TECHNISCHE
UNIVERSITÄT
DARMSTADT

Inter-Model Consistency Checking and Restoration with Triple Graph Grammars

DEM FACHBEREICH ELEKTROTECHNIK UND INFORMATIONSTECHNIK
DER TECHNISCHEN UNIVERSITÄT DARMSTADT
ZUR ERLANGUNG DES AKADEMISCHEN GRADES
EINES DOKTOR-INGENIEURS (DR.-ING.)
GENEHMIGTE DISSERTATION

VON

ERHAN LEBLEBICI

GEBOREN AM

31. MAI 1988 IN ISTANBUL, TÜRKEI

ERSTGUTACHTER: PROF. DR. ANDY SCHÜRR
ZWEITGUTACHTER: PROF. DR. BERNHARD WESTFECHTEL

TAG DER EINREICHUNG: 30.01.2018
TAG DER DISPUTATION: 04.05.2018

DARMSTADT 2018

The work of Erhan Leblebici was supported by the Graduate School of Excellence Computational Engineering (CE) at the Technische Universität Darmstadt.

The GraTraM project presented in this thesis was funded by the German Federal Ministry of Education and Research, funding code 01 | S12054, in the context of the Software Campus (www.softwarecampus.de).

Leblebici, Erhan
Inter-Model Consistency Checking and Restoration with Triple Graph Grammars
Darmstadt, Technische Universität Darmstadt
Year of publication at TUprints: 2018
Disputation date: May 4, 2018

DECLARATION OF AUTHORSHIP

I warrant that the thesis presented here is my original work and that I have not received outside assistance. All references and other sources that I used have been appropriately acknowledged in the work. I further declare that the work has not been submitted anywhere else for the purpose of academic examination, either in its original or similar form.

I hereby grant the Real-Time Systems Lab the right to publish, reproduce and distribute my work.

Darmstadt, 30.01.2018

Erhan Leblebici

Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.

Edsger Wybe Dijkstra

ACKNOWLEDGMENTS

I wish to thank

Prof. Dr. Andy Schürr for his supervision which has always made me feel lucky in both personal and technical ways,

Prof. Dr. Bernhard Westfechtel for kindly accepting to serve the PhD thesis committee and for his valuable comments on an earlier version of this document,

Dr. Anthony Anjorin, *Dr. Gergely Varró*, and *Lars Fritsche* for many fruitful hours of co-desinging, co-developing, and co-authoring,

all those who shared my PhD highs and lows at the university canteen in Darmstadt or during my time-constrained but intense visits in Istanbul,

finally yet foremost, my parents *Mehmet* and *Nursen Leblebici* and my brother *İhsan Metin Leblebici* for their continuous and unconditional support, no matter the distance and no matter the circumstance.

You all made this work possible.

Darmstadt, 2018

ABSTRACT

Software development is a complex task. The success of a software project highly relies on the involvement of domain experts in the development process. In recent years, therefore, the field of software engineering has been striving to elevate the level of abstraction towards domain-specific concepts (instead of the computation-oriented nature of classical programming languages). *Model-Driven Engineering* (MDE), a novel software development methodology, lies at the heart of these efforts. In MDE, a model represents an abstraction of a system with one specific goal in mind. Hence, the MDE strategy does not only deal with abstractions but also advocates the co-existence of related models capturing different aspects of the same system. While this supports separation of concerns, *consistency management* between related models becomes a crucial challenge as models are changed throughout their life cycle. Two basic building blocks of consistency management are (i) *consistency checking* to indicate whether or to what extent two related models are consistent and (ii) *consistency restoration* to suitably handle discrepancies between models.

To address consistency management tasks in a formally-founded manner, *bidirectional transformations* (BX) have been established as a research area. Among the diverse BX approaches, *Triple Graph Grammars* (TGGs) represent a prominent technique with various implementations and industrial applications. In this setting, models are formalized as graphs and consistency is described as a grammar constructing two consistent models together with a correspondence model. Consistency management tools are then automatically derived from this description.

Current TGG approaches (and in fact also BX approaches in general) focus on consistency scenarios where only one model is maintained by human intelligence at the same time and the other one is automatically updated by consistency restoration. Consistency management between two concurrently developed models, however, is not sufficiently supported as practical solutions for consistency checking are essentially missing. Strategies for consistency restoration, furthermore, range from heuristics to auxiliary model analyses which constitute the most complex and least understood part of TGGs. Despite sharing the same basic goal and the same formal foundation, it is difficult to exchange ideas amongst the different TGG approaches.

In this thesis, therefore, we first establish consistency checking as a novel use case of TGGs. We identify search space problems in consistency checking and overcome them by combining TGGs with *linear optimization* techniques. Second, we propose a novel consistency restoration procedure that exploits *incremental pattern matching* techniques to address the intermediate steps of consistency restoration in a simplified manner. Furthermore, we present a TGG tool that implements our formal results and experimentally evaluate its scalability in real-world consistency scenarios. Finally, we report on an industrial project from the mechanical engineering domain where we applied this tool for maintaining consistency between computer-aided design and mechatronic simulation models.

ZUSAMMENFASSUNG

Softwareentwicklung ist eine komplexe Aufgabe. Der Erfolg eines Softwareprojektes ist in hohem Maße auf die Beteiligung der Domänenexperten im Entwicklungsprozess angewiesen. Das Gebiet Softwaretechnik strebt daher höhere Abstraktionsstufen mit Fokus auf domänenspezifische Konzepte an (anstatt der auf Rechnen orientierten Natur der klassischen Programmiersprachen). *Model-Driven Engineering* (MDE), eine neue Softwareentwicklungsmethodik, liegt im Mittelpunkt dieser Bestrebungen. In MDE stellt ein Modell eine Abstraktion eines Systems mit einem spezifischen Ziel dar. Folglich beschäftigt sich MDE nicht nur mit Abstraktionen, sondern befürwortet auch die Koexistenz verwandter Modelle, die dasselbe System beschreiben. Während dies die Trennung der Zuständigkeiten unterstützt, wird *Konsistenzverwaltung* zwischen verwandten Modellen eine wichtige Herausforderung, da Modelle sich im Laufe ihres Lebenszyklus ändern. Zwei Grundbausteine der Konsistenzverwaltung sind (i) *Konsistenzprüfung*, um zu bestimmen, ob oder inwiefern zwei verwandte Modelle konsistent sind und (ii) *Konsistenzwiederherstellung*, um Diskrepanzen zwischen den Modellen zu beseitigen.

Um Konsistenzverwaltung mit formalen Mitteln anzugehen wurden *bidirektionale Transformationen* (BX) als ein Forschungsfeld etabliert. Unter den BX-Ansätzen stellen *Tripel-Graph-Grammatiken* (TGGen) eine prominente Technik mit verschiedenen Implementierungen und industriellen Anwendungen dar. Dabei werden Modelle als Graphen und Konsistenzbeziehungen als eine Grammatik beschrieben, die zwei konsistente Modelle mit einem Korrespondenzmodell aufbaut. Werkzeuge zur Konsistenzverwaltung werden aus dieser Beschreibung generiert.

Existierende TGG-Ansätze (sowie BX-Ansätze generell) setzen weitgehend voraus, dass gleichzeitig nur das eine Modell durch menschliche Intelligenz gepflegt und das andere durch Konsistenzwiederherstellung automatisch aktualisiert wird. Das Problem der Konsistenzerhaltung zwischen zwei konkurrierend gepflegten Modellen bleibt dagegen offen, da praktische Lösungen zur Konsistenzprüfung fehlen. Des Weiteren reichen die Strategien zur Konsistenzwiederherstellung von Heuristiken bis hin zu zusätzlichen Modellanalysen, die den unübersichtlichsten Teil der TGGen bilden. Das erschwert auch den Ideenaustausch zwischen unterschiedlichen TGG-Ansätzen, obwohl diese dieselben Ziele und Grundlagen haben.

In dieser Thesis wird erstens Konsistenzprüfung als ein neuer Anwendungsfall der TGGen eingeführt. Suchraumprobleme in Konsistenzprüfung werden identifiziert und durch *lineare Optimierungstechniken* bewältigt. Zweitens wird eine neue Prozedur zur Konsistenzwiederherstellung vorgestellt, die von Techniken zur *inkrementellen Graphmustersuche* Gebrauch macht, um Konsistenzwiederherstellung zu vereinfachen. Des Weiteren wird ein TGG-Werkzeug vorgestellt, das die formalen Ergebnisse umsetzt und dessen Skalierbarkeit mit realen Beispielen evaluiert wird. Zum Abschluss wird über ein Industrie-Projekt aus der Maschinenbau-Domäne berichtet, in dem dieses Werkzeug zur Konsistenzerhaltung zwischen Modellen von rechnergestütztem Konstruieren und mechatronischer Simulation benutzt wurde.

CONTENTS

API	Application Programming Interface
BX	Bidirectional Transformations
CAD	Computer-Aided Design
EMF	Eclipse Modeling Framework
HTML	Hypertext Markup Language
ILP	Integer Linear Programming
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MOF	Meta-Object Facility
NAC	Negative Application Condition
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform-Independent Model
PSM	Platform-Specific Model
QVT-R	Query/View/Transformation - Relations
TGG	Triple Graph Grammar
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1

OVERVIEW AND MOTIVATION

Since the appearance of first computers in the 1930s, the rapid increase in computing power of hardware has made it imaginable to tackle more and more complex tasks with computers. The difficulty of writing useful software meeting the complex requirements, however, has always been the limiting factor in exploiting computers. In the 1960s, even the term *software crisis* [111] was coined at the intergovernmental level referring to software products that are of low quality, inefficient, not meeting the requirements, or undelivered, and referring to software development processes that run over time and over budget. To this end, the then practically unknown field of *software engineering* has been established to apply systematic engineering methods to the development of software. Today, software engineering has a strong impact on our daily life as computers are employed in almost every context, be it a workstation, an industrial process, a medical treatment, a means of transportation, or a consumer good.

The essence of software engineering is the division between hardware and software by introducing appropriate *abstractions* to deal with the increasing complexity. High-level programming languages provide a means for this abstraction where human-readable source code is compiled to or interpreted as executable machine code. Although such languages have made a significant contribution to the technological progress in the last decades, they still have a computation-oriented focus. Hence, the achieved level of abstraction is still not sufficient to incorporate non-developer domain experts (in general users of a software product) into the development process. Indeed, recent *chaos reports* [29] on information technology projects identify user involvement as the key factor for success and lack of user input as currently the most challenging problem.

There are educational measures to bridge the gap between domain experts and software engineers including, e.g., integrating a specific field as secondary subject into computer science courses (or vice versa), and even inherently combined courses such as business informatics or bioinformatics. While such interdisciplinary education can provide qualified workforce for the interface between software and its application domain, there is still a need for technological change (with regard to tools, languages, and formalisms) to facilitate software development with domain experts. With this in mind, the field of software engineering has been striving to elevate the level of abstraction towards domain-specific concepts rather than computational details in recent years. The software development methodology *Model-Driven Engineering* (MDE) lies at the heart of these efforts.

In an MDE context, a *model* is considered as an abstraction of a system built for a specific goal in mind and provides information only related to this goal [20, 23, 90].

The most prominent and standardized set of guidelines for how to structure and organize models in MDE has been established by the Object Management Group (OMG) under the name of *Model-Driven Architecture* (MDA) [110]. The MDA strategy comes along with a set of standards enjoying broad acceptance such as *Meta-Object Facility* (MOF) [107] to define domain-specific modeling languages, *Unified Modeling Language* (UML) [141] to create object-oriented design models, and *XML Metadata Interchange* (XMI) [149] to persist and exchange models in a common format.

1.1 Co-existing Models and Bidirectional Transformations (BX)

Defining a model as intended for a specific goal, the MDA strategy strongly advocates separation of concerns via co-existing models for a system instead of capturing the whole system as one global model. Depending on the abstraction level of models, MDA distinguishes between the two basic concepts of *Platform-Independent Models* (PIM) and *Platform-Specific Models* (PSM). While the term platform here might be associated with different abstraction levels such as hardware architecture, operating system, or programming language, PIMs and PSMs of a system remain in all cases relative to each other in the following sense: PIMs abstract from technological details of the platform and represent business logic of the system while PSMs stipulate how to realize the system in the respective platform. Hence, PIMs are rather expected to integrate domain experts with less software engineering knowledge into the development process while PSMs are executable artifacts (or should directly lead to executable artifacts). Therefore, the MDE vision is not only concerned with introducing higher abstraction levels but also suggests to *regard everything as a model* consequently, e.g., even source code written in a programming language that exists longer than the MDE vision itself is a model.

Figure 1.1 shows an excerpt of different PIMs and PSMs that can be involved in the development of a software system. The models cover overall three different goals, namely documentation, implementation, and testing of the system. At the highest abstraction level, i.e., at the level of PIMs, there exists a documentation model that abstracts from the format of the documentation. Similarly, an object-oriented design in UML as well as a test suite model abstract from the programming language used for implementation and testing, respectively. Considering the level of code as platform in our concrete case, the models at the PSM level represent details concerning the choice of the programming language for each individual goal. At the documentation side, for example, the Hypertext Markup Language (HTML) is chosen to be the language of documentation artifacts, and the HTML model (basically a PSM) organizes the documentation data in the form of HTML elements (called HTML tags). Analogously, assuming Java as the programming language of the project, the Java model in the middle captures the object-oriented design with Java language features and possibly limitations in mind (the same applies to the Java model for testing at bottom right).

Co-existence of different models representing the same system facilitates a modular development process but requires appropriate *consistency management* as models are due to change during their life cycles. In its most broad interpretation with regard to the goals of this thesis, *consistency* of two models refers to a state where no

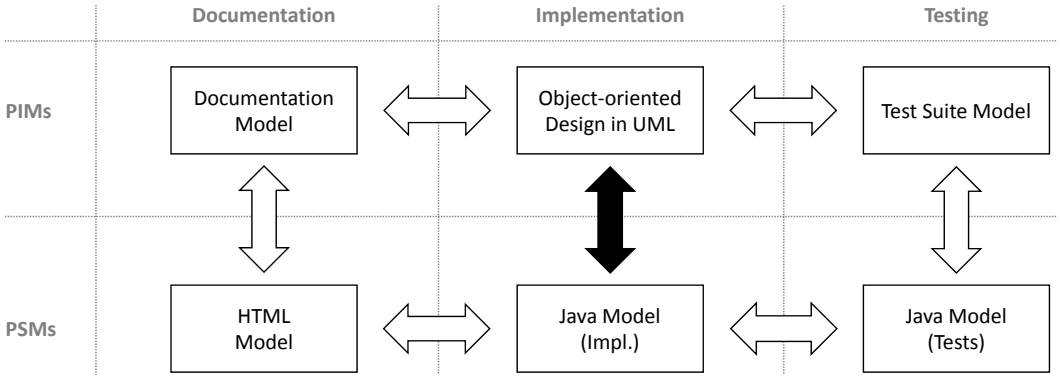


Figure 1.1: Exemplary PIMs and PSMs in a software engineering project

contradiction exists between the models. In other words, consistent models agree on the information they are meant to represent in common. Consistency becomes a relevant challenge especially in large software projects when different domains are involved with their own models, and models are developed *concurrently* by their owners. In general, consistency must be addressed *vertically* between different abstraction levels, e.g., between a PIM and a PSM from the same domain, as well as *horizontally*, e.g., between two PIMs or between two PSMs from different domains.

