete syntax in the *Converge* programming language, which is a special host language for embedding. It is different from the other staging-based approaches in that language developers describe the syntax of the embedded language in a BNF-like DSL, they generate a parser from this, and they specify transformation rules to rewrite AST nodes to Converge code. A Converge program can use a quotation operator to embed DSL code in concrete syntax into a so-called *DSL block*, which will *reify* a corresponding AST representation that is then rewritten by the rewrite rules, which *reflects* the AST nodes to executable code.

In comparison, in Converge, language end users can write the program in any concrete DSL syntax in CFGs. Similar to TIGERSEYE/REA, Tratt uses Earley parsers. However, the Converge approach is limited with respect to extensibility, whenever adding a new expression into an existing language, the Earley parser needs to be re-generated. In REA, since BNF and the Earley parser are both homogeneously embedded in Java, no code need to be generated. Further, in TIGERSEYE/REA, the generic pre-processor does not rely on exclusive host language features. Since only uses TIGERSEYE/REA features that are available in most OO languages, its techniques can be transferred to other host languages. In Converge, transformation happens at compile-time, and therefore, the DSL code can be expected to execute rather fast, while TIGERSEYE/REA has a runtime overhead. In Converge, there are no guarantees that the generated code is type-safe, and there is no support to analyze and transform code, like in REA.

9.1.4. Typed Object-Oriented Languages

The following object-oriented host languages allow modular and type-safe embeddings.

Fluent Interfaces: Evans [Eva03] and Fowler [Fow05] propose to embed DSLs into mainstream programming languages used in industry, such as Java. In Java, language end users can encode embedded programs in abstract syntax as ordinary Java programs that call the API of an embedded library. This API is structured in a special way, which Evans and Fowler call a *fluent interface*. The classes of the library define expression types in the embedded language using Java constructs. Literals are encoded as *constants*, domain-specific operations are encoded with method calls. For creating complex expressions, method calls can be *chained* together, where the return parameter of a method in the fluent interface represents the syntactic category of the next possible expression. He proposed to refer to such an embedded DSL as an *internal DSL* since the embedded DSL is implemented as a library, which contrasts it from *external DSLs* that are implemented with pre-processors or other external tools.

In comparison, fluent interfaces require no special host language features, similar to REA where often special host features can be replaced with more common host features. However, with less powerful features used in the usual hosts for fluent interfaces, it is hard to design the abstract syntax close to the domain, mostly because the Java syntax and semantics are not flexible enough to omit type annotations and delimiters. Fluent interfaces do not support sophisticated embeddings with concrete syntax, scoping, analysis or transformation.

DSL2JDT: Garcia [Gar08] addresses the problem to reduce the effort to implement an embedded DSL as a fluent interface using generative techniques. A language developer models the syntax of an embedded DSL as a tree-based model in Eclipse EMF [SBP$^+$09], and from that model, a generator generates the Java code for a fluent interface API, called *EMF2JDT*.

In comparison, the major advantage of DSL2JDT is that a generator takes over the task to encode abstract syntax in a fluent interface from the language developer. In REA, language developers have to encode the language interface, façade, and model classes on their own. In DSL2JDT, developers can combine their generator with another generator for model constraints, this enables generating constraint checks for embedded expressions. Further, standard tools services, such as code completion and debugging can be reused. The disadvantage of DSL2JDT is that language evolution is difficult. Once the embedded DSL is generated, in case there is language syntax evolves, a language developer must update the model, re-generate, and compile the whole language implementation again. Unlike REA, DSL2JDT does not support composition of DSLs. Because, DSL2JDT is confined to Java, it does not support embedding special scoping strategies, analyses, or transformations.

Embedding in Scala: Dubochet [Dub06] and Odersky et al. [OSV07] experimented with embedded DSLs in Scala [Sca] – a statically typed language that combines features of object-oriented and functional languages. Currently, extending embedded DSLs and composing independently developed embedded DSLs is not addressed by both authors. They have rather focused on demonstrating small examples of embedded DSLs, but they do not propose a disciplined approach for embedding, like REA. Language evolution, concrete syntax, scoping, analysis and transformation remain out of their scope.

Polymorphic Embedding: Hofer et al. [HORM08] propose an disciplined DSL embedding approach for Scala. To enable multiple interpretations of programs, they apply Carette's technique [CKS09] in the context of *Scala*, which they call *pluggable semantics*. DSL programs are encoded in abstract syntax. To define the syntax of a language, this is encoded into a set of method signatures of classes or traits. To define the semantics, a developer defines code in the method implementations. To evolve a language, a developer can use a trait that adds new expression types to the embedding. Hofer et al. also address the composition of independently developed languages. While composition independent languages are discussed, they do not address composition of languages that have interactions in the syntax and semantics. Composition of semantics is based on monadic composition of computations. In [HO10], Hofer et al. have adopted the idea of [ALY09] to use different forms of encodings to allow developers to simpler express new analyses and transformations, but none of the encoding is both extensible and composable. Unfortunately, since they do not address implicit isomorphic conversion from one encoding to another like [ALY09], developers can no more freely choose the best encoding after they have committed to one particular encoding.

In comparison to REA, embedding in Scala equally well supports adding syntax and semantics. For enabling conservative extensions, the type checker can check polymorphic extensions for possible syntax conflicts, which is only possible at composition-time in REA. Embedding in Scala is statically type-checked, but also in Scala, the generic type checker does not provide implicit domain-specific guarantees to prevent semantic conflicts or side-effects. Traits allow plugging in different evaluations at compile-time, which allows abstracting over semantics like REA. But in contrast to REA, polymorphic embedding does not support plugging in new semantics at runtime. In polymorphic embedding, traits allow composition semantics similar to using the linearizing combiner in REA. However, composition between two interacting DSLs can be incorrect, since the linearization implicit handles homonymous member names, which is good for implementing OO applications but counter-productive for embedding. Since syntactic conflicts are implicitly resolved using a pre-defined linearization order, the Scala compiler does not notify the language developer possible conflicts, which can result in erroneous compositions. In contrast in REA, such conflicts are detected by the BlackBoxCombiner and CCCombiner and reported before evaluating the program. In contrast to REA, polymorphic embeddings are confined by their host language's syntax, semantics, and scoping mechanism. It is not possible to provide meta-level extensions or to override the semantics of host language keywords. Hofer et al. [HORM08, HO10] support either extensible or composable analyses depending on the used program encoding. They demonstrate simple analyses, such as expression counting, but domain analysis like in REA is out of scope. They support external visitors to composable analysis of independent languages, but they explicitly exclude composing analyses of languages that have interactions. In contrast, in REA, all analyses are extensible and composable, because combiners enable handling interactions. In polymorphic embedding, only exo-transformations are supported, but it is not possible to transform a running program with an endo-transformation like in REA.

9.2. Heterogeneous Embedding

Heterogeneous embedding approaches are interesting since they try to address the weaknesses of homogeneous embedding approaches by being inspired from traditional non-embedding-based language implementation approaches. Heterogeneous embedding approaches can also be distinguished w.r.t. what kind of host language is used to implement the embedding. First, there is the *embedded compiler* approach that embeds a DSL compiler/generator into a general-purpose language that generates code in the same or another GPL. Second, there is are approaches that embed DSLs in *source translation languages* What is common for both classes is that often the host and the target language are different, therefore they do not allow reusing the host language features within the embedding—embedding can only use the target language features. Moreover, even if they generate code in the same language that implements the embedding, they do not have a uniform compile- and runtime between the host and the embedded language, therefore they cannot uniformly exchange objects between those host and embedded programs, like REA.

9.2.1. Embedded Compilers

Embedded compilers have similar qualities compared to REA but differences depend on employed techniques and therefore are discussed individually.

Embedding Source Code Generators: In [Kam98], Kamin proposes to embed languages as *program-generating languages*[1], which basically are embedded generators, for which DSL programs are actually specifications for generating programs in another language. In such an embedding, the embedded language and the host language are heterogeneous, they may have different syntax, the may even have different semantics, and both languages are processed by different infrastructure (i.e. compiler or interpreter). In such an embedding, a program of the embedded language uses the host language to rewrite its expressions into a target language.

[1]The view on program-generating languages is similar to *meta-languages*, where a program in one language generates a program in another language (cf. Section 2.3.3, p. 25).

Kamin uses *ML* as a host language and generates code in *C++*, which is the target language. To define new syntax, the language developer defines a new expression type as a new function in ML. To define semantics for an expression, the corresponding function generates and returns a code fragment in form of a string. To transform a program into its executable form, the fragments of all programs expressions are concatenated and then compiled by the target language compiler. Kamin demonstrates that his technique can be used to implement various embedded program generators, such as *FPIC* a small language for drawing pictures, a *parser generator* that is combined from smaller parsing components, i.e. *parser combinators*, and a parser generator for the LL(1) sub-class of context-free grammars. The advantage of Kamin's technique is that the execution is less bound to a specific target language, as the code of the generator can be changed to produce code in a different target language. The generated code in the target language, does not suffer from interpretative overhead like homogeneous approaches.

Compared to REA and the other homogeneous approaches that reuse their host's features and compiler, for embedded generators, there is the disadvantage that language developers have high implementation efforts, because they have to implement a new back-end in form of an embedded generator. Moreover, heterogeneous embedded languages cannot reuse the host language features in the generated code (e.g. the host compiler's optimizations like partial evaluation), but developers have to re-implement these features if a host language features is also required in the embedding but not available in the target language. Unlike TIGERSEYE/REA, the program syntax of embedded DSL can only have an abstract syntax. Further, for Kamin, scoping and analyses are out of scope. Embedded generators support exo-transformations but not endo-transformations.

Embedding Compilers: Elliot et al. [EFdM00, EFDM03] extend Kamin technique by embedding an *optimizing compiler* in that compiles Haskell to Haskell and that uses *algebraic manipulation*, which substitutes expressions by more optimal but semantically equivalent expressions. They address the problem that homogeneous embedded DSLs suffer from interpretative overhead. They first tried to speed up homogeneous embedded DSLs by adding custom optimizations using user-defined rewrite rules, which currently only special host compilers enable, such as the *Glasgow Haskell Compiler*[2]. Still, they made the experience that they could not remove this interpretative overhead. When they combined multiple of such rewrite rules, they experienced too complex interactions between the rewrite rules that could not be controlled. To solve the problems, they represent program expressions in abstract syntax as *algebraic types* and statically optimize expressions when these are constructed. For optimizing an expression, they use a *smart constructor* for this expression that pattern match on its sub-expressions to detect opportunities for optimizations, so that an optimized expression is created. They apply optimization techniques inspired from traditional non-embedding-based approaches, such as *constant folding*, *if-floating*, and *static expression type specific rewrites for domain-specific optimizations*. Finally, an embedded compiler rewrites the optimized expressions to the target language. To demonstrate, the advantages of using algebraic types in embedded compilers, they

[2]The Glasgow Haskell Compiler Homepage: `http://www.haskell.org/ghc/`.

address an efficiency problem with a first embedded compiler version that repetitively rewrites *common sub-expressions* in a program. To avoid repetitive rewrites, they perform *common sub-expressions elimination* (CSE) that identifies common sub-expressions in a program, shares them between the expressions, and rewrites them only once. They demonstrate their application of their technique by implementing the *Pan* language, a small language for image synthesis and manipulation. The major advantage is that the language developer can evolve an embedded compiler into an optimizing compiler with only a few changes made to its code. Further, they claim far better performance of programs and more efficient program generation due to CSE. Unfortunately, they do not proof this claim by evaluating the actual performance speed up with measurements.

Rule-based Generators: Cuadrado et al. [CM07] use Ruby to embed DSLs that are rule-based generators for model-driven development, called *RubyTL*. DSL programs are models from which the embedding – a Ruby class – generates code, e.g. Java code. Because only a few classes need to be implemented, Ruby TL is a light approach to implement an ad-hoc generator.

In comparison to REA, although not discussed in their paper, conceptually, RubyTL would support extensibility, since rules and generators are embedded as classes, which developer could subclass. Unlike REA, embedded compilers do not use meta-levels. The embedded compiler approaches can support several embedded DSLs in parallel, but unlike REA, this does not enable programs that can mix DSL keywords. The embedded compiler approaches do not support concrete syntax, scoping strategies, domain-specific analyses and transformation, with the exception of the one and only transformation to the target language.

9.2.2. Source Transformation Languages

The source transformation languages approaches have very similar properties, and therefore, after discussing the each of them in isolation, this section compares them together to REA.

MetaBorg/Stratego: MetaBorg [BV04] is an embedding approach that uses a source transformation language that can rewrite a heterogeneous embedded DSL to any GPL. It is the most mature approach for heterogeneously embedding DSLs with a concrete syntax [Tra08]. DSL programs are encoded in arbitrary context-free syntax. MetaBorg uses Stratego/XT as a host language, which is a DSL for generating language infrastructure consisting of *lexical patterns* and *rewrite rules* as part of a Stratego *module*. To define syntax, a language developer specifies a set of lexical patterns that recognize expressions of a CFG in a program. To define semantics, the developer specifies a set of rewrite rules. The rules can rewrite expression types into AST nodes. They can analyze or transform nodes and finally rewrite AST node into to another language. For evaluation of programs, given the syntax of a embedded and a host language, MetaBorg generates a corresponding pre-processor for parsing and transforming EDSL programs. The pre-processing logic in the pre-processor generated from the rules parses rewrite the successively annotate AST nodes, and finally applies an AST transformation that generates code in the target language. For extensibility in MetaBorg, every language definition module can extend other modules by importing their syntax rewrite rules. They demonstrate their approach by implementing a DSL – called *SWUL* – for creating Swing applications. Programs in SWUL syntax have a more concrete encoding than using Java to create Swing UIs. Additionally, they provide an embedding of Java in Java that allows generating Java programs, and another embedding of XML in Java for generating XML documents.

The advantages of using MetaBorg to implement DSL are manifold, since MetaBorg is a very mature tool. Years of investments have equipped Stratego with many useful features, for defining lexical patterns, importing grammars, priorities, quoting and unquoting, rewriting strategies, generic traversals, and advanced disambiguation with disambiguation filters. These features allow defining modular and composable syntax and semantics. MetaBorg/Stratego supports *context-free grammars* (CFG) through so-called *scannerless GLR* parsers (SGLR parsers) [Vis97a]. Such a SGLR parser creates a *forest of ASTs* that first contains all valid syntactical ambiguities of a program's parse. Later on, MetaBorg uses *disambiguation filters* [vdBSVV02] to remove ambiguities based on declarative disambiguation rules. In MetaBorg, embeddings are independent of the target language. Unfortunately, the fact that MetaBorg is not integrated with the target language disallows safe embeddings, since the generated parsers and pre-processors output code that may contain errors, which eventually later on the target language compiler can detect, but such errors are hard to trace back. Further, MetaBorg is inconvenient for incremental language evolution, since whenever a language definition changes, its complete infrastructure must be regenerated, which is disruptive.

TXL: TXL [CHHP91, Cor06] is another heterogeneous embedding approach that uses a source transformation language that is similar to Stratego in MetaBorg, however the TXL language provides a slightly different set of features. To define new concrete DSL syntax, a language developer specifies the grammar with BNF-like syntax rules (productions), which recognize expressions and create an AST representation. To define new semantics, the developer can encode transformation rules whose parameters accept nodes from the AST. TXL uses these rules to rewrite AST nodes to the target language. Furthermore, TXL allows defining *functions* that traverse the AST to extract information from it and which developer can call in their rewrite rules. To transform a DSL program, TXL parses it into an AST representation and uses the transformation rule the rewrite it into an executable form in the target language. Cordy demonstrates the applicability of TXL by implementing several language, such as a heterogeneous embedding of SQL in the *Cobol* [oD65] programming language, and a little generator that transform XML to C++ code. TXL has many advantages but important limitations. TXL's transformation rules enable both declarative and modular language definitions. In particular, rules and functional abstractions are important, since in TXL developers can precisely restrict the scope of rules and functions. They can build hierarchies of rules that have sub-rules, whereby rules can pass parameters to their sub-rules. In TXL, grammars are implicitly free of ambiguity, since every production is prioritized by the order the productions in a grammar are defined. Unfortunately, TXL does not support a composable subset of CFGs, and therefore it does not fully support composition.

Comparison with REA: In comparison, MetaBorg and TXL are very mature approaches that allow embedding large DSLs with several hundred of expression types. Although MetaBorg and TXL are heterogeneous approaches, REA/TIGERSEYE makes use of similar concepts and mechanisms. The module concepts of Stratego and TXL play a similar role like the façade class in REA. Implementing a rewrite rule in MetaBorg or TXL is similar to implementing a method in one of REA's façades. Implementing a global strategy in MetaBorg or global scoped function in TXL is similar to implementing a meta-method (e.g. invokeMethod) in a meta-façade in REA that uses a generic strategy for providing the evaluation semantics. However, implementing a global strategy in a meta-façade is more complex than implementing in a declarative language Despite this REA's advantage is polymorphism, MetaBorg and TXL do not provide late binding semantics for extensions, and therefore they do not support conservative extensions.

w.r.t. the concrete syntax, only MetaBorg/Stratego supports CFGs like REA, but TXL only supports the not composable subset of LL(*). Similarly to REA, Stratego and TXL support island grammars [Moo01], but MetaBorg and TXL do not use them for partial definition of grammars for EDSLs like in TIGERSEYE/REA. It would be interesting to investigate using island grammars for embedding in these approaches.

Like TIGERSEYE/REA approach, both MetaBorg and TXL are host-language independent, but supporting new host languages is more complicated than with TIGERSEYE/REA. To support a new host language, Stratego/XT and TXL require a complete grammar definition of the target language. However, many grammars are not available in Stratego or TXL. As we will discuss later on in Section 10.4.4 (p. 240), for example, to support Groovy in Stratego or TXL, the complete Groovy grammar would have to be (re-)written in a SDF or TXL specification that encodes approximately 950[3] BNF-like productions. To enable parsing Java and Groovy less with embedded DSL syntax in REA, the developer only needs to specify 19 productions for Java and respectively for Groovy.

In contrast to REA, MetaBorg and TXL support sophisticated analyses and transformations. In MetaBorg, Stratego support global analysis that traverse the whole AST. In MetaBorg and TXL, language developers can use analyses to implement scoping, however, the language developers must themselves implement symbol and lockup tables, which is unlike reusing the first-class environments embedded in REA. Similarly to REA, transformation can be statically plugged by importing the rules into the corresponding module, but dynamic transformations are not supported by MetaBorg and TXL.

9.3. Non-Embedding Approaches

Despite non-embedded DSL approaches are complimentary to the research on language embedding, this section gives an overview of non-embedded DSL approaches and points out important relations.

Non-embedded DSL approaches have a different focus and use different techniques than embedding DSL approaches. Usually, non-embedded approaches provide special concepts and

[3]This estimation is based on a detailed analysis of the Groovy ANTLR grammar.

techniques to realize productive-strength DSL implementations, but these approaches require that the language developer is an expert who knows the approaches' complicated concepts and techniques. These approaches a rather steep learning curve, and they have high initial development costs [KLP+08]. In contrast, most embedded approaches make use only standard language features and target at non-compiler-expert language developers who want to implement rather small languages with low initial costs.

Despite the differences between embedded and non-embedded approaches, it is in particular interesting what concepts from non-embedded approaches could improve existing embedding approaches. Therefore, the remainder of this section gives links to concepts that could be interesting for future investigations, but that have been out of the scope of this thesis.

9.3.1. Extensibility

Non-embedded approaches have a broad variety of mechanisms for extensibility that could compliment the extensibility mechanisms in embedded DSL approaches.

Language Implementation Approaches: In particular interesting is the research on extensibility mechanisms for language implementation.

There is a myriad of language implementation approaches with a closed set of extensibility mechanisms. There are special language implementation approaches that provide special support for syntactic and semantic extensions, which are discussed in the following.

For syntactic extensions, *grammar inheritance* [AMH90, KRV08], *importing grammars* [BV04], *overriding grammars* [Par08], and *transformation rules* [Cor06] is in the same vein as REA's language polymorphism. Generally, the extensibility mechanisms in meta-languages are more mature ans specialize for language implementation than the extensibility mechanisms borrowed from the host in REA. However, import/inheritance mechanisms in meta languages do not provide established concepts like late binding. In contrast, in REA every inherited syntax need to be fully in conformance with the base syntax, which provides sub-type guarantees for languages.

For semantic extensions, there are special language implementation approaches that allow extensions to a closed set of language mechanisms. Most works focus on extending OO mechanisms. *Meta-object protocols* [KRB91] enables experimenting with OO semantics, but combining several extensions with meta-objects for one object is hard. In [HBA10], Havinga et al. present a model for composable composition operators that allow experimenting with inheritance mechanisms. In [AKMP10], Axelsen et al. present a meta-protocol for packages and class inheritance that is inspired by the meta-aspect protocol of REA. Compared to these approaches, they only consider extensions to a closed set of mechanisms in one individual languages. In contrast, REA is open and allows embedding various new language mechanisms with a flexible strategy, such as scoping strategies and meta-aspects.

Reflective Architectures: Various reflective systems have been proposed, which build upon different reflective concepts. Their implementations have subtle differences, but these are not important for a comparison.

An extensive overview of the different architectures and techniques they use is given in [Tan04]. Smith have proposed *procedural reflection* [Smi82] in 3-LISP, which is implemented as an continuation passing interpreter. In [RS84], Smith et al. proposes *reflective procedures*, which allow executing code from one level to execute at a higher level, which is more "meta". Friedman et al. [FW84] propose the concept of a *reifier*. With reifiers, one can gain access to the state of the interpreter by making a meta-level entity available at a base-level, which they call *reification*. Conversely, one can manipulate values and pass them back to the interpreter, which they call *reflection*. Maes [Mae87] proposes a uniform architecture for reflective OO languages. She propose the concept of a *meta-object* that interprets the semantics of base-level objects. Ferber [Fer89] has adopted Maes concepts for class-based OO languages. Central to those reflective languages is that there is an interpretation level that executes the base language concepts, and that for this level, there is a meta-level that allows gaining explicit access to what is interpreted below. There may be additional levels that interpret the meta-levels. Theoretically, there can be an *infinite of tower* such meta-levels [WF88].

In comparison, in REA as shown in Section 4.1.1.4 (p. 77), every language implementation implicitly follows REA's structure with a meta-level and causally connected entities in the language model. It is trivial to reify a first-class representation, since one can obtain a reference to an instance of the language model. It is also trivial to reflect changes made to a first-class representation back, since such changes directly manipulate the first-class representation. Similarly to many other dedicated reflective architecture, there are no special optimizations. A few reflective architectures have special optimizations, such as Reflex [Tan06] that optimizes composition of aspects and CLOS [KRB91] for which compilers can remove indirections due to meta-objects.

Semantic Adaptations in Languages: Non-embedding approaches support semantic adaptations for individual languages, which are GPLs or extensions to GPLs.

Adaptable programming languages allow adapting language constructs and mechanisms. They have a closed built-in indirections in their infrastructures or program code, still developers can configure these indirections from the outside to adapt them. Adaptable languages have differences in what language constructs developers can adapt and how. For syntax adaptations, For semantic adaptations, developers can adapt the language infrastructure to change the evaluation process. There are various realizations, discussed in the following.

Adaptable grammars enable developers to adjust the parser for new expression types. However, adapting the syntax does not address adapting semantics.

Today most reflective languages are individual solutions that extend non-adaptable and non-reflective language with reflective constructs. Only a few languages are built with support for reflection from the beginning. Because of the high investment necessary, most reflection mechanisms are only built in general-purpose languages (e.g., procedural, logic-based, and OO languages) and for general-purpose language constructs (e.g., procedures or objects). However, there is a lack of generic approaches for implementing reflection into DSLs.

Kiczales et al. [KRB91] proposes using meta-object protocols (MOPs) that each provides an abstract meta-interface to end users allowing them adapt the semantics of object-oriented language constructs. Adapting the semantics of domain-specific constructs is out of scope. Relying on the semantic flexibility that MOPs enable for object-oriented language, REA enables a similar flexibility for adapting the semantics of DSLs.

Bracha et al. [BU04] propose a mirror-based design for reflective OO languages that decouples reified language constructs from their implementation. In comparison, in REA, it would be possible to follow a mirror-based design for an embedded DSL. However, the Groovy MOP does not follow this design. If a language developer would follow the mirror design pattern in a non-mirror-based reflective language, the developer does not yield the advantages of mirror-based design. After all, what would be left are the addition complexity and implementation costs of mirrors. Therefore, REA does not use mirrors in Groovy.

Meta-object protocols can be distinguished with respect to when adaptations to objects are allowed: at *runtime* or *compile-time*. The next two paragraph elaborate on representatives of these two categories.

There are several languages with a *runtime MOP*, which allow adapting language semantics during the execution of programs. CLOS [KRB91] is a MOP for LISP, SchemeTalk [VR09] for Scheme, Joose[4] for JavaScript, Moose[5] for Perl, Groovy [KG07], and dynalang[6] is a generic MOP for dynamic programming languages on the JVM. In comparison, while the above languages are general-purpose languages, REA focuses on providing semantic adaptations of DSLs. Still adapting DSL semantics in REA is in the same vein as adapting GPL and OO semantics in MOPs. Like them, REA is inspired by the use of the open implementation principle in MOPs. The meta-protocols and reflection mechanism allow runtime adaptations like runtime MOPs. In contrast to the MOPs for these languages that have been implemented from scratch, in REA, developers do not implement a MOP from scratch—instead they embed a domain-specific meta-protocol on top of the MOP of the host language.

Further, there are *compile-time MOPs* that limit semantic adaptations to be applied before runtime, which leverages compile-time optimizations. Compile-time MOPs have been implemented as extensions for main-stream languages: there is OpenC++ [CM94] for C++; and there are OpenJava [TCKI00] and Reflex [Tan04] for Java. Compile-time MOPs allow special optimizations that remove parts of the costs of MOPs. They may even virtually eliminate all costs of indirections at compile-time, when no adaptation is required and those indirections are not required. In comparison, REA does not limit adaptation to compile-time only, the compiler of the host language is not aware of the MOP and optimizes indirections only with standard optimizations. Still in REA, developers can use familiar mechanisms for optimizations, such as *partial evaluation*. In REA, performance issues are addressed like [KRB91] proposes: to implement *sub-protocols* that take care of optimizations in the runtime MOP. REA implements such a sub-protocol by optimizing the execution of aspects.

[4]The Joose homepage: `http://code.google.com/p/joose-js/`.
[5]The Moose homepage: `http://www.iinteractive.com/moose/`.
[6]The dynalang.org homepage: `http://dynalang.sourceforge.net/index.html`

9.3.2. Composability of Languages

Many language approaches only support implementations of isolated languages, hence they do not support language composition. There are a few language implementation approaches that propose special composition mechanisms for language composition. Language composition has been most prominently discussed in the context of syntax composition. While the theory of syntax composition is mature, there is a lack of good theories and mechanisms for semantic composition. This thesis does not even attempt to give a complete overview of syntactic and semantic composition. Rather, to suggest all possible paths of research, it highlights the most important composition scenarios already available in various non-embedded approaches.

It is common to use some form of an import and inheritance mechanism in heterogeneous approaches that use special frameworks, meta-languages, or generators for composing languages [AMH90, HM03, KRV08, Par08]. Single inheritance (and single imports) support only language extensibility, but it is inadequate for language composition. Only approaches that support either multiple imports or multiple inheritance are adequate for language composition.

In comparison to REA, language inheritance and imports hierarchically decompose a language into sub-modules, but to compose languages there always need a common top-level module to compose the languages into explicitly one. Therefore, when composing languages with a meta-language, the developer must enumerate all possible points where expressions can flow from one language module to another module.

In contrast to this explicit composition, although REA also supports hierarchically decomposition, it inclines composing languages as equal peers. REA is closer related to *union grammars* and *agile grammars* [Cor06] that use implicit rules, make assumptions and use conventions when composing languages, and provide composition operators for composing grammars. In addition to grammar composition, REA provides semantic composition operators.

In addition to the related work discussed so far, there are only a few more classes of language approaches that support semantic composition, which are discussed in the remaining paragraphs of this section.

Attribute grammars [Knu68] have been found good for composing semantics [HM03]. Moreover, there are *source code* and *bytecode weavers* [Aßm03].

In *model-driven engineering* (MDE), languages are encoded as models. When models have interactions, they need to be composed by *model* and *meta-model weaving* [Dav03]. There is no fundamental difference between MDE and DSLs in REA, except that most model use visual languages and that REA focuses on composing textual languages. When composing two interacting models, similar composition mechanisms as they are used in REA need to be used.

In *generative programming* [CE00], developers use *templates* to implement language components, which can parameterized by other templates. Although developers can compose languages, all possible interactions must be explicitly defined by each language component beforehand.

In virtual machines (cf. Section 2.3.1, p. 25), in particular interesting is that developers can decompose single technical components of one language into several dimensions [SHH09]. VMs need a decomposition that does not only decompose the primary functionality for evaluation, but also they also need to compose secondary technical concerns, such as memory management. As VMs need to decompose all aspects of their functionality, support for *multi-dimensional* separation of concerns becomes interesting.

The finding of the approaches above and the open task to embed such mechanisms are a valuable source of ideas for future work.

9.3.3. Enabling Open Composition Mechanisms

This section compares POPART/REA to related language composition mechanisms.

Embedding of Crosscutting Composition: Reflection and MOPs have been used to implement AOP technology by adapting the semantics of objects. This approach has been followed by Sullivan [Sul01], *AspectS* [Hir01, Hir03], Lorenz and Kojarski [LK03, KL04], *Composition Filters* [BA01, BA04], Tanter [Tan04], *AspectLua* [CBF05], *Aquarium* [Wam08], and *GroovyAOP* [KG08]. Embedding AOP on top MOPs is in the same vein as embedding AO concepts in REA. However none of the other MOP-based solutions comes with a well-designed meta-aspect protocol that complements the MOP with a meta-interface that allows programmers to adapt the semantics of aspects. Moreover, adapting the semantics of aspect composition both at the application-level and at run-time has not been in the focus.

Kojarski and Lorenz [LK03, KL04] analyze the relation of reflection and AOP, they argue that AOP is a first-class computational reflection mechanism and therefore is at the same level as reflection. In comparison, REA/POPART uses its MAP as a first-class mechanism like a MOP.

Furthermore, other crosscutting concepts have been embedded into host languages.

In object-oriented programming, Havinga et al. [HBA08] have embedded object-oriented composition operators into a language called *Coop*. The composition operators enable extensibility and composability for objects. Developers can extend composition operators to embed new inheritance mechanisms for objects, which is not in the focus of REA. Unfortunately, Coop excludes meta-level extensions. In contrast, instead of providing extensibility and composability for objects, REA's language combiner focus on extending and composing languages as objects. Still, Coop is similar to REA with respect to embedding AO composition. Unfortunately, in Coop, aspects for DSLs are out of scope.

In *context-oriented programming* (COP), e.g. *ContextL* embed layers into CLOS, *ContextS* into Smalltalk, *ContextR* into Ruby [AHH^{+}09]. These works are in the same vein as embedding aspects, and in addition embedded layers have a good support for handling interactions [CD08], similar to handling interactions in REA/POPART.

In sum, on the one hand, embedding the above paradigm-specific mechanisms into REA would be interesting. On the other hand, it would be interesting to equip those paradigms a meta-level like the one in REA.

Aspect-Oriented Programming: To review the support for crosscutting composition, this section explores related work from AOP, which is complimentary to embedding AO concepts in POPART/REA.

Most prominently, in existing AO languages, there are variations how they handle interactions and what base language they support, whether it is general-purpose or domain-specific, as elaborated below.

Support for Composition Conflict Resolution:

There are numerous general-purpose aspect languages that allow handling interactions, which have been surveyed by Kniesel [Kni07]. Aspect interactions occur between general-purpose aspects that advise the same program. On the one hand, an interaction can be direct, i.e. it directly manifests on the same advised part, e.g. two aspect advise the same join point resulting in a semantic conflict. On the other hand, an interaction can be indirect, i.e. it influences another part, e.g. one aspect advises the data flow in a way that interferes with another aspect. Existing aspect languages provide a closed set mechanisms for handling aspect interactions of which the only most important are mentioned at this point.

The advice order mechanism provide resolution logic for *aspect interactions*, that change the composition semantics of aspects. Several approaches have been proposed, such as *logical meta-programming* [BMDV02], special *composition modules* that allow to specify logic to order advice [SVJ03], as well as providing *rules* to explicitly order, include, and exclude advice [Tan04, Tan06].