The bidirectional arrows in Figure 1.1 represent the choice of model pairs between which a tool-supported consistency management is desired for the concrete scenario (the black-filled arrow will be further investigated in detail as running example throughout the thesis). The vertical arrows indicate that two models of different abstraction levels (but sharing the same goal) must conform to each other in all cases while horizontal arrows represent a consistency management strategy that considers implementation as the central point of consistency, i.e., documentation as well as test models must be kept consistent to implementation models. It should be mentioned that some of the horizontal arrows in Fig. 1.1 can be realized indirectly in practice, e.g., horizontal consistency at the PSM level can be induced via horizontal consistency at the PIM level (or vice versa) together with vertical consistency. The choice of arrows that must be realized as a distinct component or induced by other arrows is case-specific and depends usually on the internal structures of models or the selected development process.

The MDA specification already exhibits awareness of consistency challenges in MDE from the very beginning and proposes to apply *model transformations*, i.e., producing an output model from a given input model in an automated way, to keep related models consistent. Model transformations thus play a central role in MDA and systematic development of model transformations has been actively investigated with various approaches differing in their supported features, capabilities, and scopes (cf., e.g., [13, 34, 132] for classifications and comparisons of different approaches). If models are intended for specific goals and maintained by different stakeholders, moreover, model transformations are not one-way streets but must be *bidirectional*, i.e., executable in either direction. In line with this observation, a cross-disciplinary research community named *bidirectional transformations* (BX) [35] has been established whose research interests include (but are not limited to) con-

sistency tasks in MDE. Providing appropriate BX formalisms, languages, and tools has been the centre of BX-related research in recent years.

The MDA specification indeed introduces the BX language *Query View Transformation - Relations* (QVT-R) [120]. Unlike the aforementioned MDA standards such as MOF, UML, and XMI, however, QVT-R has not gained a broad acceptance and implementations are scarce [134]. Arguably, describing and maintaining consistency between two models is one of the most complex tasks in the MDE vision and the level of formalization and preciseness in the QVT-R specification does not meet the requirements of this complexity. Recently, ambiguities in the standard and the resulting conformance issues of the respective tool support for QVT-R have been stressed in [146]. The observations from the investigated case studies reveal that BX development with QVT-R turns out to be a trial-and-error process when coping with these problems. Consequently, developers might end up with complex solutions while BX research generally strives to allow for compact and readable solutions as compared to, e.g., a manual implementation.

Nevertheless, a lot of formal work to address BX in an MDE context has been done in the field of *graph grammars*. In this setting, the content of a model is formalized as a graph, and computations on a model (e.g., analyses, manipulations, or consistency management) are captured as *graph grammar rules*. Especially *Triple Graph Grammars* (TGGs) [128], a particular dialect of graph grammars dedicated to BX, have been successfully used in several industrial MDE projects [5, 21, 56, 67, 126], and present a prominent alternative to QVT-R with various implementations [46, 71, 83, 86, 96]. The ancestors of TGGs are *pair grammars* [119] whose central idea is twofold: (i) provide a pair of grammars that build up together consistent pairs of *source* and *target* models, and (ii) automatically derive consistency tools from this specification. Refining this idea, TGGs introduce a third grammar that builds up a third graph representing the *correspondence links* between consistent pairs.

In the universe of BX languages, TGGs belong, similar to QVT-R, to *declarative* approaches where the BX specification only describes what consistency means (for the particular BX scenario) but not how it is maintained. Both approaches, however, particularly differ in the way of describing consistency. A QVT-R specification, on the one hand, consists of *relations* in the form of logical predicates and consistency of two models means the satisfaction of these predicates. A TGG, on the other hand, is a graph grammar constructing consistent models and consistency is defined as membership to the language induced by the grammar. Both approaches, therefore, are commonly distinguished as relation-based and grammar-based within the BX landscape. A further representative group of BX languages is given by those based on *bidirectional programming* as proposed in, e.g., [69, 87, 153]. In this setting, consistency maintenance is programmed in one direction and the corresponding program in the reverse direction is automatically induced. Bidirectional programming languages offer more fine-grained control over consistency maintenance as compared to declarative approaches but provide less abstraction for the necessary computations thereof. Bidirectional programming, furthermore, generally tends to be more restrictive with regard to the supported class of consistency scenarios (e.g., primarily supporting one-to-one mappings between models such that the program

can be reversed unambiguously). TGGs, on the contrary, allow for specifying and maintaining consistency in a flexible manner as we shall demonstrate with our examples throughout the thesis.

1.2 Challenges of BX in an MDE context

The question of how to deal with consistency between related models is strongly coupled with how the models are created and maintained. Current BX approaches, in particular TGG approaches, focus on the case where always one of the models is maintained by human intelligence at the same time and deriving the other one is only routine work (and the goal of BX is to automate this routine work). Such a scenario is depicted in Figure 1.2 where Model A and Model B refer to two related models (e.g., two models at the ends of any bidirectional arrow in Figure 1.1). The rectangles with gearwheels in the middle represent individual runs with a BX tool (the dashed gray arrows indicate the input and the output for each run). The v.X suffixes at each model indicate its version (e.g., v.1 for the first version). Incrementing the version of a model is either the result of a transformation or is done by human intelligence. The latter case is depicted via black arrows with a stick person. Finally, white filling of models indicates that they are created by human intelligence while gray-filled ones are automatically derived via BX.

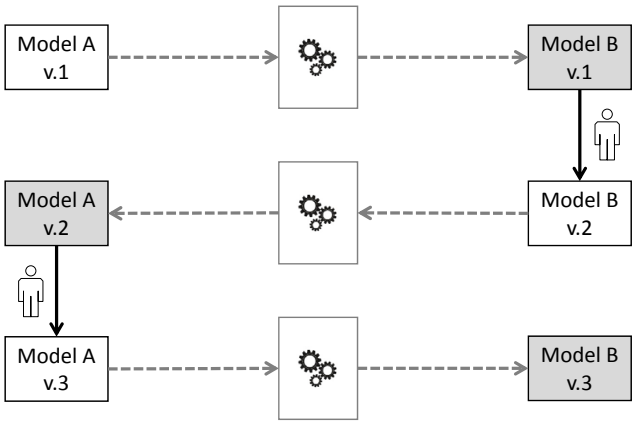


Figure 1.2: Consistency management as addressed by current BX approaches in MDE

In Figure 1.2, the first version of Model A is created by human intelligence as a first step and is transformed to the first version of Model B. After changing Model B to its second version (again by human intelligence), the second version of Model A is derived via a transformation in the reverse direction. Analogously, further changes are done on one of the models, and the other one is derived. In most of the BX approaches, after changing a model with human intelligence, the derivation of the other one is done as an *incremental update*, i.e., the previous version of the derived model is taken into consideration and only changes are propagated incrementally instead of an entire transformation from scratch. BX tools generally keep auxiliary data for the consistency history (e.g., *traces* between models) to calculate what is to be done for updating a model according to the changes on the other side.

In a scenario as depicted in Figure 1.2, there is an assumption that does not necessarily hold in many real-world consistency tasks: Only one of the models is available at the beginning and the first version of the other one is to be automatically derived. Further transformations by propagating changes in one of the models to the other rely on this initial transformation as the starting point. This requires working in a BX-supported environment and applying BX from the very beginning starting with one model. Applying BX, however, can be a decision taken in an intermediate phase of the development process where both models are existent and have already reached an advanced state.

Different models indeed belong to different stakeholders and concurrent developments speed up the development process. Figure 1.3, therefore, depicts the vision of a more general consistency management support eliminating these assumptions and defining the goals of this thesis. This time, the starting point of consistency management allows two already existing models which are concurrently developed. In the initial start, consistency management takes both models as input (indicated as v.1) and derives their updated versions (v.2) yielding a consistent state. Further runs of change propagation from one model to the other (as previously shown in Figure 1.2) are again possible upon the results of this initial start (e.g., changing Model A to its v.3 and deriving v.3 of Model B).

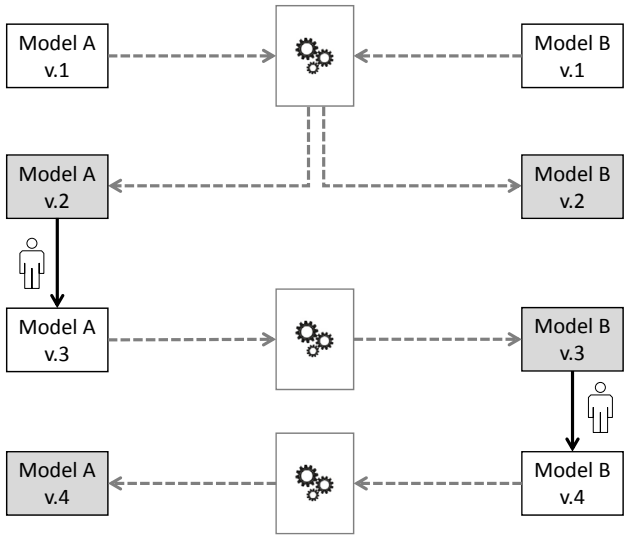


Figure 1.3: Consistency management with two existing models as the starting point

In the following, we explore in more detail (on the level of model elements) what intermediate steps are to be taken into account to realize a BX vision as depicted in Figure 1.3 (these intermediate steps shall define the individual subtasks we address throughout the thesis). For this purpose, an excerpt of the consistency between Java and UML models (the black-filled bidirectional arrow in Figure 1.1) serves as our running example. In particular, we focus on Java and UML classes with their methods. Both models represent a system from an object-oriented view where classes define a template for runtime objects. In fact, there exist UML tools [140] providing their own mechanisms for consistency with Java code but they operate, in line with the BX conception in Figure 1.2 starting with one model. Consistency

management for concurrently developed models, however, are not addressed by these custom-tailored BX solutions either.

In Figure 1.4, an exemplary pair of Java (left) and UML (right) models is depicted both representing a class List. The Java model has two remove methods, one with a parameter of type Object and the other one with a parameter of type int. The UML model has only one remove method (with a parameter of type Object) and, additionally, another method named add (again with a parameter of type Object). Note that these models represent v.1 on both sides when transferred to our BX conception in Figure 1.3.

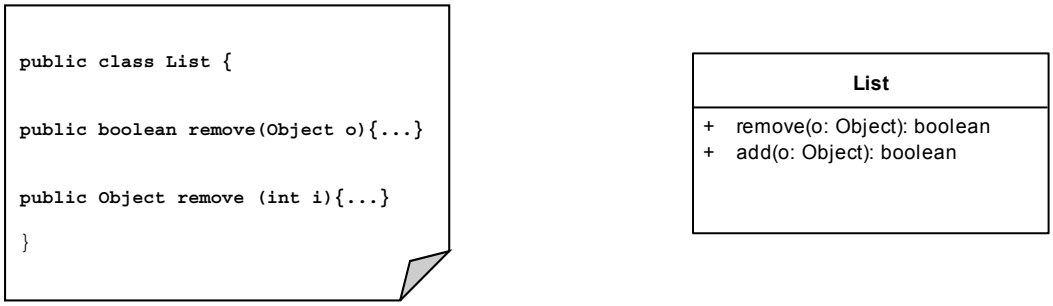


Figure 1.4: An exemplary pair of Java (left) and UML (right) models

There are obvious inconsistencies between the two models in Figure 1.4. The Java model has an additional remove method which might have been introduced by Java programmers as a result of their experience. Furthermore, the Java model misses the add method as compared to the UML model, i.e., a feature planned on the higher abstraction level has not been reflected in the code level yet. Nevertheless, the models have also consistent parts, e.g., they agree on a List class and a remove method with an Object parameter. In order to tackle consistency issues in such a case, there are two basic steps that must be supported by a fully-fledged BX approach: (i) consistency must be *checked* between the two models to detect which parts of the models already correspond to each other (and which parts violate consistency), and subsequently, (ii) consistency must be *restored* depending on the decisions of stakeholders, e.g., by applying model transformations that propagate or simply delete inconsistent parts.

Consistency checking and restoration between concurrently developed models are illustrated in Figure 1.5 based on this exemplary model pair. In addition to the models, the bidirectional dashed arrows in the middle indicate *correspondences* between model elements. New correspondences and model elements created in individual steps are depicted green with an additional «new» markup.

In a first step in Figure 1.5, consistency checking is performed and the correspondences between conforming elements are created, i.e., between the List classes and their remove methods with Object parameter (we omit the correspondence arrows on the parameter level to avoid diagram clutter). As a consequence, the remaining elements can be regarded as violating consistency and must be handled to restore consistency. One of the possibilities would be to delete these elements and to reduce the models to their consistent parts which, however, is a very strict and unsatisfactory solution. In Figure 1.5, a strategy is chosen that transforms the

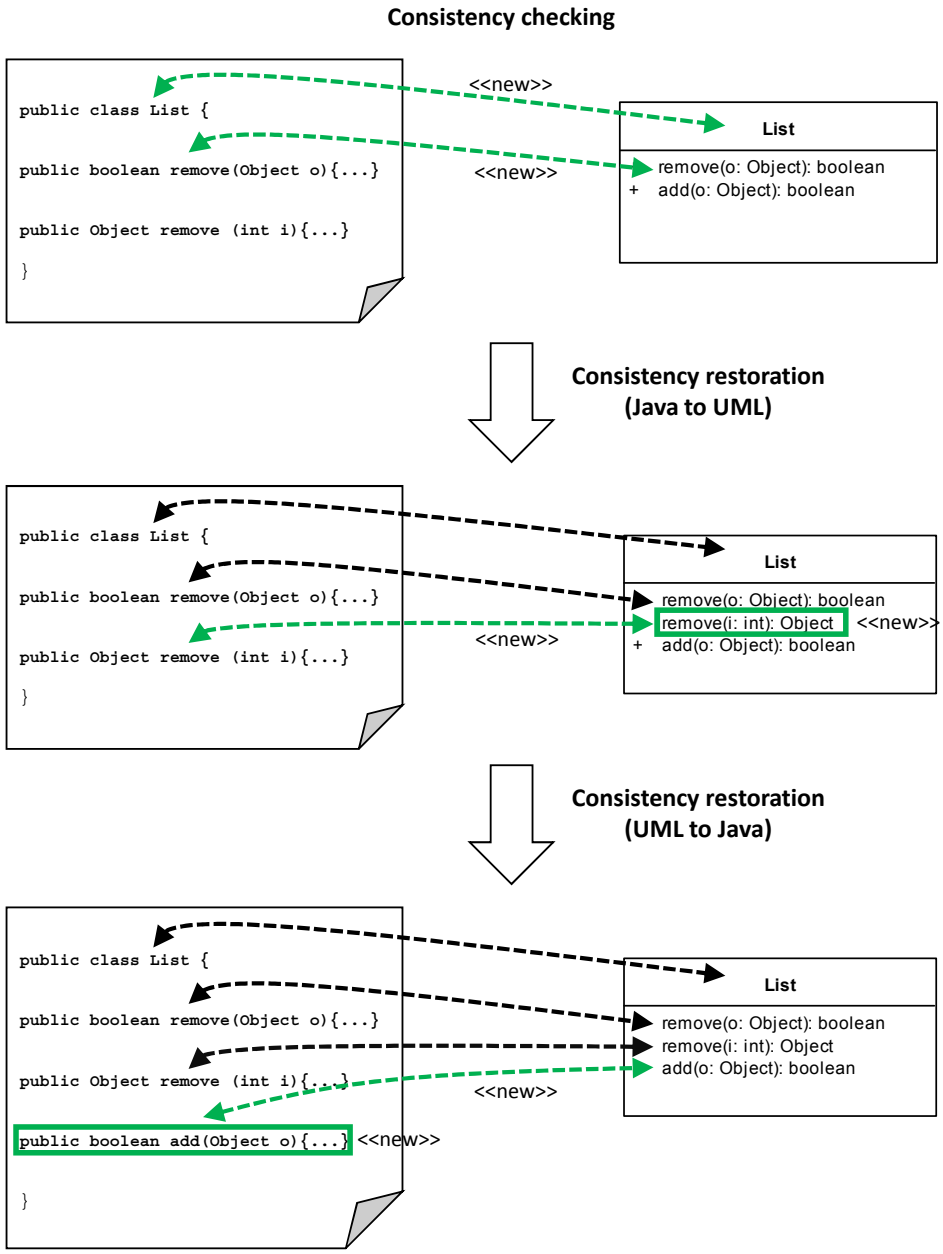


Figure 1.5: Consistency checking and restoration on the exemplary model pair

remaining Java elements to the UML model and vice versa. Note that, from a consistency restoration point of view, we consider the remaining elements as *additions* to the consistent portions of the models (detected in the initial consistency checking run) and propagate these additions to the other side. In the final state, both models are consistent and contain all three methods. This state of the models represents their v.2 when transferred to the BX conception in Figure 1.3.

Our understanding of consistency restoration, moreover, is not only limited to the remaining elements after a consistency checking run. We also handle user changes on the models in the same manner whereas user changes possibly do not only involve additions but also *deletions*. In Figure 1.6, a further run of consistency

restoration is demonstrated starting with the consistent state from Figure 1.5. This time, the user deletes one of the remove methods and adds a size method in the Java model. The propagation of these changes with consistency restoration accordingly deletes the corresponding remove method in the UML model (the deleted method and the correspondence are grayed-out). Furthermore, a size method is created in the UML model (together with its correspondence).

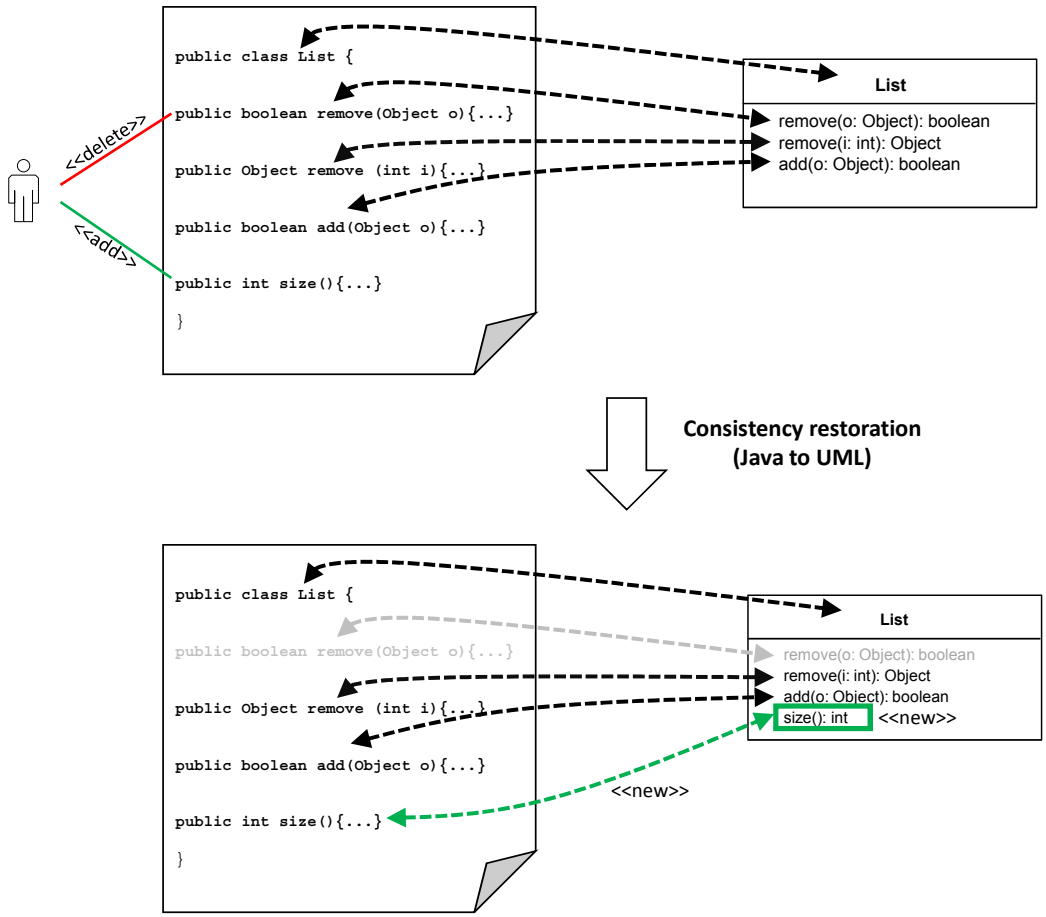


Figure 1.6: A further consistency restoration run upon user changes

To sum up, this thesis represents the following two building blocks for consistency management in MDE:

- Given two concurrently developed models (possibly without any prior consistency management support until the models reach an advanced state), a *consistency checking* run indicates which parts of the models are in conformance and which parts require action for consistency restoration. Furthermore, correspondences are created representing the relationships between individual elements of the two models.
- Given two consistent models (together with correspondences), a *consistency restoration* run propagates additions and/or deletions in one of the models to the other. Correspondences are updated in the process as well.

1.3 Stakeholders and Requirements

Throughout this thesis, consistency checking and restoration in an MDE context are discussed and formal as well as practical results based on TGGs are proposed. After having illustrated the targeted BX vision in an abstract manner (Figure 1.3) and via small concrete examples (Figure 1.5 and 1.6), we are now ready to identify stakeholders in BX and their requirements.

In Figure 1.7, the functional requirements are summarized via main use cases of different stakeholders. We distinguish between the following three different environments (and three groups of stakeholders):

- *Consistency tool*, used by *model owners* to check and restore consistency between their related models
- *Meta-tool*, i.e., a tool for building (consistency) tools, used by *consistency tool developers*
- *BX approach*, used by *meta-tool developers* (e.g., BX researchers), providing the foundations for the meta-tool.

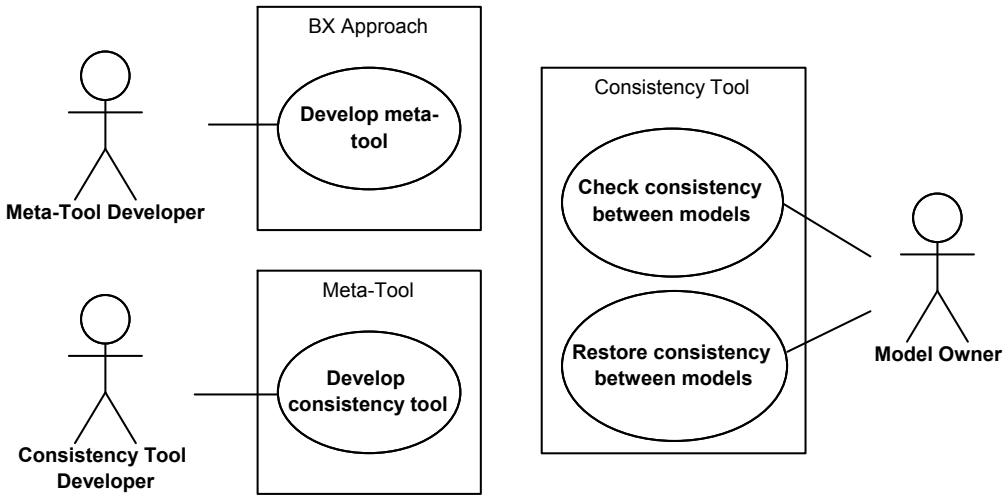


Figure 1.7: Stakeholders and their functional requirements in a BX landscape

Obviously, model owners are clients of consistency tool developers while consistency tool developers are clients of meta-tool developers. Although we consider three different environments with clear boundaries here, some of the stakeholder roles can possibly belong to the same person, e.g., when model owners develop a consistency tool for their own needs, or when meta-tool developers provide individual consistency tools using their own techniques. In all cases, solid theoretical foundations and available tool support for BX are important for all stakeholders, and define the goals of BX-related research as well as this thesis.

Besides the functional requirements, we demand the following non-functional requirements that are crucial for a scientific underpinning of a BX approach:

- *Scalability:* We have a pragmatic scalability demand from a tooling point of view and define it as the capability of dealing with industry-sized and re-

al-world scenarios of consistency checking and restoration. In this regard, the proposed (and implemented) procedures for consistency checking and restoration must consume acceptable runtime with typical hardware resources.

- *Formal properties:* A consistency checking and restoration approach must guarantee *correctness*. This means that, given a consistency specification and two models (that are not necessarily consistent), the outcome must be a consistent pair of models. While guaranteeing this for all kinds of consistency requirements cannot be feasible in general (e.g., due to scalability issues), the limitations and compromises for scalability must be defined clearly. Moreover, it must be possible to examine at specification time, whether a correct result can be guaranteed for a given consistency specification.

- *Expressiveness:* Consistency checking and restoration in BX relies on a consistency description written in the respective BX language. Hence, the BX language must be *expressive* enough to describe consistency for exactly those model pairs that are meant to be consistent. Otherwise, consistency tool developers must resort to hand-crafted pre- and post-processing of models to reduce the complexity of BX. This, however, distributes the focus of consistency tools over different components and reduces the scope of formal guarantees.

- *Usability:* The ease of use of a BX meta-tool must be provided in at least two different aspects: First, appropriate language editors must facilitate productivity when specifying consistency with a BX language. Second, execution of consistency checking and restoration based on a consistency specification must be possible with little effort (e.g., via included libraries with entry points or graphical user interface). Ease of execution has also an arguable impact on the usability of consistency tools developed with the meta-tool.

- *Validation:* The provided BX theory and tool support for consistency checking and restoration must be used and validated in different application scenarios. On the one hand, academic "toy examples" that are compact but exhibit non-trivial consistency challenges must be used to validate capabilities of a BX approach. On the other hand, this must be complemented via real-world and industrial case studies where involved models as well as consistency specifications are larger.

1.4 Contributions and the Structure of the Thesis

This thesis is mainly concerned with consistency checking and restoration based on TGGs and its respective tool support. We first discuss TGGs as a set of grammar rules that build up consistent pairs of models together with correspondences. The concrete contributions based on this formalism are:

- **Contribution I:** We formalize consistency checking with TGGs. We first *operationalize* TGG rules for consistency checking, i.e., consistency checking rules are derived that do not build up models but take existing ones as input and detect their consistent parts. Subsequently, we identify challenges with regard

to search space involved in using these rules to detect consistent parts, and incorporate *optimization techniques* into TGGs to tackle these challenges. Finally, we exploit these optimization techniques (i) to define sufficient and necessary conditions for consistency between two models and, in case of inconsistency, (ii) to detect the *largest* consistent portions of models.

- **Contribution II:** We formalize consistency restoration with TGGs that can operate upon consistency checking results. Analogously to consistency checking, we operationalize TGG rules to consistency restoration rules that take one of the models as input and update the other one consistently. We again *identify and clear search space problems* (resulting from wrong choices of rules that can lead consistency restoration to a dead end before consistency is restored). We discuss sufficient conditions (i) to avoid dead ends at rule application time and (ii) to state clearly under which circumstances a correct result of consistency restoration can be guaranteed.

- **Contribution III:** We consider the realization of consistency checking and restoration from a tool architecture point of view and provide procedures that operate with the aforementioned operational rules. Having a formalism based on graphs and graph patterns in case of TGGs, we propose to use *incremental pattern matching techniques* in these procedures. An incremental pattern matcher maintains and reports patterns in a host graph where our procedures are solely reactions to these reports. This yields a noticeable simplification for a new generation of TGG tools as additional analyses at runtime are outsourced to calculate necessary steps when checking and restoring consistency. Furthermore, this enables TGGs to profit directly from current and future progress in the scalability of incremental pattern matchers.

- **Contribution IV:** We present a *meta-tool* whose implementation is based on the formal results and tool architecture from the first three contributions. Besides investigating our running example, we report on another *industrial consistency project* in the context of computer-aided engineering where our meta-tool is used to develop a tool for consistency checking and restoration between technical drawings and their respective mechatronic simulations. We furthermore conduct experiments on different consistency scenarios to evaluate the scalability of our approach, and provide tool comparisons in some cases where solutions with other tools are available.

Table 1.1 relates these contributions to the functional and non-functional requirements. It can be observed that the contributions I-III are mainly foundational and address developing a meta-tool with formal properties in the first place. This is complemented with the provided meta-tool based on the results and its evaluation (contribution IV) allowing users to develop their own consistency tools with a validated and usable approach. In all of the contributions, consistency checking and/or restoration are of the uttermost importance. Finally, expressiveness of TGGs is out-of-scope for this thesis and is a current research topic with open questions concerning new language constructs and their operationalization.

	Functional Requirements				Non-functional Requirements				
	Check Cons.	Restore Cons.	Develop Cons. Tool	Develop Meta-tool	Scalability	Formal Properties	Expr.	Usability	Validation
Contr. I	X			X		X			
Contr. II		X		X		X			
Contr. III	X	X		X	X	X			
Contr. IV	X	X	X					X	X

Abbreviations: Contr. = Contribution | Cons. = Consistency | Expr. = Expressiveness

Table 1.1: Relation between contributions and requirements

Having so far introduced and motivated our BX vision consisting of consistency checking and restoration, the rest of the thesis is organized as follows:

- The following Section 2 provides the necessary formal background for TGGs to understand the subsequent sections. Furthermore, our running example is introduced focusing on the consistency between Java code and UML class diagrams.

The subsequent two sections form our formal contributions:

- Section 3 represents our consistency checking approach where we combine TGGs with linear optimization techniques. Our main formal results are Theorem 1 and Theorem 2 stating a sufficient condition or a sufficient and necessary condition, respectively, to conclude consistency of two models.
- Section 4 represents our consistency restoration approach where we this time combine TGGs with incremental pattern matching techniques. Our main formal results are given by Algorithm 2, which provides a consistency restoration procedure, and its formal guarantees including termination and correctness stated in Theorem 3 and Theorem 4, respectively.

Both Section 3 and Section 4, furthermore, conclude with their own discussions of related work. Our practical contributions are discussed subsequently:

- Section 5 represents our meta-tool implementing the formal results, reports on an industrial project where we have applied TGGs for consistency management in the computer-aided engineering domain, and, finally, provides an experimental and quantitative evaluation of our meta-tool.

Section 6, finally, concludes the thesis and discusses future work.

2

FUNDAMENTALS AND RUNNING EXAMPLE

This section introduces foundational aspects which are necessary to understand how models are structurally organized and discusses what consistency means in a graph grammar-based BX approach. We strive to impart intuitions with examples as well as formalizations to a sufficient extent. In our formal statements, we highly use the category theoretical foundations of graph grammars (in particular as introduced in [42]) and make our own adoptions and simplifications to capture all what we need on the way of formalizing consistency of models.