In particular interesting are two extensions that have been proposed for AspectJ.

Assaf et al. [AN08] proposes *DynamicAspectJ*, an extension to AspectJ that allows dynamic weaving and scheduling of aspects. In DynamicAspectJ, it is possible that the aspect developer reflects on part the aspect context. Advice can reflect on the aspects' context, e.g. at a shared join point, it is possible to reflect on the co-advising pointcut-and-advice. By using the following expression thisJoinPoint.context.applicablePAs, a POPART aspect's advice can reify a list of all co-advising pointcut-and-advice and manipulate the list for example to exclude a certain advice.

These approaches support to a certain degree the customization of aspect interaction semantics, however, conflict resolution is not the only semantic variation in aspect languages. Moreover there are two major problems with the above approaches. First, the advice ordering strategies cannot dynamically be changed based on dynamic context. Second, there is a technology break between the way aspects are specified and how conflicts are resolved. On the contrary, POPART allows to specify aspects and their resolution logic – as well as other semantic variations – in the same language by either using the primary interface or the meta-interface. Moreover, the MAP allows the user to define a resolution strategy for context-dependent aspect interactions.

To conclude and in comparison, the open mechanism for handling aspect interactions in REA/POPART has been inspired by related work. Moreover, REA introduces custom on-line techniques using its meta-aspect protocol, which developers can use to extend language semantics with domain-specific strategies for handling interactions.

Support for Extensible AO Semantics: Extensible aspect-oriented language infrastructures [SVJ03, Tan04, ACH+06, Tan06, KL07b] allow semantics to be adapted when building a special compiler or a special runtime.

The *abc* compiler has been used for defining new kinds of pointcuts [ACH+06] as extensions of AspectJ.

The *Aspect SandBox* (ASB) supports prototyping alternative AOP semantics and implementation techniques [MKD03], e.g., new kinds of pointcut designators and alternative weaving techniques. However, providing new concepts and semantics results in excessive changes throughout the interpreter, e.g., when a new type of join point is added [UMT04]. This is because ASB does not provide clear interfaces for controlling the underlying implementation strategy. There is no meta-interface available to programmers for tailoring AO semantics at the application level. In contrast to the extensible AO solutions above, POPART allows the adaptation of semantics at run-time by providing a meta-interface that can be extended using well-known object-oriented techniques.

Marot et al. [MW10] propose *OARTA* a declarative extension to the abc weaver that allows dynamic weaving and scheduling of aspects. OARTA extends the AspectJ language syntax so that a developer can name a pointcut, a so called *named pointcut*, which allows referring to it later on. It is possible that aspects weave on other aspects, which allows *advising* aspect-oriented concepts, such as *creation of aspect instances*, *named pointcuts*, and *execution of individual advice*. OARTA allows implementing similar functionality to what is possible with POPART's meta-aspects. However, the two approaches are very different in their concepts and techniques used. OARTA is implemented as an invasive extension of a static weaver. To weave on aspects, OARTA circumvents protective restrictions in the weaver. In contrast, in POPART/REA, aspects and meta-aspects are first-class objects that are homogeneously embedded into the host language. In OARTA, there are no means to analyze aspects or debug them. Unlike in POPART/REA, syntax extensions for specific domains are not addressed. Unlike POPART/REA support composing aspects, OARTA does not allow weaving into DSLs, since it uses the abc compiler with its AspectJ-like join point model.

Reflex [Tan04, Tan06], *JAMI* [HBA08], *MetaSpin* [BMN+06], and FIAL [BMH+07] are aspect-oriented runtimes that employ aspect-oriented meta-models that are open for alternative AO semantics at build-time. In their meta-models, interfaces for AO abstractions are provided that can be extended, e.g., with new pointcut designators or composition strategies. The four approaches differ in their objectives and implementation techniques which are, however, not relevant for a comparison to POPART. In general, they do not support changing the AO semantics at run-time.

Further, Tanter [Tan10] proposes to support *execution-levels* for aspects. Execution-levels address the problem of *infinite regression* when aspects have erroneous pointcuts that confuse the base- and the meta-level. He extends aspects with the concept of *execution-levels* following the *meta-helix architecture* by Chiba et al. [CKL96]. The execution levels help keeping track of base-level actions. The levels help not confuse base actions with actions inserted by aspects, which are at one higher level. Having information about the execution level allows restricting the visibility of action by allowing one execution level to reason only about the lower execution. Restricting the visibility of actions prevents that confused levels lead to infinite regressions by aspect advising the aspects at the same level. Embedding execution levels for aspect will be address by future work.

MetaBorg has been used to enable syntactic extensions for Reflex, in an approach called *ReflexBorg* [TT08]. ReflexBorg allows designing syntactic extensions to Reflex in order to implement DSALs with a concrete syntax. MetaBorg rewrite DSAL aspects in concrete syntax to Reflex syntax. But, in contrast to REA, since Reflex supports only weaving on Java with an AspectJ-like JPM, thus composing aspects into DSLs is not supported.

MetaAspectJ [ZHS04] is a macro extension to AspectJ, which allows meta-programming on aspects. MetaAspectJ support syntactic extensions to AspectJ. But, every generated aspect is an AspectJ aspect and must use the AspectJ semantics. Unlike in REA, dynamic aspects and weaving on DSLs is not supported.

To conclude and in comparison, the variation points in the design-space of AO languages found in related work have inspired the semantic variations of REA/POPART.

Support for Composition Conflict Resolution for DSALs: There are special aspect languages that support special conflict resolution when composing domain-specific aspect languages.

PluggableAOP [KL05] addresses the problem of integrating a (general-purpose) base language with a set of *third-party aspect extensions* that define component-based aspect mechanisms targeted for different domains and that are composed in order to provide a mechanism addressing multiple domains. They implement an interpreter-based weaver that allows to plug in aspect languages as language components. All language components are composed by whereby the plug-ins are chained in a *pipe-and-filter* architecture, which composes the language components constructs with wrapping semantics. However, they state that wrapping semantics cannot handle composition, where the aspect mechanism must reason about the interaction of all components. In comparison, pluggable AOP only composes a general-purpose aspect language with one DSAL. The authors themselves state that their mechanisms cannot handle complex compositions. In complex compositions, the composition mechanism must reason about all composed languages in order to compose them correctly. However, in the pipes-and-filters architecture of PluggableAOP, one weaving interpreter can only reason about subsequent interpreters in the pipe. In POPART, the meta-aspect protocol controls the complete evaluation of all composed aspect languages. Thanks to the abstract aspect composition process, there is a variation point in the MAP, at which developers can provide logic that can reason about interactions of any aspect regardless of what language the aspect originates from.

AweSome [KL07b] provides a similar plug-in architecture like PluggableAOP, but it follows a compiler-based approach. To compose multiple aspect languages, they propose to use an abstract weaving process that can control the weaving order at the level of pointcut-and-advice. They present a prototype implementation based on the *abc* extensible aspect compiler for static weaving. In comparison to POPART, the abstract weaving process has inspired the abstract aspect composition process in POPART's MAP. In addition, POPART supports runtime aspect composition and dynamic re-ordering of advice. Further, with AweSome, it is not possible to plug-in further aspect language after the weaver has been compiled and delivered to the end user. Consequently, the set of composed languages is closed. In contrast, POPART allows language developers to provide plug-ins and domain-specific scheduling strategies in the user domain.

9.3.4. Support for Concrete Syntax

The related work on concrete syntax in non-embedded approaches is complementary and interesting for the future work. Traditional built compilers and interpreters (cf. Section 2.1, p. 16) support concrete syntax well, but their heterogeneous nature, which hampers composition as desired in this thesis. Still, there are special non-embedding approaches that support composing concrete syntax, which are mentioned for the sake of completeness.

Language Workbenches: There are *language workbenches*. They prescribe a detailed and closed process on how to exactly implement a language's model and how to compose. First, SAFARI [CDF$^+$06] is a meta-tooling framework for generating language-specific IDEs in Eclipse. Second, IMP [CFSJ07] is another meta-tooling framework for Eclipse. Third, ANTLR-Works [BP08] is a development environment for the ANTLR parser generator, which focuses on providing tool support for ANTLR grammars. Forth, Spoofax [KKV09, KV10] generates IDE tools from language specified in the Stratego programming language. In comparison, language workbenches have a different focus than REA/TIGERSEYE, since they target at experts. TIGERSEYE allows to easily reuse existing models and APIs in the language implementation, such as Java libraries or existing EMF models, as demonstrated in [DWL09]. Language workbenches have a strong focus on tool support for end users, which is out of the scope of this thesis.

Generic Syntax Representations: Embedding requires a program representation that can be easily changed, extended, and composed. For the program representations that today's language infrastructures use, this is often not the case. Specifically, e.g. because ASTs are generated, or because the bytecode representation cannot be extended without extending the back-end. Still, there some representations that have some of these three properties that enable a comparable compositionality like in REA.

First, there are special syntactic encodings of program expressions: *attribute grammars* [Knu68], *S-expressions*, and *church encodings* [Chu40, PE88].

Second, there are ATerms – an extensible and composable AST representation where every *term* (AST node) has an open set of *annotations* to add new annotations to existing terms. With such annotation, different compiler phases can incrementally contribute to the AST. ATerms have means for pattern matching and term rewriting, which TIGERSEYE/REA uses to implement extensible and composable source code transformations.

In comparison, in REA, when encoding every expression as a method call in its abstract syntax using a prefix notation, or when mapping concrete syntax to this encoding, it is in the same vein as using the prefix notation of S-expressions. Using an encoding similar to S-expressions gives confidence that arbitrary expressions can be encoded. However, there are practical differences. In contrast, REA's program representation is rather bloated, and there is a rather large overhead since every expression is processed as a reflective method call. Therefore, future work will study also alternative representations in order to find ways how to optimize the current program representation.

9.3.5. Enabling Pluggable Scoping

There is extensive research in non-embedded-based approaches, which is complementary to the embedding of scoping. The literature on scoping mechanism for interpreters and compiler is complimentary to REA.

W.r.t. pluggable scoping in interpreters, there are interpreters with first-class environments for scoping language constructs, with which REA is in the same vein. First, in [IJF92, LF93], Friedman et al. adapt the scoping strategy of an interpreter using an interpreter that adapts the variable look up using reflection. Second, REA is in the same vein as scoping in [AS96], which structures like REA environments a map that is inter-connected with other environments. Third, Tanter [Tan09] uses first-class scoping strategies to adapt the scope of language constructs, such as variable bindings, aspects (similar to [PGA01, PAG03, AO10]) and layers (similar to [CH05, HCH08, Tan08, CH05, HPSA10]). Forth, REA's dynamic scoping and activation of aspects in DSLs are in the same vein as dynamically scoped aspects in interpreters for DSLs, such as AO4BPEL [CM04, Cha08].

For scoping in compilers, there is the compiler construction system *Eli* [KW91], which systematic supports implementing pluggable scoping as a separated compiler phase.

Such non-embedding-based implementations of scoping differs mostly in that the scoping implementation is completely controlled by the creator of the language, but developers cannot adapt the scoping strategies in the user domain. Whenever changing the scoping strategy, the creator has to recompile the language infrastructure, which makes user-extensions impracticable.

9.3.6. Enabling Pluggable Analyses

This thesis does not even attempt to summarize the literature on analyses in non-embedded approaches, which is complementary. Their literature can be distinguished on what representation the analysis performed, namely models, source code, ASTs, and bytecode.

Models: First, there are analyses of models, which allows sophisticated analyses at a high level of abstraction. There are numerous representations, such as *finite state machine* models, *petri nets*, *graph-based models*, and domain-specific models. Further, there are various approaches on model checking, simulation, and verification. In some approaches, one can plug multiple analysis in parallel, but it is complicated to compose multiple analyses. This work is complimentary to REA, this thesis refers the interested reader to look at one of the surveys [JM09].

Source Code: Second, source code analysis processes programs at their textual representation only. Simple lexical analyses can be implemented at the source code level, regular expressions are the best example of such analysis. However, source code is almost always not the right abstraction level for analyses. In particular, source code analyses does not cover semantic analyses, therefore AST analyses are often the preferred choice.

Abstract Syntax Trees: Third, traditionally program analysis is implemented by analyzing the AST of a program. Most prominently, the visitor pattern is used to compose several analysis of an AST. ASTs are a very good representation for analyses. But in many language architectures, the AST is only accessible at compile time, therefore it is not possible to implement dynamic analyses.

In contrast, AST-based interpreters [Rub] expose the AST so that developer can use inspection capabilities to embed dynamic analyses. Similarly, REA supports inspecting the abstract syntax at runtime, because the evaluation of each expression is mapped to a reflective method call.

Bytecode: Forth, there is a large body of work on bytecode analyses, such as for bytecode verification and security. It is a relatively good representation for many analyses, and bytecode allows portable analysis between different languages and platforms. However, for high-level analyses, the bytecode representation is often too low-level. In bytecode, most of the control structures have been removed and it is hard to retrieve them back from the bytecode.

9.3.7. Enabling Pluggable Transformations

The work on transformations in non-embedded approaches is complimentary to embedding. Nonetheless, non-embedding approaches have a different focus than transformations for in embeddings. Non-embedding approaches often facilitate transformations that embedding approaches often can reuse from their host compiler, such as *inlining* optimizations or β-reduction. While embedding compile-time optimizations, such as inlining, has been studied by Seefried [SCK04], adopting dynamic optimization techniques as just-in-time compilers [AFG$^+$00] use them, have been out of scope.

Despite the different focus, there are two interesting issues. First, whether in the approach, the source and target language/models are causally connected. Second, whether the approach allows interactions between transformations. These two issues are discussed in the following.

Causally Connected Source and Target: Most non-embedded approaches support transformations, but often the source and the target language or model are not causally connected.

In model-driven engineering, REA is in the same vein as executable models and causally connected models, such as *models-at-runtime* [NB09].

In compiler approaches, a similar mechanism is *debugging* and *structure preserving compilation* [BHMO04].

Interactions: There are only a few transformation approaches that support modular transformations that allow interactions.

In model-driven engineering, there is ongoing work to enable rule-based transformations (e.g. QVT) and to control interaction between rules. In particular interesting are rule-based transformation approaches in which rules compose with function composition semantics i.e. *confluence* [Vis05]. In such systems, the order of a transformation not relevant.

In compiler approaches, it is in particular interesting how JastAdd [HM03] combines transformations on the AST. JastAdd uses aspects to add attributes to an attribute grammar, whereby attributes are lazily calculated and automatically scheduled on demand.

10. Case Studies: Review of the Support for the Desirable Properties

This section qualitatively evaluates REA's support for language evolution by means of several case studies. Indeed REA suites implementing extensions, compositions, interactions, and reflections on language embeddings. For the integration of DSLs into current software, it is important that REA's concepts can be supported by other host languages. Because REA makes little assumptions about the language developer's skills and available host language features, REA's concepts can be applied by most developers and its techniques can be adopted with little changes in other host languages.

For reviewing the quality of the techniques that this thesis proposes, this section studies how good REA's core and its extension addresses the desirable properties from Section 3. For each property, the review demonstrates that REA supports the desirable property by implementing its scenarios. To determine the quality of the support, the review investigates the general support for the property by the current concepts and draws conclusions.

10.1. Extensibility

This section shows how the REA supports all of the proposed evolution scenarios.

10.1.1. Adding New Keywords

To recapitulate, in Section 3.1.1 (p. 31), we have motivated adding keywords to existing languages. In Section 4.2.1 (p. 84), we already have discussed the principle support for one language extension using language polymorphism. Still, it remains to be shown how good language polymorphism supports multiple incremental extensions.

Therefore, in this section, we review the support for multiple language extensions. The section first presents the implementations of several extensions. Then, the support for continuous language evolution is discussed, whereby we identify limitations of hierarchical decomposition using inheritance. This section shows that when using single inheritance to evolve languages, inheritance suffers from the same limitations for language inheritance as it does for ordinary class inheritance.

Example Scenarios: Consider extending SIMPLELOGO with the three example extensions EXTENDEDLOGO, CONCISELOGO, and UCBLOGO that have been proposed in Section 3.1.1 (p. 31). Although the three extensions are admittedly similar and each extension requires the same techniques to be implemented, still, we present the implementation of multiple extensions to demonstrate the limitations of language polymorphism for continuous language evolution.

As Section 5.4.1 (p. 125) has already discussed the extension for ExtendedLogo, in Listing 5.5, this section only sketches the two other extensions.

As the second example extension, consider implementing ConciseLogo, which adds shortcut commands such as fd, bd, rt, and lt. Listing 10.1 shows the language interface and implementation of ConciseLogo that defines such shortcut commands. The interface IConciseLogo declares additional keyword methods with shortcuts names. Each of these shortcut keyword methods has an identical signature like its verbose versions, except its shortened method name, as sketched by Listing 10.1 lines 9–11. The façade class implements the shortcut methods, so that every shortcut command (e.g. fd, line 9–11) calls its corresponding verbose command (e.g. forward, line 10), as described in the following mapping: fd$\rightarrow$forward, bd$\rightarrow$backward, rt$\rightarrow$right, and lt$\rightarrow$left.

```
 1  interface IConciseLogo extends ISimpleLogo {
 2    void fd(int n);
 3    void bd(int n);
 4    void rt(int a);
 5    void lt(int a);
 6  }
 7
 8  class ConciseLogoInterpreter extends SimpleLogoInterpreter implements IConciseLogo {
 9    void fd(int n) {
10      forward(n);
11    }
12    ...
13  }
```

Listing 10.1: Implementation of shortcut commands in ConciseLogo

As the third example extension, consider implementing a subset of UCBLogo that adds the repeat abstraction operator. Listing 10.2 shows an excerpt of the language interface and implementation of UCBLogo. The IUCBLogo interface (lines 1–4) declares a repeat method (line 2) that takes the number of iterations as the first and a closure as the second parameter. Since there is only one parameter to the abstraction operator in addition to the closure, the developer does not need to use a HashMap for the parameters, but the developer can use a primitive parameter. Because the last parameter is a closure, repeat expressions can pass their command sequence in the closure parameter outside the brackets, such as repeat(i){...}. The language façade UCBLogoInterpreter (lines 6–14) implements IUCBLogo with the declared repeat method (lines 7–12). Inside the implementation of repeat, as for all abstraction operators, we must take care that keywords that are nested inside the abstraction operator's body are dispatched correctly, by setting the delegate of the closure body to the bodyDelegate. Next, the implementation of repeat calls the body for n times.

Review: Language polymorphism enables the reuse and incremental extensions in the different levels in the architecture. At the language interface level, language developers can reuse the

```
interface IUCBLogo extends ISimpleLogo {
  void repeat(int n, Closure body);
   ...
}

class UCBLogoInterpreter extends SimpleLogoInterpreter implements IUCBLogo {
  void repeat(int n, Closure body) {
    body.delegate = super.bodyDelegate;
    for (int i=0; i < n; i++) {
      body.call ();
    }
  }
   ...
}
```

Listing 10.2: Implementation of the repeat command in UCBLOGO

interface of the base language, whereby they reuse parts of the abstract syntax definition, similar to what is supported by *grammar inheritance* [AMH90, KRV08, Par08]. But in contrast to only inheriting the syntax, additionally, the language polymorphism allows also reusing the execution semantics. At the language façade level, developers can reuse the method implementations of the base language's façade class, whereby the reuse parts of the syntax-to-semantics binding of the base language keywords. At the language model level, developers can inherit classes of the language model, whereby they reuse parts of the evaluation protocol of the base language.

With language polymorphism, REA supports *late binding* of syntax to its semantics. For late bound semantics, the meta-object can override the binding for one or several keywords, façades, and languages. Hence, REA fosters the *reuse* of language implementations for extensions, since parts of the language implementation can be refined selectively.

Further, the late binding enables safe extensions for which the language interface guarantees backward compatibility with old language versions as long as all semantic invariants are met. With the language polymorphism mechanisms and late binding of syntax and semantics, REA has full support for extensibility for new keywords and conservative extensions (adding keywords, conservative extensions: ●).

Further, with its meta-level, REA supports late semantic adaptations in the user domain (late semantic extensions: ●).

10.1.2. Semantic Extensions

10.1.2.1. Conservative Semantic Extensions

In Section 3.1.2.1 (p. 34), we have discussed the need for conservative semantic extensions of a language. Therefore, this section reviews how language polymorphism enables conservative semantic extensions by overriding existing keywords at the façade level and domain operations at the model level.

Example Scenario: To exemplify REA's support for conservative semantic extensions, we discuss implementing CONCURRENTLOGO proposed in Section 3.1.2.1 (p. 34). To provide support for concurrency, a language developer must associate turtle abstractions with threads that execute their commands. For this, COMPLETELOGO needs to refine several keywords.

Because CONCURRENTLOGO is a conservative semantic extension, it must not extend the ICompleteLogo language interface and preserves compatibility. The language developer only needs to extend the façade by overriding methods. Overriding an existing method binds the given syntax to new extended semantics. For example, in the case of CONCURRENTLOGO, the CONCURRENTLOGO extensions rebind part of the COMPLETELOGO keywords to a concurrent execution semantics. Still, the new façade implements the original ICompleteLogo language interface.

The subclass ConcurrentLogoInterpreter in Figure 10.3 adds two fields, namely turtles (line 2), which stores a list of all turtles, and threadToTurtle (line 3) that stores a mapping, which maps a thread to the turtle that the thread is executing.

Further, the subclass overrides several keyword methods. It overrides the method turtle (lines 5–14) to associate turtles with a thread that executes their commands, adds the turtle to the turtles list, and then sets the delegate of the turtleBody. However, the method does not directly call the turtleBody as in the super class. Instead, it wraps the turtleBody closure into a Thread (line 10). Next, it associates the thread with the turtle, by storing this information into the threadToTurtle map. To avoid that one turtle overpaints the drawing of another turtle with a different color, before starting the thread, the method checks that all currently running turtles have the same paint color, otherwise the current thread will wait until all turtles with a different pen color have finished painting. Finally, the method starts the thread (line 13), which will call the closure. Moreover, the subclass overrides all other drawing commands (e.g., forward), such that every command paints with the right turtle, when the command is executed in a thread, as exemplified in the overridden version of forward (lines 16–20).

Review: We first review the support for one conservative extension in the example scenario and then its general support for this property.

In the example scenario, we discuss what part(s) of a language implementation the developer needs to extend for a conservative extension. To implement a conservative extension, it is sufficient to inherit the base language façade and selectively override the relevant keyword methods. For backward compatibility, it is important that developers must not extend the language interface and that not all keywords need to be redefined. For example, CONCURRENTLOGO only overrides the keywords turtle and forward, and therefore it is a conservative extension and preserves backward compatibility.

To review the general support for conservative semantics extensions, we elaborate which parts of the architecture can be extended. Since a conservative extension aims to preserve the existing syntax of the language, unsurprisingly, the language interface level does not need to change. Only the façade and model level are subject to extensions.

The language façade level is central for implementing conservative extensions. The previous section has already discussed its role that fosters the *reuse* parts of an embedding implementation. While language polymorphism enables reusing parts of the syntax-to-semantics binding

```
class ConcurrentLogoInterpreter extends CompleteLogoInterpreter implements ICompleteLogo {
  List<Turtle> turtles ;
  Map<Thread,Turtle> threadToTurtle = new HashMap<Thread,Turtle>();
  ...
  void turtle (HashMap params, Closure turtleBody) {
    turtle = new Turtle(name, new Color(color));
    turtles .add(turtle );
    turtleBody.delegate = super.bodyDelegate;
    ...
    Thread closureInThread = new Thread(turtleBody as Runnable);
    threadToTurtle.put( turtle ,closureInThread);
    ... //check that currently running turtles have the same paint color, otherwise wait
    closureInThread.start ();
  }

  void forward(int n) {
    Thread thread = Thread.currentThread();
    Turtle perThreadTurtle = threadToTurtle.get(thread);
    perThreadTurtle.moveForward(n);
  }
  ...
}
```

Listing 10.3: Implementation of concurrent drawing for multiple turtles in CONCURRENTLOGO

from the base implementation, inheritance also allows overriding parts of this binding. For example, the above conservative extension reuses parts of the base façade implementation, but for concurrent drawing, all move command methods need to be overridden. This way, the developer adapts the syntax-to-semantics binding of these keywords to bind to concurrent execution semantics. Still, when overriding methods, the extension must not violate the base façade class's semantic pre-, post-conditions, and invariants.

In contrast to façades, sometimes the language model level does not need to be modified for conservative semantic extensions at all.

10.1.2.2. Semantic Adaptations

Section 3.1.2.2 (p. 35) motivated the need for semantic adaptation of languages. To cope with changing requirements in the user domain, a developer can implement a semantic adaptation for individual users at the meta-level.

Example Scenario: Reconsider implementing the support for collision detecting for CONCURRENTLOGO with the CollisionPreventingTurtleMetaClass that was already sketched in Section 5.4.2 (p. 125).

Listing 10.4 elaborates the implementation of a special meta-object for the Turtle class. The class CollisionPreventingTurtleMetaClass overrides invokeMethod in order to intercept all method calls to Turtle objects. The meta-object only changes the calls to the moveForward and the moveBackward method (line 17). When one of these two methods is called, the meta-object checks for possible collisions (line 20) by calling a helper method (line 10) that returns true if executing the commands would result in a collision. In case of a possible collision, the meta-object delays[1] the execution of the move operation to prevent the collision (line 21). Otherwise, the meta-object proceeds the call (line 24) to its super class (the default meta-object) that immediately executes the call.

```java
 1 class CollisionPreventingTurtleMetaClass extends MetaClassImpl {
 2
 3   IConcurrentLogo interpreter;
 4
 5   CollisionPreventingTurtleMetaClass(Class theClass, IConcurrentLogo interpreter) {
 6     super(theClass);
 7     this. interpreter = interpreter ;
 8   }
 9
10   boolean isTurtleFacingACollision(Turtle turtle , List<Turtle> otherTurtles) { ... }
11
12   Object invokeMethod(Class sender, Object object, String methodName, Object[] originalArguments, ...) {
13     assert (object instanceof Turtle);
14     Turtle turtle = (Turtle)object;
15     Object result;
16     synchronized (interpreter) {
17       if (methodName.equals("moveForward") || methodName.equals("moveBackward")) {
18         List<Turtle> turtles = new LinkedList<Turtle>(interpreter.getTurtles ());
19         turtles .remove(turtle);
20         while (isTurtleFacingACollision( turtle , turtles )) {
21           Thread.currentThread().sleep(100);... //preventing a collision by waiting at the current position
22         }
23       }
24       result = super.invokeMethod(sender, object, methodName, originalArguments, ...);
25     }
26     return result;
27   }
28 }
```

Listing 10.4: Excerpt of the meta-level extension for collision detection

Review: To implement the semantic adaptation, a developer only needs to implement a special meta-object that adapts the semantics of the default implementation of CONCURRENTLOGO. Indeed, the developer does not have to modify the code of the default language implementation, neither the façade nor the model has to be changed. Because the domain type Turtle defines the methods moveForward and moveBackward as two implicit extension points inside the language model, the developer who adapts the language can override the methods. These extensions points open up the language implementation providing a well-defined interface. Using the extension mechanism, language developers can later provide new variations. The language developers deliver the resulting open implementation of the language to the end user domain.

[1]The turtle is set into a waiting position, i.e., it puts its pen up, it could additionally step aside, and waits until the turtles passing by are out of its way, then it would steps back to the old position and continue drawing.

Note that the developer who provides the extensions and the developer who sets up the extension can be expected to be different people. The first developer who implements the meta-level extension needs to have a good understanding of the Groovy MOP and needs to know the language implementation details. This role can be either a language developer in the user domain or an end user with advanced skills. The second developer who set up the meta-level extension only needs to know how to replace meta-objects. Again, this can be either a language developer or an end user. To enable the new meta-object for executing a program under alternative semantics, the second developer simply needs to set up the meta-level extension before evaluating the program.

To review the general support for semantic adaptations, we elaborate what parts of the architecture can be adapted by meta-level extensions. To cope with changing requirements in the user domain, the language developers of the default language implementation needs to built-in the implicit extension points into the language implementation. Only if in the language implementation a corresponding method or respectively field has been declared, it can be extended at the meta-level.

The developer can adapt all parts in the base architecture. Specifically, the developer can use meta-object to adapt all levels—the interface level, the façade level, and the model level. The base levels do not need to be invasively modified for a late semantic adaptation. Only new code for the meta-level needs to be provided in order to realize a late semantic adaptation.

The support for late semantic adaptation is similar to late semantic adaptation in other software systems. The meta-level extensions support *planned reuse* [Bos00], i.e., the extension points have been explicitly decided by the providing language developer. When the methods and fields in the levels of the architecture have not been designed with possible extensions in mind, it might be possible that the adapting developer cannot provide the right code for a concrete semantic adaptation.

Meta-level extensions enable a great flexibility for language developers and end users, but one can argue that meta-level extensions are too dangerous to be used by end users. Since every class, method, and field in a language implementation can be adapted, a meta-level extension may also corrupt the existing language semantics resulting in incorrect semantics. Unfortunately, in Groovy, there is no special mechanism available that helps with enforcing that meta-level extension must respect possible semantic invariants. Therefore, REA needs to address the problem of missing guarantees for semantic adaptations.

To address the problem of missing guarantees for semantic adaptations, this thesis argues that one can provide generic and specific abstractions for semantic adaptations. First, a language developer can provide pre-defined meta-objects that encompass extension points themselves. Other language developers can extend these meta-objects, whereby they rely on the abstractions built into these meta-objects. Effectively, in this case the language developers who extend pre-defined meta-objects, are implementing a *sub-protocol* [KRB91] of the meta-protocol. Such a sub-protocol provides a specific abstraction on what a developer can adapt with this sub-protocol. Second, a language developer can use aspects instead of meta-objects to adapt the base levels in the architecture. In this case, the generic abstractions built into pointcut and advice languages restrict what can be modified by language developers and users, which limits adaptation to well-defined join points. Third, a language developer can provide end users with a DSL for implementing high-level semantic adaptations with a domain syntax. The DSL's syntax and abstractions provide a domain guarantees for meta-level adaptations. When providing such abstractions on top of the architecture, there a well-defined guarantees. Those guarantees tame the meta-objects in the architecture, so that semantic adaptations cannot lead to incorrect program semantics.

While meta-level extensions provide great flexibility, there is an important limitation: there may be only one meta-level extension for a particular language instance at a time. For example, we could implement the validation of the domain-specific constraint in a meta-object in a modular way. However, in real scenarios, there could be multiple constraints to be validated for the same language and its domain objects. For example, we do not only want to prevent collision but we would like to prevent turtle robots doing overtime or running out of energy. When multiple of such constraints need to be enforced on the same language constructs and on the same domain object, their enforcement logic needs to be combined. But because the meta-link can point only to one meta-object, only one meta-object (or resp. enforcement point) is allowed to be used for one object at a time. Unfortunately in this setting, it is hard to implement one meta-object in a modular way, because the enforcement logic of all constraints is tangled with each other. There is a need for a more composable mechanism.

To overcome the issue that meta-objects are not modular, in REA, it is possible to use aspect-oriented programming for semantic adaptations. Since REA is homogeneous and causally connected, a language developer can deploy an aspect on a language interface, façade, or model class, in the same way, as an user can deploy an aspect on an application-level class. Instead of implementing a tangled meta-object, in REA, the developer can implement multiple modular aspects that perform the enforcement. To enforce a constraint, each aspect defines a set of pointcut-and-advice. The pointcut matches the relevant enforcement points at which the aspect adds the advice, which contains the enforcement logic. While each constraint is localized in a separate aspect module, REA composes all aspects with the base level at runtime, whereby aspects at the language level have the full support of dynamic deployment and scoping of aspects, as well as full quantification capabilities including control-flow sensitive predicates and if-pointcuts.

10.2. Composability of Languages

REA supports compositionality of languages because they consist of a set of first-class objects and there are language combiners that can reason about those first-class objects in order to compose them, whereby preventing, detecting, and resolving interactions between those objects.

10.2.1. Composing Languages without Interactions

To recapitulate, in Section 3.2.1 (p. 37), we have motivated composing languages without interactions. We already have discussed the principle support for composition of independent languages using the BlackBoxCombiner and its implementation in Section 4.2.3 (p. 102). Therefore, this section only sketches composing multiple independent languages.