Before delving into the formal part of our fundamentals, it is important to understand what MDE (and in particular the MDA strategy) prescribes from a technical point of view to structure models. While models are the centre of attention in an MDE context, the structure of a model is defined by a *meta-model*. The prefix "meta" here (in analogy to its previous usage in meta-tool) is used for self-referencing of a term and thus leads us to the notion of a *model for models* in this concrete case. A meta-model basically resorts to basics of object-oriented analysis and defines concepts with their allowed relations to construct a valid model.

Considering our running example, a Java meta-model should adapt the Java language specification [75] to the technical space of MDE and define concepts such as classes, methods, parameters, and their relationships (e.g., classes can have methods and methods can have parameters). Analogously, a UML meta-model should do the same for UML models. In this thesis, we get our inspiration from the MDE tool MoDisco [25] for the Java meta-model and from the OMG standard [141] for the UML meta-model. We also introduce a correspondence metamodel between these two as our ultimate goal is to deal with consistency of two models together with their correspondences. Figure 2.1 depicts excerpts from the Java (left) and UML (right) meta-model connected by a correspondence meta-model (middle).

In the Java and UML meta-model in Figure 2.1, we only focus on classifiers (which can either be classes or interfaces) and their methods with parameters. While all of these concepts have a name property represented by a string value, parameters (in Java as well as in UML) additionally have a pos property represented by an integer value indicating the position of a parameter in the parameter list of the containing method. For brevity, we omit the typing information of methods and parameters. Concepts such as packages or class fields, furthermore, are omitted as well. The correspondence meta-model in the middle, moreover, maps the concepts from the Java and UML meta-model (C2C for classifier-to-classifier, M2M for method-to-method, and P2P for parameter-to-parameter correspondences).

While the Java and UML meta-model seem identical on the depicted concepts, they differ in representing inheritance relations between classifiers: In the Java

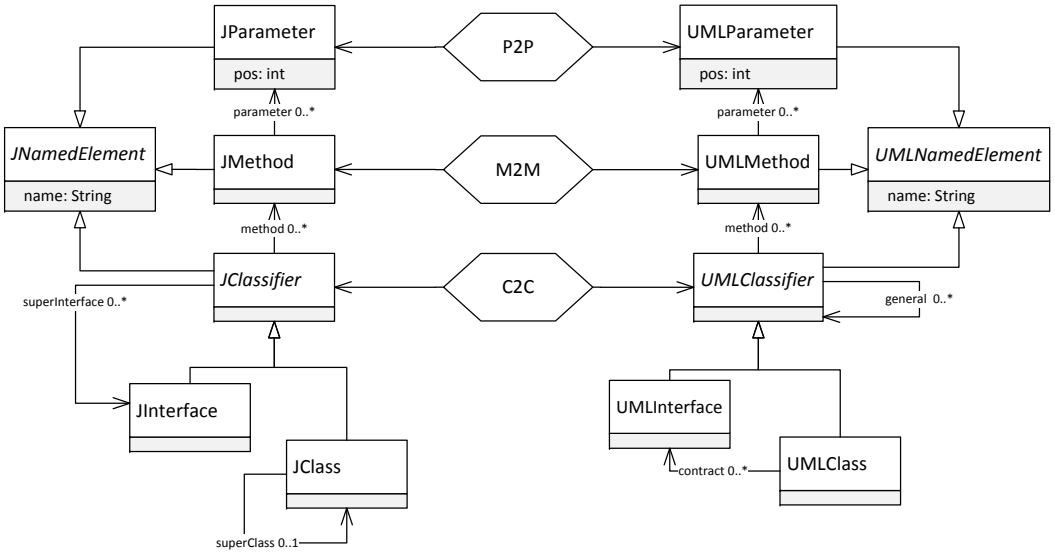


Figure 2.1: Excerpts from the Java metamodel (left), the UML metamodel (right), and the correspondence metamodel (middle)

meta-model, a class may have at most one super class (note the superClass reference with 0..1 multiplicity) while classes as well as interfaces may have an arbitrary number of super interfaces (note the superInterface reference with 0..* multiplicity). This reflects the restrictions in Java with respect to multiple inheritance (inheriting from multiple classes is not allowed). The UML metamodel, on the contrary, is more relaxed regarding inheritance relations and allows arbitrarily many inheritance relations between classifiers (be it a class or an interface) via the general reference with 0..* multiplicity. A second kind of inheritance relation from a class to an interface is represented by the contract reference (again with 0..* multiplicity).

In general, meta-models focus on defining concepts for models but not how these concepts are encoded in an actual representation of models. As already shown in Figure 1.4 and 1.5, Java models are encoded in the textual Java syntax while UML models have a visual notation. This is referred to as *concrete syntax* while the entirety of concepts defined in a meta-model describes an *abstract syntax* (which "abstracts" from the concrete representation). Abstract syntax, moreover, is generally visualized in the well-known notation of object diagrams [141]. As from now on, we normalize most of the future figures using abstract syntax when depicting models (and go back to the concrete syntax in certain cases where this increases the readability of illustrations). Figure 2.2 depicts an excerpt of the consistent model pair in Figure 1.5 and their correspondences in the abstract syntax. The Java model (left), the UML model (right) as well as the correspondence model (middle) instantiate the respective meta-models in Figure 2.1.

In the rest of this section, we formalize triples of (meta-)models as triples of graphs and subsequently define consistency as a grammar over such triples. We first focus on rather simple examples and then extend our consistency specification. Finally, we summarize this section and discuss open issues as well as possible extensions for future with regard to the consistency specification.

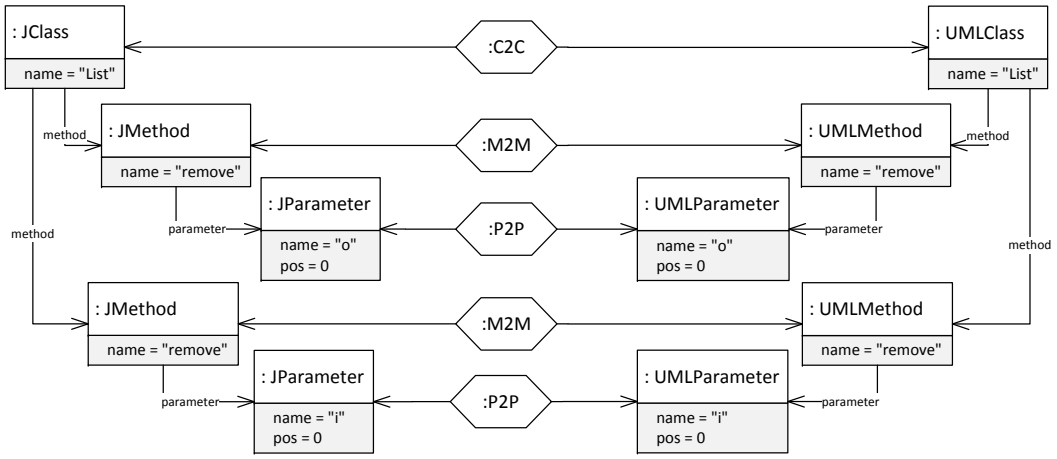


Figure 2.2: A triple of Java model (left), UML model (right), and correspondence model (middle) conforming to the triple of meta-models in Figure 2.1

2.1 Graphs and Triple Graphs

The following definitions and statements are adopted from the category theoretical foundations of graph grammars [42]. To formalize (triples of) models and their consistency, we shall start with defining the most basic mathematical structure that comprises "objects" and "mappings" (*morphisms*) between them, namely a *category*.

Definition 1 (Category)**.** [42]
A *category* $\mathbf{C} = (Ob_{\mathbf{C}}, Mor_{\mathbf{C}}, \circ)$ consists of:

- a class $Ob_{\mathbf{C}}$ of *objects*
- for each pair of objects $X, Y \in Ob_{\mathbf{C}}$, a set $Mor_{\mathbf{C}}(X, Y)$ of morphisms, where each $m \in Mor_{\mathbf{C}}(X, Y)$ is denoted as $m : X \to Y$
- for all objects $X, Y, Z \in Ob_{\mathbf{C}}$, a composition operation $\circ : Mor_{\mathbf{C}}(Y, Z) \times Mor_{\mathbf{C}}(X, Y) \to Mor_{\mathbf{C}}(X, Z)$

such that the following properties are fulfilled:

1. Associativity.
   For all objects $X, Y, Z, W \in Ob_{\mathbf{C}}$, and all morphisms $f : X \to Y$, $g : Y \to Z$, $h : Z \to W$, it holds that $(h \circ g) \circ f = h \circ (g \circ f)$
2. Identity.
   For all objects $X, Y \in Ob_{\mathbf{C}}$ and morphisms $f : X \to Y$, there exists a morphism $id_X : X \to X$ and a morphism $id_Y : Y \to Y$ such that $f \circ id_X = f$ and $id_Y \circ f = f$.

Hence, a category is an abstract representation of "structured things" and their morphisms that serve as mappings. This applies independently of the concrete structure of the objects and forms a means for using and stating mathematical results for different kinds of objects in a common way.

Throughout this thesis, the category **Sets** whose objects are sets and whose morphisms are total functions will be the most basic category that shapes the underlying structure for further categories. When considering sets as categorical objects, the probably most noticeable difference to the usual definition of sets is how functions are defined (cf. Example 2 in [118]): In a categorical setting, the output set of a function is not defined by its range (i.e., not by the elements that are a value for some elements in the input set). Instead, a function is a mapping from an input set to an output set (and the range of the function does not necessarily cover all elements in the output set).

Definition 2 (**Sets**).
Sets = $(Ob_{\mathbf{Sets}}, Mor_{\mathbf{Sets}}, \circ)$ is defined as:

- a class $Ob_{\mathbf{Sets}}$ of finite sets
- total functions $Mor_{\mathbf{Sets}}$
- for all sets $X, Y, Z \in Ob_{\mathbf{Sets}}$, the function composition operation $\circ : Mor_{\mathbf{Sets}}(Y, Z) \times Mor_{\mathbf{Sets}}(X, Y) \to Mor_{\mathbf{Sets}}(X, Z)$.

Fact 1 (The category **Sets**).
Sets forms a category where the associativity property is induced by the associativity of function composition, and identities are the identity functions.

As the definition of a category (Definition 1) and its instantiation (Definition 2) might imply, categories focus on morphisms and their properties rather than on objects. We shall exploit this "mapping-oriented" flavor of categories in at least two critical ways in this thesis: First, if our goal is to set two models in relation for consistency purposes, relations can be captured via morphisms. As exemplified in Figure 2.1 on the meta-model level and in Figure 2.2 on the model level, correspondence structures set the other two structures in relation in our case. Second, existence or absence of morphisms allows us to draw conclusions about the consistency of models (indicating what actions to take when dealing with consistency).

Furthermore, the following types of morphisms are of special interest:

Definition 3 (Monomorphism, Epimorphism, Isomorphism). [42]
Given a category **C**,

1. A morphism $m : Y \to Z \in Mor_{\mathbf{C}}$ is called a *monomorphism*, if for all morphisms $f, g : X \to Y \in Mor_{\mathbf{C}}$, it holds that $m \circ f = m \circ g$ implies $f = g$
2. A morphism $e : X \to Y \in Mor_{\mathbf{C}}$ is called an *epimorphism*, if for all morphisms $f, g : Y \to Z \in Mor_{\mathbf{C}}$, it holds that $f \circ e = g \circ e$ implies $f = g$
3. A morphism $i : X \to Y \in Mor_{\mathbf{C}}$ is called an *isomorphism* if there exists a morphism $i^{-1} : Y \to X$ such that $i \circ i^{-1} = id_Y$ and $i^{-1} \circ i = id_X$.

Example 1. An intuition in **Sets** already suffices in this thesis to understand these special types of morphisms. In **Sets**, injective functions are monomorphisms. As an injective function m maps all elements in its input set distinctly to the elements of the output set, $m \circ f = m \circ g$ can only hold if $f = g$. Furthermore, surjective functions are epimorphisms. As a surjective function e covers all elements in the output set, $f \circ e = g \circ e$ can only hold if $f = g$. Bijective functions, finally, are isomorphisms.

Remark 1. In **Sets**, bijective functions are those functions which are injective and surjective. Hence, a morphism which is a monomorphism as well as an epimorphism is an isomorphism. This, however, does not necessarily hold in the most general setting of the category theory as i^{-1} might not exist for a morphism i in a category (cf. Remark 2.14 in [42]). Hence, we do not state in Definition 3 isomorphism as being monomorphism and epimorphism at the same time. As our structures shall entirely be derived from **Sets** throughout this thesis, nevertheless, this intuition is indeed valid for our purposes.

In our context, graphs are the fundamental structures representing (meta-)models. A graph is typically defined as a quadruple (E, V, s, t) consisting of

- a set E of *edges*,
- a set V of *vertices*,
- a function $s : E \to V$ assigning each edge a *source* vertex,
- a function $t : E \to V$ assigning each edge a *target* vertex.

While this is the most common definition of a graph that can be found in the literature, an alternative definition is to define graphs as *functors*. A functor is basically a mapping between two categories remaining compatible with compositions and identities. Functors allow us to build new (and, roughly spoken, more complex) categories from existing ones, e.g., graphs from sets and triple graphs from graphs. To capture all relevant structures in a unified manner, we use functors consequently from the very beginning (starting with the definition of a graph) and exemplify them for the reader who is not familiar with this concept.

Definition 4 (Functor)**.** [42]
Let $\mathbf{C}$ and $\mathbf{D}$ be two categories. A *functor* $F : \mathbf{C} \to \mathbf{D}$ is given by two mappings $F_{Ob} : Ob_{\mathbf{C}} \to Ob_{\mathbf{D}}$ and $F_{Mor} : Mor_{\mathbf{C}}(X, Y) \to Mor_{\mathbf{D}}(F_{Ob}(X), F_{Ob}(Y))$ with $X, Y \in Ob_{\mathbf{C}}$, such that the following properties are fulfilled:

1. For all morphisms $f : X \to Y$ and $g : Y \to Z \in Mor_{\mathbf{C}}$, it holds that $F_{Mor}(g \circ f) = F_{Mor}(g) \circ F_{Mor}(f)$
2. For each object $X \in Ob_{\mathbf{C}}$, it holds that $F_{Mor}(id_X) = id_{F_{Ob}(X)}$.

Example 2. The classical definition of a graph, i.e., the quadruple (E, V, s, t) mentioned above, states nothing but two sets and two total functions in the same direction between these sets. Such a structure can be constructed as a functor which maps a "small" category $O_1 \rightrightarrows O_2$, i.e., a category consisting of the two objects O_1 and O_2 and two morphisms from O_1 to O_2, to the category **Sets**. The result of this mapping consists of two sets and two functions in the same direction exemplified in Figure 2.3 using a Java model. At the top, the small category $O_1 \rightrightarrows O_2$ is depicted, and at the bottom two sets (E and V) with two functions ($s, t : E \to V$). The functor (represented with bold and gray arrows) maps O_1 and O_2 to E and V, respectively, while the two morphisms $O_1 \rightrightarrows O_2$ are mapped to the two functions s and t. Though not depicted explicitly, the functor furthermore maps the id morphisms id_{O_1} and id_{O_2} to the id functions id_E and id_V, respectively, fulfilling thus the two conditions stated over morphisms in Definition 4.

The set E consists of two elements, namely the method and parameter edges, where the set V consists of three elements representing a class List, a method remove, and a parameter o. The function s (represented with dashed arrows) assigns the class List to the method edge and the method remove to the parameter edge as their sources. Furthermore, the function t (represented with dotted arrows) assigns the method remove to the method edge and the parameter o to the parameter edge as their targets.

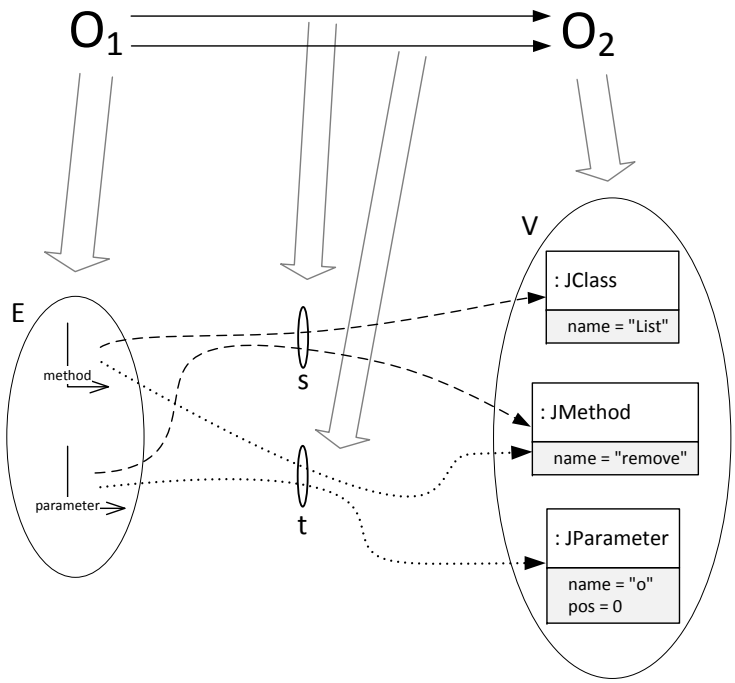


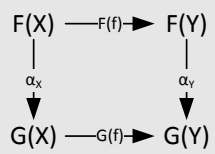
Figure 2.3: A graph considered as a functor from $O_1 \rightrightarrows O_2$ to **Sets**

To sum up, a functor $(O_1 \rightrightarrows O_2) \to$ **Sets** captures all what a quadruple (E, V, s, t) with the required properties stated before intends to describe for defining a graph.

While a functor is one single mapping between two categories (that leads to, e.g., one single graph as depicted in Figure 2.3), all functors together with their

compatible morphisms yield a *functor category*. This leads us to the category **Graphs** and subsequently **TripleGraphs**.

Definition 5 (Functor Category)**.** [42]
Given two functors $F, G : \mathbf{C} \to \mathbf{D}$, a *natural transformation* $\alpha : F \Rightarrow G$ is a family of morphisms $\alpha = (\alpha_X)_{X \in Ob_\mathbf{C}}$ with $\alpha_X : F(X) \to G(X) \in Mor_\mathbf{D}$, such that for all morphisms $f : X \to Y \in Mor_\mathbf{C}$ the diagram to the right commutes, i.e., $\alpha_Y \circ F(f) = G(f) \circ \alpha_X$.



A *functor category* **[C,D]** is given by all functors $F : \mathbf{C} \to \mathbf{D}$ as the objects and by the natural transformations as the morphisms. Composition of two natural transformations $\alpha : F \Rightarrow G$ and $\beta : G \Rightarrow H$ is the componentwise composition in $\mathbf{D}$, i.e., $\beta \circ \alpha = (\beta_X \circ \alpha_X)_{X \in Ob_\mathbf{C}}$. Identities are identical natural transformations given by componentwise identities in $\mathbf{D}$.

Definition 6 (The Category **Graphs**)**.**
The category **Graphs** of graphs is the functor category $[O_1 \rightrightarrows O_2, \mathbf{Sets}]$. Given a graph G, the sets $G(E)$ and $G(V)$ are referred to as *edges* and *vertices* of G, respectively. We refer to the union set $G(E) \cup G(V)$ as *elements* of G, denoted as elements(G).

Example 3. Each of the Java, UML, and correspondence meta-models in Figure 2.1 and models in Figure 2.2 represents an individual graph. The concepts (e.g., JClass and JMethod) and their references (e.g., the method reference) in the meta-models as well as their instantiations in the models represent vertices and edges, respectively. Note that we further on use the visualization in these figures (referred to as abstract syntax at the beginning of this section) to represent graphs and do not explicitly depict the small category $O_1 \rightrightarrows O_2$ or its mappings induced by the functor (which we have done once in Figure 2.3 to understand functors).

Additionally, Figure 2.4 shows two exemplary morphisms α and β in **Graphs**, depicted via dashed arrows between their respectively mapped graphs. Both α and β are natural transformations, i.e., a family of morphisms in **Sets** (in particular a pair of morphisms in **Sets** as the small category $O_1 \rightrightarrows O_2$ consists of two objects). Considering the individual components of α and β according to Definition 5, α_{O_1} and β_{O_1} refer to those parts of morphisms that map edges while α_{O_2} and β_{O_2} map vertices. Most importantly, a morphism in **Graphs** commutes with the s and t functions of the individual graphs. For example, when α_{O_1} (or β_{O_1}) maps an edge of a graph to an edge of another graph, the source and target vertices of these edges are also mapped accordingly by α_{O_2} (or β_{O_2}). Note that this is inherently required in Definition 5 which states that natural transformations commute with the morphisms in the "host" category (**Sets** in the case of **Graphs**). Furthermore, the composition of natural

transformations as stated in Definition 5 can be exemplified by $\beta \circ \alpha$ which is given by the componentwise compositions $\beta_{O_1} \circ \alpha_{O_1}$ and $\beta_{O_2} \circ \alpha_{O_2}$.

Finally, α and β are monomorphisms as they map vertices and edges injectively while β, being surjective at the same time, is additionally an isomorphism.

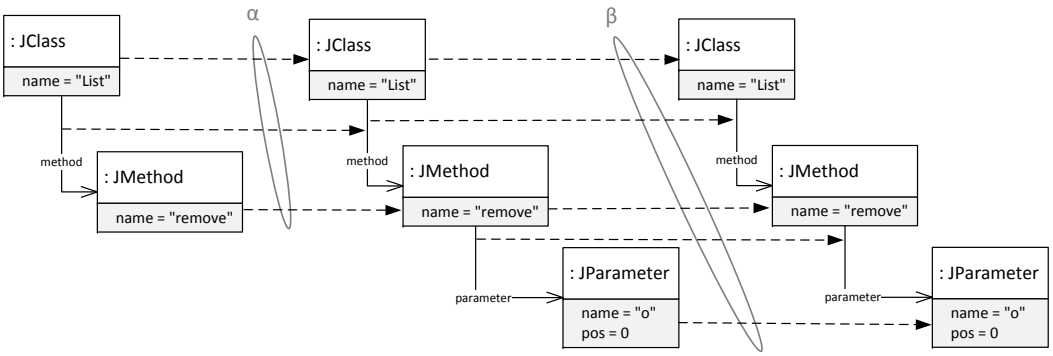


Figure 2.4: Two morphisms in **Graphs**

Remark 2. For brevity, we restrict our formalization to graphs with vertices and edges but without attributes (which are actually used on the meta-model and model level in Figure 2.1 and 2.2, respectively) and vertex type inheritance (used on the meta-model level in Figure 2.1). Although we implicitly use attributes and inheritance in our examples, formalizations reduced to vertices and edges suffice to represent the contributions of this thesis with respect to consistency checking and restoration (and help us to avoid notational inflation). The formalization, nevertheless, can compatibly be extended to *attributed graphs with vertex type inheritance* by constructing new kinds of categories which still retain all desired properties. We refer the interested reader to [42] for the respective extensions in the general theory of graph grammars, and to [8, 92] for attribute handling dedicated to the case of TGGs. While there is no doubt on the solid formal foundation of these extensions, an interesting discussion that also merits mentioning here is that of "the awkwardness of attributes" [124] as manipulating attributes requires different techniques than manipulating vertices and edges. To this end, there have been recent efforts to come up with more lightweight approaches to integrating attributes into graph grammars [73].

Analogously to graphs, *triple graphs* are also constructed as functors which map a small category $S \leftarrow C \rightarrow T$ to **Graphs** (the letters S, C, and T indicating the source, correspondence, and target domain, respectively).

Definition 7 (The Category **TripleGraphs**).
The category **TripleGraphs** of triple graphs is the functor category $[S \leftarrow C \rightarrow T, \mathbf{Graphs}]$. Given a triple graph G, the graphs $G(O_S)$, $G(O_C)$, $G(O_T)$ are referred to as the *source graph*, the *correspondence graph*, and the *target graph* of G, respectively. G is denoted by $G_S \leftarrow G_C \rightarrow G_T$ where $G_X = G(X), X \in \{S, C, T\}$.

Example 4. The triple of meta-models in Figure 2.1 and the triple of models in Figure 2.2 are exemplary triple graphs where the individual components (Java, UML, and correspondences) represent single graphs. The connections from correspondences to the other two graphs are captured as morphisms in **Graphs** as Definition 7 implies. We further on depict these morphisms via solid (and horizontal) lines with open arrows.

In Figure 2.5, moreover, a morphism (in fact, a monomorphism) α in **TripleGraphs** is depicted via vertical dashed arrows. Basically, α is a natural transformation consisting of three morphisms α_S, α_C, and α_T in **Graphs** (represented in Figure 2.5 by the arrows placed at the left, middle, and right part, respectively). Most importantly, as a result of Definition 5 and as is the case in Figure 2.5, α_S, α_C, and α_T commute with the morphisms from the correspondence graphs to the source and target graphs of the respective triple graphs. That is, for each correspondence mapped by α_C, its source and target vertex are mapped accordingly by α_S and α_T, respectively.

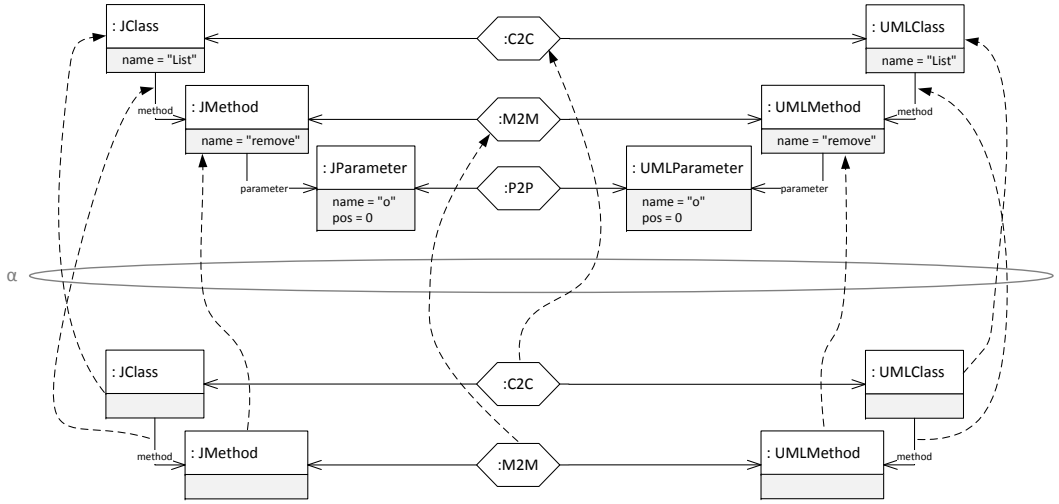


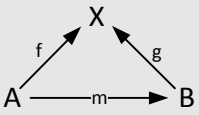
Figure 2.5: A morphism in **TripleGraphs**

While meta-models and models are graphs, meta-models are in fact "distinguished" graphs that represent *typing* information for their instantiating models. This also applies analogously to triples. For consistency checking and restoration purposes, we only deal with triples of models whose types conform to a given triple of meta-models (e.g., the one in Figure 2.1 for our running example). Typing, moreover, is captured as a morphism, commonly referred to as a *type morphism*. Intuitively, a type morphism maps vertices and edges in models to vertices and edges in meta-models and induces this way a typing information.

In a categorical setting, typing can be captured via *slice categories*, i.e., a "slice" of a category that solely consists of objects that have a type morphism to a distinguished object and morphisms that are compatible with these type morphisms. In our concrete case, triple graphs representing Java and UML models with correspondences (i.e., triples of models typed over the triple of meta-models given in Figure 2.1) form a "slice" of **TripleGraphs**.

Definition 8 (Slice Category)**.** [42]
Let $\mathbf{C}$ be a category and X an object in $\mathbf{C}$. The slice category $\mathbf{C}_X$ is defined as follows:

1. $Ob_{\mathbf{C}_X} = \{f : A \to X \mid f \in Mor_{\mathbf{C}}\}$
2. $Mor_{\mathbf{C}_X}(f : A \to X, g : B \to X) = \{m : A \to B \mid g \circ m = f\}$ (i.e., the diagram to the right commutes for each $m \in Morc_{\mathbf{C}_X}$)



3. composition operation $\circ$ as defined in $\mathbf{C}$
4. $id_{f:A\to X} = id_A \in Mor_{\mathbf{C}}$.

Example 5. Although we ultimately deal with a slice of **TripleGraphs** throughout this thesis, we first exemplify objects and morphisms from a slice of **Graphs** to impart an intuition for Definiton 8.

At the top of Figure 2.6, an excerpt from the Java meta-model is given which is now considered to be a distinguished graph and thus represents the object X in Definition 8. Note that, as already stated in Remark 2, we provide our formalization without vertex type inheritance on the meta-model level. Hence, the inheritance hierarchy shown in Figure 2.1 is now "flattened" from a formal point of view and represented this way in our excerpt in Figure 2.6 (e.g., JClass has now a direct edge to JMethod which was initially inherited from JClassifier in Figure 2.1).

At the bottom of Figure 2.6, two graphs and a morphism are given which actually repeat Figure 2.4. In addition to Figure 2.4, the vertices and edges in both graphs are now mapped to the distinguished graph by the type morphisms (depicted via vertical and grayed out arrows). In all cases, the type morphisms commute with the morphism between these graphs (horizontal dashed arrows) and induce typing information for the individual vertices and edges. The entirety of such graphs (for which the type morphisms are given) and their morphisms (which commute with the type morphisms) forms a slice of **Graphs**. In the case of Figure 2.6, this slice represents models typed over the Java meta-model (and excludes all other graphs representing other models).

Example 5 discusses a slice of **Graphs** that leads us to the notion of *typed graphs*. This rather serves as an introductory example to understand slice categories. As mentioned before, we deal with a slice of **TripleGraphs** in our context. Hence, although exemplified, we skip the definition of typed graphs and apply the concept of slice categories directly to **TripleGraphs** in the following definition. Lifting Example 5 to triples is left to the reader as exercise.

Definition 9 (Typed Triple Graph)**.**
Let TG be a distinguished triple graph, referred to as *type triple graph*. The category of *triple graphs typed over* TG is the slice category $\mathbf{TripleGraphs}_{TG}$.