Example Scenario: For the review, reconsider composing CompleteLogo and Functional into the composite language FunctionalLogo, as it has been already discussed in Section 5.4.3.1 (p. 128).

Review: When languages do not have interactions in the specifications, such as CompleteLogo and Functional, the BlackBoxCombiner can be used to compose the two languages, without requiring language developers to provide additional code for the language composition. Each language component remains encapsulated (without interactions: ●), which is advantageous. On the one hand, the BlackBoxCombiner fosters the reuse. On the other hand, the language developer is supported when composing the languages, which is because unnoticed syntax conflicts are automatically detected. This encapsulation is a limited control for interactions.

10.2.2. Composing Languages with Interactions

10.2.2.1. Syntactic Interactions

In Section 3.2.2.1 (p. 39), we have motivated composing languages with syntactic interactions. When languages have syntactic interactions, the language developer should be supported by automatically detecting and handling those interactions. Therefore, this section reviews the BlackBoxCombiner's support for dealing with syntax conflicts.

Example Scenario: Consider the program in Listing 10.5 that tries to compose the two syntactically conflicting languages Functional and Structural, which both declare the ambiguous keyword define. But, when a developer tries to compose the two conflicting languages in line 2, the BlackBoxCombiner raises an exception that states the signatures of conflicting keyword methods.

```
1 def structural  = new Structural(); def functional  = new Functional();
2 def combinedLanguage = new BlackBoxCombiner(functional,structural); //raises an exception
3  ...
4
5 Closure program = {
6    ...;  define  (...);   ...
7 }
8 program.delegate = combinedLanguage;
9 program.call();
```

Listing 10.5: A program excerpt that tries to compose Functional and Structural with the BlackBoxCombiner

Review: The BlackBoxCombiner can automatically detect and handle syntactic conflicts between constituent languages in the a composition. Possible syntactic conflicts are detected and handled at composition time, when the developer tries to compose the languages, thus conflicts are detected before evaluating the program closure. For example, in Listing 10.5 (p. 219), the syntactic conflict is detected in line 2 before even reaching the program closure in lines 5–7). Hence, developers are patronized from ambiguous language compositions if they disregard syntax conflicts while composing languages (syntactic interactions: ●).

To detect the syntax conflict, before executing the program closure, it is crucial that the BlackBoxCombiner can use the meta-level of the languages to detect the conflicts. Specifically, the BlackBoxCombiner uses introspection to check whether the set of method signatures from all façade objects passed to the BlackBoxCombiner's constructor are disjoint. Since conflicting signatures between language façades are reported, developers can easily find the syntax conflicts in their corresponding language interfaces and façades implementations.

Still, there are two limitations. First, the BlackBoxCombiner assumes that syntactic conflicts cannot be resolved and always raises an error, which ensures safe compositions, but it does not support resolving syntactic conflicts. Second, unfortunately, syntactic conflicts are not detected and reported at compile-time.

10.2.2.2. Semantic Interactions

In Section 3.2.2.2 (p. 40), we have motivated composing languages with semantic interactions. For such compositions, one needs a combiner that is more liberal than the BlackBoxCombiner, therefore, the language developer can use e.g., the LinearizingCombiner (cf. Section 4.2.3.2, p. 107) that does not prevent interactions between languages. When composing language with this combiner, on the one hand, the language developer needs to extend the languages that are influenced by the interaction. On the other hand, each language implementation should stay encapsulated. Therefore, this section reviews the LinearizingCombiner's support for dealing with semantic interactions, dependencies and conflicts.

Example Scenario: Consider implementing the composition of FUNCTIONAL, STRUCTURAL, and CONSOLE that provides the operation writeln to print out string representations of either functions or data types, but for this, writeln has to look up in the correct environment, as argued in Section 3.2.2.2 (p. 40). First, we discuss the implementation of CONSOLE. After that, we discuss how to compose this implementation with other language implementations.

Listing 10.6 presents the language façade of Console whose functionality depends on having access to the environments of the two other languages FUNCTIONAL and STRUCTURAL. To allow the Console accessing their environments, one has to pass references to their façade to one of the constructors of Console in lines 5–7 that store references to those façades. To expose access to the environments, the developer has extended both façades with a getter for retrieving the environment via the method getEnvironment(), which is declared by the extended interfaces IFunctionalEnv and respectively IStructuralEnv. In lines 9–17, the implementation of the writeln operation looks up the object in one of the environments using the name identifier and prints its string representation. Note that the code uses Groovy's "?"-operator to safely navigate when null values are present.

```
class Console extends Interpreter {
  IFunctionalEnv functional;
  IStructuralEnv structural;

  public Console(IFunctionalEnv functional, IStructuralEnv structural) { ... }
  public Console(IFunctionalEnv functional) { ... }
  public Console(IStructuralEnv structural) { ... }

  public void writeln(String name) {
    if (functional?.getEnvironment()?.get(name) != null) {
      println "FUNCTION: "+functional.getEnvironment().get(name).toString();
    } else if (structural?.getEnvironment()?.get(name) != null) {
      println "TYPE: "+structural.getEnvironment().get(name).toString();
    } else {
      println "UNKNOWN OBJECT $name";
    }
  }
}
```

Listing 10.6: The implementation of the CONSOLE façade that depends on FUNCTIONAL and STRUCTURAL

Next, consider implementing the compositions of FUNCTIONAL with CONSOLE and STRUCTURAL with CONSOLE. Listing 10.7 shows how the LinearizingCombiner can be used to compose the two combinations of the languages. When evaluating the two program fragments (lines 5–8 and lines 16–19) , LinearizingCombiner does not prevent possible interactions. Therefore, the lines 7 and 18 in Listing 10.7 can print out the objects from the two composed languages as the syntax conflict is resolved in both cases.

Review: The LinearizingCombiner enables dependent compositions by allowing possible interactions between the languages. The language developers can extend the façades of the constituent languages to allow them to interact with each other. Although meta-object make adaptation possible, each language component remains weakly-encapsulated (semantic interactions: ●).

So far, there are still limitations with handling semantic conflicts in compositions using the LinearizingCombiner. Although, each interaction reflects in the interfaces of the façades, in general, there can be data flow between façades. When passing an object from one language to another, there can be subtle semantic interactions that do not reflect on the interface level and that are hard to detect. While the LinearizingCombiner prioritizes the expression types of languages to prevent syntactic conflicts, it does not prevent any semantic conflicts. Thus, they still need to be checked by the developer.

10.3. Enabling Open Composition Mechanisms

In REA, extending existing composition mechanisms is possible, since language combiners are first-class objects that can be extended and compose with each other using OO composition. When there are special requirements and assumptions in a composition scenario, language developers can easily define new language combiners in REA. They can extend existing composition

```
DSL functional = new Functional();
DSL console = new Console(functional)
DSL functionalConsole = new LinearizingCombiner(functional,console);

Closure program = {
  define(name:"sort") {  list  −> ... }
  writeln("sort");  // prints  "FUNCTION: name 'sort' parameters: 1 ..."
}
program.delegate = functionalConsole;
program.call ();
 ...
DSL structural  = new Structural();
console = new Console(structural)
DSL structuralConsole = new LinearizingCombiner(structural,console);

program = {
  define(name:"sort"){  ...  }
  writeln("sort");  // prints  "TYPE: name 'sort' slots:  kind=String  ..."
}
cl2.delegate = structuralConsole;
cl2.call ();
```

Listing 10.7: Two programs using the LinearizingCombiner for a dependent composition

mechanisms and reuse common parts of the composition logic. Fortunately, the costs associated with defining a new mechanism are rather low compared to implementing a new composition mechanism from scratch. We discuss handling different conflict cases, for each of these, an open composition mechanism is provided in the following.

10.3.1. Open Mechanisms for Handling Syntactic Interactions

When the syntax of constituent languages is non-orthogonal in a composition, the language developer can extend existing combiners for handling the syntactic interactions. In the following, we review how far existing combiners can be extended w.r.t. guaranteeing absence of syntactic conflicts (in Section 10.3.1.1), by resolving them by renaming conflicting keywords (Section 10.3.1.2), and by defining priorities between languages (in Section 10.3.1.3).

10.3.1.1. Generic Mechanism for Conflict-Free Compositions

Section 3.3.1.1 (p. 42) has motivated to prevent syntactic conflicts when independent languages are composed. Section 10.2.1 has reviewed the support of the BlackBoxCombiner to allow composing independent languages, such as SIMPLELOGO and FUNCTIONAL. In contrast, Section 10.2.2.1 has reviewed the support of the BlackBoxCombiner to prevent erroneous composition of syntactically conflicting languages, such as FUNCTIONAL and STRUCTURAL. It remains to be shown how the BlackBoxCombiner can be used to prevent special syntactic conflicts that are specific to a certain composition scenario, such as context-sensitive syntax conflicts.

Example Scenario: Reconsider composing FUNCTIONAL and COMPLETELOGO such that it is forbidden to use the COMPLETELOGO keywords that have side effects in a function body, as motivated in Section 3.3.1.1 (p. 42). Listing 10.8 shows a program that has an context-sensitive syntax error, since in the "square" function's body in line 10, the forward keyword is used.

```
 1  define(name:"dble") { length ->
 2    return length * 2; //okay to define a function without Logo keywords
 3  }
 4
 5  turtle (name:"Triangle", color:red) {
 6    fd 100; rt 120; fd 100; rt 120; fd 100; rt 120; //okay to use Logo keywords outside of functions
 7  }
 8
 9  define(name:"square") { length ->
10     ...; forward length; .... //context−sensitive syntax error (Logo keywords not allowed in functions)
11  }
```

Listing 10.8: A program excerpt with a context-specific syntax conflict

To detect and handle such context-sensitive syntax conflicts, a language developer needs to specialize BlackBoxCombiner so that it prevents those conflicts in a certain lexical region. Listing 10.9 presents a custom combiner that prevents using COMPLETELOGO keywords in functions by refining the define keyword in lines 6–10. The combiner's define method forwards to the original define method of the Functional façade in order to store the function in the environment. In addition, in line 8, it changes the delegate (i.e., the bodyDelegate) for the function body by wrapping it into a transformation[2] that filters out the keywords defined in ICompleteLogo. Further, in line 10, it adapts the resolveStrategy of the function body to Closure.DELEGATE_ONLY, so that the closure resolves contained keywords only over the filtered delegate, and not over its owner[3].

Review: By inheriting from the BlackBoxCombiner, it is easy for developers to implement the custom combiner to prevent context-sensitive syntax conflicts. Because the custom combiner inherits from BlackBoxCombiner, the subclass reuses the common composition logic, which fosters reuse. In addition, the subclass prevents context-sensitive syntactic conflicts only by refining the implementation of those abstraction operations for which nested expression types need to be restricted (e.g., in the above scenario, only for one abstraction operator method).

[2]The KeywordFilterTransformation constructor takes two parameters. The first parameter is a façade object to be filtered. The second parameter specifies a set of language interfaces that each defines a set of method signatures to be filtered out. At runtime, the KeywordFilterTransformation intercepts every method call. In case the intercepted method's signature is contained in one of the specified interfaces, the transformation does not forward to the call to the wrapped façade. Otherwise, the keyword method call is simply proceeded to the wrapped façade. Details of how to implement transformations are elaborated in Section 10.7

[3]When setting up the resolveStrategy to Closure.DELEGATE_ONLY, this has the effect that keywords in the functions are not resolved over the closure's owner, which may be allowed to use COMPLETELOGO keywords.

```
public class MyContextSensitiveCombiner extends BlackBoxCombiner {
  IFunctional functional; ...
  public MyContextSensitiveCombiner(IFunctional functional, ISimpleLogo logo) {
    super(functional,logo); this.functional = functional; ... }

  public void define(HashMap params, Closure body) {
    functional.define(params, body);
    body.delegate = new KeywordFilterTransformation(body.delegate, [ICompleteLogo]);
    body.resolveStrategy = Closure.DELEGATE_ONLY;
  }
}
```

Listing 10.9: A custom combiner that filters the Logo keywords inside function bodies

In case of a function body being evaluated that contains an error, the keyword method will not be found in the function closure's delegate because it is filtered out via the KeywordFilterTransformation.

Note that the custom combiner is limited to compose the two languages. For other context-sensitive syntax conflicts and different languages, in REA, the developer has the possibility to implement custom combiners.

In general, it can be considered a design error to use the BlackBoxCombiner for interacting languages, which indeed are not black-box. Still, semantic interactions that originate from erroneous design of language components are not detected and prevented by the BlackBoxCombiner. Therefore, one could argue that it is disadvantageous that, unlike functional embedding approaches, the BlackBoxCombiner does not give any automatic semantic guarantees that composition are semantically conflict-free, since theoretically language components could have side-effects.

However, this thesis argues that, in REA, language developers can easily manage semantic interactions between the language components because of several reasons. First, every semantic interaction is well reflected in classes that implement the language, thus language developers can easily detect interaction in the source code by following the import of the classes of language components, which is supported by IDE tools, such as *iSpace*[4] for Eclipse. Second, to prevent conflicts, not only language components but any interacting component should always respect interfaces and its semantic contracts of other components. Third, in POPART/REA, if there are special semantic interactions, these can still be controlled using aspects at the language-level. For example, an aspect could track and prevent values from one language component to flow into another language component. For this, a *data flow aspect* [MK03] could be deployed on the language façade that *taints* every object returned by a keyword method with a tag for the corresponding language. When calling keyword methods, another aspect could check that only objects that hold the tag of the same language are allowed as parameters to keyword methods. Therefore, also semantic conflict can be prevented (conflict-free: ●).

[4]The iSpace Homepage: `http://ispace.stribor.de/`

10.3.1.2. Supporting Keyword Renaming

Section 10.2.2.1 has reviewed the BlackBoxCombiner's support for dealing with syntax conflicts, such as the conflicting keyword define between FUNCTIONAL and STRUCTURAL. However, as motivated by Section 3.3.1.2 (p. 43), one can resolve syntactic conflicts by renaming the keywords in the constituent languages.

Example Scenario: Consider defining a custom combiner that renames the define keyword in FUNCTIONAL to defun and in STRUCTURAL to make. Listing 10.10 presents the implementation of a custom combiner that the language developer who wants to compose the two languages must provide. The combiner MyRenamingCombiner inherits from LanguageCombiner that allows combining conflicting languages. Further, the combiner defines a restricted constructor that only accepts composing two language façade objects that implement IFunctional and respectively IStructural. Note that since combiner is also a façade class, it can define new keyword methods. Therefore, to rename the keywords, the combiner defines two new keyword methods. First, it defines the method defun—calls to this method will be forwarded to Functional.define. Second, it defines the method make—conversely, calls to this method will be forwarded to Structural.define. Third, it overrides the method define so that using the old define keyword will result in an error. Consequently, the custom combiner extends the syntax-to-semantic binding of LanguageCombiner with the renamed keywords.

```
public class MyRenamingCombiner extends LanguageCombiner {
  IFunctional functional; IStructural structural;

  public MyRenamingCombiner(IFunctional functional, IStructural structural) {
    super(functional, structural); this.functional = functional; this.structural = structural;
  }
  public void defun(HashMap params, Closure body) { functional.define(params, body); }
  public void make(HashMap params, Closure body) { structural.define(params, body); }
  public void define(HashMap params, Closure body) { throw new Exception("Invalid keyword. Use define/make."); }
}
```

Listing 10.10: A custom combiner that renames the conflicting keywords of FUNCTIONAL and STRUCTURAL

Listing 10.11 shows a modified program version (of the erroneous program in Listing 3.5). Instead of define, it uses the renamed keywords defun of FUNCTIONAL and make of STRUCTURAL. The program defines a new data type tuple that has two Integer slots. It defines a function tupleSquare that maps a tuple to another tuple with the square values of its components. Since the program uses the renamed keywords, there is no syntax conflict in the program, because the conflicting define is never used.

Review: With the help of inheriting from the LanguageCombiner, it is easy for the developer to implement the custom combiner. The implementation of the MyRenamingCombiner is straightforward. It inherits all meta-methods that define the default keyword method dispatch from its

```
def structural  = new Structural(); def functional  = new Functional();
def combinedLanguage = new MyRenamingCombiner(functional,structural);

Closure program = {
  make(name:"tuple") { x = Integer; y = Integer; }
  def t1  =  init (name:"tuple",x:3,y:5);

  defun(name:"tupleSquare") { tuple ->
    return  init (name:"tuple",x:(tuple.x * tuple.x),y:(tuple.y * tuple.y));
  }
  return apply("tupleSquare")(t1);
}
program.delegate = combinedLanguage;
 println  program.call();   // will print tuple [x:9,y:25]
```

Listing 10.11: Program using the renamed keywords of FUNCTIONAL and STRUCTURAL

super class. Consequently, the developer does not have to implement special meta-methods for the MOP himself/herself.

With such custom combiners, syntactic conflicts can successfully be resolved for a pre-defined set of languages. However, one could criticize that these custom combiners have the limitation that they are only for a specific set of languages. To address this problem, this thesis argues that generic strategies can be embedded to resolve syntactic conflicts. First, REA would allow implementing a generic renaming combiner that allows rule-based definition of renaming of keyword methods of arbitrary languages. In this case, this combiner redefines the meta-methods (e.g., invokeMethod, getProperty and setProperty) to intercept any method call and field access. It would have to check for possible renaming, and possibly manipulate the signature of a keyword method call before forwarding it to the right façade. Second, POPART/REA would allow deploying aspects on the language façades. These language-level aspects can introduce new keyword methods as well as intercept and manipulate keyword method calls. When multiple aspects are defined, their interaction can be controlled such that aspect-oriented extensions are applied in the right order. Because REA is uniform the embedded mechanisms can be used to detect and prevent interactions (syntactic interactions: ●).

10.3.1.3. Supporting Priority-Based Conflict Resolution

As motivated in Section 3.3.1.3 (p. 43), since the language extensions can rely on the same base language theorems, often it is sufficient to resolve conflicts using a priority. Therefore, this section reviews the support of the LinearizingCombiner (cf. 5.4.3.2) to resolve syntactic conflicts by prioritizing expression types of extensions.

Example Scenario: Reconsider composing CONCISELOGO, UCBLOGO, and EXTENDEDLOGO extension of SIMPLELOGO, by composing the independent extensions using the LinearizingCombiner. One can compose the implementations of ConciseLogo, UCBLogo, and ExtendedLogo. Because all three extensions inherits from the same base language, there is a diamond inheritance problem (e.g. all three façades define the keyword turtle). Still, with the LinearizingCombiner, one can compose the extensions. All that a language developer needs to do is to pass all instances of the language extensions' facçades to the LinearizingCombiner's constructor as demonstrated by Listing 10.12 in line 8. Since the LinearizingCombiner prioritizes the façades in the order they are passed to the constructor, the LinearizingCombiner will delegate keywords to the ExtendedLogo façade first, which resolves the conflicts, e.g. the keyword turtle is evaluated only by the first façade.

```
1 Closure program = {
2   turtle (name:"squareComplete", color:red) {
3     setpencolor(green);
4     repeat (4) {
5       fd 50
6       rt 90
7 } } }
8 program.delegate = new LinearizingCombiner(new ExtendedLogo(), new ConciseLogo(), new UCBLogo());
9 program.call ();
```

Listing 10.12: Implementation of composing the independent extensions to execute a COMPLETELOGO program

Review: With the LinearizingCombiner, in the above example, a language developer can consolidate multiple extensions without changing existing code, but developers can use this combiner only in simple scenarios, since there are certain limitations, which are explained next.

The combiner assumes that all expression types of a language have the same priority as expression types of other languages. Therefore, the combiner does not allow to prioritize single expression types individually. To address this limitation, one can always implement a more generic combiner that resolves keyword conflicts by using individual priorities per expression type is conceivable (linearization/priorities: ●).

10.3.2. Open Mechanisms for Handling Semantic Interactions

10.3.2.1. Specific Mechanism for Controlling Semantic Interactions

When the semantics of constituent languages have conflicts, the language developer can extend combiners for handling those semantic interactions correctly.

Example Scenario: Consider preventing conflicts for the composition of FUNCTIONAL, STRUCTURAL, and CONSOLE, such that identifiers in the composed language are always unique.

Listing 10.13 presents a custom combiner that extends of the LinearizingCombiner. Like the custom combiner presented in Section 10.3.1.2, this combiner resolves syntactic conflicts. Before adding a new function, in line 14, the defun method ensures that there is no data type defined with the same. Similarly, before adding a new data type, in line 21, the make method prevents name clashes with functions. Line 25 prevents using the old keyword define.

Review: In case of the above composition scenario, the presented solution resolves the interaction with little effort, still it is undesirable for two reasons. First, the enforcement logic is

```
public class MySemanticAdaptationCombiner extends LinearizingCombiner {
  IFunctional functional ;
   IStructural structural ;

  public MySemanticAdaptationCombiner(IFunctional functional, IStructural structural, DSL console) {
    super(functional, structural ,console);
    this. functional = functional ;
    this. structural = structural ;
  }

  public void defun(HashMap params, Closure body) {
    if ( structural .getEnvironment().get(params.name) != null)
      throw new SemanticConflictException("Function name is not unique '$params.name'.")

    functional .define(params, body);
  }

  public void make(HashMap params, Closure body) {
    if ( functional .getEnvironment().get(params.name) != null)
      throw new SemanticConflictException("Data type name is not unique '$params.name'.")

    structural .define(params, body);
  }

  public void define(HashMap params, Closure body) { throw new Exception("Invalid keyword. Use define/make."); }
   ...
}
```

Listing 10.13: Excerpt of a custom combiner that enforces identifiers to be unique

scattered in the methods of the custom combiner. Second, it is unclear how to prevent or handle complex and more general cases of semantic interactions. Third, defining a new custom combiner for every new conflicting semantic composition is not a scalable solution, when there are more that one semantic conflict to handle the combiner implementation can be expected to become scattered and tangled. Although semantic conflict resolution is fully supported when using custom combiners, implementing a resolving combiner lead to high effort.

10.3.2.2. Generic Mechanism for Crosscutting Composition of DSLs

To separate crosscutting concerns in overlapping domains, REA's means for *domain-specific join points* allows creating semantic interacting compositions of languages, based on the AO concepts presented in Section 7. When multiple domains crosscut each other, REA allows defining multiple join point models for the composed aspect language.

To enable aspects for a DSL, the developer needs to implement the DSL base language with a domain-specific aspect language (DSAL), which consists of three building blocks: (a) a *domain-specific join point model* (DS-JPM), (b) a *domain-specific pointcut language* (DS-PCL), and (c) a *domain-specific advice language* (DS-AdvL) [DMM09].

10.3.2.2.1. A Domain-Specific Join Point Model for Ao4Logo Consider growing SIMPLE-LOGO into its aspect-oriented version AO4LOGO, whereby the language developer can completely reuse the non-AO version for the extended language. To implement a DSAL for SIMPLELOGO, the developer incrementally implements the three DS-JPM, DS-PCL, and DS-AdvL building blocks by taking into account SIMPLELOGO's domain semantics.

For the DS-JPM, the developer defines the following subclasses of JoinPoint. ForwardJoinPoint and BackwardJoinPoint both hold a reference to thisTurtle. They have an attribute steps to store the steps to move. LeftJoinPoint and RightJoinPoint also hold a reference to thisTurtle and have an attribute degrees. The four join points are contained in a package which encapsulates in DS-JPM.

For the DS-PCL, the language developer defines a modular language façade that has a set of pointcut designators as its language model. To implement the language model, the developer defines the pointcut designators classes TurtleMovingForwardPCD, TurtleMovingBackwardPCD, TurtleTurningLeftPCD, and TurtleTurningRightPCD, which are subclasses of PointcutDesignator. Further each class defines three operations for defining *range patterns* to select specific join points. The first constructor does not have any parameters, it creates a designator instance that will match any instance of the corresponding joint point type, e.g., left(a), where a can be any positive integer. The second has one value parameter that creates an instance that only matches JP of the corresponding type if the parameter to the command matches the exact value, e.g., left(50). To implement the DS-PCL, the developer defines a façade LogoPointcutLanguage class. The façade class defines corresponding operation keywords of which each creates an instance of the pointcut designator, namely pforward, pbackward, pleft, and pright. For each pointcut designator keyword, there are three overloaded methods for passing the different range patterns to its constructor. Further, the façade class defines the more abstract designator keywords, such as pmotion, by composing the atomic designator classes. Further, in line 4, the LogoPointcutLanguage declares the join point model for SIMPLELOGO.

For the advice language, the developer can use any other embedded DSL of REA. In the example scenario, the developer implements a special tracing DSL for *Logo*, which is presented in Listing 10.15. The developer implements the ITracingDSL language interface in the façade class TracingLogoDSL, which implements the keyword methods show nand trace for *Logo*.

To compose the modular language components, the developer uses the CCCombiner, as shown in Listing 10.16 that implements the AO program from Listing 3.9 (p. 47). In line 2, the program creates an instance of the LogoPointcutLanguage façade. Next in line 3, the program creates an instance of the LogoPointcutLanguage façade. To setup the composition, the façades are passed to the constructor of the CCCombiner. Internally, the CCCombiner invokes the LogoPointcutLanguage to declare the DS-JPM, which opens up the encapsulated SIMPLELOGO language component to expose its join points. After the setup, to define the *tracing aspect*, line 16 evaluates the aspect program (lines 7–14). Finally, to trace the program, line 29 evaluates the DSL program (lines 19–27) with the aspect.

10.3.2.2.2. Multiple Join Point Models To demonstrate that POPART/REA supports multiple join point model at a time, consider embedding an aspect-oriented TINYSQL dialect into an

```
class LogoPointcutLanguage extends Interpreter implements LogoPointcutLanguage {
  void setMetaClass(MetaClass mc) {
    super.setMetaClass(mc); //mc is an instance of CCCombiner
    mc.declareJoinPoint(SimpleLogoInterpreter.class, "forward", TurtleForwardJPInstrumentation.class);
     ...
  }

  Pointcut pforward() { return new TurtleMovingForwardPCD(); }
  Pointcut pforward(int steps)   { return new TurtleMovingForwardPCD(steps); }
  Pointcut pforward(int minSteps, int maxSteps) {
    return new TurtleMovingForwardPCD(minSteps, maxSteps);
  }
  Pointcut pbackward() {...}
  Pointcut pleft ()   {...}
  Pointcut pright ()   {...}
  Pointcut pmotion() { return new OrPCD( new TurtleMovingForwardPCD(),
    new TurtleMovingBackwardPCD(), new TurtleTurningLeftPCD(), new TurtleTurningRightPCD());
  }
}
```

Listing 10.14: Excerpt of the pointcut language implementation of AO4LOGO

```
interface ITracingDSL {
  void show(String message);
  void trace ();
}

class TracingLogoDSL extends Interpreter implements ITracingDSL {
   ...
  void show(String message) { println message; }
  void trace()  {
    //the concrete facade class assumes to be used in a composition with Logo exposing
    println bodyDelegate.thisTurtle.toString ();
  }
}
```

Listing 10.15: Excerpt of the advice language implementation of AO4LOGO

OO language, called AO4SQL. For example, in this dialect, the entity bean in Listing 10.17[5] defines a business object with *bean-managed persistence* with embedded TINYSQL statements, comparable to Hibernate[6] or J2EE [JBC02].

For the embedded TINYSQL dialect in Groovy, there are two JPM—one for Groovy and one for TINYSQL. For Groovy, there are method_execution and method_call. For TINYSQL, the prototype defines defines a new join point model in POPART/REA with the following domain-

[5]For the sake of a clearer presentation, the example code in Listing 10.17 uses concrete syntax, although the current prototype uses abstract syntax, but the current prototype could be easily extended.

[6]The Hibernate Homepage: `http://www.hibernate.org/`.

```
//set up
def dspcl = new LogoPointcutInterpreter();
def dsadvl = new SimpleLogoInterpreter();
def ccc = new CCCombiner([dsadvl,dspcl]);

//evaluate aspect program
def aoDSLProgram = {
  aspect(name:"TracingAspect") {
    ...
    before  (pmotion()) { show "$command $args"; }
    after    (pforward() | pbackward() | pleft () | pright () | ) { trace(); }
    ...
  }
}
aoDSLProgram.delegate = ccc;
aoDSLProgram();

//evaluate DSL program (with tracing aspect)
def logoDSLProgram = {
  define(name:"polygon") { length, edges -> (
    ...
  }

  turtle (name:"Octagon",color:green) {
    apply("polygon")(8,25)
  }
}
logoDSLProgram.delegate = ccc;
logoDSLProgram();
```

Listing 10.16: Using the CCCombiner to compose the language components of AO4LOGO

specific join points (DS-JPs) that cast on executions of the corresponding TINYSQL statements: (1) SelectJoinPoint, (2) InsertJoinPoint, (3) RemoveJoinPoint, and (4) UpdateJoinPoint.

Further, there is a declarative domain-specific advice language, which defines corresponding pointcut designators that match only their corresponding kinds of TINYSQL statements: (1) pSELECT, (2) pINSERT, (3) pREMOVE, and (4) pUPDATE. Similar to other domain-specific advice languages, the designator has corresponding parameters to select only DS-JPs, e.g., that access certain columns. Further, there is the pFROM designator that matches only queries for a particular table. For example, consider the pointcut “pSELECT("Forename") & pFROM("Persons")” that uses the designators and parameters to quantify over TINYSQL queries, and select only the executions of queries like:
SELECT "Forename", "Lastname" FROM "Persons"
but not
SELECT "Lastname", "Age" FROM "Persons".

As an example runtime schema evolution, consider changing the schema of the "Person" table. For example, the database administrator executes a *data definition language* (DDL) query, such

```
class PersonBean {
  PersonKey personKey;
  String forename;
  String lastname;
  int age;

  PersonBean(PersonKey personKey, String forename, String lastname, int age) { ... }
  PersonBean(Object... params) { ... }

  void ormCreate() {
    INSERT INTO Persons ("PrimaryKey","Forename","Lastname","Age")
      VALUES (primaryKey,forename,lastname,age)
  }

  void ormRemove() { DELETE FROM ... }

  PersonBean ormFindByPrimaryKey(PersonKey pkey){
    return new PersonBean(
      SELECT "p.PrimaryKey", "p.Forename", "p.Lastname", "p.Age" FROM "Persons" p
    );
  }
   ...
}
```

Listing 10.17: An entity bean with bean managed persistence

as “ALTER TABLE "Persons" ADD "Job"...”. After the DDL query, the entity bean implementation is not longer valid.

Without AOP support for TINYSQL, the developer of the entity bean would have to invasively change the entity bean implementation. For example, to manage the additional field in the table, the developer has to add a new String job field to the PersonBean class, a getter and a setter. In addition, the developer has to change the code of all embedded TINYSQL queries.

To support schema evolution, in the context of this thesis an aspect-oriented version of TINYSQL was implemented, called AO4SQL. With AO4SQL, there is support for aspect-oriented programming for embedded TINYSQL/Groovy, the developer can implement an aspect that updates the PersonBean definition without having to change the code of the bean implementation.

Listing 10.18 presents example code aspect for TINYSQL/Groovy. In the aspect, line 4 adds the PersonBean.job field, and line 7 adds the getter for the field. To adapt the bean managed persistence implementation, the TINYSQL advice in lines 10–14 adapts the TINYSQL statements that are embedded into the PersonBean class. The advice reifies the query at a select-from join point (line 11). It changes the list of select columns query.selectList in the reified query (line 12), by adding a new column selection for the column “Person.job”. And finally, the advice proceeds the execution of the SQL query (line 13).

```
 1 aspect(name:"BeanAOPTest"){
 2
 3   // inter−type declaration that adds a field "Person.job" (with default value "jobless")
 4   introduce_field(is_type(PersonBean), "job", "jobless")
 5
 6   // inter−type declaration that adds a getter for "Person.job"
 7   introduce_method(is_type(PersonBean), "getJob") { return job; }
 8   ...
 9   //pointcut−and−advice
10   around(pSELECT() & pFROM("Persons")){ //pointcut matches all SELECT queries
11     Query query = thisJoinPoint.query;
12     query.selectList.add(Col("job"));
13     return proceed();
14   }
15 }
```

Listing 10.18: An AO4SQL aspect that changes an entity bean and its TINYSQL queries

In REA, a special runtime aspect weaver for TINYSQL was implemented. To implement AO4SQL, several language components were implemented. There is a base language component for TINYSQL, and there are several language components for the aspect languages.