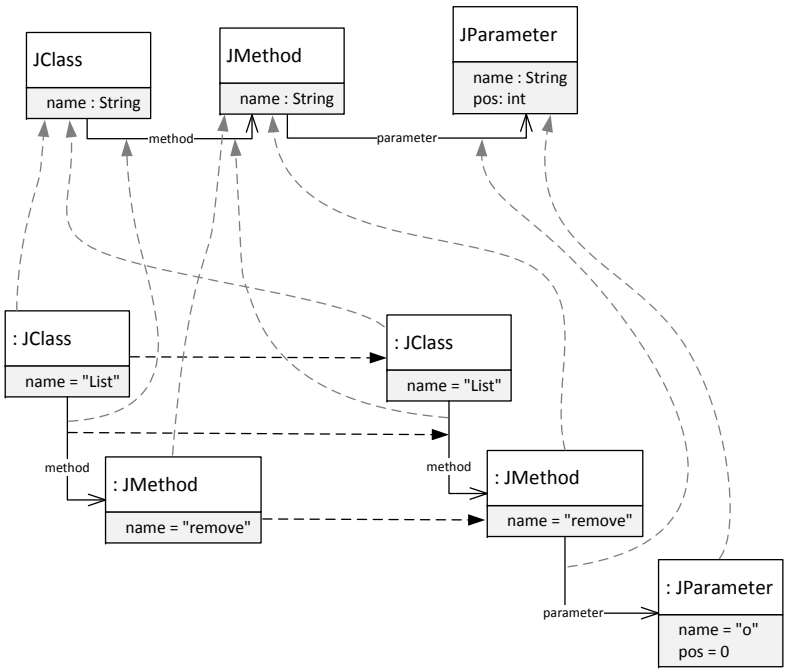


Figure 2.6: A distinguished graph (top) and two graphs together with a morphism from a slice of **Graphs** derived via this distinguished graph (bottom)

Example 6. The triple of meta-models in Figure 2.1 is the type triple graph TG for our running example, and the triple of models in Figure 2.2 represents a triple graph typed over TG. We further on use the visual notation of Figure 2.2 and do not explicitly depict TG or type morphisms when discussing triple graphs typed over TG. Type information, nevertheless, is explicitly given via the labels on the vertices and edges (e.g., :JClass for a vertex which is mapped to the JClass vertex in the Java meta-model).

Remark 3. In our definitions and statements from now on, we assume a type triple graph TG is always given (and thus do not require it explicitly). In the absence of an explicit TG symbol, the term triple graph refers to a triple graph typed over TG, and **TripleGraphs** denotes the slice category $\mathbf{TripleGraphs}_{TG}$.

2.2 Triple Graph Grammars (TGGs)

The term *grammar* has its roots in linguistics where it refers to a set of *rules* that describe how to construct valid words, phrases, and sentences in a natural language. In the 1950s, formal grammars have been introduced (e.g., [30, 31]) as an approach to studying the structure of a language. It then came as no surprise that formal grammars became highly relevant for "computational" linguistics, e.g., describing what is syntactically possible in a programming language by using *string grammars*. In the 1970s, *graph grammars* have followed evolving into a community with dedicated scientific events [33] as graphs are (just like strings) ubiquitous in computer

science to represent data. Around that time, *pair grammars* [119] have shown that grammars are also useful to describe consistency between two structures.

The main idea of a pair grammar is to provide a set of rule pairs where each rule pair constructs the two structures simultaneously. Though only illustrated on string-to-graph translators, possible use cases include graph-to-string, string--to-string, and graph-to-graph translators. This is the source of inspiration for TGGs, introduced in the 1990s in [128], where pairs are lifted to triples by introducing correspondences in the middle. All in all, a TGG describes how triple graphs are constructed.

If a grammar is a set of rules, our first step is to define what is a rule in the context of TGGs. A rule describes how a triple graph can be extended to another triple graph by creating new vertices and edges on the individual graphs of the triple. Although it might seem to be a severe limitation at a first glance that the rules can only create something but never delete (which is generally allowed in graph grammars), it should be noted that our goal is not to capture all possible modifications to triple graphs but only to describe which triple graphs are consistent. Indeed, a consistency restorer derived from a TGG is able to handle creations as well as deletions when propagating changes as we shall see later (though the consistency description itself is only constructive).

Definition 10 (Rule)**.**
A *TGG rule*, short *rule*, is a monomorphism $r : L \to R$ in **TripleGraphs**. The triple graphs L and R are referred to as the *left-hand side* and the *right-hand side*, respectively, of r.

Example 7. Figure 2.7 depicts a set of rules that construct triple graphs representing Java and UML models together with their correspondences.

As a rule, being a monomorphism, "embeds" its left-hand side L injectively into its right-hand side R, we use a compact syntax merging both L and R in the same diagram. Elements (vertices and edges) that are both in L and R are depicted black. These elements represent the triple graph which is to be extended and are referred to as *context* elements required by the rule. Elements that are in R but not in L are depicted green and additionally with a (++)-markup for monochrome printing. These elements represent the extensions intended by the rule and are referred to as *created* elements. We also use *attribute constraints* in some rules (within a note icon) that require solely name equality between Java and UML elements (and position equality between parameters) that are created together. Our rules have the following purposes:

- ClassRule (r_1) does not require any context elements and creates a pair of Java and UML classes together with a correspondence in the middle. The names of the Java and UML class, furthermore, must be equal.
- InterfaceRule (r_2) is similar to the previous rule and creates a corresponding pair of Java and UML interfaces without requiring a context.

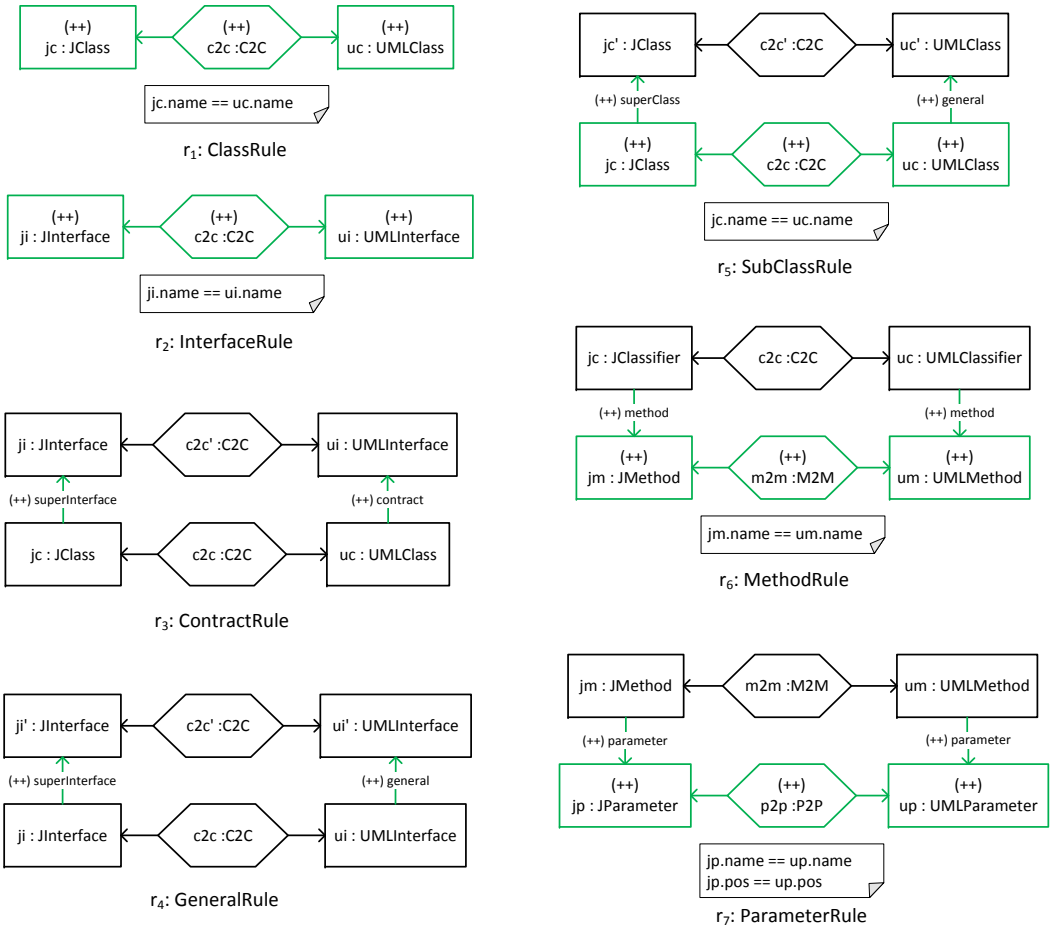


Figure 2.7: TGG rules for our running example

- ContractRule (r_3) requires a corresponding pair of Java and UML classes as well as a corresponding pair of Java and UML interfaces as context (which, e.g., can be created by the previous two rules), and creates a superInterface edge from the Java class to the Java interface as well as a contract edge from the UML class to the UML interface.

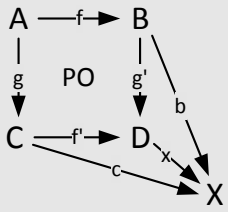
- GeneralRule (r_4) requires two corresponding pairs of Java and UML interfaces as context, and creates a superInterface edge on the Java side as well as a general edge on the UML side. Both edges are created in the same direction.

- SubClassRule (r_5) requires a corresponding pair of Java and UML classes, and creates a corresponding pair of subclasses with equal names. Note that the rule crates a Java class together with its superClass edge. As we do not have any other rule creating a superClass edge for an existing Java class, it is ensured that a Java class has either one super class (when created with this rule) or no super class (when created with ClassRule) but never multiple super classes which is forbidden in Java.

- MethodRule (r_6) requires a corresponding pair of Java and UML classifiers (which can be classes or interfaces) as context and creates a corresponding pair of Java and UML methods with equal names.
- ParameterRule (r_7), finally, requires a corresponding pair of Java and UML methods as context and creates a corresponding pair of Java and UML parameters with equal names and positions.

As the reader might have noted, our definition of a rule (Definiton 10) is nothing but a monomorphism in **TripleGraphs** but we discuss them with the intention of creating vertices and edges. In fact, creations are realized by *applying* a rule to a given triple graph. A rule application is basically gluing the right-hand side of a rule with the given triple graph along the left-hand side of the rule. Constructions over gluings is generalized as a *pushout* in the sense of category theory.

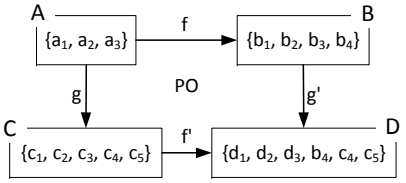
Definition 11 (Pushout)**.**
Given morphisms $f : A \to B$ and $g : A \to C$ in a category **C**, a pushout is defined by a pushout object $D \in Ob_{\mathbf{C}}$ and morphisms $f' : C \to D$ and $g' : B \to D$ with $f' \circ g = g' \circ f$, such that the following property, referred to as the *universal property*, is fulfilled: For all morphisms $b : B \to X$ and $c : C \to X$, there is a unique morphism such that $x \circ g' = b$ and $x \circ f' = c$.



Future diagrams representing a pushout are labeled with PO (as is the case in Definition 11).

Example 8. As the category **Sets** is the underlying category for **Graphs** (and consequently **TripleGraphs**), we again start with an intuition on the level of sets to understand how a pushout is constructed in our future examples. The pushout object over two morphisms $f : A \to B$ and $g : A \to C$ in **Sets** (i.e., two total functions) can be constructed as the quotient $B \uplus C|_{\equiv}$ where $\uplus$ denotes disjoint union of sets and $\equiv$ is the smallest equivalence relation such that $\forall a \in A$, $(f(a), g(a)) \in \equiv$.

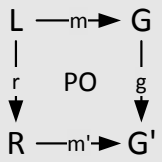
In the exemplary diagram below, all functions map the elements in the input set to the elements with the same index in the output set, e.g., $f(a_1) = b_1$ and $g(a_1) = c_1$. The pushout object, finally, is the set of equivalence classes with respect to $\equiv$. Equivalence classes that contain two elements (i.e., $\{b_1, c_1\}$, $\{b_2, c_2\}$, and $\{b_3, c_3\}$) are collapsed to a single representative (i.e., to d_1, d_2, and d_3, respectively) while all other equivalence classes (i.e., $\{b_4\}, \{c_4\}$, and $\{c_5\}$) are represented with the symbol of their single element.



Pushouts in **Graphs** and consequently **TripleGraphs** are induced by componentwise pushouts in **Sets**. We refer to [42] (Fact 2.17 for graphs and Fact A.37 for functor categories in general) for a detailed proof of the following fact.

Fact 2 (Pushouts in **TripleGraphs**)**.**
Pushouts exist in **Graphs** and **TripleGraphs**. In **Graphs**, pushouts can be constructed componentwise for vertices and edges in **Sets**, where pushouts in **TripleGraphs** can be constructed componentwise for the source, correspondence, and target graph in **Graphs**.

Definition 12 (Rule Application)**.**
A *rule application* with a rule $r : L \to R$ and a triple graph G over a monomorphism $m : L \to G$, denoted as $G \overset{r@m}{\Longrightarrow} G'$, is a pushout as depicted in the diagram to the right. For a rule application, we refer to m and m' as *match* and *comatch*, respectively. A sequence $d : G_1 \overset{r_1@m_1}{\Longrightarrow} G_2 \overset{r_2@m_2}{\Longrightarrow} \ldots \overset{r_n@m_n}{\Longrightarrow} G_n$ of rule applications is referred to as a *derivation*.



Example 9. Figure 2.8 depicts a derivation consisting of two rule applications (represented as two pushout diagrams) with two of our exemplary rules from Figure 2.7. The vertical arrows on the left side represent the two rules, namely ClassRule (r_1) and MethodRule (r_6). Note that, for the first time, we explicitly show the rules as a morphism and do not use the compact syntax with (++)-markup. Accordingly, L_1 (R_1) and L_6 (R_6) refer to the left-hand side (right--hand side) of r_1 and r_6, respectively. The matches and comatches of the single rule applications, furthermore, are represented as m and m', respectively.

We start with the *empty* triple graph (G_0) and find a match of L_1 which is also an empty triple graph, and create a pair of Java and UML classes by gluing R_1 over the empty triple graph, resulting in the pushout object G_1. We then find a match of L_6 in G_1, glue R_6 over this match resulting in a corresponding pair of Java and UML methods in the pushout object G_2.

Finally, a TGG is a set of rules and its *language* consists of triple graphs that can be constructed via derivations with these rules starting with the empty triple graph (as illustrated, e.g., in Figure 2.8).

Definition 13 (Triple Graph Grammar)**.**
A *triple graph grammar*, denoted and abbreviated as TGG, is a set of rules.
The *language* $\mathcal{L}(TGG)$ of a TGG is defined as:
$\mathcal{L}(TGG) = \{G_\emptyset\} \cup \{G \mid \exists\, d : G_\emptyset \overset{r_1@m_1}{\Longrightarrow} G_1 \ldots \overset{r_n@m_n}{\Longrightarrow} G_n \text{ with } r_1, \ldots, r_n \in TGG\}$
where $G_\emptyset = \emptyset \leftarrow \emptyset \rightarrow \emptyset$ is the empty triple graph.
The *source language* $\mathcal{L}_S(TGG)$ and the *target language* $\mathcal{L}_T(TGG)$ are defined as:
$\mathcal{L}_S(TGG) = \{G_S \mid \exists\, G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)\}$
$\mathcal{L}_T(TGG) = \{G_T \mid \exists\, G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)\}$.

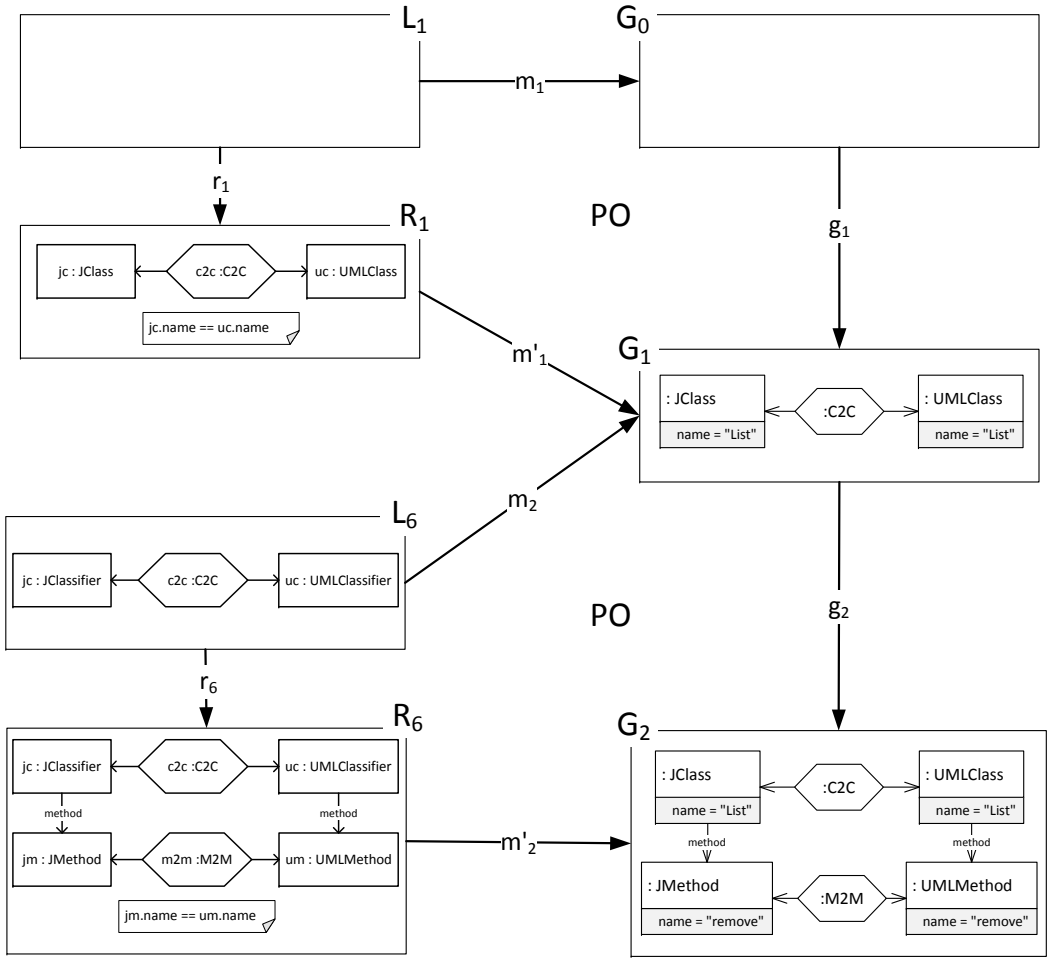


Figure 2.8: A derivation with two rule applications

Example 10. The set of rules in Figure 2.7 forms a TGG for our running example.

The language of a TGG induces a notion of consistency for source and target graphs which is central in the upcoming sections.

Definition 14 (Inter-model Consistency)**.**
Given a TGG, two graphs $G_S \in \mathcal{L}_S(TGG)$ and $G_T \in \mathcal{L}_S(TGG)$ are said to be *consistent* with respect to TGG if $\exists\, G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)$.

2.3 An Extended Consistency Specification for the Running Example

So far, we have discussed the consistency of Java and UML models on minimal examples with a rather simple set of rules (Figure 2.7). Our rules seem symmetric and basically describe a one-to-one mapping: For each Java classifier, method, parameter, and inheritance relation, there must be a counterpart in UML. Consistency can be considered as a *bijective function* in this case, i.e., for a given Java model in the source language of our TGG there is exactly one UML model in the target language simply consisting of the same classifiers, methods, parameters, and in-

heritance relations (this also applies reversely for a given UML model). Experience over the years with industrial consistency projects, however, shows that a more flexible conception of consistency is needed such that consistency of two models is a *relation* (and not necessarily a function). That is, for a given a source model in the source language of a TGG, there might be several target models that are consistent to the given source model (and vice versa).

In fact, consistency of Java and UML models is also a typical scenario requiring a broad consistency understanding that goes beyond bijections. If UML models are intended to be rather high-level models abstracting from implementation details, the extent of abstraction can be considered as a design choice leading to a certain degree of freedom. For example, assume that some classifiers, methods, and further elements in a Java model are used for good Java programming practices but do not necessarily contribute to a platform-independent understanding of the developed system (and this platform-independent understanding is probably the main motivation of using UML). In this case, a UML model representing "less" than the Java model can still be considered as consistent. In the following, we focus on one of these situations and extend our consistency specification which (together with our former set of rules in Figure 2.7) can construct several UML models (differing in their extent of abstraction) for the same Java model. Reversely, several Java models can be constructed for the same UML model (differing in their usage of programming practices that are irrelevant to the UML model).

A well-known practice in Java, which indeed is used by several tools such as [39, 140] when generating Java code from PIMs, is to represent each entity systematically as an interface together with an implementation class. A client application which makes use of such Java code should only interact with interfaces while implementation classes exist only for hiding behavioral complexity. In fact, there is usually technological support to enforce this by making only interfaces available to clients. Generally, the implementation classes adopt a naming convention (e.g., an "Impl"-suffix to the name of the respective interface) to easily assign them to their interfaces when reading code. It is then a reasonable abstraction for the UML model to allow that pairs of interfaces and implementation classes (adhering to the naming convention) in Java are represented as single UML classes. We consider this as a *decision* for each individual case (e.g., more abstraction in UML for a certain group of Java elements and less abstraction for others depending on what the stakeholders expect from a UML model).

In Figure 2.9, two Java models and two UML models are depicted in concrete syntax, all representing a Rectangle entity once as an interface together with an implementation class and once solely as a class. We use an Impl-suffix for naming an implementation class, e.g., RectangleImpl, and refer to such classes as Impl-class (the naming convention generally can be matter of a predefined configuration when generating code or matter of taste when programming). According to our extended, and actually relaxed, understanding of consistency, the following three pairs of models must be regarded as consistent (these pairs are also indicated by the bidirectional arrows in Figure 2.9):

1. The Java model containing only one Rectangle class is consistent to the UML model containing only one Rectangle class, i.e., not using the aforementioned

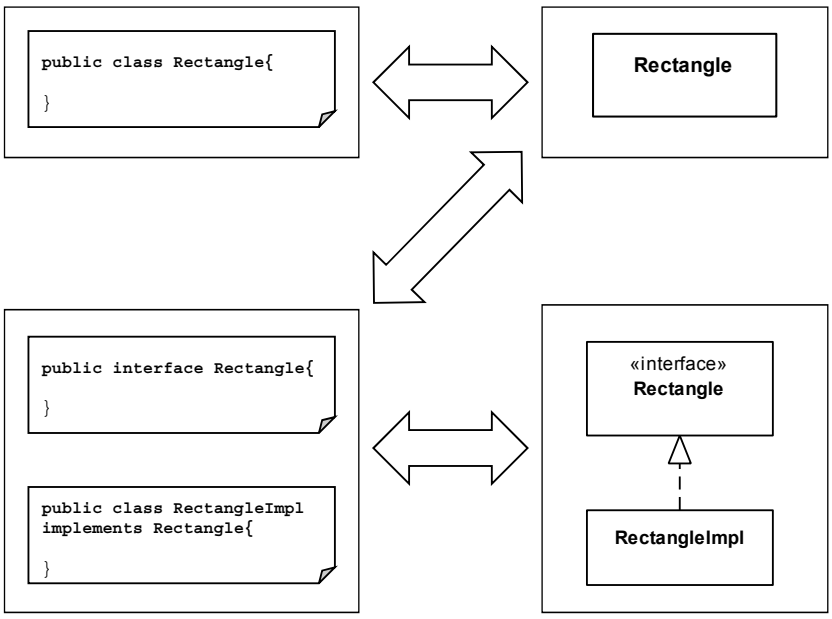


Figure 2.9: Two Java models and two UML models where three pairs are consistent (indicated with ⇔ between models)

programming practice in Java models is allowed (this case is already supported with our rules so far).

2. Also the Java model containing a Rectangle interface and a RectangleImpl class is consistent to the UML model containing only one Rectangle class, i.e., the UML model may abstract from Impl-classes (and this is exactly the extension we are intending in the following).

3. The Java model containing a Rectangle interface and a RectangleImpl class is also consistent to the UML model containing both, i.e., the UML model does not necessarily have to abstract from Impl-classes (this case is already supported with our rules so far).

Moreover, systematic use of interfaces and Impl-classes in Java is not only a good practice but also a necessity if a corresponding UML model describes multiple inheritance between classes (which is not allowed in Java as already taken into account by our rules). Consider, for example, the UML model in Figure 2.10 consisting of Rectangle, Clickable, and Button classes where the latter has inheritance relations to the first two. In this case, demanding for each UML class a corresponding Java class (as we have done so far) is not appropriate anymore. The Java model in Figure 2.10, therefore, represents all these classes as an interface and an Impl-class where multiple inheritance is represented via inheritance relations between interfaces. While such a strategy might seem to be an unsatisfactory solution in many cases leading to redundant method implementations and attributes as these are not inherited from interfaces, it is also the most common means for dealing with the discrepancy between Java and UML. The strategy is especially useful for entities that are rather meant for maintaining and querying data but not for complex com-

putations. Default method implementations in interfaces since the introduction of Java 8, nevertheless, mitigates the problem of redundant method implementations.

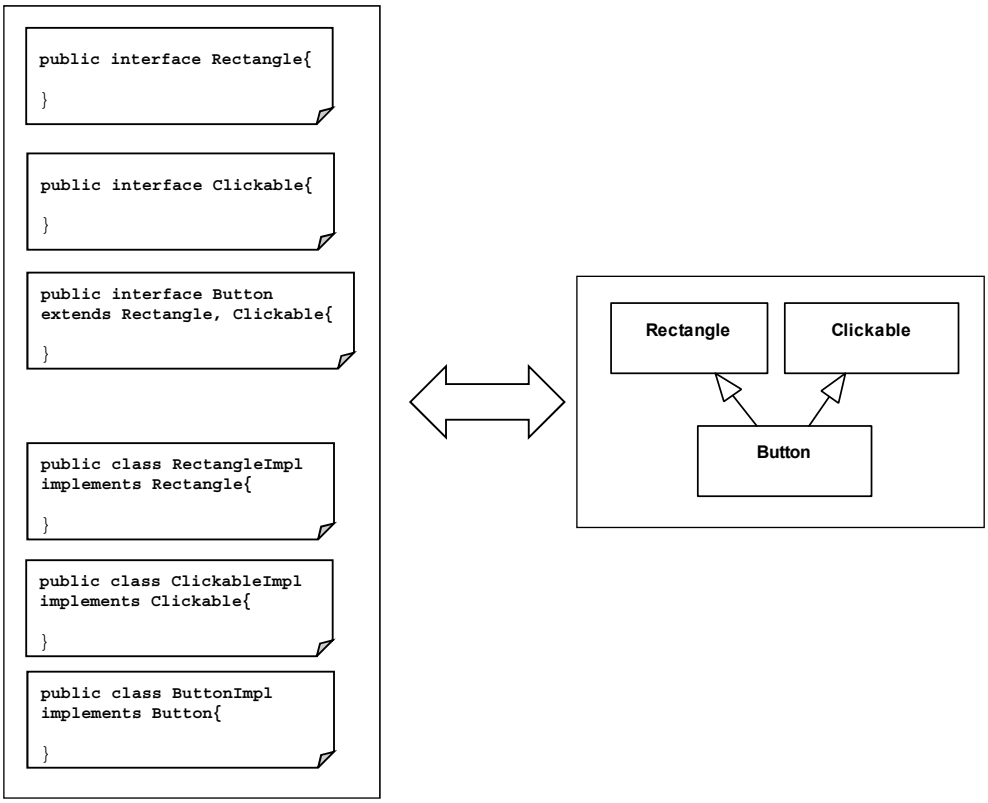


Figure 2.10: A model pair representing multiple inheritance

Having discussed some examples with additional Impl-classes in Java without explicit UML counterparts, we are now ready to extend our consistency specification to capture this. While we need new rules that create a Java interface and a corresponding UML class, the additional Java Impl-class will be connected to the same UML class. To this end, we first extend our correspondence meta-model and introduce a new type of correspondence, namely C2C*, between Java and UML classes as depicted in Figure 2.11 (all other meta-model parts that have already been shown in Figure 2.1 are grayed out). This new correspondence type will be used to connect Impl-classes with the corresponding UML class of their interfaces and, furthermore, allow us to distinguish Impl-classes from other Java classes.