For the base language component, to allow weaving of aspects into TINYSQL, it was not possible to use the default homogeneous embedding of TINYSQL that was presented in Section 8 (p. 171). Instead an alternative implementation of TINYSQL was embedded as an embedded compiler, which follows Kamin's style of embedding [Kam98], and that generates standard SQL code [DD89] in concrete syntax at runtime. The implementation of the TINYSQL-to-SQL generator consists of a language façade that implements the TINYSQL language interface. The embedded compilers language model implements an AST for standard SQL, containing expression types for select-, insert-, update-, and delete queries.

When evaluating an embedded TINYSQL program, the embedded compiler first creates an AST from the embedded program. After the program was completely evaluated, the complete AST is pretty printed in standard SQL code. Next, the embedded compiler executes the generated standard SQL code using a JDBC connection to the corresponding database management system.

To support aspect weaving on TINYSQL, for the aspect language, a DS-JPM, DS-PCL, and DS-AdvL was implemented. The join point model captures points in the generation of the SQL code inside the embedded compiler. e.g. SelectJoinPoint object cast on the generation of a select-statement in the generator. Part of the generators context, e.g. the selected columns and tables and the where expressions of the select-query are reified from the generator's context and can be altered by aspects. The pointcut is implemented as a language façade with pointcut designators that match those join points in the generator. The advice language makes the context at the join point available, e.g., such as query.selectList (in Listing 10.18 line 12) exposes a first-class representation of the selected column in a select query.

10.3.2.2.3. Review of Crosscutting Composition In REA, the composition architecture fully supports crosscutting composition with pointcut-and-advice semantics similar to AspectJ. Moreover, the implementation of the CCCombiner is agnostic of the embedded DSLs to be composed. The combiner is parameterized by the embedded DSLs to compose; crosscutting combiners in addition are parameterized by a set of pointcut languages and advice languages, which are embedded DSLs.

Effectively, the CCCombiner enables a gray-box composition model for well encapsulated embedded DSL with crosscutting semantics. Domain-specific join point models and domain-specific pointcut languages define an abstraction layer over the interpretation of embedded DSLs to compose. Hence, only the relevant points for the crosscutting composition are exposed. Still, the CCCombiner enables invasive composition of the embedded DSL's evaluation protocols. For this, the CCCombiner exploits the MOP-based interception of method calls inside embedded DSL implementation, join points can be reified from any method call inside an embedded DSL interpreter that constitutes a composition point. Each join point's context can refer to any object of an embedded DSL program and to the internal state of embedded DSL interpreters. Pointcut expressions can even mix sub-expressions from different domains because the CCCombiner composes all pointcut languages that are passed to its constructors.

As a result, embedded DSLs that participate in a composition are defined independent of each other, while at the same time being composable even in an invasive way.

To validate that complex domain-specific aspect languages can be implemented with POPART/REA, in the context of this thesis, the existing domain-specific aspect language Cool [Lop97] has been designed and re-implemented on top of MAP.

In the context of this thesis, the following domain-specific aspect languages (DSALs) have been implemented as prototypes:

(a) Cool DSAL: The Cool DSAL [LK97], which weaves coordination concerns into OO program

(b) Caching DSAL: The caching DSAL [HBA08], which caches values of certain fields when reading them from their objects.

(c) Zip DSAL: A DSAL that allows compressing string parameters in method calls, which allows compressing all strings before they are written into a file, and it allows decompressing strings when reading from a file.

For the following DSL base languages to be advised by aspects, the corresponding DSALs was implemented without needing to change the CCCombiner:

(i) Ao4Logo: homogeneous embedding

(ii) AOP for ProcessDSL: homogeneous embedding

(iii) Ao4Sql: heterogeneous embedding

Because POPART/REA supports multiple join point models, the join point models of TINYSQL and Groovy are integrated without restriction. Aspect may define pointcuts that quantify over the join point of different languages, e.g., the pointcut "pSELECT() & cflow(method_execution("orm.*"))" would only match the embedded SQL statements inside bean-managed persistence methods.

For all aspect-oriented languages that have been implemented in the context of this thesis, the default AO semantics of the CCCombiner could be reused for all crosscutting compositions without changing its code (crosscutting: ●).

10.3.2.3. Supporting Composition Conflict Resolution

As discussed in Section 3.3.2.2 (p. 47), when multiple aspect languages are composed there can be composition conflicts.

In [DMB09], a generic mechanism for resolving aspect interactions is presented for POPART. While the section presents how the language developer can use the generic AO mechanism to resolve aspect interactions, it is out of the scope to repeat the implementation details of the aspect interaction resolution mechanism itself. The interested reader is referred to [DMB09].

Meta-aspects allow defining aspect interaction resolution strategies that can be either application-specific or domain-specific. Technically, there is the OrderedMetaAspectManager which allow language developers to define a domain-specific pointcut-and-advice comparator that resolves interactions between several DSALs by ordering advice (or resp. deleting some advice). However, it is a little inconvenient to implement as pointcut-and-advice comparator, for every new DSAL composition. Therefore, this section presents a generic extension for the CCCombiner that can resolve any static co-advising interaction between DSALs.

Example Scenario: To validate co-advising interactions in REA, this thesis uses POPART/REA to compose the following DSALs with each other and analyze their interactions:

(a) Cool DSAL

(b) Caching DSAL

(c) Zip DSAL

To compose the three DSAL into a language called COCAZIP, developers simply can use REA generic InductiveWrappingCCCombinerComparator, while they do not need to change the implementation of the CCCombiner. This comparator orders the advice in the order in which their façades are passed to the constructor of the CCCombiner. Listing 10.19 presents the implementation of the InductiveWrappingCCCombinerComparator (lines 2–9), how it is set up (line 18), and how it is used to evaluate aspect-oriented program that mixes keyword from the 3 DSALs (lines 25–31). To compose the DSALs and resolve their composition conflicts, it is crucial that in line 18 the façades of the 3 DSLs are passed in the right order to the CCCombiner's constructor.

```
 1  //The domain−specific extension needed for the composition of the 3 DSALs
 2  class InductiveWrappingCCCombinerComparator<...> extends ApplicationSpecificComparator<...> {
 3    public int compare(T to1, T to2) {
 4      //determines the ordering by accessing the CCCombiner
 5      //when before advice: advice of first facade (e.g., Cool) come before others (e.g., caching)
 6      //when around advice: first (e.g., Cool) wraps around the others (e.g., caching wraps compression)
 7      //when after advice: last facade (e.g., caching) come before others (e.g., Cool)
 8    }
 9  }
10  AspectManager.defaultComparator = new InductiveWrappingCCCombinerComparator<...>();
11
12  // set up
13  def coolDSL = new CoolDSL(isPerClass);
14  def cachingDSL = new CachingDSL(isPerClass);
15  def zipDSL = new ZipDSL();
16
17  //the order in which the developer passes the DSAL facades to the CCCombiner defines their advice ordering
18  def coolcachingzip = new CCCombiner(coolDSL, cachingDSL, zipDSL);
19
20  Closure cocazip = { String className, HashMap aspectBody −>
21    return coolcachingzip.aspect([name:className], body) };
22  }
23
24  //the composed DSAL program
25  cocazip("DBFile") {
26    selfex { write };  mutex { read; write } //two pieces of advice from Cool DSAL
27
28    memoize "read", { invalidated_by_calling "write" } //an advice from Caching DSAL
29
30    compress_args "write"; decompress_result "read" //two pieces of advice from Compression DSAL
31  }
```

Listing 10.19: Handling composition conflicts by defining a static order on DSALs

Review: To validate the support for handling interactions, the DSALs on POPART/REA were tested with a test suite for concurrently executing test cases, which is supported by *Grobo Utils*[7]—an extension to *JUnit*[8]. First, each DSAL was tested in isolation. For each example proposed in the thesis of Lopes [Lop97], a test was implemented for the embedded Cool DSAL. Second, POPART/REA was tested with respect to the tests that have been proposed in related

[7]The GroboUtils Homepage: `http://groboutils.sourceforge.net/`
[8]The JUnit Homepage: `http://www.junit.org/`.

work [HBA08]. Third, additional tests have been implemented for the Zip DSAL for compression and its interactions with the Caching DSAL. These tests give a high confidence that the aspect interactions/conflicts are resolved correctly (composition-conflict resolution: ●).

10.4. Support for Concrete Syntax

In TIGERSEYE/REA, concrete syntax can be rewritten only because the AST of an embedded language is reified, transformed, and reflected back at compile time. In most cases, the developer often does not have to specify the complete formal syntax, but TIGERSEYE automatically extracts the necessary information from the embedded objects.

10.4.1. Converting Concrete to Abstract Syntax

Section 8.2 (p. 179) has already elaborated how the TIGERSEYE generic pre-processor can map arbitrary concrete syntax to concrete syntax in the host languages, therefore the remainder of this section only reviews this support.

Review: A developer only needs to add some annotations in order to enable mapping concrete to abstract syntax, the mapping itself is automatic. For making this automatic conversion possible, the generic architecture of TIGERSEYE is crucial. For the conversion of concrete syntax to abstract syntax, it is important that all steps in the conversion or transformation process are generic. In the TIGERSEYE, all transformation steps make no special assumptions about the language to be composed. Further, the generic pre-processor transformation logic is parameterized by their meta-data: the annotations. Since none of the steps in this process make any assumption about particular languages, one can enable arbitrary languages and their programs on top of TIGERSEYE.

Further, to support multiple languages, in the transformation process, it is crucial that every intermediary representation is composable and the representation builds on the previous steps. Further, the pre-processor uses a composable representation in each step, which is elaborated next.

First, composing meta-data for a syntactic transformation is simple, since the pre-processor uses context-free grammars consisting of BNF-rules that are a composable. The pre-processor extracts BNF-rules from the annotations defined by language components and converts them to first-class objects of the embedded BNF dialect, which allows composition of these syntax rules.

Second, the pre-processor parses a program into a generic and composable AST representation. It is important that this AST representation does not use a closed set of types for the AST nodes, but that a new kind of AST node can be introduced as an annotation of an AST node, which make the AST extensible and independent of the concrete languages. Therefore, technically the pre-processor uses untyped ATerms [vdBK07] to implement the AST, which are a well-established representation widely used in source code analysis and transformation tools [vdBK07, BV04].

Third, the AST transformation that rewrites the AST elements to host language syntax must be composable. An important detail is that the pre-processor does not have to perform any semantic transformations that depend on a concrete embedded language. The transformation process must only know how to transform an AST node to a method call in the host language, which is rather a simple lexical transformation than a semantic transformation. Since there are various lexical transformation steps that are necessary, it is important to correctly compose them. Therefore, the pre-processor ensures the correct ordering of each lexical transformation steps by taking into account the pre- and post-condition of each lexical transformation.

Fourth, it is crucial that method calls in the host language are composable. In Groovy, encoding compositional method calls is simple for all kinds of compositionality of expression types. For terminal expressions, the encoding uses parameterless method calls or properties respectively. For non-terminal expressions, one can put a method call into the parameters of another method call for its sub-expressions. For nested expressions in a body, the encoding nests method calls in a Groovy closure of which the delegate is set to an instance of the corresponding façade class. For compositionality of all three expression types, arbitrary method calls can be put in a sequence. In Java, encoding compositional method calls is similar to Groovy, except for nested expressions in a body. Nested expressions are encoded so that the method call always is calling the method in abstract syntax directly on an instance of the corresponding façade class.

With the above support for an intermediary representation that supports compositionality of expressions, there are no practicable restrictions for Groovy and Java, TIGERSEYE supports a generic transformation that transform concrete to abstract syntax. The generic transformation is extensible, i.e. subclasses of façades can add new keywords with concrete syntax. The generic transformation is composable, i.e. the concrete syntax of several languages can be composed into one combined syntax. The generic transformation is automatic integrated into the compilation process of embedded program. With these characteristics, there is full support for concrete-to-abstract syntax transformation (concrete-to-abstract syntax: ●).

10.4.2. Supporting Prefix, Infix, Suffix, and Mixfix Operators

To incrementally add support, using TIGERSEYE/REA, the language developer can add annotations for concrete syntax as described in Section 8.1.2 (p. 174) directly to the existing source code of an existing language interface or façade class with abstract syntax, or the developer can subclass the façade class and add annotations in the incremental extension.

Example Scenario: Consider implementing the prefix, infix, suffix, and the mixfix operators from Section 3.4.2 (p. 50). Listing 10.20 presents three classes that incrementally add more concrete syntax to a small example language with the four kinds of operators. First, the class AbstractOperators (lines 1–6) implements the operator in abstract syntax. Second, ConcreteOperators (lines 8–17) extends the first class to define annotations for concrete syntax. Third, UnicodeOperators (lines 19–28) incrementally extends the second class to give the part of the operators with concrete syntax a Unicode syntax.

Review: To validate the support for prefix operators, in the context of this thesis, various embeddings with concrete syntax have been embedded into TIGERSEYE, such as sets with the Unicode operators ∪ and ∩, an extended BNF-dialect with support for regular expressions, and many other small examples.

```
class AbstractOperators extends Interpreter {
  boolean not(boolean pred) {...} // textual syntax for prefix operator is already okay
  Closure compose(Closure f, Closure g) {...}
  Formula prime(Formula k) {...}
  void selectFromWhere(String[] columns, String[] tables, WhereClause preds) { ... }
}

class ConcreteOperators extends AbstractOperators {
  @DSLMethod(production="p0_compose_p1")
  Closure compose(Closure f, Closure g) {...}

  @DSLMethod(production="(p0)prime")
  Formula prime(Formula k) {...}

  @DSLMethod(prettyName = "SELECT__p0__FROM__p1__WHERE__p2")
  void selectFromWhere(String[] columns, String[] tables, WhereClause preds) { ... }
}

class UnicodeOperators extends ConcreteOperators {
  @DSLMethod(production="¬p0")
  boolean not(boolean pred) {...} // Unicode syntax must be declared

  @DSLMethod(production="p0__∘__p1")
  Closure compose(Closure f, Closure g) {...}

  @DSLMethod(production="p0′")
  Formula prime(Formula k) {...}
}
```

Listing 10.20: An incremental extensions to FUNCTIONAL for concrete syntax

In the aforementioned examples, the abstract syntax always could be replaced by concrete syntax (*-fix operations: ●).

10.4.3. Supporting Overriding Host Language Keywords

With TIGERSEYE/REA, it is possible to override host language keywords, in a similar way as concrete syntax is defined.

Example Scenario: Consider implementing the following two small DSLs that define expression types that override reserved keywords in the host language.

Listing 10.21 presents the ConditionalDSL class that overrides Groovy's if keyword. However, it overrides the keyword only when the DSL program uses the then and else keywords, which allows mixing DSL code and Groovy code, and use different semantics in parallel without special quotations, as long as the syntax-to-semantic binding is context-free.

Listing 10.22 presents the SetDSL class that overrides the curly bracket delimiters of Groovy. The annotation in line 4 defines concrete syntax for the asSet method, whereby the production can use the reserved Groovy delimiter keywords "{" and "}". For example, using the curly brackets in line 16 does not define a Groovy closure, but the curly brackets are syntactic sugar that define new sets.

```
@DSL(whitespaceEscape=" ")
class ConditionalDSL implements Interpreter {
  @DSLMethod(production="if ( p0 ) then { p1 } else { p2 }")
  boolean ifThenElse(boolean check, Closure thenBlock, Closure elseBlock) {
    if (check) { thenBlock.call (); } else { elseBlock.call (); }
    return check;
  }
}
```

Listing 10.21: Overriding the host language keyword if

```
@DSL(whitespaceEscape=" ")
class SetDSL implements Interpreter {
  ...
  @DSLMethod(production="{ p0 }")
  Set asSet(MyList a) {
    return new HashSet(Arrays.asList(a.toArray()));
  }

  @DSLMethod(production="p0 , p1")
  private MyList multiElementedList(String head, MyList tail ) {
    return new MyList(head, tail);
  }
}
...
//DSL file using the support for sets
Set M = {"a", "b", "c"};
```

Listing 10.22: Overriding the curly brackets delimiter of Groovy

Review: TIGERSEYE's support for concrete syntax does not make a difference between reserved keywords and other literals, since the transformation is independent of the host language. It simply overrides keywords in the concrete syntax of the embedded DSL to method calls in the host language. Note that reserved keywords are only overridden if they exactly match the declared production. Further, note that after transforming the reserved keyword to a method call, REA's meta-level and POPART's aspects could even advise the keyword.

To validate, overriding host language keywords, test cases where written that in particular test whether different nested structures are rewritten correctly (overriding host keywords: ●).

10.4.4. Supporting Partial Definition of Concrete Syntax

Example Scenario: Consider embedding TINYSQL into Java and Groovy. With TIGERSEYE, as elaborated in Chapter 8, only part of the Java and Groovy syntax needed to be specified.

Review: In TIGERSEYE/REA, Table 10.1 shows that the integration of the EDSLs with new host languages also requires only negligible effort. For Java and Groovy, in both cases, it was sufficient to specify only 19 productions. As elaborated in Section 8.1.4 (p. 178), to allow parsing the host languages with island grammars, TIGERSEYE had to define only the most important productions that recognize the host languages' basic syntactic structure.

Table 10.1.: Number of grammar rules that need to be specified for the concrete syntax

EDSL	Complete Grammar	Island Grammar	Savings
TINYSQL	6	1	83 %
BNF	9	9	0 %
State Machines	10	2	80 %
Simple Logo	11	1	91 %

Table 10.2.: Number of grammar rules that need to be specified to enable the integration with the host language

Host Language	Complete Grammar	Island Grammar	Savings
Java	1700	19	98.9 %
Groovy	950	19	98.0 %

Furthermore, as Table 10.2 shows, the effort that is necessary to support concrete syntax for a homogeneous embedded EDSL that can be used across different host languages, is reduced. In case of the four discussed DSLs, no additional effort was necessary to directly support concrete syntax for the discussed DSLs when we embedded them also in Java (they were first embedded into Groovy). This, however, requires that we can reuse the same DSL interpreter for the new target host languages and that support for the target languages is readily available in the TIGERSEYE pre-processor. If this is the case, it is in general only necessary to adapt the EDSL's grammar to the new host language grammar. Due to using island grammars and the syntactical similarities between Groovy and Java, no adaptation was necessary (partial syntax: ●).

10.5. Enabling Pluggable Scoping

Section 3.5 (p. 53) has motivated several scenarios that require controlling scoping in embedded languages. Therefore, this section presents how language developers can embed new scoping schemes on top of REA. In REA, the native host language concepts can be replaced by first-class concepts that are embedded. In particular, name bindings can be embedded as first-class environments, which are in the same vein as [AS96]. In REA, various scoping strategies for language constructs are possible, since language constructs are first-class and their scope can be embedding restrictions in the part of the evaluation protocol that executes the language construct.

10.5.1. Supporting Dynamic Scoping

Section 3.5.1 (p. 53) has discussed the problem that embedding approaches have not yet shown how to embed different scoping schemes. One important reason for this open issue is that most host languages only support one closed scoping scheme that cannot be changed.

Although Groovy is also a lexically-scoped host language, the fact that in Groovy closures do not completely close over their lexical context (cf. Section 5.1) allows embedding various scoping schemes. Therefore, the language developer can embed a dynamically scoped language into Groovy. For dynamic scoping, the language developer embeds a new scoping scheme for variables, so that they are bound to a global context, not to the lexical enclosing context.

Example Scenario: Consider embedding dynamically scoped *Logo* into Groovy. To implement dynamic scoping (or other schemes) for various embedded languages, language developers can extend or compose the ENVIRONMENT language presented below. ENVIRONMENT provides an *interpreter environment* (after [AS96]) to bind names to values via two operations **let(**$name$**,**$value$**)** and **get(**$name$**)**. The keyword let binds the $name$ to the $value$ in the current environment, and the keyword get retrieves the value for $name$ either directly from the current environment or from a possible enclosing environment (i.e., enclosing abstraction operators).

Listing 10.23 presents the implementation of an environment interpreter in REA. It shows the IEnvironment interface (lines 1–4) and an excerpt of its implementation the EnvironmentInterpreter façade (lines 6–24). Note that the façade has an adjustable scoping Strategy that can be either LEXICAL_SCOPING or DYNAMIC_SCOPING (line 7) that the developer or user can change. Most importantly, the façade holds a map from Strings to Object values (line 8), into which one can bind a name to a value using method let (line 10), and from which one can retrieve a value using method get (lines 12–17). Whenever the get method cannot find a value in the current environment, it tries to retrieve the value from an enclosing environment via method getFromEnclosingEnvironment (lines 21). The getFromEnclosingEnvironment methods looks up the special key "##enclosingEnv##" whose value is another environment. The methods getEnvironmentBodyDelegate (line 22) and getEnvironmentBodyResolveStrategy (line 23) are necessary to create new environments for abstraction operators.

Consider implementing the dynamic scoping example in Listing 10.24, which is motivated from Section 3.5.1 (p. 53), and which overrides the variable binding from line 3 in line 10. To execute the example code, all the developer has to do is to instantiate a new EnvironmentInterpreter (line 15), select the dynamic scoping strategy for it (line 16), pass it to some combiner (line 17) to compose it with other languages, and use the combiner to evaluate programs.

When evaluating the program with dynamic scoping, it is crucial that the methods let and get always operate on the enclosed environment, which is in case of dynamic scoping the global environment. Dynamic scoping is the simpler scoping strategy to implement, since there is no special propagation of bindings, and every binding is available from the global environment.

In contrast, the language developer can switch to lexical scoping, e.g., by replacing the selection the strategy in Listing 10.24 line 16 when evaluating the program with lexical scoping.

Review: To review the support of scoping, we discuss several important issues with respect to design, implementation, as well as issues related to performance and usability.

First, the design of the pluggable scoping mechanism is modular, extensible, and composable. It was successfully implemented on top of existing host language mechanisms as a language

```
interface IEnvironment {
  void let (String  name, Object value);
  Object get(String  name);
}

class EnvironmentInterpreter extends Interpreter implements IEnvironment {
  int  scopingStrategy =  LEXICAL_SCOPING; // alternative: DYNAMIC_SCOPING
  HashMap<String,Object> environment = new HashMap<String,Object>();

  void let (String  name, Object value) { environment.put(name, value); }

  Object get(String  name) {
    if  (environment.containsKey(name)) {
      return environment.get(name);
    else {
      return getFromEnclosingEnvironment(name) {
    }
  }
   ...
  HashMap<String,Object> getEnvironment() { return environment; }
  public Object getFromEnclosingEnvironment(String name) { ... }
  protected IEnvironment getEnvironmentBodyDelegate(Closure body) { ... }
  protected IEnvironment getEnvironmentBodyResolveStrategy(Closure body) { ... }
}
```

Listing 10.23: Implementation of pluggable scoping EnvironmentInterpreter

```
Closure program = {
  turtle (name:"Teacher", color:red) {
    let ("length",50);
    define(name:"complexShape") {
      fd get("length");  rt  120; fd get("length");  rt  120; fd get("length");  rt  120;
    }
  }

  turtle (name:"Pupil", color:red) {
    let ("length",100);
    apply("complexShape")();
  }
}

def env = new EnvironmentInterpreter();
env.scopingStrategy = EnvironmentInterpreter.DYNAMIC_SCOPING;
program.delegate = new LinearizingCombiner(env, new Functional(), new CompleteLogo());
```

Listing 10.24: Example from Listing 3.11 defining a dynamically-scoped variable using let and get

façade. It is configurable, because the developer can select between dynamic and lexical scoping. To extend the scoping strategy, a language developer can extend EnvironmentInterpreter.

With the embedding of pluggable scoping into Groovy, one can enable a different scoping scheme by adjusting the scopingStrategy property of the EnvironmentInterpreter. Figure 10.1 shows the differences for the environments with lexical and dynamic scoping. With lexical scoping, shown on the left hand side, for each closure, there is a dedicated environment that is linked to enclosing environments. The indices (right of the box) indicate at what regions in the code the corresponding bindings are effective. Note that in contrast to let, which defines a local binding, the keyword define establishes a global binding, therefore, the function is defined in the top-level environment. Conversely as shown on the right hand side, with dynamic scoping, there is only one global environment. The indices in this case indicate at which line a new version of the global environment is established.

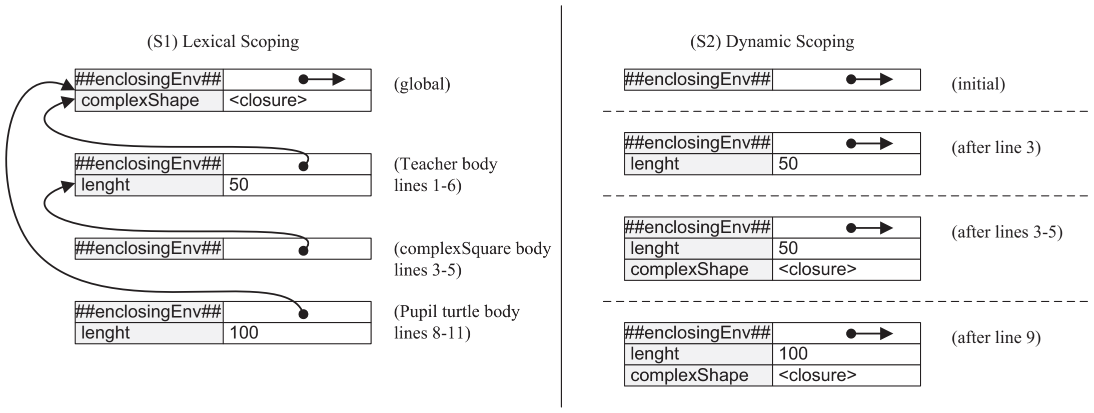


Figure 10.1.: Lexical scoping vs. dynamic scoping

With reflective embedding new schemes can be implemented for embeddings simply by extending the EnvironmentInterpreter. When such a specialize EnvironmentInterpreter provides different keywords for different kinds of (variable) bindings, such as the keyword slet for static binding, dlet for dynamic binding, various scoping schemes are possible in one program.

Second, the implementation of the architecture's scoping mechanism was tested under isolation and in compositions with other languages.

Third, the performance is far from being optimal, because every binding is dynamically resolved at runtime. Developers can use the current implementation prototyping new scoping mechanism. Yet, implementing efficient scoping schemes are not addressed. Since the scoping mechanism is embedded as a library and the Groovy compiler is not aware of this, it cannot optimize the embedded bindings. Currently, there are no special optimizations, and the embedded scoping mechanisms do not pass special information about bindings to the compiler. It would be interesting to apply well-known techniques of non-embedding-based approaches for optimization. For example, it is well-known that in case of lexical scoping, there is the possibility to statically resolve bindings, because every binding is known at compile time by taking into account the lexical context. But, it is out of the scope of this thesis.

Fourth, the usability of the scoping mechanism has two limitations. First, end users are required to use the keywords let and get to access variables with a special scoping. This makes writing variable expressions for end users more verbose. Note that, later on in Section 10.7, this chapter discusses how the keywords let and get can be omitted by using a simple transformation and the Groovy MOP. Second, there is no way to enforce that end user actually use these keywords for dynamic scoping. Specifically, when an end user uses the Groovy keyword def to define a local variable, the so defined variable is lexically scoped according to Groovy semantics. The Groovy MOP does not intercept access to local variables, therefore, so far it is impossible to override the scoping of local Groovy variables.

Finally, embeddings are not *scope-safe* [Stu09]. By default, in Groovy, variables and methods in closures can be captured and escape scopes unintended. There are only little means for restrict scoping; one can adjust the default resolveStrategy (which is OWNER_FIRST, cf. Section 5.1, p. 118) of closures to DELEGATE_ONLY in order to restrict possible bindings of variables and methods to an environmental interpreter, or respectively select OWNER_ONLY to restrict possible bindings to the lexical scope.

In sum, although the current pluggable scoping mechanism is limited for general scoping strategies, it fully supports embedding an alien dynamic scoping strategy into a lexically scope host languages. In addition, the current embedding of scoping is extensible by users who can easily adapt scoping when they have special needs (dynamic-scoping: ●).

10.5.2. Supporting Implicit References

In REA, language developers can easily define new implicit references in façades using ordinary literal keyword methods and compose these references into existing languages.

Example Scenario: Consider implementing the implicit reference thisTurtle as motivated in Section 3.5.2 (p. 55). Listing 10.25 presents the implementation of the implicit reference as part of the façade LogoImplicitReference that depends on an evaluation context of a CompleteLogo façade passed to its constructor. The façade implements the literal keyword method getThisTurtle (line 4) that returns a first-class representation of the enclosing turtle abstraction operator. This method retrieves the enclosing turtle from the evaluation context of the CompleteLogo façade via method getTurtle, through which its language model is accessible. To directly access properties of thisTurtle, such as color instead of writing thisTurtle.color, the LogoImplicitReference façade define other getters with the corresponding names prefixed by get. Thanks to the convention that when Groovy encounters an unknown property, when evaluating programs, Groovy will automatically call the corresponding getter methods, which enables implicit references.

Consider implementing the example program from Listing 3.12. Listing 10.26 only has to instantiate the LogoImplicitReference and compose it with the other façade to enable the implicit reference in COMPLETELOGO.

Review: As demonstrated, language developers can enable implicit references without much effort. Note that the literal keyword methods that implement implicit references can contain complicated resolution logic. With such logic, it is possible to implement sophisticate resolutions of

```
class LogoImplicitReference extends Interpreter {
  CompleteLogo logo;
  public LogoImplicitReference(CompleteLogo logo) { this.logo = logo; }
  public Turtle getThisTurtle () { return logo.getTurtle (); }
  public Color getColor() { getThisTurtle (). getPenColor(); }
   ...
}
```

Listing 10.25: Excerpt of the implementation of EnvironmentInterpreter

```
Closure program = {
  define(name:"print") { str −> ... }
  define(name:"colorToStr") { color −> ... }
  turtle (name:"Turtle1",color:red) {
    apply("print ")( thisTurtle.name)
  }
  turtle (name:"Turtle2",color:blue) {
    apply("print ")( apply("colorToStr")( color ))
  }
}
CompleteLogo logo = new CompleteLogo(); LogoImplicitReference lir = new LogoImplicitReference(logo);
Functional functional = new Functional(); program.delegate = new LinearizingCombiner(lir,functional,logo);
program.call ();
```

Listing 10.26: A Logo program using the implicit reference thisTurtle

implicit references (implicit references: ●), e.g., from the enclosing interpreter environments, or dependent on the dynamic runtime context.

10.5.3. Supporting Activation of Language Constructs

Reconsider the schema evolution example from Section 10.3.2.2.2 (p. 229), where the bean managed persistence logic was updated with a static aspect. Although the code of the application must no more be manipulated. It is still inconvenient that the developer has to stop the application, add the aspect, and restart the application together with the aspect, which updated the bean implementation. Unfortunately, stopping the application for installing the aspect causes a downtime the bean's business application, which result may result in various loss of business values, such as loosing business opportunities during the downtime of the business application's services, loosing clients' trust in the availability, and risking service-level agreement penalties [CDM09].

Example Scenario: Therefore, consider implementing runtime schema evolution with dynamic AOP. Runtime schema evolution is enabled through adapting the entity bean at runtime using a dynamic AO4SQL aspect. Listing 10.27 presents the implementation of a dynamic aspect. The dynamic aspect implements the same adaptation as the static aspects, but it is not initially deployed when starting the application, since the AO programs declares this via the deployed:false parameter in line 3. All what is necessary to adapt the bean at runtime is to deploy the aspect in line 11, which immediately adapts the running application without downtime.

```
 1 //implements dynamic AO4SQL aspect
 2 DynamicAsepct myAo4SqlAspect =
 3   aspect(name:"BeanAOPTest",deployed:false){
 4     //the code from
 5     ...
 6     introduce_field (...)
 7     introduce_method(...) { ... }
 8     around(pSELECT()...){ ... }
 9   }
10 ...
11 myAo4SqlAspect.deploy(); //dynamically deploy aspect (adapt the bean class and embedded SQL)
12 ...
13 PersonBean person = PersonBean.ormFindByPrimaryKey(new PersonKey(1));
14 assert "musician".equals(person1b.getJob()); //access to introduced field
```

Listing 10.27: An AO4SQL aspect that changes an entity bean and its TINYSQL queries

Review: Although TINYSQL is a very limited subset of SQL, dynamic aspect-oriented programming for SQL becomes conceivable. For the implementation of dynamic scoping for AO4SQL aspects, it was not necessary to change the implementation of the CCCombiner in POPART/REA (activation: ●).

10.6. Enabling Pluggable Analyses

Section 3.6 (p. 56) has motivated to use language embeddings for scenarios where syntactic and semantic analysis is needed, as it is supported by traditional non-embedding-based approaches. Therefore, this section discusses embedding pluggable analyses on top of the architecture. Developers can implement rich domain analyses that perform composable pluggable syntactic analyses on DSLs, which Section 10.6.1 elaborates, as well as composable pluggable semantic analyses, which Section 10.6.2 elaborates.

10.6.1. Syntactic Analyses

For a syntactic analysis, it is crucial that the meta-objects of code blocks create method calls for every encountered expression. This enables interpretation of program expressions under abstract semantics.

Each syntactic analysis consists of an analyzer façade that implements a syntax-to-semantics binding for each expression type in the language to analyze to its abstract semantics. Further, it consists of an alternative language model that implements the abstract domain of the analysis. Optionally, it may consist of a meta-level that allows implementing generic analyses that adapt themselves to the context in which they are used.

Once a syntactic analyzer is implemented, its language façade can be used to evaluate programs consisting of expressions in abstract syntax. Evaluating a program with the façade of the syntactic analyzer is similar to traversing an AST. However, instead of visiting AST nodes, the analyzer façade receives a sequence of method calls for each expression. By analyzing the method calls resulting from the evaluation of the program closure, the analyzer calculates the result of the analysis.

The remainder of this section demonstrates how this architecture's model allows implementing flexible syntactic analysis.

Example Scenario: Reconsider implementing a syntactic analysis that checks that abstraction operators in FUNCTIONALLOGO contain not more than 10 expressions, as motivated in Section 3.6.1 (p. 57). For this analysis, a language developer needs to implement several parts. First, there is an expression counting part that performs a modular abstract interpretation that counts the number of expression contained inside a closure. Second, there is a checking part that performs a modular abstract interpretation that checks that the abstraction operator has not more than 10 expressions. The second part of the analysis uses the first part to determine the number of expressions. Note that the task of implementing the syntactic analysis is complicated by the fact that this analysis must be performed for a composite language, hence it is desirable to also implement the analysis for its constituent languages in a modular way. We present the implementation of these two parts of the analysis in the next two paragraphs.

Example Scenario – Expression Counter Analysis: To implement the first analysis, the developer implements an analyzer façade to count expressions for each constituent language. Listing 10.28 and Listing 10.29 presents the implementation of the analysis. There are two analyzer façades, namely ExpressionCounterCompleteLogo and ExpressionCounterFunctional. Both façades hold an expressionCount property (Listing 10.28, line 8 and Listing 10.29 line 3), which initially is 0. They implement their corresponding language interfaces, namely ExpressioCounter-CompleteLogo implements abstract semantics for ICompleteLogo and ExpressionCounterFunctional implements IFunctional. Note that there is no need to define a special language model for this analysis, because the *abstraction domain* can readily be mapped to the domain of Integers. For each expression type defined in the language interfaces, both façades provide a keyword method implementation that increases expressionCount (e.g., Listing 10.28 in lines 12, 14, 17, and Listing 10.29 line 6 and 12), so that when the expression type is used in a program the total number of expressions is increased. In addition to this, for each abstraction operator, the implementation calls the body in order to count its nested elements (e.g., cf. turtle in Listing 10.28 line 19 and define in Listing 10.29 line 8). The result of an analysis is accessible through the interface IExpressionCounter (cf. Listing 10.28) that each of the analyzers implements.

Next, to count expressions for both constituent languages, the developer has to combine the two modular analyzers. For this, the developer implements a custom combiner that composes the analyzer and their results. All the developer has to do is to implement the class Expression-CounterCombiner that is shown in Listing 10.30 that extends LinearizingCombiner. Note that the subclass can completely reuse the implementation of its super class, therefore no MOP specific code needs to be implemented. The developer only needs to implement the IExpressionCounter

```
interface IExpressionCounter extends DSL {
  int  getExpressionCounter();
  void reset ();
}

class ExpressionCounterCompleteLogo extends Interpreter
                                   implements ICompleteLogo, IExpressionCounter {
  protected int expressionCount = 0;
  public int  getExpressionCounter() { return expressionCount; }
  public void reset()  {  expressionCount = 0; }

  public int  getBlack()  {  expressionCount++; return 0; }
   ...
  public void forward(int n)  {  expressionCount++; }
   ...
  public void  turtle (HashMap params, Closure body) {
    expressionCount++;
    body.delegate = bodyDelegate;
    body.call ();
  }
   ...
}
```

Listing 10.28: Implementation of expression counting for COMPLETELOGO

```
public class ExpressionCounterFunctionalLogo extends Interpreter
                                             implements IFunctionalLogo,IExpressionCounter {
  protected int expressionCount = 0;
   ...
  public void define(HashMap params, Closure body) {
    expressionCount++; ...
    body.delegate = bodyDelegate;
    body.call ();
  }

  public Closure app(String name) {
    expressionCount++; ...
    return {  return null  };  //must return  an empty closure  to  avoid double counting
  }
}
```

Listing 10.29: Implementation of expression counting for FUNCTIONAL

(cf. Listing 10.28), so that the reset method resets all analyzers (line 10), and so that the getExpressionCounter combines the results of all analyzers (lines 4–8).

To recapitulate, the developer can implement the expression counting part of the analysis as a separate reusable analysis. Note that the analysis remains extensible through language polymorphism. Further, other analyses can use this analysis to perform a subordinate analysis in one of their steps, such as the checking parts done in the coding convention scenario as demonstrated in the following.

```
public class ExpressionCounterCombiner extends LinearizingCombiner implements IExpressionCounter {
  public ExpressionCounterCombiner(IExpressionCounter... dslDefinitions) { super(dslDefinitions); }

  public int getExpressionCounter() {
    return dslDefinitions . inject (0) { int partSum, IExpressionCounter ec −>
        return partSum + ec.getExpressionCounter()
    };
  }

  public void reset() { dslDefinitions .each { IExpressionCounter ec −> ec.reset() }; }
}
```

Listing 10.30: A custom combiner for combining the expression counting analyses

Example Scenario – Clean Code Analysis: To implement the checking part, the developer implements another analyzer that depends on the checking parts. Note that there are two design choices that can be made for implementing the second part. First, the language developers can implement this analysis as a *partial analysis*, since for the analysis only abstraction operators need to be analyzed. Second, by reusing an existing analysis, part of the implementation can be reused from existing analyses. Due to the above two design choices, there is less implementation effort, as demonstrated below.

When implementing several partial analyses for a language, language developers can define once an empty analyzer that completely implements the language interface, and let partial analyzers extended the empty analyzer. For example, for COMPLETELOGO, a language developer can implement the empty analyzer AbstractInterpretationCompleteLogo. This analyzer façade implements the ICompleteLogo interface with empty keyword methods[9]. This class has to be implemented only once, and language developers can inherit from the class and inherit its methods for convenience.

Listing 10.31 presents the implementation of CleanCodeAnalysisCompleteLogo that enforces that a maximum of 10 expressions in each abstraction operator. To reuse the existing expression counter analysis, the constructor takes an instance of the corresponding analyzer (exprCounterAnalyzer[10] in lines 2–5). The class inherits from a class AbstractInterpretationCompleteLogo, so that it does not have to implement the complete language interface. The subclass defines a partial analysis by overriding part of the keyword methods. For checking the coding convention in COMPLETELOGO, the subclass only has to override the methods for the abstraction operators turtle and repeat. To enforce a maximum number of nested expressions, these two methods use the method performSubordinateStepExpressionCounting (cf. lines 21–28), from which the both methods retrieve the number of nested expressions in the variable exprCnt. Whenever exprCnt exceeds the threshold of 10 expressions, the keyword methods raise an exception (lines 10

[9]To analyze expressions inside abstraction operators, their keyword methods are not completely empty, but they call their body.

[10]At this point, the exprCounterAnalyzer does not commit to a specific language interface. This allows developers to later combine it with other expression counter.

and 17). Finally, the methods propagate the analysis to nested expressions by calling their super methods that then call the abstraction operators body (lines 11 and 18).

The logic that reuses the expression counting analysis is contained in method performSubordinateStepExpressionCounting (lines 21–28). To count nested sub-expressions, this method performs the following steps. In line 22, it clones the body parameter. Note that it is crucial that a clone of the closure is created, so that the original delegate of the body is not altered. In line 23, it resets the counter of the exprCounterAnalyzer. In lines 24–27, this method uses the exprCounterAnalyzer to evaluate the clone of body parameter under the corresponding abstract semantics. Note that it is crucial, in line 25, that the resolveStrategy of the clone is set to DELEGATE_FIRST (cf. Section 5.1, p. 118), which changes the dispatch priorities of the closure's meta-object to dispatch keyword method calls first to the delegate and then to the owner. This is necessary in order to define the right semantics for the keyword methods in this step, because setting the right resolveStrategy enforces that the subordinate analyzer (i.e., the expression counter analyzer) has a higher priority than the enclosing analyzer (i.e., this) for this step.

```
 1 public class CleanCodeAnalysisCompleteLogo extends AbstractInterpretationCompleteLogo {
 2   private DSL exprCounterAnalyzer;
 3   public CleanCodeAnalysisCompleteLogo(DSL exprCounterAnalyzer) {
 4     this.exprCounterAnalyzer = exprCounterAnalyzer;
 5   }
 6
 7   public void turtle (HashMap params, Closure body) {
 8     def exprCnt = performExpressionCounting(body);
 9     if (DEBUG) println "turtle $params.name: "+exprCnt;
10     if (exprCnt > 10) throw new AnalysisException("Too much expressions in turtle '$params.name'.");
11     super.turtle (params, body);
12   }
13
14   public void repeat(int n, Closure body) {
15     def exprCnt = performExpressionCounting(body);
16     if (DEBUG) println "repeat: "+exprCnt;
17     if (exprCnt > 10) throw new AnalysisException("Too much expressions in repeat.");
18     super.repeat(n, body);
19   }
20    ...
21   protected int performExpressionCounting(Closure originalBody) {
22     Closure body = originalBody.clone();
23     exprCounter.reset();
24     body.delegate = exprCounterAnalyzer;
25     body.resolveStrategy = Closure.DELEGATE_FIRST;
26     body.call ();
27     return exprCounterAnalyzer.getExpressionCounter();
28   }
29 }
```

Listing 10.31: Implementation of coding convention analyzer for COMPLETELOGO

Listing 10.32 presents the implementation of the coding convention checker for FUNCTIONAL. Similarly, it only has to override the define (lines 5–9).

Note that, because the two modular coding convention checkers for COMPLETELOGO and FUNCTIONAL do not have dependencies, the language developers do not have to define a special combiner for this analysis. CleanCodeAnalysisCompleteLogo and CleanCodeAnalysisFunctional can be composed using the LinearizingCombiner.

```
 1  public class CleanCodeAnalysisFunctional extends Interpreter implement IFunctional {
 2     ...
 3    protected int performExpressionCounting(Closure originalBody) { ... }
 4
 5    public void define(HashMap params, Closure body) {
 6      def exprCnt = performExpressionCounting(body);
 7      if  (exprCnt > 10) throw new AnalysisException("Too much expressions in function '$params.name'.");
 8      super.define(params.name,body);
 9    }
10     ...
11  }
```

Listing 10.32: Implementation of coding convention checking analyzer for FUNCTIONAL

To apply the coding convention analysis, a developer only needs to compose the above modular analyses. Listing 10.33 demonstrates how to apply the complete analysis to a program. In lines 13–14, the setup step uses the ExpressionCounterCombiner to compose instance of ExpressionCounterCompleteLogo and ExpressionCounterFunctional into one composite analyzer that can be used for counting expressions. Next, the setup step uses the LinearizingCombiner to compose instances of CleanCodeAnalysisCompleteLogo and CleanCodeAnalysisFunctional to which it passes a reference to the composite analyzer for expression counting. This composition of analyzers is used as the delegate for evaluating the program under abstract semantics. When evaluating the program in Listing 10.33, the analyzer reports an violation of the coding convention by function dblsquare (lines 6–8).

Review: To partially answer the question raised by related work whether embedding approaches allow syntactic analyses [MHS05] and composable analyses [HORM08], the above demonstration shows that modular, extensible, and composable syntactic analyses can be implemented using REA and its support for composition. To review the current support, this section puts in relation what REA provides for embedded languages, what is already available in non-embedded approaches.

To investigate the general support, several more syntactic analyses have been implemented on top of the architecture:

Provided Keywords Analysis: A syntactic analysis that analyzes the language interface of an arbitrary reflective embedding for the keywords the language defines. Internally, this analysis uses *Java reflection* [AGH05, FF04] to introspect defined keywords.

```
Closure program = {
  turtle (name:"ThreeSquares",color:green) {
    define(name:"square") {
      fd 50; rt 90; fd 50; rt 90; fd 50; rt 90; fd 50; rt 90;
    }
    define(name:"dblsquare") {
      fd 50; rt 90; fd 50; rt 90; fd 50; rt 90; fd 50; rt 90; rt 45; fd 50; rt 90; ...
    }
    apply("square")();
    apply("dblsquare")();
  }
}
IExpressionCounter exprCntr = new ExpressionCounterCombiner(
  new ExpressionCounterCompleteLogo(),new ExpressionCounterFunctional());
program.delegate = new LinearizingCombiner(
  new CleanCodeAnalysisCompleteLogo(exprCntr),new CleanCodeAnalysisFunctional(exprCntr));
program.call (); // will raise an exception
```

Listing 10.33: Applying the coding convention analysis

Used Keywords Analysis: A syntactic analysis that analyzes an arbitrary DSL program for the keywords that the program uses. Internally, this analysis defines an analyzer as a meta-façade that implements MOP methods, such as methodMissing, in order to look up keywords.

Syntax Checker: A syntactic analysis that checks whether the abstract syntax of a DSL program is syntactically correct for a certain language or language composition. Internally, this analysis uses the provided keywords analysis to determine all available keywords. Then, it checks with the used keywords analysis whether all keyword used in the program are defined. Note that, the syntax checker provides more guarantees for programs than the default Groovy parser. The default Groovy parser cannot detect such errors, since it accepts every expression type in abstract syntax, which constitutes a legal method call.

To review pluggable syntactic analyses, one can investigate their support for possible evolutions and compare it to relate work, such as (1) extending analyses, (2) binding time, (3) composing analyses, and (4) generalizations of analyses:

(1) Extending Analyses: Obviously one can conclude from extensibility of façades reviewed in Section 10.1 that analyzers are extensible. Developing polymorphic analyzers is possible, since every expression type in abstract syntax yields a method call that the language polymorphism mechanism dispatches. Instead of dispatching to one closed analyzer façade, language polymorphism dispatches these method calls to the most concrete analyzer. The architecture's analysis model is based on polymorphic method calls—a key property that leverages the flexibility for (syntactic) analyses. Since each expression type casts on one specific signature, the meta-objects of a program's closures can create method call from them. Since each expression is dynamically dispatched as a method call, polymorphism provides type guarantees when extending analyzers by more specific ones.

(2) Binding Time: In REA, developers can plug analyses to their programs at any time. Because dispatch only takes into account an expression's signature, but not the type of the analyzer that receives the call, developers can dynamically plug various syntactic analyses on the same program representation. The use of a dynamic language is crucial to enable dynamically pluggable analyses. The architecture's program representation based on abstract syntax and dynamic syntax-to-semantic binding leads to this loose coupling.

(3) Composing Analyses: Developers can compose several independent analyses (e.g., where each of them analyzes one of several independent languages). Second, because method calls can be delegated to combiners, several syntactic analyses can by combined from modular analyses. The language developer can compose analyzer façades like composing the default façades with execution semantics. While it is problematic to add new expression types into an AST, in the reflective architecture new expression types can be mixed in easily using the language combiners.

However, it remains unclear so far, how several analyses decompose at their fix points, as it is usual in non-embedded approaches. Although it is possible to extend existing combiners, it is not a feasible solution for developers to implement a custom combiner with complex composition logic for each new combination of analyses.

(4) generalizations of analyses: Although the analysis model is similar to traversing an AST, since method calls in the architecture's analysis model are intercepted by the meta-level, this leverages more flexibility for syntactic analyses. While ASTs are generally bound to specific languages, thanks to the presence of the meta-level in the architecture, generic analyses can be implemented that can analyze several languages. For example, a generic analyzer can count expression for arbitrary embedded languages, which basically implements methodMissing and propertyMissing and increases the counter, whenever the MOP intercepts an arbitrary keyword method call and executes one of these methods. By implementing such a generic analyzer with a meta-level, one can intercept method calls to the façade using the MOP. Instead of executing concrete methods to calculate the analysis, the MOP can answer the calls directly via a meta-interpretation, or it can dispatch the calls to other methods.

While the above results are very encouraging, it remains unclear whether REA supports implementing complicated *context-specific* syntax analyses and *backward analyses*, which is a possible direction of future work (syntactic analyses: ◐).

10.6.2. Semantic Analyses

Section 3.6.2 (p. 58) has motivated evaluating programs under abstract semantics in order to determine semantic properties of programs. Therefore, this section elaborates how to implement semantic analyses as language façades that calculate semantic properties. To semantically analyze programs of a language, a language developer can define new alternative semantics from its expression types to any abstract domain.

Example Scenario: Reconsider validating a semantic contract for FUNCTIONALLOGO programs by checking a domain-specific constraint that requires that a *Logo* program (e.g., Listing 3.14) may use only positive integer values as parameters to *Logo*'s move commands, as motivated in Section 3.6.2. To calculate this semantic analysis, one does not have to actually execute the program, but one can perform an abstract interpretation that analyzes several facets of the program. Whereby developers can implement the analysis of each independent facet in isolation and compose those partial analyses later on.

There are several facets that are relevant for this analysis: (1) it needs to execute the *semantics of variables* with the scoping strategy, (2) it needs to *calculate arithmetics* (e.g., mathematical operations like "+" and "-") to determine the runtime values of variables used as parameters, (3) it needs to analyze functions, and (4) it needs to *check each move command* for semantic constraints. While certain facets of the semantic analyses are rather concrete others are rather abstract.

For the analysis, the first three facets are *concrete* without abstract semantics. Because it has to calculate the results of arithmetic operations of integers and variables, the analysis needs to calculate the scope and value of the used variables. Indeed, this section demonstrates that one can reuse concrete semantics for implementing part of the analysis, namely to analyze the semantics of variables and to calculate arithmetics. Further, it shows that one only needs to newly implement the part of the analysis that provides abstract semantics for checking the constraint as demonstrated below.

First, a developer can completely reuse the dynamically-scoped variable semantics from Section 10.5.1. For this facet of the analysis, the developer can reuse the implementation of EnvironmentInterpreter with DYNAMIC_SCOPING.

Second, thanks to homogeneous embedding in Groovy, the developer can reuse the host language mathematical operations for determining the values resulting from arithmetic calculations in FUNCTIONALLOGO. For this facet, the developer does not need to implement a special façade.

Third, the developer can reuse the concrete semantics of FUNCTIONAL to analyze functions. For this facet, the developer can reuse the implementation of Functional from Listing 5.10.

Forth, the developer needs to implement an analyzer with abstract semantics for checking the constraint. Consider implementing an abstract interpretation that raises an exception, whenever a negative value is passed to one of the move commands. For this, the developer implements the class PositiveValueAnalysisCompleteLogo inherits the empty abstract semantics from its super class AbstractInterpretationCompleteLogo (cf. Section 10.6.1). To check the constraint, the class only overrides the move operation. The overridden implementations only throw an exception in case the corresponding parameter has a negative value, otherwise move commands are just ignored. This analyzer abstracts over the concrete integer values processed by the move operations and only interprets negative values as errors.

Finally, the developer composes the four independent facet of this semantics analysis. Whereby technically, there is no difference between the composing concrete and abstract domains. As long as their domains are conflict free, the developer can use one of the existing language com-

```
class PositiveValueAnalysisCompleteLogo extends AbstractInterpretationCompleteLogo {
  public void forward(int n) { if (n < 0) throw new AnalysisException("n=$n to fd is negative."); }
  public void backward(int n) { if (n < 0) throw new AnalysisException("n=$n to bd is negative."); }
  public void right(int a) { if (a < 0) throw new AnalysisException("a=$a to rt is negative."); }
  public void left(int a) { if (a < 0) throw new AnalysisException("a=$a to lt is negative."); }
}
```

Listing 10.34: Semantic analysis for COMPLETELOGO (abstract semantics)

biners to compose the facets of the semantic analysis. Consider Listing 10.35 that compose the analyzers to semantically analyze a program. First, in lines 13–16, the program composes a composite semantic analyzer from instances of the above partial analyzers that analyze only their facet. When evaluating the EDSL program (lines 1–12), the composite semantic analyzer successfully detects the semantic error in line 11.

```
 1  Closure program = {
 2    define(name:"hexagon") { length ->
 3      turtle(name:"hexagon",color:red) {
 4        repeat (6) {
 5          forward length
 6          right 60
 7        }
 8      }
 9    }
10    i = 100
11    apply("hexagon")(50-i) //it detects the semantic error in this line
12  }
13  EnvironmentInterpreter env = new EnvironmentInterpreter();
14  env.scopingStrategy = EnvironmentInterpreter.DYNAMIC_SCOPING;
15  program.delegate = new LinearizingCombiner(
16    new PositiveValueAnalysisCompleteLogo(), env, new Functional());
17  program.call(); // will raise an exception
```

Listing 10.35: Applying the semantic analysis for Listing 3.14

Review: In the architecture, semantic analyses have the same qualities w.r.t. extensibility and composability as syntactic analyses. In the above scenario, the developer could reuse existing semantics and implement new facets of semantics as separate modules.

To investigate the general support, several more semantic analyses have been implemented on top of REA:

Position Tracker Analysis: A semantic analysis for FUNCTIONALLOGO that calculates the position of a turtle after executing all COMPLETELOGO commands inside a function body. The abstract domain is the 2D vector space. Internally, without drawing on a canvas, the position tracker calculates the relative position after each drawing command in the function's body. The result of this analysis is a relative position to the current position, and

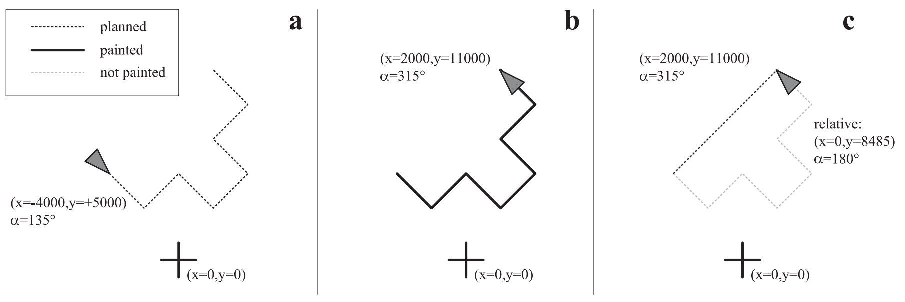


Figure 10.2.: Illustration of the position tracker analysis

a relative rotation to the current orientation. With the information of this analysis, the drawing of a function can be skipped. For example, Figure 10.2 illustrates the calculations of the position tracker analysis for an example. Initially, the turtle is at position $(x = -4000, y = 5000)$ and has an orientation of $\alpha = 135°$. It calls a function with the following commands: fd 2828;lt 90;fd 2828;rt 90;fd 2828;lt 90;fd 2828;lt 90;fd 2828;rt 10;fd 2828;. In Figure 10.2, index a presents the turtle at its initial position, index b shows the turtle at the final position after drawing the function, index c shows the relative path of the turtle to skip the function application. To skip painting the function, the turtle must execute the commands: (i) forward 0: to goto relative position $y = 0$, (ii) left 90; forward 8485; right 90: to goto relative position $x = 8485$, and (iii) right 180: to turn to relative orientation $\alpha = -180° \equiv 180°$.

Time Estimation Analysis: A semantic analysis for FUNCTIONALLOGO estimates the amount of time to draw a complete turtle program or only a function with a sequence of commands. The abstract domain is the absolute time unit in ticks. Internally, for each FUNCTIONALLOGO command, the analysis assumes a constant number of time units to draw it. The time estimation integrates over the complete program, without drawing, and sums up the estimates time per command. The result of this analysis can be used to estimate the cost to draw a function or FUNCTIONALLOGO program.

The above analyzes demonstrate the feasibility of implementing simple semantic analyses for DSLs. For the COMPLETELOGO keywords, implementing a complete semantic analysis is not a problem, since the analyzer can map expressions straight-forward to the abstract domain. Even, repeat expressions are no big problem. Since the number of iterations in COMPLETELOGO's repeat is constant and not based on a dynamic condition, the analyzer can easily map them to the abstract domain. However, for the FUNCTIONAL semantics, the above analysis implementation is not a complete and sound abstract interpretation, since the current implementation disregards recursive function calls. Loops and recursion are a general problem for such analyses that is extensively discussed in the literature [CC77, Cou96]. In the presence of loops and recursion, generally, analyses need to use fix points to abstract over loops and recursive calls. Still, implementing sophisticated abstract interpretations with fix points, it is out of the scope of this thesis.

While the above results are very encouraging, it remains unclear whether REA supports implementing complicated semantic analyses that are composed at fix points and backward analyses, which needs to be addressed in future work (semantic analyses: ◐).

10.7. Enabling Pluggable Transformations

With language polymorphism and transformer façades, language developers can define custom transformations, as these have been motivated in Section 3.7 (p. 59). Similarly to an analyzer façade, the transformer first generates objects of a language model from expressions of a program. Then, it analyzes these objects, and finally transforms the model to an output model or to target code. Compared to most other transformation approaches, what is special with transformation in REA is that original expressions and transformed expressions are first-class method calls in the host language. It is this property that all expressions are polymorphic method calls that enables extending and composing transformations with the REA's mechanisms.

In REA, language developers can embed transformations as first-class objects. What is special in REA, it that it supports both exo- and endo-transformations (cf. Section 3.7, p. 88). Therefore the developer has the choice whether to endo-transform on the causally connected program representation or to exo-transform the program to another target language with a textual representation that is not causally connected.

In the following, this section demonstrates how language developers can implement both static and dynamic pluggable transformations.

10.7.1. Support for Static Transformations

As discussed in Section 3.7.1 (p. 60), static transformations can either take into account syntactic or semantics information of the program.

10.7.1.1. Syntactic Transformations

REA supports transforming expressions in abstract syntax, which are represented as method calls, by transforming these method calls to alternative method calls that represent different expressions in abstract syntax.

Example Scenario: Reconsider implementing the desugarizing transformation from Section 3.6.2 (p. 58) that replaces user-defined command expressions in FunctionalLogo by a corresponding function applications, such as the expression "hexagon 50" is mapped to a function call apply("hexagon")(50).

Consider implementing this transformation by adding a meta-level extension to Functional. Since the transformation for shortcut applications of functions is not only relevant for FunctionalLogo, but generally interesting for Functional, one can modify the Functional façade with an inline meta-level, which turns the class Functional into a transformer/meta-façade and completes the excerpt of its implementation from Listing 5.10. To transform user-defined command expressions to function calls, the meta-façade intercepts every usage of a user-defined command expression and transforms it to a function application. To intercept user-defined command expressions that cast on method calls to the meta-façade, the meta-façade overrides methodMissing (lines 8–14). Since the façade cannot define concrete methods for each user-defined command expression, it uses the methodMissing method to reify every method call to a pretended method. It analyzes the reified information of such a call in order to determine the abstract syntax of the user-defined command expression of which this call was created from. It uses the methodName to look up the corresponding function (line 9). If a function with a corresponding name is defined, methodMissing applies this function using the reified parameter as parameter to the function (line 10) and returns its result. If no function is defined, methodMissing calls its super method, which by default will raise an exception that an undefined method is called.

```
 1 class Functional extends Interpreter implements IFunctional {
 2   HashMap<String,Closure> definedFunctions = new HashMap<String,Closure>();
 3
 4   void define(HashMap params, Closure body) { ... /* code from Listing 5.10 */ }
 5
 6   Closure apply(String name) { ... /* code from Listing 5.10 */ }
 7
 8   Object methodMissing(String methodName, Object args) {
 9     if (definedFunctions.get(methodName) != null) {
10       return apply(methodName).call(*args);
11     } else {
12       return super.methodMissing(methodName, args);
13     }
14   }
15 }
```

Listing 10.36: Completed implementation of FUNCTIONAL from Listing 5.10

To evaluate programs, the meta-façade with the transformation is set up in the same way as a normal façade. When evaluating a program and encountering user-defined command expressions, such as hexagon 50 from Listing 5.10, the code block's meta-object produces a method call for the user-defined command to the meta-façade. Then, the methodMissing will loop back the transformed version of the method call to itself, e.g., it calls apply("hexagon"), that returns a closure. The subsequent evaluation then applies the returned closure with the parameters from the original call, e.g., 50.

Review: To generally review the support for static transformations, it is important to investigate a potential transformation w.r.t. (1) what language it transforms to, (2) whether it abstracts over concrete expression types, (3) the locality where programs are transformed, and (4) how transformations can rewrite expression types. In the following, this section elaborates each point.

First, the architecture distinguishes a transformer w.r.t. what target language it transforms expressions to. When a transformation is an exo-transformation from one language to a completely different language, a developer needs to implement a completely new façade class and language model for the transformer. Conversely, when the transformation is an endo-transformation, then the transformer can be implemented as an extension to an existing language façade, which reuses part of the language model.

Second, the architecture distinguishes a transformer w.r.t. if it can abstract over concrete expression types. When it only transforms one concrete expression type, the developer needs to provide a concrete keyword method implementation for this expression type's signature in the transformer façade. When it transforms several expression types in a similar way, the developer needs to implement a meta-method or meta-object to realize a meta-transformer façade. Either, one of the inline meta-methods can be overridden, such as invokeMethod or methodMissing, or a separate meta-object with a corresponding implementation can be implemented for the façade or a domain type in its model.

Third, the architecture distinguished local and global transformations. Local transformations need to rewrite one concrete expression into another concrete expression (like a rewrite/transformation rule). Therefore, the developer can implement a local transformation as a concrete method in the transformer façade, whereby the transformer can ignore the enclosing lexical context of the expression. This method receives an original expression's method call and manipulates it to a corresponding transformed call. Conversely, global transformations need to quantify over multiple expression types and rewrite them according to a certain strategy (like traversal functions [Cor06] or strategies [Vis97b]). Therefore, the developer needs to implement a keyword method that takes into account its context, which can either be extracted from the internal state of transformer or inspected from the stack. Global transformations on several expression types are enabled, when a transformer façade implements a meta-method, such as methodMissing() or invokeMethod(). To allow abstracting over expression types in abstract syntax, these meta-methods allows intercepting several calls to the keyword methods in abstract syntax.

Fourth, in REA, every expression type in abstract syntax can be transformed via the method call it produces. A meta-transformer façade receives this method call and produces another method call, which again represents an arbitrary expression in abstract syntax. In the transformation, the meta-transformer uses reflection to transform method calls on the façade or inside its model. Using introspection, it can analyze the embedded source and the target languages. Using intercession, it can manipulate the call method name and parameter and receiver. Finally, the possibly repetitively manipulated expressions can be reflected back into the system.

If syntactic transformation in REA are feasible, still, the developer can use TIGERSEYE/REA pre-processor to transform it in an appropriate form (syntactic transformations: ●).

10.7.1.2. Semantic Transformations

REA supports transforming a program's semantics by manipulating the receivers and parameters of method calls.

Example Scenario: Reconsider implementing OPTIMIZINGLOGO that optimizes applications of functional abstractions by transforming them such that these expressions are executed in parallel. Since this transformation does not transform to another language, it is an endo-transformation.

To implement OPTIMIZINGLOGO, a language developer needs to implement a new parallelizing transformation that transforms function applications. Listing 10.37 presents an excerpt of the OptimizingLogo class that implements this transformation. This endo-transformer executes FUNCTIONALLOGO code and transforms the code for optimization. For optimization, the transformer needs to specialize the semantics of functions, so that functions are optimized. Therefore, in line 1, the OptimizingLogo class inherits from ConcurrentLogo in order to reuse the semantics of CONCURRENTLOGO for implementing ICompleteLogo, and in addition, it reimplements IFunctional in order to specialize the semantics of its FUNCTIONAL part.

To parallelize the semantics of functions, first and foremost, the OptimizingLogo façade overrides the method apply (Listing 10.37, lines 3–41) to transform function applications. To transform a function application, there are several tasks to accomplish in order to replace a direct function application by a parallelized one. The transformation looks up the called function (line 5) and clones this (line 6). It looks up the first-class representation for turtle of the current thread (tTurtle in line 9). It wraps the original function into a closure (wrappedFunction in lines 12–39) that will parallelize the call.

The wrapped closure (Listing 10.37, lines 12–39) transforms a wrapped function application as follows. It delegates the tasks of executing the function from the original turtle (i.e., tTurtle) that calls the function to a new turtle that is called delegateTurtle (cf. lines 22–30). Internally, it uses the position tracking analysis to calculate the relative position the original turtle would have after itself would paint the function. It also stores the old position of the original turtle (line 18). It creates the delegateTurtle, which will go to the original turtle's old position (lines 13–16), and which then will call the function for the original turtle. Knowing that the delegate will take care of calling the function, the original turtle can skip the function call by moving to the relative position obtained by the position tracking analysis (lines 32–36). Finally, instead of returning the function itself, the apply method returns the wrapped closure that will parallelize the function application when the optimize function is applied (line 40).

When evaluating a program that is written with only one turtle, the OptimizingLogo can parallelize and therefore speed up its execution. For example, Listing 10.38 shows an example program that only defines one turtle (lines 3–12), and therefore it would be executed, e.g., by ConcurrentLogo in only one thread. In contrast, when using OptimizingLogo to evaluate the same program, the transforming façade parallelizes the program to speed it up.

Figure 10.3 compares the sequential to the parallel execution of the same program. While the left figure illustrates an execution with ConcurrentLogo with only one turtle ("IceFlower"), the right side illustrates executing the same program with OptimizingLogo, which create four additional turtles ("Opt#polygonX") that paint the blossoms of the flower in parallel.

Note that in this example optimization, there are several assumptions that hold for the scenario of controlling a fleet of physical turtle robots, as motivated in Section 10.7.2. First, in the example scenario, there are fixed time constants[11] to execute a certain move operation of a

[11] In the simulation of optimizing the move operation of physical turtles, forward and backward take 2 time units per steps to move, and left and right take 1 time unit per degree.

```
class OptimizingLogo extends ConcurrentLogo implements ICompleteLogo, IFunctional {
  ...
  public Closure apply(String name) {
    if  (!definedFunctions.containsKey(name)) throw ...
    Closure function  = definedFunctions.get(name);
    def functionForWrapper = function.clone();
    ...
    Thread thread = Thread.currentThread();
    Turtle  tTurtle  = threadToTurtle.get(thread);

    //calculating  position  after  drawing the  turtle
    Closure wrappedFunction = { Object... args ->
      def positionTracker  = new PositionTrackingLogo(); //calculate  position  after  drawing the function
      functionForWrapper.delegate = positionTracker;
      functionForWrapper.resolveStrategy = Closure.DELEGATE_ONLY;
      functionForWrapper(*args); //position  tracking  (the  *−operator resolves args  as  parameters of  the  call )

      int oldX = tTurtle .position .x;  int oldY =  ...;   int oldAngular =  ...;

      //create  new concurrent  turtle   that  draws the  function
      def delegateTurtle  = super.turtle (name:("Opt#"+(tTurtle.name+(count++))),
                                          color :( tTurtle .penColor.value)) {
        penup(); //set  up delegate  turtle  to  current  position  of  original  turtle
        fd oldY;  //( i )  bring delegate  turtle  in  y  position  of  original  turtle
        rt 90;  fd oldX;  lt  90;  //( ii )  bring delegate  turtle  in  x  position  of  original  turtle
        rt oldAngular; //( iii )  bring delegate  turtle  in  orientation  of  original  turtle
        pendown();

        functionForWrapper.call(*args);  //( the  *−operator resolves args  as  parameters of  the  call )
      }

      penup() //set  up original  turtle  to  relative  position  after  painting  the  function
      fd  positionTracker .deltaY  //( i )  bring delegate  turtle  in  new y position
      rt 90;  fd  positionTracker .deltaX;  lt 90  //( ii )  bring delegate  turtle  in  new x position
      rt positionTracker .deltaAngular //( iii )  bring delegate  turtle  in  new orientation
      pendown()

      return delegateTurtle ;
    }
    return wrappedFunction;
  }
  ...
}
```

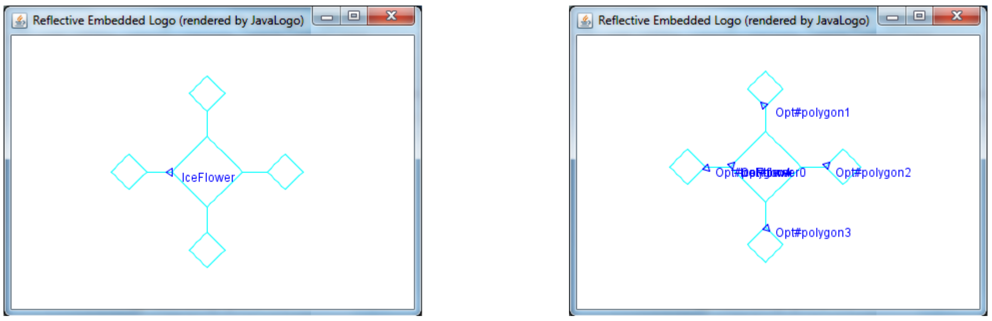
Listing 10.37: Excerpt of the OPTIMIZINGLOGO transformer façade

turtle that is controlled by the program. The execution of the entire program is finished, after all remote turtle robots have finished their drawings. Second, it assumes that there is no data flow or side effects between functions. Scheduling and isolating multiple threads of execution is a complicated problem that is out of the scope of this thesis.

Review: Thanks to the flexibility that a transformer façade has access to the first-class representations of the program expressions, it can transform the program. The example optimization

```
Closure singleTurtleProgram = {
  define(name:"polygon") { int length, int edges -> ... }
  turtle (name:"IceFlower",color:blue+green) {
    edges = 4; length = 50; angle = (int)(360 / edges);
    halfAngle = (int)(angle / 2); halfLength = (int)(length / 2);
    right 45;
    repeat (edges) {
      forward length; left(90-halfAngle); forward halfLength; //paint peduncle
      left(90-halfAngle); polygon(halfLength,edges); right(90-halfAngle); //adjust for blossom
      backward halfLength; right(90-halfAngle); right angle;
    }
  }
}
singleTurtleProgram.delegate = new OptimizingLogo();
singleTurtleProgram.call ();
```

Listing 10.38: Using OptimizingLogo to optimize a program with only one turtle


Reflective Embedded Logo (rendered by JavaLogo)
IceFlower
Reflective Embedded Logo (rendered by JavaLogo)
Opt#polygon1
Opt#polygon2
Opt#polygon3

(a) Sequential execution with ConcurrentLogo

(b) Parallel execution with OptimizingLogo

Figure 10.3.: Comparison of sequential and parallel execution of Listing 10.38

implemented is a very simple one, but more sophisticated transformations are conceivable. Despite this, there are practicable limitations. Although only static information is accessed, in REA, transformations must be actually performed at runtime. Performing a static transformation at runtime is rather awkward, since there are costs that are associated with the optimization itself that may slow down the execution. From the perspective of the cost of the transformation, it is disadvantageous that the static transformation is performed dynamically. Still, the concrete example scenario the runtime overhead is no problem, since the bottleneck to be optimized is greater than this overhead.

To review the support for parallelization in OPTIMIZINGLOGO, execution times of Listing 10.38 with and without parallelization were measured. Figure 10.4 illustrates how OptimizingLogo delegates the function applications of the “IceFlower” turtle to its delegate turtles. The first line illustrates the program execution by a single turtle, as the program would be painted without transformation. The second line illustrates the paint operation by the original turtle, which delegates painting the blossoms to its delegate turtles. The third line illustrates the paint

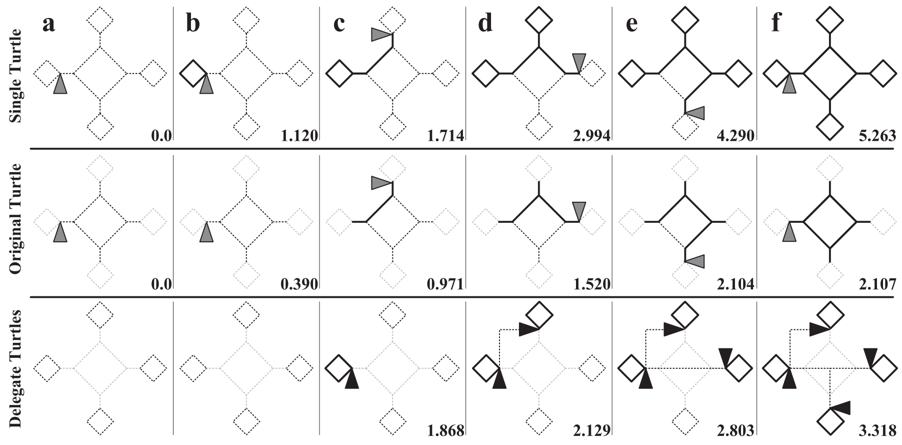


Figure 10.4.: Parallelization through delegating function calls to delegate turtles

operations by all delegate turtles. In numbers in the lower right corner of the boxes indicate the time that was measured for each snapshot. For this experiment, there is a measured speed up of $S_p = \frac{T_1}{T_p} = 1.586$, which was calculated from the sequential execution time $T_1 = 5.263$ (right most figure in upper line) and the parallel execution time $T_p = 3.318$ (right most figure in lower line).

REA does not target for high-performance optimizations, but for maximum flexibility. Still, if high-performance optimizations are required by a language, the language developer can implement a transformer façade that transforms first-class expression and finally generates the transformed code. The developer then can compile the generated code using the Groovy compiler or another compiler, depending on the target language. In this case, the costs of the transformation must not be paid at runtime. For example, in REA, there is a generator that embeds a first-class representation of SQL expressions in Groovy and generates SQL queries. A language developer could extend this generator to write transformations on SQL queries.

Despite this, in the end, it is expected that the current implementation of REA in Groovy is not suitable for high-performance optimizations, in particularly, because there is a large execution overhead due to indirections of the Groovy MOP (e.g., for every method call, for autoboxing primitive values, and for arithmetic operations). In case, there are hard performance requirements, it would be recommended to re-instantiate REA in a compiled language where the compiler can remove the MOP indirections, such as CLOS.

While the above results are very encouraging, it remains unclear whether REA supports traditional semantic transformations, as they are supported by compilers. Usually such transformations need access to the basic-block level, such as *loop unrolling*. In the current implementation of REA in Groovy, reflecting on the basic-block level is currently not addressed, and therefore, it needs to be addressed in future work (static transformations: ◖).

10.7.2. Support for Dynamic Transformations

Section 3.7.2 (p. 62) has motivated dynamic adaptive optimizations for DSLs. Therefore, consider implementing such adaptive optimizations using a dynamic transformation of which the optimization decision depends on the runtime context.

Example Scenario: Reconsider implementing the dynamic transformation AdaptiveOptimizingLogo motivated in Section 3.7.2 (p. 62) that takes into account whether it is worth optimizing function applications. Listing 10.39 shows an excerpt of the implementation of this transformation. It overrides the apply keyword method to wrap function applications into a closure (lines 6–19). Before optimizing a function application, in line 7, the wrapping closure estimates the cost (i.e., baseCost) for drawing the wrapped function. This cost estimation needs to take into account the function's dynamic arguments, which are obtained from the parameters to the wrapped closure args. Next, in line 8, it calculates the potential initialization costs (i.e., initCost) to bring a delegate turtle from position $(x = 0, y = 0)$ to the position of the original turtle. After estimating the costs, in line 10, there is the optimization decision that depends on the dynamic condition, since the actual costs depend on the current position of the turtle and the parameters to the function call. The transformation only optimizes the program, when the condition is met that the potential saving (of the costs to draw the function) exceeds the costs for starting a potential new turtle. After the decision, in case the decision was to optimize the corresponding function application, the optimizer calls the apply of the super class OptimizingLogo. Otherwise, it calls the non-optimized function.

```
 1  class AdaptiveOptimizingLogo extends OptimizingLogo {
 2    ...
 3    Closure apply(String name) {
 4      Closure function = functionNamesToClosure.get(name).clone();
 5
 6      Closure wrappedFunction = { Object... args ->
 7        int baseCosts = ... // estimate time to draw function with time estimation analyses (cf. Section 10.6.2)
 8        int initCosts = ... // calculate time for a delegate turtle to goto (x,y) position
 9
10        if (baseCosts > initCosts) {
11            // returning the optimized function
12            Closure optimizedFunction = super.apply(name).clone();
13            return optimizedFunction.call(*args);
14        } else {
15            // returning the non-optimized function
16            return function.call(*args);
17        }
18      }
19      return wrappedFunction;
20    }
21  }
```

Listing 10.39: Excerpt of the AdaptiveOptimizingLogo transformer façade

Review: Thanks to the flexibility that one can dynamically transform the first-class representations of program expressions at runtime, an adaptive optimization could be implemented using an endo-transformation that transform a program while it executes. Indeed, the above optimization implements an *on-line feedback system* [AFG$^+$00].

Note that there can be also costs associated with having such a feedback system at runtime. Still in this scenario, the computation costs of the adaptive optimization's *on-line feedback system* [AFG$^+$00] that calculates the optimization decision takes far less time to compute, compared to performing the move operations of the turtles.

While the computing time for an analysis performed at runtime is such as that it assumes that there are always enough resources. In contrast, parallelizing the physical moves can lead to a speed up in magnitudes, depending on the concrete program and number of available robots.

For example, to determine the execution times, one can perform an abstract interpretation or measure sample executions at runtime. Both analyses come with certain costs – either for executing the abstract interpretation or the overhead of collecting samples at runtime. Further, there are costs due to the transformation of the program. With an adaptive optimization, it is important to make sure that the costs of the feedback system and performing the optimization do not actually exceed the potential savings.

While dynamic transformations are technically supported in REA, when programs use such transformations, it poses a constant overhead for their execution. Currently, REA does not address partial evaluation and optimization of dynamic analyses. Optimizing dynamic analyses is required for acceptable performance of programs. It therefore needs to be addressed in future work (dynamic transformations: ◐).

10.8. Review Result

Table 10.3 presents a detailed results of this review and compares it to related work. The right most column shows the marks of REA's support for the desirable properties. The results are presented in comparison with the qualities by related embedding approaches at the left-hand side. REA has support fully support for the properties, except the mentioned limitations with implementing analysis and transformation.

Table 10.3.: Comparison with related embedding approaches

Desirable Property (○ : partial support, ◖ : important limitations, ●: full support)	Pure Embed. [Hud96]	Tagless Embed. [CKS09]	Unembedding [ALY09]	Jargons [Pes01]	EDSL Groovy [KG07]	EDSL Ruby [TFH09]	TwisteR [AO10]	π Pattern Lang. [KM09]	Helvetia [RGN10]	Staged Interpr. [COST04]	Ext. Meta-Prog. [SCK04]	Converge [Tra08]	Fluent [Eva03, Fow05]	EMF2JDT [Gar08]	Scala EDSL [OSV07]	Polymorphic [HORM08]	Embed. Gen. [Kam98]	Embed. Compiler [EFDM03]	Ruby Gen. [CM07]	MetaBorg [BV04]	TXL [Cor06]	**Reflective Embed. (REA)**
3.1. Extensibility	○	○	○	○	◖	◖	◖	◖	○	○	○	○	○	○	○	○	○	○	○	○	○	●
3.1.1. New Keywords	●	●	●	◖	●	●	●	●	◖	○	◖	○	○	○	●	●	◖	◖	◖	◖	◖	●
3.1.2. Semantic Extensions	○	○	○	○	◖	◖	◖	◖	○	○	○		○	○	○	○				○	○	●
3.1.2.1. Conservative Extensions	●	●	●	○	◖	◖	◖	◖	○	●	●		○	◖	◖	◖				○	○	●
3.1.2.2. Semantic Adaptations					◖	◖	◖	◖	○	○												●
3.2. Composability of Languages	○		○		○	○	○	○	◖				○	○	○	○						●
3.2.1. Languages without Interactions	●		●		●	●	●	●	●				◖	◖	●	●						●
3.2.2. Languages with Interactions	○		○		○	○	○	○	◖													●
3.2.2.1. Syntactic Interactions	◖		◖		○	○	○	◖	◖													●
3.2.2.2. Semantic Interactions					○	○	○		◖													●
3.3. Composition Mechanisms	○		○		○	○	○	○	○				○	○	○	○				○	○	●
3.3.1. Syntactic Interactions	◖		◖		○	○	○	○	○				○	○	◖	◖				○	○	●
3.3.1.1. Conflict-Free Compositions	◖		◖					◖	○				◖	◖	◖	◖				●	◖	●
3.3.1.2. Resolving with Renaming	◖		◖		◖	◖	◖	◖							◖	◖				●	◖	●
3.3.1.3. Resolving with Priorities					◖	◖	◖		●						◖	◖				●	◖	●
3.3.2. Semantic Interactions							○		○											○	○	●
3.3.2.1. Crosscutting Composition							◖		◖											○		●
3.3.2.2. Resolving Composition Conflicts																				○	◖	●
3.4. Concrete Syntax	○	○		○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●
3.4.1 Concrete-to-Abstract							○	●	●								○			●	◖	●
3.4.2 Mixfix Operators	○	○	○	○	○	○	○	●	●	●	○	○	○	○	◖	◖	○	○	○	●	●	●
3.4.3. Overriding Host Keywords					●			●	●	●	●	●										●
3.4.4. Partial Syntax																						●
3.5. Pluggable Scoping				○			○	○												○	○	●
3.5.1. Dynamic Scoping				○																○	○	●
3.5.2. Implicit References								●												●	●	●
3.5.3. Activation of Constructs							○	○												○	○	●
3.6. Pluggable Analyses		○	◖				◖		○	○	○	○				◖		○	○	◖	◖	◖
3.6.1. Syntactic Analyses		○	◖				◖		○	○	○	○				◖		○	○	●	●	◖
3.6.2. Semantic Analyses		○	◖				◖		○	○	○	○				◖		○	○	◖	◖	◖
3.7. Pluggable Transformations		○	○				○		○							○		○	○	○	○	◖
3.7.1. Static Transformations		○	○				○		◖	●	●	●				◖		○	○	◖	◖	◖
3.7.1.1. Syntactic Transformations		○	○				○		◖	●	●	●				◖		○	○	◖	◖	●
3.7.1.2. Semantic Transformations		○	○				○		◖	●	●	●				◖			○	◖	◖	◖
3.7.2. Dynamic Transformations							○															◖

10.9. Summary

This section has evaluated the quality of the support for language evolution of the reflective embedding architecture. REA has an excellent support for five of the seven desirable properties:

1. REA supports *extensibility*.
2. REA supports *composability*.
3. When existing extensibility or composability mechanisms are inadequate, REA allows to define new appropriate mechanisms on top of existing *open composition mechanisms*.
4. With the generic TIGERSEYE pre-processor, REA supports also incrementally and partially defined *concrete syntax* for embedded DSLs.
5. REA supports *pluggable scoping* to allow different scoping than the host language.
6. Although REA support modular, composable, and *pluggable analyses* for different languages, it remains to be shown whether REA supports composition of analyses with fix point semantics.
7. There is support for *pluggable transformations*, but it remains to be shown how traditional compiler analyses can be implemented and how the runtime overhead of dynamic analyses can be improved.

With the above support for the desirable properties, REA is readily equipped for language evolution.

11. Costs of Reflective Embedding

This chapter measures the costs of reflective embedding and compares them with the costs of related work.

Benchmarks for GPLs are broadly available for comparing mainstream languages, such as the SPECjvm2008 benchmark[1] for Java. Unfortunately, there are no sophisticated benchmarks for DSLs [KLP$^+$08]. Even for standard DSLs that are broadly used in industry, such as the BPEL workflow language [AAB$^+$07], there are currently no benchmarks available. Having a benchmark for DSLs would be interesting, because it would allow comparing the costs of different DSL approaches.

An open source benchmark for DSLs: To address the lack of a DSL benchmark, in the context of this thesis, an open source benchmark was implemented that compares different DSL technologies. The thesis uses this benchmark to compare related DSL implementation approaches—also approaches that are not based on embedded DSLs. The basic idea of the open source benchmark is to implement the same DSL with different technologies and then to compare the qualities of these resulting DSL implementations.

To compare the costs of the different DSL technologies, the benchmark's experiment is always to implement the same DSL[2] with different technologies. Instead of comparing only embedded DSL approaches, the benchmark compares also to traditional DSL approaches that are popular and/or often mentioned in related work. Each selected approach is a representative of a certain category of technologies for DSL implementation in related work. The benchmark compares the following categories of DSL approaches:

(1) parser generators: Bison/C [LMB92], ANTLR/Java [Par93];

(2) extensible DSLs: MontiCore/Java [KRV08];

(3) embedded DSLs: DSL2JDT/Java (homogeneous) [Gar08]; MetaBorg/Stratego (heterogeneous) [BV04]; REA/Groovy (homogeneous); TIGERSEYE/Groovy (hybrid)

(4) commercials-of-the-shelves: Apache SCXML[3].

[1]SPECjvm2008: `http://www.spec.org/jvm2008/`.

[2]As a starting point, the benchmark uses the DSL for specifying state machines (FSMDSL) from Section 6.2. Note that, for the measurements, it can be expected non-significant which particular DSL is selected for comparison. But for a fair benchmark, various types of DSLs with different complexities would need to be selected.

[3]Apache SCXML executes a state machine definition in abstract XML syntax. More information can be found on the Internet. The Apache SCXML Homepage: `http://commons.apache.org/scxml/`.

To compare the qualities of the resulting DSL implementations, the benchmark compares various metrics:

(a) size of *hand written code* for DSL implementation in effective lines of code (eLOC),

(b) size of *generated code* for DSL implementation in eLOC,

(c) *execution time of the same DSL program* under various problem sizes,

(d) *memory usage of the DSL program* under various problem sizes,

(e) *correctness* of expected output,

(f) *syntactic noise* in DSL programs, which is visible for end users.

It is out of the scope to present all details of the benchmark results. Only the most important results are presented here. The complete report with all benchmark results can be downloaded from: `http://www.stg.tu-darmstadt.de/research/ao4dsls`.

The benchmark compares the implementation costs of the various technologies by implementing FsmDSL with them, by measuring the above metrics for each of the different implementations, and finally by comparing their metrics, which is similar to the approach taken in Kosar et al. [KLP$^+$08]. Similar to Kosar et al., the benchmark of this thesis measures the eLOC metric to compare implementation costs and execution times. Like them, we also compare the implementations of the same DSL with different technologies, but Kosar et al. use another DSL for the specification of feature models. Unfortunately, Kosar et al.'s benchmark is neither open source nor available for download. In contrast, the benchmark proposed in this thesis allows other researchers to validate and reproduce results more easily.

Note that in all experiments presented in this chapter, the benchmark uses same problem setting and problem size to compare the different technologies. All quantitative benchmark measurements were executed on an Intel Pentium 4, with 2.99 GHz, and 4GB RAM, Debian Linux, kernel version 2.6.2. Further, note that the benchmark is not completely fair for REA, because it compares REA to technologies that do not have the same flexibility for language evolution, partial definition of concrete syntax, and so on like they are supported by REA. Still, the benchmark results can be used as a rough estimation of the implementation and runtime costs.

11.1. Implementation Costs of Reflective Embedding

This section evaluates the costs for implementing DSLs, for compiling and for executing DSL programs.

11.1.1. Implementation Costs of Domain-Specific Languages

The discussion about the implementation costs for embedding DSLs distinguishes costs for syntax (Section 11.1.1.1) and semantics (Section 11.1.1.2).

11.1.1.1. Domain-Specific Syntax

This section compares the costs for DSL syntax. Table 11.1 shows the syntactic qualities and compares the support for partial syntax. It compares this thesis' approaches both with compromised syntax (REA) and with concrete syntax (TIGERSEYE) to related work. The support for partial syntax and the syntactic qualities determines syntactic costs of the corresponding embedding approach. Only, when partial syntax is supported, the cost of implementing the grammar is significantly reduced, as it was shown in Section 10.4.4.

Specifically, Table 11.1 compares the other embedding approaches with TIGERSEYE (from left to right) w.r.t.: (1) what host languages are supported out of the box, (2) if the approach is host language independent, and, if so, (3) if the grammar of the complete host language has to be available. Further, (4) it compares the "embedding style" and (5) if the approach supports "concrete syntax", (6) whether a complete specification of the DSL's grammar is required. Finally, (7) it compares the class of languages supported by the approach. It classifies the TIGERSEYE approach as a *hybrid* embedding approach. Since TIGERSEYE is a heterogeneous but well-integrated pre-processor for homogeneous embedded DSLs. For the table one can see that currently TIGERSEYE is the only approach that reduces the costs for concrete syntax by employing special techniques like island grammars.

To sum up, the TIGERSEYE/REA approach reduces the costs of EDSLs with concrete syntax by building upon partial syntax definitions and its advantage is that it leaves it up to the developers to decide when they want to incrementally add concrete syntax. Although, the Stratego language and TXL have principle support for island grammars, they currently do not use island grammars to reduce the costs for embedding.

11.1.1.2. Domain-Specific Semantics

Table 11.2 gives a rough estimation of the implementation effort with the different DSL approaches. The table compares the eLOC of the FSMDSL implementations with the different technologies. The benchmark counts lines of user-defined and generated code.

W.r.t. comparing the implementation effort, only SCXML—the commercial-of-the-shelves approach—has less eLOC than the embedding in REA. Naturally for a COTS-based approach, virtually no code has to be implemented for a DSL that comes from the shelves. It is not surprising that re-using a DSL is cheaper than implementing it. For SCXML, only a thin wrapper needed to be implemented in order to integrate the SCXML implementation into the benchmark. REA was the second cheapest approach in term of eLOC. Only a few line more were added to enable concrete syntax for FSMDSL. For the embedded approaches, no code had to be generated. All other DSL technologies required more eLOC. Fortunately, the DSL specifications in their meta-language are very concise, still a vast amount of code is generated from it. Note that the benchmark does not compare different EDSL approaches.

11.1.2. Implementation Costs of Domain-Specific Aspect Languages

To quantitatively evaluate the implementation cots of aspect languages, in the context of this thesis the *Cool* domain-specific aspect languages was implemented. Cool was a demonstrator

Table 11.1.: Comparison language embeddings' support for concrete syntax

	Host Language			Embedding			
Embedding Approach	Supported Host Languages (1)	Host Language Independent (2)	Grammar Size for Host Language (3)	Embedding Style (4)	Concrete Syntax (5)	Grammar Size for Embedding (6)	Supported Languages (7)
Pure Embedding							
[Hud96], [ALY09]	Haskell			homogeneous	no		
[CKS09]	OCaml, Haskell			homogeneous	no		
Dynamic Languages							
Jargons [Pes01]	Scheme			homogeneous	no		
[TFH09]	Ruby			homogeneous	no		
[KG07]	Groovy			homogeneous	no		
TwisteR [AO10]	Ruby			homogeneous	no		
(Multi-)Staging							
[COST04]	MetaOCaml			homogeneous	no		
[COST04]	TemplateHaskell			homogeneous	no		
[SCK04]	TemplateHaskell			homogeneous	no		
Typed OO							
[Eva03], [Fow05]	Java			homogeneous	no		
DSL2JDT [Gar08]	Java			homogeneous	no		
Polym. [HORM08]	Scala			homogeneous	no		
REA	Groovy, Ruby			homogeneous	no		
Dynamic Languages							
π [KM09]	π			homogeneous	**yes**	complete	**CFG**
Helvetia [RGN10]	Smalltalk			homogeneous	**yes**	complete	PEG
Staging/Meta-Progr.							
Converge [Tra08]	Converge			homogeneous	**yes**	complete	**CFG**
Source Trans.							
MetaBorg/Stratego [BV04]	Java, C, C++, ...	✓	complete	heterogeneous	**yes**	complete	**CFG**
TXL [Cor06]	C++, C#, Java, ...	✓	complete	heterogeneous	**yes**	complete	LL(*)
TIGERSEYE/REA	Java, Groovy, ...	✓	**partial**	**hybrid**	**yes**	**partial**	**CFG**

Table 11.2.: Comparison of FSMDSL implementation costs

Approach	Bison	ANTL	MontiCore	DSL2JDT	Stratego	SCXML	**REA**	**TigersEye**
Style	Parser Generator	Parser Generator	Ext. Compiler	Homogen. EDSL	Heterogen. EDSL	COTS	Homogen. Reflect. EDSL	Heterogen. Reflect. EDSL
User-defined	353	226	190	206	226	112	136	142
Generated	2642	1443	3816	1210	1443	0	0	0

for AOP since the beginning, it is a well-established DSALs [KL07b, HBA08] to implement in order to evaluate an extensible AO implementation approach.

Table 11.3 compares the support of the Cool features that Lopes proposes [Lop97] in different AO implementation approaches. Unfortunately, the original Cool implementation [Lop97] as parts of the AspectJ source code weaver v0.1 is no longer available, which is why the results are in brackets. Unfortunately, AweSome [KL07b] is not available for download. JAMI [HBA08] is the only Cool implementation available for comparison. Compared to JAMI, POPART's Cool implementation is more complete.

Table 11.3.: Comparison of the support for the Cool DSAL features

Framework	Impl. Apporach	per obj/class	self-/mutex	condition	var	guard
Cool/D [Lop97]	Pre-Processor	(●)	(●)	(●)	(●)	(●)
AweSome	Ext. Compiler		(●)	(●)	(●)	(●)
JAMI	Interpreter	◖	●			
Popart/REA	Embedding	●	●	●	●	●

Table 11.4 compares the eLOCs of the DSALs that have been implemented, only user-written code is compared. For the Cool DSAL implementation, compared to REA, JAMI has less eLOC, but the Cool/JAMI is not comparable to the Cool/POPART, since it implements only one out of the five Cool features completely. For the Caching DSAL, JAMI and REA are comparable, because JAMI and REA completely implement it. To implement the Caching DSAL, REA requires significant less code than JAMI (36.0 % less eLOCs). For the Zip DSAL, there is only an implementation in REA with 192 eLOCs.

Table 11.4.: Comparison of DSAL implementation costs

DSAL	JAMI 0.3	**POPART/REA**
COOL	(532)	1083
Caching	491	314
Zip	(N/A)	192

11.2. Compilation Costs of Reflective Embedding

The following section measures the costs for compiling DSL programs and compares compilation time to traditional and related embedding approaches.

The experiment compares the compilation time of a DSL programs for the same DSL implemented with the different technologies. The benchmark uses a state machine DSL specification that was implemented then implemented with each corresponding technology. The experiment uses a generated input file which contains a large state machine and input events with 10 different input sizes (1 to 10 chunks, with the maximum of approx. 1500 eLOCs). The same state machine was executed with different implementations of the DSLs and the times for executing the same state machine program were measured. Finally, the experiment compares execution times between the different technologies.

The benchmark results in Figure 11.1 shows that bison+flex (+), AntLR (□), Monticore (△), and SCXML (○) have low compilation times. Heterogeneous embedding with MetaBorg/Stratego (✖) has a good compilation time. In contrast to this, homogeneous embeddings with

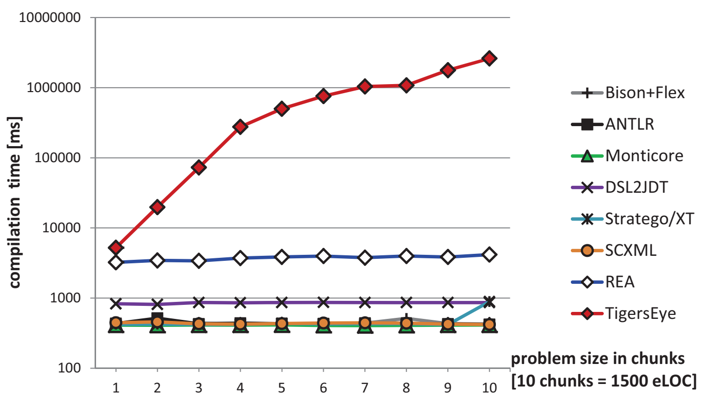


Figure 11.1.: Benchmark results for DSL program compilation time

DSL2JDT ($\times$) or REA (unfilled $\diamond$) have moderate compilation times. Only TIGERSEYE (filled $\diamond$) has a bad compilation performance.

For TIGERSEYE, the compilation time quickly increases with the size of the DSL program, and therefore it is currently not appropriate for DSL programs with large files. The low compilation performance is due to using the Earley algorithm for parsing, which has its worst case complexity of $O(n^3)$ [Ear70], since there are many ambiguities introduces by the island rules. Despite the disadvantageous results for TIGERSEYE, note that the parser's performance has not yet been optimized, which was out of scope. Currently, parsing and transforming DSL programs increases exponentially with the DSL program file size. While processing 500 effective lines of code (eLOC) takes approx. 10 seconds, TIGERSEYE needs 43 minutes to rewrite 1500 eLOC. Therefore, it is only practicable if developers use TIGERSEYE for DSL programs with small files.

11.3. Runtime Costs of Reflective Embedding

The following section measures costs for implementing DSLs and aspect languages and compares them to traditional and related embedding approaches.

11.3.1. Runtime Costs of Domain-Specific Languages

Although providing productive qualities for EDSLs is outside the scope of this thesis, it is interesting to reveal the runtime costs of the current prototype. Therefore, an experiment was conducted that compares the performance of REA to related work.

The experiment compares the execution times of a large DSL program for the same DSL implemented with the different technologies. The benchmark provides a state machine DSL specification that was implemented then implemented with each corresponding technology. The experiment uses a generated input file which contains a large state machine and input events with 5 different input sizes (chunks between 0 to 500 eLOCs). The same state machine was executed with different implementations of the DSLs and the times for executing the same state machine program were measured. Finally, the experiment compares execution times between the different technologies.

When using a declarative language (e.g., a DSL) an input file with 500 eLOCs can be considered a rather large DSL program. Still, the benchmark's input size is very small, since there are many DSL e.g. HTML that use large input files. Therefore, with the current benchmark version, one cannot draw general conclusion about DSL performance. Despite this, these results represents a rough estimation.

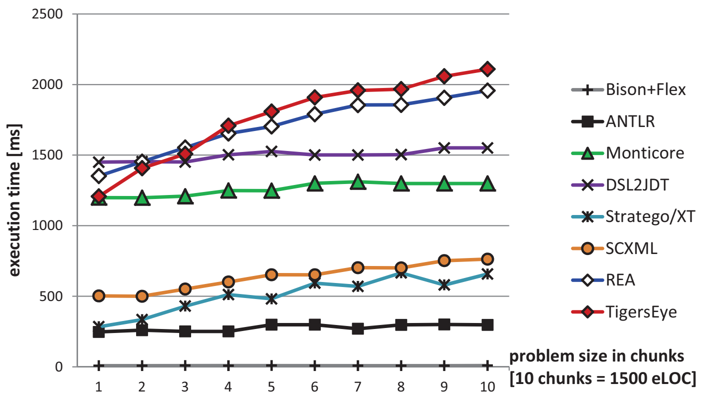


Figure 11.2.: Benchmark results for DSL program execution time

The measurements presented in Figure 11.2 indicate that most DSL technologies have good execution times for FSMDSL programs. The parser generator approaches perform the best, but they are not easily extensible. Bison (+) is the fastest, followed by AntLR (□). Although one can expect a large overhead due to XML parsing in SCXML (○), it performs good. MontiCore (△) allows some extensions but it still performs moderate.

The heterogeneous embedded approach MetaBorg/Stratego (✱) had a good performance. In contrast, the homogeneous embedded approaches that systematically support both extensions and composition are slower than the other DSL approaches. This is in same line as the benchmark results in [KLP+08]. DSL2JDT (×) is faster than both REA (unfilled ◇) and TIGERSEYE (filled ◇). Although REA and and TIGERSEYE are only a little slower than DSL2JDT, they are much more flexible than DSL2JDT. Due to the overhead of parsing and rewriting concrete syntax with the Earley parser, the same state machine DSL program executes far more slower than in REA with abstract syntax.

For REA and TIGERSEYE, despite the presence of the meta-level, they execute embedded DSL programs with a similar moderate performance lie the extensible DSL approach Monticore. Despite the indirections due to the meta-object, REA executes embedded DSL programs with a similar performance as the DSL2JDT approach that embeds a fluent interface in Java, but DSL2JDT does not address extensibility and composability.

11.3.2. Runtime Costs of Aspect Languages

This section reports on experiments about the runtime penalty of reflective embedding. For an estimation of the runtime cost, this thesis compares the performance of tangled implementations with an experiment that implemented the same program with and without crosscutting composition and compared the execution time. Currently, the CCCombiner exploits *partial evaluation* of static parts in pointcuts (cf. [HH04, Tan04]), i.e., POPART only reifies join point contexts and evaluate the dynamic parts of pointcuts when aspects define pointcuts that potentially match a given join point. The observed overhead is up to 3 times. In sum, AO language embeddings in POPART/REA the flexibility comes at a price due to the indirections used in the meta-aspect protocol.

A comprehensive treatment of performance issues of reflective embedding remains to be done in future work. It is a complicate task particularly for two reasons.

First, there are no fair standard benchmarks available for aspect languages [HM04]. There are benchmarks that evaluate the runtime costs of general-purpose aspect languages [HM04], but no benchmarks are available for domain-specific aspect languages. Designing a fair benchmark for DSALs was not in the scope of this thesis.

Second, for determining the runtime costs an embedding approach, it is not clear what costs are due to the host language and due to the embedding. The above performance measurements are primarily determined by the performance of the Groovy MOP implementation, only part of the overhead is due to reification of join points and to advice composition. Note that the Groovy MOP was designed for convenience and flexibility, not optimized for performance. In Groovy, the Groovy MOP as well as boxing and unboxing of primitive values has a large overhead on application execution [KG08]. Although Groovy code is compiled to Java bytecode, which is subject to aggressive dynamic runtime optimization in the JVM, the compiler inserts many indirections into bytecode to allow the flexibility of the MOP. In Groovy, method calls are reflective calls for which up to several hundreds of ordinary Java methods calls have to be executed. Groovy's caching facilities for reflective method calls effectively reduce the number of necessary method calls, but there is a large remaining overhead even with caching. Further, Groovy has a large footprint on the call stack, because when Groovy makes a method call this leads to between 9 and 13 more activations of Java stack frames that remain on the stack until the Groovy method returns.

Part V.

Conclusion

12. Conclusion

12.1. Summary of Results

The objective of this thesis was to address problems of existing embedding approaches for domain-specific languages (DSLs). While existing approaches have a limited support for extensibility and composability of embedded languages, they do not address embedding of many well-established language concepts that are supported by traditional non-embedded language implementation approaches. An investigation conducted on related work on embedded domain-specific languages showed that each of the existing approaches supports only a subset of the traditionally supported language concepts. In particular, even the most advanced embedding approaches lack support for (1) concrete syntax, (2) crosscutting composition, and (3) fine-grained compositions. Therefore, they do not adequately support sophisticated language compositions that have interactions in the syntax and semantics of constituent parts of the language.

To address these problems, this thesis has presented the reflective embedding architecture (REA)—a novel language embedding approach that equips embedded languages with a meta-level architecture. The approach consolidates two well-established ideas in language research, namely *embedded domain-specific languages* and *meta-object protocols*. Hudak's idea was to embed a language into another language as a library. Kiczales et al.'s idea [KRB91] was to provide meta protocols that open up a language implementation. The reflective embedding architecture contributes to both ideas by combining them into a uniform architecture for embedding and composing various language concepts.

Language developers can use the uniform architecture to reuse existing languages and to embed new language concepts, such as functions, state machines, and aspects. A language is embedded as component that remains encapsulated and that has a well-defined interface. The reflective embedding architecture supports language evolution in various forms:

Extensibility: The architecture supports extensibility through language polymorphism, which enables late binding for syntax and semantics. When required, developers can open up (white-/gray-box) language components to tailor them for their domains. Whereby, a language implementation is opened up revealing only the necessary implementation details.

Compositionality: It supports composition of languages that have interactions in their syntax and semantics. The architecture provides means for language composition in form of so-called language combiners that are first-class objects for composing languages. When required, developers can extend the available set of combiners to support special compositions.

Pluggable Scoping: There is support for scoping language constructs with various strategies on several levels. First, at the expression level, one can scope expression types to the lexical or dynamic context. Second, at the module-level, one can dynamically scope aspects. Third, at the language level, one can scope language semantics to individual programs using language polymorphism.

Pluggable Analyses: The architecture can analyze languages and programs under different semantics. In particular, it supports abstract interpretations to analyze programs before they are executed. Additionally, it supports dynamic analyses. Modular abstract interpretations of different languages can be composed.

Pluggable Transformations: The approach supports for two kinds of transformations. First, it supports exo-transformations from one language into another language, whereby the program in the source language is used as a specification for generated program in the target language. Second, it supports endo-transformations within one causally connected runtime model of a language, even when this model is currently under execution. Such transformation that is executed online enables dynamic adaptive optimizations.

Further, the applications of this thesis's techniques contribute new concepts and generalizes existing concepts in different areas of programming language research.

Reflective Languages: The reflective embedding architecture allows growing an embedded domain-specific language into a *reflective DSL* enables DSL programs that reason about their own structure and behavior, as proposed by Maes [Mae87] for general-purpose languages.

Meta Protocols: The approach applies the concept of meta-protocols and open implementations by Kiczales et al. in the context of domain-specific languages. In the reflective embedding architecture, every embedded language can provide a dedicated interface to its programs through which an end user program can adapt the implementation strategy of a language for special needs. In the context of this thesis, meta-protocols have been validated for DSLs [DMM10] and for aspect-oriented programming languages [DMB09]. The concepts presented in Section 4.2.3.3 and in [DMB09] are now adopted by two other groups of researchers: First, Achenbach et al. [AO10] have implemented a *meta-aspect protocol* in other aspect-oriented programming language. They have adopted the MAP for implementing an open aspect-oriented language with a meta-level for Ruby, which architecture is inspired by POPART. Second, Axelsen et al. [AKMP10] have designed a meta-protocol for class extensions whose design is inspired by the design of POPART's meta-aspect protocol, which allows type-safe extensions to classes.

Domain-Specific Aspect Language: Because all concepts proposed in the thesis are embedded in a uniform way as first-class objects, developer can use embedded concepts in concert with new embeddings. In particular, it is possible to integrate DSL constructs with aspects. This allows implementing *domain-specific aspect languages* in a reusable way, which is in the same vein as extensible aspect languages, but the reflective embedding architecture allows extensions that are not possible with existing extensible aspect languages. In particular, it allows composing multiple domain-specific aspect languages that have interactions or composition conflicts, whereby defining a reusable aspect conflict resolution strategy as a successor to the concepts that have been proposed by Kosar et al. [KL07b] and Havinga et al. [HBA08].

The concepts of this thesis have been qualitatively evaluated by using them to implement various language concepts. To quantitatively evaluate the concepts, several experiments have been conducted that indicate that the implementation effort to implement individual language components is comparable to the effort in language approaches. This thesis proposes to use open source benchmarks to compare the qualities of the DSL implementations. Preliminary benchmark measurements of the implementation with reflective embedding indicate a moderate overhead compared to the implementation with traditional and related approaches. Special optimizations were studied particularly for aspect languages. Only with partial evaluation of pointcuts, the meta-aspect protocol could achieve a moderate execution overhead for object-oriented program. There is a high confidence that the performance can be further improved by using additional optimization techniques from related work. It would be possible to exploit more concepts from traditional approaches. Alternatively, the reflective embedding architecture could be instantiated in a host language, such as CLOS, that optimizes indirections of meta-objects.

The code of POPART, TIGERSEYE, and the DSL benchmark can be downloaded from: `http://www.stg.tu-darmstadt.de/research/ao4dsls`.

12.2. Concluding Discussion

This thesis concludes with a discussion about the dialectics in the philosophies of programming languages. The last decades has led to a continuously quest about what the best language constructs are that a language should provide for its users. Different paradigms try to give an answer to this question, such as pure functional languages advocate taming side-effects, concurrent languages tame parallel threads of execution, object-oriented programming try to model the world with set of discrete inter-connected objects, and event-based programming try to model the world's objects as more or less independent and decoupled actors.

Thus, in common, there is a broad disagreement of what the right language construct and paradigm is. Naturally is no general answer to the question as the answer depends at least on the concrete application and problem domain. Still, asking this question is an important driver for language research.

Often the criticisms of existing language constructs and their paradigms led to an initiative for a new language. Whereby, the language evolutions established by such initiatives can be understood as *dialectic* movements in the sense of Hegel [HNP30]. For example, aspects in aspect-oriented programming raises object-oriented modules to a new more sophisticated level of OO programming. First, often such an initiative accepts the existing language concepts and constructs, e.g., aspects accept modules as an important means for separation (*"das Abstrakte und Verständige"*, i.e., the *abstract reasoning*). Second, they criticize those concepts and constructs for their contradictions, e.g., by encapsulating functional concerns, modules scatter and tangle crosscutting concerns like non-functional concerns (*"Dialektische oder Negativ-Vernünftige"*, i.e., the *dialectic negative-reasoning*). Third, they speculate for new language constructs to improve the state-of-the-art, e.g., aspects negate strong encapsulation of modules, to encapsulate the crosscutting concerns (*"das Spekulative and Positiv-Vernünftige"*, i.e., the *speculative positive-reasoning* or the *negation of negation*).

Such movements for new languages or new versions of languages often came along with the implementation of a new language from scratch. However, as it was criticized in the introduction of this thesis that implementation from scratch contradicts with the economics of language developers, therefore there is a need for language implementation approaches that can cope with various language evolution scenarios.

In the last years, the merger of object-oriented languages and functional languages in the Scala language [OSV07] has created a tremendous interest within the ranks and files of programmers from various paradigms. This merger alone has created a significant impact on the collaboration of programmers from various paradigms. Still today, most languages and their paradigms remain isolated from each other, and it is not possible to easily merge their syntax and semantics. This is not possible for a specific reason: because most existing language architectures are not do not have access to a meta-level and therefore cannot use it to reason about languages and manipulate them. Only heterogeneous language implementation approaches have meta-levels, but they only support evolution at compile time and disallow runtime evolution.

Consider what it would be like to have language architectures embedded into today's languages. In such architectures, language researchers and developers could contribute to a common asset of language constructs and paradigms, which could co-exist in the same environment. Whereby the language end user is empowered to choose the best construct in need without changing the host language. The reflective embedding architecture can be thought of a first step into this direction. Complementing this vision would allow a new style of developing programming languages, which can be called *dialectic programming*. In *dialectic programming*, there is no commitment to a language construct with mechanic semantics or to a particular paradigm, but developers are in the full control of the semantics of every language concept. What is new is that developers can analyze existing constructs (the abstract reasoning), they can criticize or break with existing constructs that do not fit their needs (the negation), and they can adapt constructs to speculate for new alternative semantics in these constructs to reach their goals (negation of negation).

12.3. Future Work

The following challenges are possible directions of future work.

DSL Implementation: Currently it is unclear whether TIGERSEYE/REA scales for implementing large DSLs with thousands of productions. Therefore, it would be interesting to conduct a case study that implements a large DSL.

Currently the input size of a DSL program is limited by the maximum file size of a Java class (64KB). To avoid this limitation, it would be interesting to allow splitting up a DSL file into several parts of DSL modules.

Static Guarantees: As usual for dynamic approaches, there is a trade-off between adaptability and statically checked correctness. REA supports a maximal adaptability, but therefore it mostly lacks special static guarantees.

DSL programmers who override parts of the default DSL semantics might violate contracts and responsibilities that are implicit in the DSL design. In particular, future work will investigate using aspects for design-by-contract for languages. The principle idea is to deploy an aspect on the language façade to enforce that extending languages do not violate any contracts.

Future work will also explore explicit contracts that can be checked at runtime to ensure the semantic consistency of adaptations.

Benchmarking: It would be desirable to exploit standard optimizations for Groovy, such as those discussed in [KG08], as soon as they become available for newer versions of Groovy. The JIT optimization could further reduce the overhead for execution, but the Groovy JIT is only available for an older version[1] of Groovy that is incompatible to the Groovy version used in this thesis.

It would be interesting to use a DSL for which there is a standard benchmark and compare to other implementation technology. Currently the input size of the open source benchmark is too small to allow general conclusions. It would be also interesting to benchmark different kinds of DSLs, which different grammar sizes, semantic complexity, and possibilities for optimizations.

Aspect Languages: With respect to possible applications of the meta-aspect protocol, future work needs to explore the ability of the MAP to support further AO language features.

With respect to the performance of the meta-aspect protocol, still new special concepts for meta-aspect protocol need to be invented to minimize their overhead. This thesis has implemented certain *optimistic optimizations* similar to those proposed for MOPs [KRB91, Sul01].

Performance: It has been shown that partial evaluation of pointcut designators and structural designators helps to reduce the overhead of the MAP. While when no aspects are present, the performance is at a moderate level and slows down the performance of domain-specific and object-oriented applications at a factor of less than three, there is a large overhead when aspects and meta-aspects are present.

[1] In [KG08], the Groovy JIT eliminates the overhead of boxing and unboxing as well as special optimizations have been implemented for aspects. However, the GIT is only available for the Groovy version 1.6-beta2 snapshot, but the reflective embedding architecture uses specific feature of the Groovy MOP in version 1.7.x.

Currently, the meta-aspect protocol currently does not perform enough optimizations to eliminate the complete overhead when composing aspects into programs. Although the POPART code is subject to *adaptive optimization* provided by the just-in-time compiler of the Java VM due to the tight integration of Groovy into Java, a large run-time overhead has been measured for the Groovy-specific features especially for the Groovy MOP.

Although the above optimizations are interesting from a research perspective and for integrating with the current Java bytecode platform, but they would introduce additional complexity which could interfere with existing infrastructure, such as *garbage collectors*, in an unpredictable way. Further, when there are hard performance requirements, it would be more interesting to reimplement the complete meta-aspect protocol on another host language that systematically supports removing indirections at compile time, such as CLOS.

Special Extensions: It would also be interesting to investigate special optimization techniques for aspects. It would be interesting to further investigate whether the compile-time optimizations using *partial reflection* [Tan04] could be used for this. It could be interesting to investigate on-demand weaving techniques [Hir03] to transform the base system at runtime in such as way that still meta-aspects can be involved, whereby analyzing the aspects at runtime in order to identify what indirections are actually needed by aspects, and removing indirections of the meta-aspect protocol if they are currently not needed.

Bibliography

[AAB$^+$07] Assaf Arkin, Sid Askary, Ben Bloch, et al. Web Services Business Process Execution Language 2.0, OASIS Standard. Technical report, Organization for the Advancement of Structured Information Standards (OASIS), April 2007.

[ACH$^+$06] P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. ABC: An Extensible AspectJ Compiler. *Transactions on Aspect-Oriented Software Development*, 3880:293, 2006.

[ADDH10] Toheed Aslam, Jesse Doherty, Anton Dubrau, and Laurie Hendren. AspectMatlab: an Aspect-Oriented Scientific Programming Language. In *Proceedings of AOSD'10 [JS10]*, 2010.

[AET08] Pavel Avgustinov, Torbjörn Ekman, and Julian Tibble. Modularity First: A Case for Mixing AOP and Attribute Grammars. In *Proceedings of AOSD'08 [D'H08]*, pages 25–35, 2008.

[AFG$^+$00] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Proceedings of OOPSLA'00 [OOP00]*, 2000.

[AGH05] K. Arnold, J. Gosling, and D. Holmes. *The Java (TM) Programming Language*. Addison-Wesley Professional, 2005.

[AHH$^+$09] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A Comparison of Context-Oriented Programming Languages. In *International Workshop on Context-Oriented Programming. In conjunction with the 23rd European Conference on Object-Oriented Programming (ECOOP'09)*, COP '09, pages 6:1–6:6, New York, NY, USA, 2009. ACM.

[AKMP10] Eyvind W. Axelsen, Stein Krogdahl, and Birger Møller-Pedersen. Controlling Dynamic Module Composition through an Extensible Meta-Level API. In *Proceedings of the 6th Dynamic Languages Symposium (DLS'10)*, pages 81–96, New York, NY, USA, 2010. ACM.

[Aks03] Mehmet Aksit, editor. *Proceedings of the 2nd ACM International Conference on Aspect-Oriented Software Development (AOSD'03)*, Boston, Massachusetts, USA, 2003. ACM Press, New York, NY, USA.

[Alm] D. Almear. SQL AOP becomes SQXML AOP. `http://almaer.com/blog/sql-aop-becomes-sqxml-aop`.

[ALSU07] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools (2nd Edition)*. Addison-Wesley, 2007.

[ALY09] Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding Domain-Specific Languages. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell'09)*, pages 37–48, New York, NY, USA, 2009. ACM.

[AMH90] Mehmet Akşit, Rene Mostert, and Boudewijn Haverkort. Compiler Generation based on Grammar Inheritance, 1990.

[AN08] Ali Assaf and Jacques Noyé. Dynamic AspectJ. In *Proceedings of the 2008 Dynamic Languages Symposium*, DLS'08, pages 8:1–8:12, New York, NY, USA, 2008. ACM.

[AO10] Michael Achenbach and Klaus Ostermann. A meta-aspect protocol for developing dynamic analyses. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 153–167. Springer Berlin / Heidelberg, 2010.

[AS96] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1996.

[Asp] AspectJ Home Page. `http://www.eclipse.org/aspectj/`.

[Aßm03] U. Aßmann. *Invasive Software Composition*. Springer-Verlag, New York, 2003.

[Atk09] Robert Atkey. Syntax for Free: Representing Syntax with Binding Using Parametricity. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications*, volume 5608 of *Lecture Notes in Computer Science*, pages 35–49. Springer Berlin / Heidelberg, 2009.

[BA01] L. Bergmans and M. Aksit. Composing Crosscutting Concerns using Composition Filters. *Communications of the ACM*, 44(10):51–57, 2001.

[BA04] L. Bergmans and M. Aksit. Principles and Design Rationale of Composition Filters. *Aspect-Oriented Software Development*, pages 63–95, 2004.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-Oriented Programming and Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA/ECOOP'90)*, pages 303–311, New York, NY, USA, 1990. ACM.

[BdGV06] Martin Bravenboer, Rene de Groot, and Eelco Visser. MetaBorg in Action: Examples of Domain-Specific Language Embedding and Assimilation Using Stratego/XT. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 297–311. Springer Berlin / Heidelberg, 2006.

[BHMO04] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *Proceedings of AOSD'04 [Lie04]*, pages 83–92, 2004.

[BI82] A.H. Borning and D.H.H. Ingalls. Multiple Inheritance in Smalltalk-80. In *Proceedings of the National Conference on Artificial Intelligence*, pages 234–237, 1982.

[BKHV03] O.S. Bagge, K.T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost Transformation System for Domain-Specific Optimisation of C++ Programs. In *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75, 2003.

[BMDV02] J. Brichau, K. Mens, and K. De Volder. Building Composable Aspect-Specific Languages with Logic Metaprogramming. In *GPCE*, pages 110–127, 2002.

[BMH+07] Christoph Bockisch, Mira Mezini, Wilke Havinga, Lodewijk Bergmans, and Kris Gybels. Reference Model Implementation. Technical Report AOSD-Europe-TUD-8, Technische Universität Darmstadt, 2007.

[BMN+06] Johan Brichau, Mira Mezini, Jacques Noyé, Wilke Havinga, Lodewijk Bergmans, Vaidas Gasiunas, Christoph Bockisch, Johan Fabry, and Theo D'Hondt. An Initial Metamodel for Aspect-Oriented Programming Languages. `http://www.aosd-europe.net/deliverables/d39.pdf`, 2006.

[Bon03] J. Bonér. AspectWerkz - Dynamic AOP for Java. `http://codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf`, 2003.

[Bos00] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach.* Addison-Wesley Professional, 2000.

[BP01] Jonthan Bachrach and Keith Playford. The Java Syntactic Extender (JSE). *SIGPLAN Not.*, 36(11):31–42, 2001.

[BP08] J. Bovet and T. Parr. ANTLRWorks: an ANTLR Grammar Development Environment. *Software: Practice and Experience*, 38(12):1305–1332, 2008.

[BU04] Gilad Bracha and David Ungar. Mirrors: Design Principles for Meta-Level Facilities of Object-Oriented Programming Languages. *SIGPLAN Not.*, 39(10):331–344, 2004.

[BV04] Martin Bravenboer and Eelco Visser. Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions. In *Proceedings of OOPSLA'04 [VS04]*, pages 365–383, 2004.

[CBF05] N. Cacho, T. Batista, and F. Fernandes. AspectLua-A Dynamic AOP Approach. *Journal of Universal Computer Society*, 11(7):1177–1197, 2005.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 238–252, New York, NY, USA, 1977. ACM.

[CCR10] William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors. *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10)*, Reno/Tahoe, Nevada, USA, 2010. ACM Press, New York, NY, USA.

[CD08] Pascal Costanza and Theo D'Hondt. Feature Descriptions for Context-oriented Programming. In *2nd International Workshop on Dynamic Software Product Lines (DSPL'08)*, 2008.

[CDF$^+$06] Philippe Charles, Julian Dolby, Robert M. Fuhrer, Stanley M. Sutton, Jr., and Mandana Vaziri. SAFARI: A Meta-Tooling Framework for Generating Language-Specific IDE's. In *Proceedings of OOPSLA'06 [TC06]*, pages 722–723, 2006.

[CDM09] A. Charfi, T. Dinkelaker, and M. Mezini. A Plug-in Architecture for Self-Adaptive Web Service Compositions. In *Proceedings of the 2009 IEEE International Conference on Web Services*, pages 35–42. IEEE Computer Society, 2009.

[CE00] K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press, New York, NY, USA, 2000.

[CFSJ07] P. Charles, R.M. Fuhrer, and S.M. Sutton Jr. IMP: A Meta-Tooling Platform for Creating Language-Specific IDEs in Eclipse. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 485–488, New York, NY, USA, 2007. ACM.

[CH05] Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-Oriented Programming: An Overview of ContextL. In *Proceedings of the Dynamic languages Symposium (DLS'05)*, pages 1–10, New York, NY, USA, 2005. ACM.

[Cha08] Anis Charfi. *Aspect-Oriented Workflow Management: Concepts, Languages, Applications*. PhD thesis, Technische Universität Darmstadt, Germany, 2008.

[CHBB09] Steve Counsell, Tracy Hall, David Bowes, and Sue Black. Program Slice Metrics and Their Potential Role in DSL Design. In *Proceedings of Workshop on Knowledge Industry Survival Strategy Initiative (KISS'09)*, 2009.

[CHHP91] James R. Cordy, Charles D. Halpern-Hamu, and Eric Promislow. TXL: A Rapid Prototyping System for Programming Language Dialects. *Computer Languages*, 16(1):97 – 107, 1991.

[CHR$^+$03] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and C. Pu. Spidle: a DSL Approach to Specifying Streaming Applications. *Generative Programming and Component Engineering*, 2830:1–17, 2003.

[Chu40] A. Church. A Formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5(2):56–68, 1940.

[CJ03] Ron Crocker and Guy L. Steele Jr., editors. *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, Anaheim, CA, USA, 2003. ACM Press, New York, NY, USA.

[CKL96] Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding Confusion in Metacircularity: The Meta-Helix. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *Object Technologies for Advanced Software*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer Berlin / Heidelberg, 1996.

[CKS09] J. Carette, O. Kiselyov, and C. Shan. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

[Cle07] T. Cleenewerck. *Modularizing Language Constructs: A Reflective Approach*. PhD thesis, Vrije Universiteit Brussel, Belgium, 2007.

[CM94] S. Chiba and T. Masuda. A Reflective Language OpenC++ and its Application to Distributed Computing. *Computer Software*, 11(3):33–48, 1994.

[CM04] Anis Charfi and Mira Mezini. Aspect-Oriented Web Service Composition with AO4BPEL. In *European Conference on Web Services*, 2004.

[CM06] A. Charfi and M. Mezini. Aspect-Oriented Workflow Languages. *Lecture Notes in Computer Science*, 4275:183, 2006.

[CM07] Jesús Sánchez Cuadrado and Jesús García Molina. Building Domain-Specific Languages for Model-Driven Development. *IEEE Softw.*, 24(5):48–55, 2007.

[CNF$^+$08] Thomas Cleenewerck, Jacques Noyé, Johan Fabry, Anne-Françoise Lemeur, and Éric Tanter. Summary of the Third Workshop on Domain-Specific Aspect Languages. In *Proceedings of the 2008 Workshop on Domain-Specific Aspect Languages (DSAL'08)*, pages 1–5, 2008.

[CNW01] F. Curbera, W. Nagy, and S. Weerawarana. Web Services: Why and How. In *Workshop on Object-Oriented Web Services*, 2001.

[Coi87] Pierre Cointe. Metaclasses are First Class: The ObjVlisp Model. *SIGPLAN Not.*, 22(12):156–162, 1987.

[Cor06] James R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 61(3):190 – 210, 2006. Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA '04).

[COST04] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. DSL Implementation in MetaOCaml, Template Haskell, and C++. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 51–72. Springer Berlin / Heidelberg, 2004.

[Cou96] Patrick Cousot. Abstract Interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.

[CR06] Lori A. Clarke and David S. Rosenblum. A Historical Perspective on Runtime Assertion Checking in Software Development. *SIGSOFT Softw. Eng. Notes*, 31(3):25–37, 2006.

[Dav03] James Davis. GME: the Generic Modeling Environment. In *Companion of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03 [CJ03])*, pages 82–83, New York, NY, USA, 2003. ACM.

[dB00] Hans de Bruin. A Grey-Box Approach to Component Composition. In Krzysztof Czarnecki and Ulrich Eisenecker, editors, *Generative and Component-Based Software Engineering*, volume 1799 of *Lecture Notes in Computer Science*, pages 195–209. Springer Berlin / Heidelberg, 2000.

[DD89] C.J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley New York, 1989.

[DDLM07] Marcus Denker, Stéphane Ducasse, Andrian Lienhard, and Philippe Marschall. Sub-Method Reflection. *Special Issue TOOLS Europe 2007*, 6(9), 2007.

[DDMW00] M. D'Hondt, W. De Meuter, and R. Wuyts. Using Reflective Logic Programming to describe Domain Knowledge as an Aspect. *Generative and Component-Based Software Engineering*, 1799:16–23, 2000.

[DEM10] Tom Dinkelaker, Michael Eichberg, and Mira Mezini. An Architecture for Composing Embedded Domain-Specific Languages. In *Proceedings of AOSD'10 [JS10]*, pages 49–60, 2010.

[DFS02] Rï£¡mi Douence, Pascal Fradet, and Mario Sï£¡dholt. A Framework for the Detection and Resolution of Aspect Interactions. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 173–188. Springer Berlin / Heidelberg, 2002.

[D'H08] Theo D'Hondt, editor. *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD'08)*, Brussels, Belgium, 2008. ACM Press, New York, NY, USA.

[Dic92] K. Dickey. Scheming with Objects. *AI Expert*, 7(10):24–33, 1992.

[Din10] Tom Dinkelaker. Review of the Support for Modular Language Implementation with Embedding Approaches. Technical Report TUD-CS-2010-2396, Department of Computer Science, Technische Universitaet Darmstadt, Germany, November 2010.

[DJKN07] Tom Dinkelaker, Alisdair Johnstone, Yuecel Karabulut, and Ike Nassi. Secure Scripting Based Composite Application Development: Framework, Architecture, and Implementation. In *Proceedings of the International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2007)*, pages 85–94. IEEE, 2007.

[DMB09] Tom Dinkelaker, Mira Mezini, and Christoph Bockisch. The Art of the Meta-Aspect Protocol. In *Proceedings of AOSD'08 [D'H08]*, pages 51–62, New York, NY, USA, 2009. ACM.

[DMM09] T. Dinkelaker, M. Monperrus, and M. Mezini. Untangling Cross-Cutting Concerns in Domain-Specific Languages with Domain-Specific Join Points. In *Proceedings of the 4th Workshop on Domain-Specific Aspect Languages (DSAL)*, pages 1–6, New York, NY, USA, 2009. ACM.

[DMM10] Tom Dinkelaker, Martin Monperrus, and Mira Mezini. Supporting variability with late semantic adaptations of domain-specific modeling languages. *CEUR-WS.org*, 564, 2010.

[DN09] N.A. Danielsson and U. Norell. Parsing Mixfix Operators. In *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages (IFL'08)*. Springer Berlin / Heidelberg, 2009.

[Dub06] G. Dubochet. On Embedding Domain-Specific Languages with User-friendly Syntax. In *ECOOP Workshop on Domain-Specific Program Development*, 2006.

[DWL09] Tom Dinkelaker, Christian Wende, and Henrik Lochmann. Implementing and Composing MDSD-Typical DSLs. Technical Report TUD-CS-2009-0156, Department of Computer Science, Technische Universitaet Darmstadt, Germany, October 2009.

[Ear70] J. Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[EFdM00] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, pages 9–26. Springer Berlin / Heidelberg, 2000.

[EFDM03] C. Elliott, S. Finne, and O. De Moor. Compiling Embedded Languages. *Journal of Functional Programming*, 13(03):455–481, 2003.

[EH07a] Torbjörn Ekman and Görel Hedin. The JastAdd Extensible Java Compiler. In *Proceedings of OOPSLA'07 [GBLJ07]*, pages 1–18, 2007.

[EH07b] Torbjörn Ekman and Görel Hedin. The JastAdd System – Modular Extensible Compiler Construction. *Science of Computer Programming*, 69(1-3):14–26, 2007. Special Issue on Experimental Software and Toolkits.

[ELW98] R. Eckstein, M. Loy, and D. Wood. *Java Swing*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 1998.

[Eva03] Eric Evans. *Domain-Driven Design: Tackling Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[EVI05] Jacky Estublier, German Vega, and Anca Daniela Ionita. Composing Domain-Specific Languages for Wide-scope Software Engineering Applications. In *Proceedings of MODELS/UML*, 2005.

[FC09] Jose Falcon and William Cook. GEL: A Generic Extensible Language. In Walid Taha, editor, *Domain-Specific Languages*, volume 5658 of *Lecture Notes in Computer Science*, pages 58–77. Springer Berlin / Heidelberg, 2009.

[FD06] J. Fabry and T. D'Hondt. KALA: Kernel Aspect Language for Advanced Transactions. In *Symposium on Applied Computing*, 2006.

[Fel90] M. Felleisen. On the Expressive Power of Programming Languages. *3rd European Symposium on Programming (ESOP'90)*, 432:134–151, 1990.

[Fer89] J. Ferber. Computational Reflection in Class Based Object-Oriented Languages. In *Proceedings of OOPSLA'89 [Mey89]*, pages 317–326, 1989.

[FF00] R.E. Filman and D.P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced separation of Concerns*, 2000.

[FF04] Ira R. Forman and Nate Forman. *Java Reflection in Action (In Action series).* Manning Publications Co., Greenwich, CT, USA, 2004.

[Fla02] David Flanagan. *JavaScript: The Definitive Guide, 4th Edition.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[For04] B. Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *ACM SIGPLAN Notices*, 39(1):122, 2004.

[Fow05] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages. `http://www.martinfowler.com/articles/languageWorkbench.html`, 2005.

[FP06] Steve Freeman and Nat Pryce. Evolving an Embedded Domain-Specific Language in Java. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06 [TC06])*, pages 855–865, New York, NY, USA, 2006. ACM.

[FPB$^+$70] W. Feurzeig, S. Papert, M. Bloom, R. Grant, and C. Solomon. Programming-Languages as a Conceptual Framework for Teaching Mathematics. *SIGCUE Outlook*, 4(2):13–17, 1970.

[FS99] P. Fradet and M. Südholt. An Aspect Language for Robust Programming. *LNCS*, 1743:291–292, 1999.

[FW84] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without Metaphysics. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP'84, pages 348–355, New York, NY, USA, 1984. ACM.

[Gar08] Miguel Garcia. Automating the Embedding of Domain Specific Languages in Eclipse JDT. `http://www.eclipse.org/articles/Article-AutomatingDSLEmbeddings/`, July 2008.

[GBLJ07] Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors. *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, Montreal, Quebec, Canada, 2007. ACM Press, New York, NY, USA.

[GBNT01] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling Crosscutting Constraints in Domain-Specific Modeling. *Communications of the ACM*, 44(10):87–93, 2001.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (Second Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[Glu91] Robert Glueck. Towards Multiple Self-Application. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '91)*, pages 309–320, New York, NY, USA, 1991. ACM.

[Gro] The Groovy Home Page. `http://groovy.codehaus.org/`.

[GV07] Iris Groher and Markus Voelter. XWeave: Models and Aspects in Concert. In *Proceedings of the 10th International Workshop on Aspect-Oriented Modeling (AOM '07)*, pages 35–40, New York, NY, USA, 2007. ACM.

[GWDD06] Kris Gybels, Roel Wuyts, Stéphane Ducasse, and Maja D'Hondt. Inter-Language Reflection: A Conceptual Model and its Implementation. *Computer Languages, Systems & Structures*, 32(2-3):109–124, 2006.

[HB04] Rachid Hamadi and Boualem Benatallah. Recovery Nets: Towards Self-Adaptive Workflow Systems. In Xiaofang Zhou, Stanley Su, Mike P. Papazoglou, Maria E. Orlowska, and Keith G. Jeffery, editors, *Web Information Systems (WISE'04)*, volume 3306 of *Lecture Notes in Computer Science*, pages 439–453. Springer Berlin / Heidelberg, 2004.

[HBA08] W. Havinga, L. Bergmans, and M. Aksit. Prototyping and Composing Aspect Languages using an Aspect Interpreter Framework. In *ECOOP*, pages 180–206, 2008.

[HBA10] Wilke Havinga, Lodewijk Bergmans, and Mehmet Aksit. A Model for Composable Composition Operators: Expressing Object and Aspect Compositions with First-Class Operators. In *Proceedings of AOSD'10 [JS10]*, pages 145–156, 2010.

[HCH08] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An Introduction to Context-Oriented Programming with ContextS. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 396–407. Springer Berlin / Heidelberg, 2008.

[HH04] Erik Hilsdale and Jim Hugunin. Advice Weaving in AspectJ. In *Proceedings of AOSD'04 [Lie04]*, pages 26–35, New York, NY, USA, 2004. ACM.

[HHJ$^+$08] J. Henriksson, F. Heidenreich, J. Johannes, S. Zschaler, and U. Aßmann. Extending Grammars and Metamodels for Reuse: the Reuseware Approach. *IET Software*, 2(3):165–184, 2008.

[HHJZ09] Florian Heidenreich, Jakob Henriksson, Jendrik Johannes, and Steffen Zschaler. On Language-Independent Model Modularisation. *LNCS*, 5560:39–82, 2009.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF—Reference Manual. *SIGPLAN Not.*, 24(11):43–75, 1989.

[Hir01] R. Hirschfeld. AspectS: AOP with Squeak. In *Workshop on Advanced Separation of Concerns in OO Systems*, 2001.

[Hir03] R. Hirschfeld. AspectS: Aspect-Oriented Programming with Squeak. In *Netobjectdays (NODe)*, pages 216–232, 2003.

[Hir09] Robert Hirschfeld. AspectS - Aspect-Oriented Programming with Squeak. In Mehmet Aksit, Mira Mezini, and Rainer Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *Lecture Notes in Computer Science*, pages 216–232. Springer Berlin / Heidelberg, 2009.

[HJZ07] F. Heidenreich, J. Johannes, and S. Zschaler. Aspect Orientation for Your Language of Choice. In *Workshop on Aspect-Oriented Modeling (at MoDELS)*, 2007.

[HM03] G. Hedin and E. Magnusson. JastAdd - An Aspect-Oriented Compiler Construction System. *Science of Computer Programming*, 47(1):37–58, 2003.

[HM04] M. Haupt and M. Mezini. Micro-Measurements for Dynamic Aspect-Oriented Systems. In M. Weske and P. Liggesmeyer, editor, *Net.ObjectDays*, volume 3263 of *LNCS*, 2004.

[HNP30] G.W.F. Hegel, F. Nicolin, and O. Pöggeler. *Enzyklopädie der philosophischen Wissenschaften im Grundrisse*. Meiner Verlag (1991), 1830.

[HO07] C. Hofer and K. Ostermann. On the Relation of Aspects and Monads. In *Workshop on Foundations of Aspect-Oriented Languages*, pages 27–33, 2007.

[HO10] Christian Hofer and Klaus Ostermann. Modular Domain-Specific Language Components in Scala. In *Proceedings of the International Conference Generative Programming and Component Engineering (GPCE)*. ACM, 2010.

[Hoa78] CAR Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):677, 1978.

[Hor01] P. Horn. Autonomic Computing: IBMï£¡s Perspective on the State of Information Technology. *IBM TJ Watson Labs, NY, 15th October*, 2001.

[HORM08] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic Embedding of DSLs. In *Generative Programming and Component Engineering (GPCE'08)*, pages 137–148. ACM New York, NY, USA, 2008.

[HPSA10] Robert Hirschfeld, Michael Perscheid, Christian Schubert, and Malte Appeltauer. Dynamic Contract Layers. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC'10)*, pages 2169–2175, New York, NY, USA, 2010. ACM.

[Hud96] P. Hudak. Building Domain-Specific Embedded Languages. *ACM Computing Surveys*, 28(4es):196–196, 1996.

[Hud98] Paul Hudak. Modular Domain Specific Languages and Tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[IJF92] John Wiseman Simmons II, Stanley Jefferson, and Daniel P. Friedman. Language Extension via First-class Interpreters. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.1396`, 1992.

[Jav02] The JavaLogo Homepage. http://sourceforge.net/projects/javalogo/, April 2002.

[JBC02] Eric Jendrock, Jennifer Ball, and Debbie Carson. *The J2EE Tutorial*. Addison-Wesley Longman, Amsterdam, 2002.

[JM09] Ranjit Jhala and Rupak Majumdar. Software Model Checking. *ACM Comput. Surv.*, 41(4):1–54, 2009.

[JS10] Jean-Marc Jézéquel and Mario Südholt, editors. *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD'10)*. ACM Press, New York, NY, USA, 2010.

[Kam98] S.N. Kamin. Research on Domain-Specific Embedded Languages and Program Generators. *Electronic Notes in Theoretical Computer Science*, 14:149–168, 1998.

[KAR$^+$93] G. Kiczales, J. M. Ashley, L. Rodriguez, A. Vahdat, and D.G. Bobrow. Metaobject Protocols: Why we want them and what else they can do. *Object-Oriented Programming: The CLOS Perspective*, pages 101–118, 1993.

[KG07] D. König and A. Glover. *Groovy in Action*. Manning, 2007.

[KG08] C. Kaewkasi and J.R. Gurd. Groovy AOP: A Dynamic AOP System for a JVM-based Language. In *Workshop on Software Engineering Properties of Languages and Aspect Technologies*, 2008.

[KHH$^+$01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP*, volume 2072 of *LNCS*, pages 327–353, 2001.

[Kic96] G. Kiczales. Beyond the Black Box: Open Implementation. *IEEE Software*, 13(1):8–11, 1996.

[KKV09] Lennart C. L. Kats, Karl Trygve Kalleberg, and Eelco Visser. Domain-Specific Languages for Composable Editor Plugins. In T. Ekman and J. Vinju, editors, *Proceedings of the 9th Workshop on Language Descriptions, Tools, and Applications (LDTA'09)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, March 2009.

[KL04] S. Kojarski and D.H. Lorenz. AOP as a First Class Reflective Mechanism. In *Proceedings of OOPSLA'04 [VS04]*, pages 216–217, 2004.

[KL05] S. Kojarski and D.H. Lorenz. Pluggable AOP: Designing Aspect Mechanisms for Third-Party Composition. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 247–263. ACM, 2005.

[KL07a] S. Kojarski and D.H. Lorenz. Awesome: An Aspect Co-Weaving System for Composing Multiple Aspect-Oriented Extensions. In *Proceedings of OOPSLA'07 [GBLJ07]*, pages 515–534, 2007.

[KL07b] Sergei Kojarski and David H. Lorenz. AweSome: an Aspect Co-Weaving System for Composing Multiple Aspect-Oriented Extensions. In *Proceedings of OOPSLA'07 [GBLJ07]*, pages 515–534, 2007.

[KLL$^+$97] Gregor Kiczales, John Lamping, Christina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open Implementation Design Guidelines. In *Proceedings of the 19th International Conference on Software Engineering*, ICSE'97, pages 481–490, New York, NY, USA, 1997. ACM.

[KLM$^+$97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[KLP$^+$08] T. Kosar, M. López, E. Pablo, P.A. Barrientos, and M. Mernik. A Preliminary Study on Various Implementation Approaches of Domain-Specific Language. *Information and Software Technology*, 50(5):390–405, 2008.

[KM06] Roman Knöll and Mira Mezini. Pegasus: First Steps toward a Naturalistic Programming Language. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06 [TC06])*, pages 542–559, New York, NY, USA, 2006. ACM.

[KM09] R. Knöll and M. Mezini. π: a Pattern Language. *ACM SIGPLAN Notices*, 44(10):503–522, 2009.

[Kni07] G. Kniesel. Detection and Resolution of Weaving Interactions. *TAOSD: Dependencies and Interactions with Aspects*, LNCS, 2007. Special Issue on Aspect Dependencies and Interactions, edited by R. Chitchyan.

[Knu68] D.E. Knuth. Semantics of Context-Free Languages. *Theory of Computing Systems*, 2(2):127–145, 1968.

[KRB91] Gregor Kiczales, Jim d. Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: Modular Development of Textual Domain Specific Languages. *Objects, Components, Models and Patterns*, 11:297–315, 2008. 10.1007/978-3-540-69824-1_17.

[KV10] Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In *Proceedings of OOPSLA'10 [CCR10]*, 2010.

[KW91] U. Kastens and W. M. Waite. An Abstract Data Type for Name Analysis. *Acta Informatica*, 28:539–558, 1991.

[Lan66] P. J. Landin. The Next 700 Programming Languages. *Commun. ACM*, 9(3):157–166, 1966.

[LF93] Shinn-Der Lee and Daniel P. Friedman. Quasi-Static Scoping: Sharing Variable Bindings across Multiple Lexical Scopes. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'93)*, pages 479–492, New York, NY, USA, 1993. ACM.

[LH06] Andres Löh and Ralf Hinze. Open Data Types and Open Functions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pages 133–144, New York, NY, USA, 2006. ACM.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad Transformers and Modular Interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343, New York, NY, USA, 1995. ACM.

[Lie04] Karl Lieberherr, editor. *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD'04)*, Lancaster, UK, Mar 2004. ACM Press, New York, NY, USA.

[LK97] C. Lopes and G. Kiczales. D: A Language Framework for Distributed Programming. Technical report, Northeastern University, Boston, MA, USA, 1997.

[LK03] D.H. Lorenz and S. Kojarski. Reflective Mechanisms in AOP Languages. Technical report, Northeastern, 2003.

[LLMS00] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. Implicit Parameters: Dynamic Scoping with Static Types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, pages 108–118, New York, NY, USA, 2000. ACM.

[LM00] Daan Leijen and Erik Meijer. Domain Specific Embedded Compilers. *Proceedings of the 2nd Conference on Conference on Domain-Specific Languages (DSL'99)*, 35(1):109–122, 2000.

[LMB92] John Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O'Reilly, 1992. 2nd edition.

[LN04] C.V. Lopes and T.C. Ngo. The Aspect-Oriented Markup Language and its Support of Aspect Plugins. Technical report, ISR Technical Report UCI-ISR-04-8, University of California, Irvine, 2004.

[Lop97] Cristina Lopes. *D: A Language Framework For Distributed Programming*. PhD thesis, College of Computer Science of Northeastern University, 1997.

[LY99] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley, Boston, 1999.

[Mae87] P. Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987.

[Mae88] P. Maes. Computational reflection. *The Knowledge Engineering Review*, 3(01):1–19, 1988.

[Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[MC02] Finbar McGurren and Damien Conroy. X-Adapt: An Architecture for Dynamic Systems. In *Finbar McGurren and Damien Conroy*, 2002.

[ME00] J. Melton and A. Eisenberg. *Understanding SQL and Java together: a Guide to SQLJ, JDBC, and related Technologies*. Morgan Kaufmann, 2000.

[Mey88] B. Meyer. Harnessing Multiple Inheritance. *Journal of Object-Oriented Programming*, 1(4):48–51, 1988.

[Mey89] Norman K. Meyrowitz, editor. *Proceedings of the th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'89)*, SIGPLAN Notices 24(10), New Orleans, Louisiana, 1989. ACM Press, New York, NY, USA.

[MHS05] M. Mernik, J. Heering, and A.M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.

[MJ98] W.S. Miles and L. Johnson. Implementing Generalized Operator Overloading. *Software-Practice and Experience*, 28(6):593–610, 1998.

[MK03] H. Masuhara and K. Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. *Asian Symposium on Programming Languages and Systems*, 2003.

[MKD03] H. Masuhara, G. Kiczales, and C. Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs. In Görel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin / Heidelberg, 2003.

[MLAZ00] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. Compiler/Interpreter Generator System LISA. *Hawaii International Conference on System Sciences*, 8:8059, 2000.

[MMP89] O. L. Madsen and B. Moller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Proceedings of OOPSLA'89 [Mey89]*, New York, NY, USA, 1989. ACM.

[Mog89] E. Moggi. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[Moo01] L. Moonen. Generating Robust Parsers using Island Grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 13–22. IEEE Computer Society, October 2001.

[Mos80] Peter Mosses. A Constructive Approach to Compiler Correctness. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 449–469. Springer Berlin / Heidelberg, 1980.

[MS02] SL Peyton Jones MB Shields. First Class Modules for Haskell. In *9th International Conference on Foundations of Object-Oriented Languages (FOOL'02)*, pages 28–40, Jan 2002.

[MW10] Antoine Marot and Roel Wuyts. Composing Aspects with Aspects. In *Proceedings of AOSD'10 [JS10]*, pages 157–168, 2010.

[NB09] Robert France Nelly Bencomo, Gordon Blair, editor. *1st International Workshop on Models at Run-Time*, 2009.

[NCM03] N. Nystrom, M.R. Clarkson, and A.C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 138–152, Berlin, Heidelberg, 2003. Springer-Verlag.

[New73] William M. Newman. An Informal Graphics System based on the LOGO Language. In *AFIPS '73: Proceedings of the June 4-8, 1973, national computer conference and exposition*, pages 651–655, New York, NY, USA, 1973. ACM.

[oD65] Department of Defense. COBOL Edition 1965. Technical Report Formnr. 0-795-689, US-Government Printing Office, 1965.

[OMG04] OMG. UML 2.0 Superstructure. Technical report, Object Management Group, 2004.

[OOP00] *Proceedings of the th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, SIGPLAN Notices 35(10), Minneapolis, Minnesota, USA, 2000. ACM Press, New York, NY, USA.

[OSV07] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, Mountain View, CA, 2007.

[PAG03] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-In-Time Aspects: Efficient Dynamic Weaving for Java. In *Proceedings of AOSD'03 [Aks03]*, pages 100–109, New York, NY, USA, 2003. ACM.

[Par72] D.L. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1058, 1972.

[Par93] T.J. Parr. *Obtaining Practical Variants of LL (k) and LR (k) for k $>$ 1 by Splitting the Atomic k-Tuple*. PhD thesis, Purdue University, 1993.

[Par08] T. Parr. The Reuse of Grammars with Embedded Semantic Actions. In *Proceedings of the 16th IEEE Conference on Program Comprehension (ICPC 2008)*, pages 5–10, 2008.

[Par10] T. Parr. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages. *The Pragmatic Bookshelf*, 2010.

[PE88] F. Pfenning and C. Elliot. Higher-Order Abstract Syntax. *SIGPLAN Not.*, 23(7):199–208, 1988.

[Pes01] F. Peschanski. Jargons: Experimenting Composable Domain-Specific Languages. In *Workshop on Scheme and Functional Programming*, Firenze, Italy, 2001.

[PGA01] A. Popovici, T. Gross, and G. Alonso. Dynamic Homogenous AOP with PROSE. Technical report, Department of Computer Science, ETH Zürich, Zürich, Switzerland, March 2001.

[PK08] V. Pieterse and D. Kourie. Reflections on Coding Standards in Tertiary Computer Science Education. *South African Computer Journal*, 41:29–37, 2008.

[Plo81] G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[PP08] M. Pfeiffer and J. Pichler. A Comparison of Tool Support for Textual Domain-Specific Languages. In *Proceedings of the 8th Workshop on Domain-Specific Modeling*, pages 1–7, New York, NY, USA, 2008. ACM.

[Pre97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer Berlin / Heidelberg, 1997.

[PS83] H. Partsch and R. Steinbrüggen. Program Transformation Systems. *ACM Comput. Surv.*, 15(3):199–236, 1983.

[RDN10] Lukas Renggli, Marcus Denker, and Oscar Nierstrasz. Language Boxes. In Mark van den Brand, Dragan Gasevic, and Jeff Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 274–293. Springer Berlin / Heidelberg, 2010.

[Res90] M. Resnick. MultiLogo: A Study of Children and Concurrent Programming. *Interactive learning environments*, 1(3):153–170, 1990.

[RGN10] L. Renggli, T. Gîrba, and O. Nierstrasz. Embedding Languages without Breaking Tools. *ECOOP*, 6183:380–404, 2010.

[RMH^{+}06] D. Rebernak, M. Mernik, P.R. Henriques, D. da Cruz, and M.J.V. Pereira. Specifying Languages using Aspect-Oriented Approach: AspectLISA. In *28th International Conference on Information Technology Interfaces*, pages 695–700, 2006.

[RMHP06] D. Rebernak, M. Mernik, P.R. Henriques, and M.J.V. Pereira. AspectLISA: an Aspect-Oriented Compiler Construction System based on Attribute Grammars. *Electronic Notes in Theoretical Computer Science*, 164(2):37–53, 2006.

[RMWG09] D. Rebernak, M. Mernik, H. Wu, and J. Gray. Domain-Specific Aspect Languages for Modularising Crosscutting Concerns in Grammars. *Software, IET*, 3(3):184–200, june 2009.

[RS84] Jim des Rivières and Brian Cantwell Smith. The Implementation of Procedurally Reflective Languages. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP'84)*, pages 331–347, New York, NY, USA, 1984. ACM.

[Rub] Ruby Programming Language. `http://www.ruby-lang.org/`.

[SBP99] Tim Sheard, Zine-el-abidine Benaissa, and Emir Pasalic. DSL Implementation using Staging and Monads. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 81–94, New York, NY, USA, 1999. ACM.

[SBP$^+$09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks, Rochard C. Gronback, and Mike Milinkovich. *EMF: Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, Upper Saddle River, NJ, 2nd edition, 2009.

[Sca] The Scala Programming Language. `http://www.scala-lang.org/index.html`.

[SCD03] Nikita Synytskyy, James R. Cordy, and Thomas R. Dean. Robust Multilingual Parsing using Island Grammars. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research*, pages 266–278. IBM Press, 2003.

[SCK04] Sean Seefried, Manuel Chakravarty, and Gabriele Keller. Optimising Embedded DSLs Using Template Haskell. *Generative Programming and Component Engineering*, 3286:201–211, 2004.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behaviour. In *ECOOP 2003 – Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 327–339. Springer Berlin / Heidelberg, 2003.

[She04] Tim Sheard. Languages of the Future. *SIGPLAN Not.*, 39(12):119–132, 2004.

[SHH09] Hans Schippers, Michael Haupt, and Robert Hirschfeld. An Implementation Substrate for Languages Composing Modularized Crosscutting Concerns. In *Proceedings of the 2009 ACM symposium on Applied Computing (SAC'09)*, pages 1944–1951, New York, NY, USA, 2009. ACM.

[Sim95] Charles Simonyi. The Death of Computer Languages, The Birth of Intentional Programming. Technical Report MSR-TR-95-52, Microsoft Research, Redmond, WA, USA, September 1995.

[SK97] J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4):110–111, 1997.

[SLS03] Macneil Shonle, Karl Lieberherr, and Ankit Shah. XAspects: An Extensible System for Domain Specific Aspect Languages. In *Proceedings of OOPSLA'03 [CJ03]*, pages 28–37, 2003.

[Smi82] B.C. Smith. Reflection and Semantics in a Procedural Language. Technical Report 272, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts, USA, 1982.

[Smi84] Brian Cantwell Smith. Reflection and Semantics in LISP. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming languages (POPL'84)*, pages 23–35, New York, NY, USA, 1984. ACM.

[SN05] J.E. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Pub, 2005.

[SP06] P. Shaker and D. Peters. Design-level detection of interactions in aspect-oriented systems. In *Aspects, Dependencies and Interactions Workshop at ECOOP 2006*, 2006.

[SS71] D. Scott and C. Strachey. Toward a Mathematical Semantics for Computer Languages. In *Proceedings of the Symposium on Computers and Automata*, volume 21, page 19. Polytechnic Press of the Polytechnic Institute of Brooklyn, Wiley-Interscience, New York, 1971.

[SS78] G. Steele and G. Sussman. The Revised Report an SCHEME, a Dialect of IJSI. Technical Report Memo AIM-452, M.I.T. Artificial Intelligence Imboratory, 1978.

[SS98] Gerald Jay Sussman and Guy L. Steele. Scheme: A Interpreter for Extended Lambda Calculus. *Higher-Order and Symbolic Computation*, 11:405–439, 1998.

[Ste87] Lynn Andrea Stein. Delegation is Inheritance. In *Proceedings of the International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 138–146, New York, NY, USA, 1987. ACM.

[Ste94] Guy L. Steele. Building Interpreters by Composing Monads. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 472–492. ACM New York, NY, USA, 1994.

[Ste99] G.L. Steele. Growing a Language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.

[Stu09] Aaron Stump. Directly Reflective Meta-Programming. *Higher-Order and Symbolic Computation*, 22:115–144, 2009.

[Sul01] G.T. Sullivan. Aspect-Oriented Programming using Reflection and Metaobject Protocols. *Communications of the ACM*, 44(10):95–97, 2001.

[SVJ03] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: An Aspect-Oriented Approach tailored for Component based Software Development. In *Proceedings of AOSD'03 [Aks03]*, pages 21–29, 2003.

[Tan04] É. Tanter. *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*. PhD thesis, Université de Nantes, France, 2004.

[Tan06] E. Tanter. Aspects of Composition in the Reflex AOP Kernel. *LNCS*, 4089:98, 2006.

[Tan08] Éric Tanter. Expressive Scoping of Dynamically-Deployed Aspects. In *Proceedings of AOSD'08 [D'H08]*, pages 168–179, 2008.

[Tan09] Éric Tanter. Beyond Static and Dynamic Scope. *SIGPLAN Not.*, 44(12):3–14, 2009.

[Tan10] Éric Tanter. Execution Levels for Aspect-Oriented Programming. In *Proceedings of AOSD'10 [JS10]*, pages 37–48, 2010.

[Tay92] P. Taylor. The Future of TEX. *Proceedings of the Conference EuroTEX*, 92:235–254, 1992.

[TC06] Peri L. Tarr and William R. Cook, editors. *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, Portland, Oregon, USA, 2006. ACM Press, New York, NY, USA.

[TCKI00] M. Tatsubori, S. Chiba, M.O. Killijian, and K. Itano. OpenJava: A Class-Based Macro System for Java. *Reflection and Software Engineering*, 1826, 2000.

[Ten76] RD Tennent. The Denotational Semantics of Programming Languages. *Communications of the ACM*, 19(8):453, 1976.

[TFH09] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby 1.9: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2009.

[TN05] Éric Tanter and Jacques Noyé. A Versatile Kernel for Multi-language AOP. *Generative Programming and Component Engineering*, 3676:173–188, 2005.

[Tra08] Laurence Tratt. Domain Specific Language Implementation via Compile-Time Meta-Programming. *ACM Trans. Program. Lang. Syst.*, 30(6):1–40, 2008.

[TT08] R. Toledo and E. Tanter. A Lightweight and Extensible AspectJ Implementation. *Journal of Universal Computer Science*, 14(21):3517–3533, 2008.

[UMT04] N. Ubayashi, H. Masuhara, and T. Tamai. An AOP Implementation Framework for Extending Join Point Models. In *ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2004.

[US87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. *SIGPLAN Not.*, 22(12):227–242, 1987.

[vdBK07] M.G.J. van den Brand and P. Klint. ATerms for Manipulation and Exchange of Structured Data: It's all about Sharing. *Information and Software Technology*, 49(1):55–64, 2007.

[vdBSV97] M. van den Brand, M. Sellink, and C. Verhoef. Obtaining a Cobol Grammar from Legacy Code for Reengineering Purposes. *Proceedings of the Second International Workshop on Theory and Practice of Algebraic Specifications*, 1997.

[vdBSVV02] Mark van den Brand, Jeroen Scheerder, Jurgen Vinju, and Eelco Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In R. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 21–44. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45937-5_12.

[vdBvDK^{+}96] M. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. van Der Meulen. Industrial Applications of ASF+SDF. *Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology (AMAST'96)*, 1101:9–18, 1996.

[vDKV00] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[vG00] J. van Gurp. *Variability in Software Systems: The Key to Software Reuse*. PhD thesis, Dept. of Software Engineering & Computer Science, Blekinge Institute of Technology, Karlskrona, Sweden, 2000.

[VGBS01] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[Vis97a] E. Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, University of Amsterdam, The Netherlands, 1997.

[Vis97b] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, Universiteit van Amsterdam, The Netherlands, 1997.

[Vis97c] Eelco Visser. A Family of Syntax Definition Formalisms. Technical Report P9706, Programming Research Group, 1997.

[Vis05] E. Visser. A Survey of Strategies in Rule-Based Program Transformation Systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.

[Vis08] E. Visser. WebDSL: A Case Study in Domain-Specific Language Engineering. *Generative and Transformational Techniques in Software Engineering II*, 5235:291–373, 2008.

[VR09] T. Verwaest and L. Renggli. Safe Reflection through Polymorphism. In *Proceedings of the first International Workshop on Context-Aware Software Technology and Applications*, pages 21–24. ACM, 2009.

[VS04] John M. Vlissides and Douglas C. Schmidt, editors. *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, Vancouver, BC, Canada, 2004. ACM Press, New York, NY, USA.

[VWBGK08] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an Extensible Attribute Grammar System. *Electron. Notes Theor. Comput. Sci.*, 203(2):103–116, 2008.

[VWBH06] E. Van Wyk, D. Bodin, and P. Huntington. Adding Syntax and Static Analysis to Libraries via Extensible Compilers and Language Extensions. In *2nd International Workshop on Library-Centric Software Design (LCSD'06)*, page 35, New York, NY, USA, 2006. ACM.

[Wad90] Philip Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming (LFP'90)*, pages 61–78, 1990.

[Wam08] Dean Wampler. Aquarium: AOP in Ruby. In *Proceedings of AOSD'04 [Lie04]*, 2008.

[WF88] Mitchell Wand and Daniel P. Friedman. The Mystery of the Tower Revealed: A Nonreflective Description of the Reflective Tower. *LISP and Symbolic Computation*, 1:11–38, 1988.

[WfM] The Workflow Management Coalition. `http://www.wfmc.org/`.

[WGM09] Hui Wu, Jeff Gray, and Marjan Mernik. Unit Testing for Domain-Specific Languages. *Domain-Specific Languages*, 5658:125–147, 2009.

[WKBS07] Eric Van Wyk, Lijesh Krishnan, Derek Bodin, and August Schwerdfeger. Attribute Grammar-Based Language Extensions for Java. *ECOOP 2007–Object-Oriented Programming*, 4609:575–599, 2007.

[WKD04] M. Wand, G. Kiczales, and C. Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. *Transactions on Programming Languages and Systems*, 26(5), 2004.

[ZGL05] Jing Zhang, Jeff Gray, and Yuehua Lin. Metamodel-Driven Model Interpreter Evolution. In *Proceedings of the Domain-Specific Modeling Workshop*, 2005.

[Zha06] G. Zhang. Towards Aspect-Oriented State Machines. In *Asian Workshop on Aspect-Oriented Software Development (AOAsia'06)*, 2006.

[ZHS04] David Zook, Shan Shan Huang, and Yannis Smaragdakis. Generating aspectj programs with meta-aspectj. In Gabor Karsai and Eelco Visser, editors, *Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*, pages 583–605. Springer Berlin / Heidelberg, 2004.

Scientific Career

Since Jun 2005
Technische Universität Darmstadt, Germany
Research Assistant in the Software Technology Group of Prof. Mira Mezini

Apr 2007 – Sep 2007
SAP Research, Palo Alto, CA, USA
Internship Abroad

Mar 2005
Technische Universität Darmstadt, Germany
Diploma in Computer Science

Sep 2002 – Mar 2003
University of Birmingham, England
Study Abroad in Computer Science

Aug 1998 – Mar 2005
Technische Universität Darmstadt, Germany
Studies in Computer Science