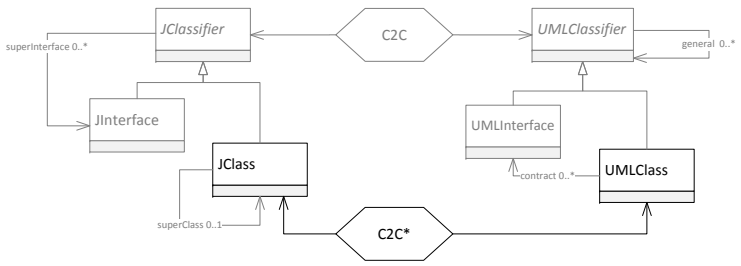


Figure 2.11: New correspondence type C2C* for Impl-classes

Finally, Figure 2.12 depicts further rules that describe how consistent Java and UML models are constructed where Java models may have additional Impl-classes. The individual rules in Figure 2.12 have the following purposes:

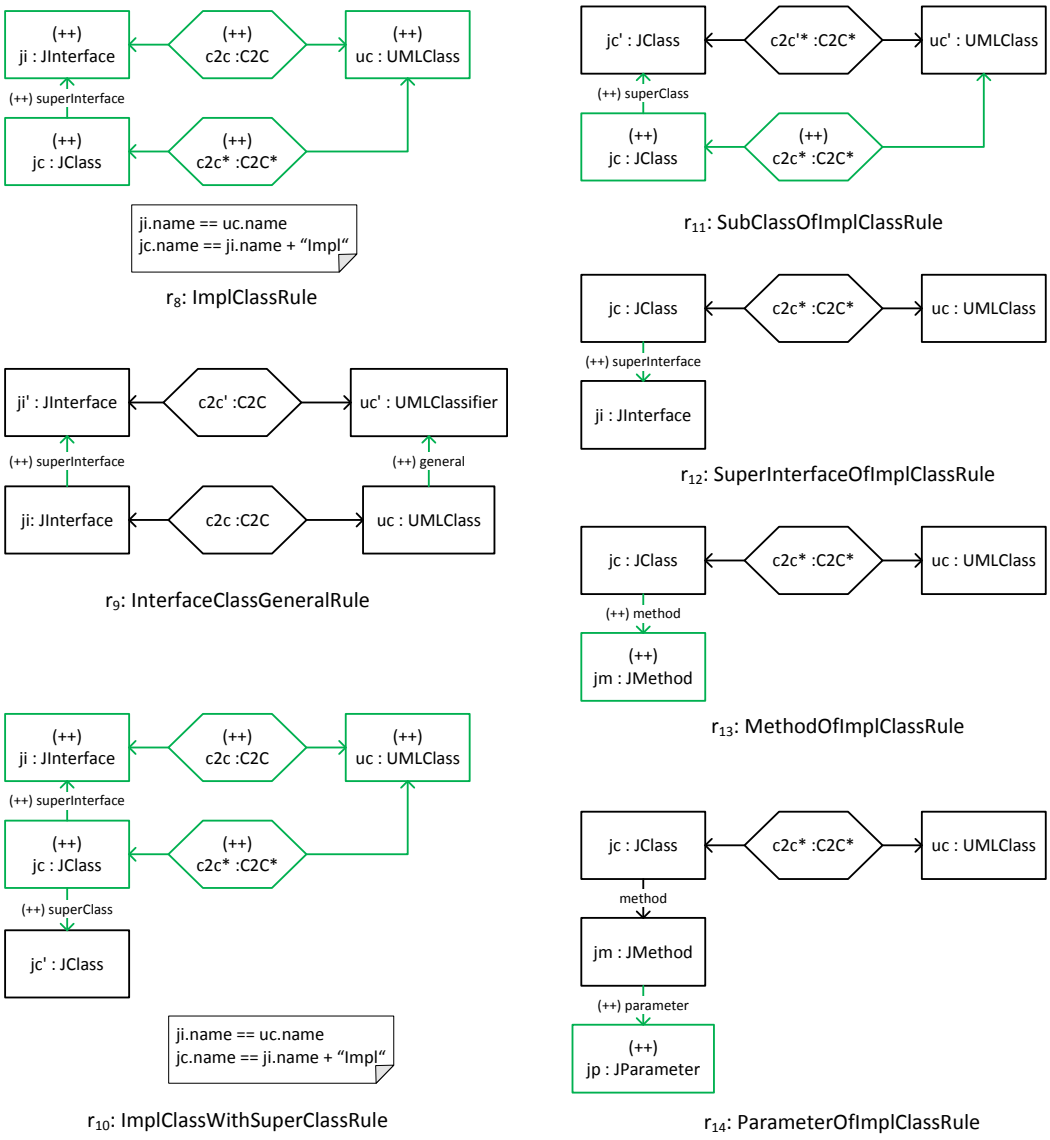


Figure 2.12: New rules to capture additional Impl-classes in Java models

- ImplClassRule (r_8) creates a Java interface and a corresponding UML class with the same name. Additionally, an Impl-class is created on the Java side and connected to the UML class with a correspondence of type C2C*. The name of the Impl-class has a suffix "Impl".
- InterfaceClassGeneralRule (r_9) is similar to GeneralRule (r_4) as it creates a superInterface on the Java side and a general edge on the UML side. The Java interfaces, however, can now have correspondences to UML classes (and not necessarily to UML interfaces) as allowed by the previous rule. Note that this rule enables, for the first time, creating multiple inheritance among UML classes (while still only interfaces might have multiple inheritance in Java).

The remaining rules in Figure 2.12 are provided for the sake of completeness and play rather a minor role in our future examples and discussions. These rules basically describe how further Java elements concerning Impl-classes in Java are created without having a counterpart in UML:

- ImplClassWithSuperClassRule (r_{10}) creates all elements that can also be created by the previously mentioned rule ImplClassRule. The only difference as compared to ImplClassRule is that the Impl-class is created as a subclass of an existing Java class. This inheritance relation is not reflected on the UML side (as the Impl-class itself is not explicitly represented on the UML side).
- SubClassOfImplClassRule (r_{11}) creates a subclass of an Impl-class on the Java side (note that we indicate an Impl-class by requiring a correspondence of type C2C*). The new subclass in Java does not have any counterpart on the UML side but is also connected to the UML class corresponding to its super class. We again use a correspondence of type C2C* for this connection. That is, we handle subclasses of Impl-classes in the same manner and consider them as further Impl-classes that can be omitted on the UML side.
- SuperInterfaceOfImplClassRule (r_{12}) creates a superInterface edge from an existing Impl-class to an existing interface on the Java side. While Impl-classes inherently have at least one superInterface edge as they are created together with their interfaces, this rule creates further superInterface edges. There is no UML counterpart for these edges as we abstract from Impl-classes in UML.
- MethodOfImplClassRule (r_{13}) creates a method for an Impl-class without any UML counterpart.
- ParameterOfImplClassRule (r_{14}) creates parameters for Impl-class methods, again without any UML counterpart.

Overall, the two sets of rules from Figure 2.7 and 2.12 constitute together a consistency specification that allows different degrees of abstraction in UML models for the same Java model (depending on whether a Java class with an Impl-suffix is created as an Impl-class or as a normal class). While such extensions can admittedly blow up the size of a consistency specification, e.g., doubling the number of rules in our concrete case, modularity and reuse concepts for rules as proposed in [11] provide viable means to reduce specification and maintenance effort in practice. We shall exploit the diversity of our rules to exemplify different situations concerning consistency checking and restoration in the upcoming sections. In each individual case, the examples will be reduced to a minimal subset of the rules.

2.4 Summary, Open Issues, and Existing Extensions to TGGs

In this section we have

- discussed meta-models that define the structure of models in an MDE context,

- formalized (meta-)models and triples of (meta-)models as graphs and triple graphs, respectively, using functor categories consequently to derive both structures basically from the well-known category **Sets** of sets,
- captured typing information in triple graphs using slice categories,
- introduced TGGs as a set of rules that describe how to construct triple graphs yielding a language of consistent models.

In all cases, we have exemplified the concepts based on our running example concerning the consistency between Java and UML models. Overall, a consistency specification is provided first as a one-to-one mapping between Java and UML elements and later extended to cater for more complex situations where UML models abstract from some details in Java.

The notion of consistency with respect to a TGG (basically defined as a language membership) is of uttermost importance in the upcoming sections representing our contributions. First, we start with a given TGG and two models and discuss how to check their consistency with respect to the TGG. Besides detecting consistent portions of the models, we create correspondences between them in accordance to the TGG. Second, given a triple graph representing a consistent pair of models together with correspondences, we discuss how to restore consistency when one of the models is changed via added and/or deleted elements. This also captures the case where some model portions are detected to be consistent via consistency checking and the remaining elements are considered as additions to the consistent portions (and thus requiring actions for consistency checking).

We have discussed attributes and vertex type inheritance in graphs only informally (in particular, only on the level of our examples) and left them out in our formalization to avoid notational inflation. These concepts, however, are and remain orthogonal to our statements and are indeed supported in the meta-tool that represents the practical contribution of this thesis.

On the one hand, the entirety of concepts introduced in this section (be it formally or informally) defines the scope for our consistency checking and restoration support in the upcoming sections. On the other hand, nonetheless, some possible extensions to TGGs that are intentionally left out (and possibly missed by a reader who is familiar with TGGs) might represent what is not yet supported by our consistency checking and restoration approach and define the future work. We, therefore, dedicate the remaining of this section to these open issues and existing extensions of TGGs. Most of the issues discussed here are not specific to our consistency scenario but point at some general *expressiveness* challenges with open research questions. While these challenges are currently investigated in an ongoing research project on TGGs (and thus go beyond the scope of this thesis as mentioned in the introduction), we discuss them in order to understand what can or cannot be expected from our consistency checking and restoration approach.

In general, given a source and target meta-model, the source and target language of a TGG covers a subset of all possible source and target models, respectively. In fact, the extension to our consistency specification via additional rules in Figure 2.12 can be regarded as an attempt to enlarge the target language of our TGG (these rules can create multiple inheritance between UML classes which was not the case in our

first set of rules in Figure 2.7). While enlarging the source and/or target language of a TGG is one of the reasonable goals when specifying consistency (i.e., consistency should be specified for as many models as possible), restricting these languages to *valid* models is the other side of the coin.

The validness of a model in an MDE context is examined with regard to a set of *constraints* formulated over the respective meta-model. The most typical type of constraints which we also have implicitly used in our meta-models (Figure 2.1) are multiplicities of edges. For example, Java models are constrained such that multiple inheritance between classes is not allowed (due to the 0..1 multiplicity at the superClass edge). Our TGG respects this constraint: If a superClass edge is created at all in our rules, it is created with the Java class itself. We do not have any rules that would create further superClass edges for existing Java classes. Constraints, however, are not limited to multiplicities and can describe more complex requirements that demand or forbid the occurrence of certain (groups of) elements in a model. Indeed, there exists a further OMG standard, namely the *Object Constraint Language* (OCL) [115], to formulate constraints in the form of predicates. An alternative concept to describe constraints for models, furthermore, is that of *graph constraints* [41] where demanded or forbidden structures over model elements are traced back to existence or absence of graph morphisms in a model. Given a set of constraints for the source and target meta-models, be it formulated as OCL predicates, graph constraints, or just with natural language, the question is whether a TGG constructs only valid models that fulfill the formulated constraints.

In Figure 2.13, the relation between the sets of all possible models, valid models, and the language of a TGG is depicted: The sets $\mathcal{L}_S$ and $\mathcal{L}_T$ represent all possible source and target models, respectively, without considering any constraint while the subsets $\mathcal{L}_S^*$ and $\mathcal{L}_T^*$ represent valid source and target models, respectively (we omit these sets for correspondence models as constraints over correspondences, being only auxiliary structures for the consistency, are not of practical relevance).

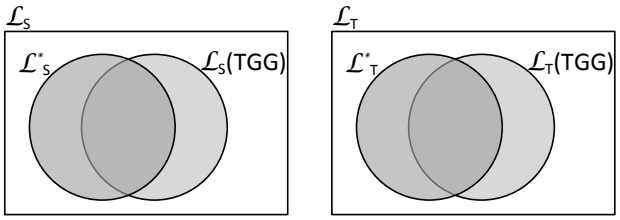


Figure 2.13: Sets of all source and target models ($\mathcal{L}_S, \mathcal{L}_T$), valid source and target models ($\mathcal{L}_S^*, \mathcal{L}_T^*$), and the source and target language of a TGG ($\mathcal{L}_S(TGG), \mathcal{L}_T(TGG)$)

The intersections $\mathcal{L}_S(TGG) \cap \mathcal{L}_S^*$ and $\mathcal{L}_T(TGG) \cap \mathcal{L}_T^*$ in Figure 2.13 indicate valid models which can also be constructed by a TGG. The relative complements $\mathcal{L}_S(TGG) \setminus \mathcal{L}_S^*$ and $\mathcal{L}_T(TGG) \setminus \mathcal{L}_T^*$, however, indicate models that can be created by the TGG but violate some constraints. That is, the consistency specification is not expressive enough to create only valid models.

The depiction of languages in Figure 2.13 also reflects the capabilities of the TGG for our running example, i.e., the TGG can indeed create Java and UML models that belong to the relative complements $\mathcal{L}_S(TGG) \setminus \mathcal{L}_S^*$ and $\mathcal{L}_T(TGG) \setminus \mathcal{L}_T^*$. In the

following list, we state two exemplary constraints for Java and/or UML models in natural language and describe why our TGG violates these constraints:

- Java and UML classifiers (classes or interfaces) may not have cyclic inheritance relations. This also applies transitively. Some rules of our TGG (in particular those which are meant to allow multiple inheritance such as ContractRule), however, create inheritance relation between two existing classifiers. When such a rule, e.g., is applied once and later again applied the other way around, cyclic inheritance relations are created.
- If a Java class inherits from some interfaces, all (transitively or directly) inherited methods must be represented in the Java class (in practice, this does not apply to abstract Java classes which, however, are not captured by our TGG anyway). When creating an inheritance relation between a Java class and and a Java interface, therefore, all inherited methods (whose number is arbitrary and model-specific) must be created which is not provided by our TGG.

The first constraint is of symmetric nature where a model pair created by our TGG can satisfy or violate the constraint on both sides at the same time. The second constraint, however, is specific to the Java side and can be violated by a model pair where the UML model is in fact valid.

In both cases, moreover, the necessity for at least two additional mechanisms can be observed: First, applications of some TGG rules must be blocked to prevent invalid model elements if certain circumstances occur (the first constraint). Second, applications of some TGG rules must be enforced to complement missing elements in models (the second constraint). The following two possible extensions to TGGs (which go beyond what we have formalized so far) can potentially cater for these required mechanisms:

- *Application conditions* that require and/or forbid certain morphisms when applying a rule, i.e., a rule is applied over a match only if the application conditions are satisfied,
- *Multi-amalgamation* that enforces the application of further rules (over certain matches) when a rule is applied, i.e., multiple rule applications are amalgamated (consolidated) in one single step of a rule application (hence the term multi-amalgamation).

There already exist some primary work to integrate application conditions [7, 85] and multi-amalgamation [98, 101] into TGGs. Lifting these concepts to a high level of expressiveness for TGGs is a current research objective where integrating them into consistency checking and restoration procedures is the next logical step. We, therefore, leave these concepts out of our formalization, and consequently out of the scope of this thesis. While the combination of consistency checking and restoration discussed in the upcoming sections is a novel progress already for basic TGGs, we also believe to provide a foundation to comply with when future extensions are developed for TGGs.

3

CONSISTENCY CHECKING WITH TGGS

This section presents our first main contribution based on [95, 100], namely consistency checking between two models that are possibly developed and maintained concurrently (and independently) by their owners. Different than consistency management support as provided by existing MDE tools, and in particular TGG tools, we do not require any available traces or similar information on consistency history between two models but produce these as a result of consistency checking. This is the first step towards a BX vision as depicted in Figure 1.3 where model owners can start using BX in an advanced stage of their development process, and not necessarily from the very beginning as assumed so far.

The grammatical characteristics of TGGs lead to a constructive, precise, and direction-agnostic notion of consistency as stated in Definition 14: A source graph G_S and a target graph G_T are consistent to each other with respect to a TGG if a triple graph $G_S \leftarrow G_C \rightarrow G_T$ can be constructed by the TGG. Hence, given G_S and G_T, the main goal of a consistency checking procedure is to find a valid G_C if there exists one. In the case of inconsistency, moreover, a triple graph $G'_S \leftarrow G_C \rightarrow G'_T$ must be explored where G'_S and G'_T are subgraphs of G_S and G_T, respectively, indicating their consistent portions.

Establishing consistency checking with TGGs is crucial as practical yet formally founded support for consistency checking is scarce in MDE. The *checkonly* mode of QVT-R is currently the only available standard for consistency checking whereas the standard documentation [120] is intentionally not specific on how to realize this. We shall qualitatively compare TGGs and QVT-R in detail for consistency checking purposes when discussing the related work at the end of this section. To mention the main advantages of TGGs in advance, nevertheless, the following points provide the driving force behind our motivation: precise semantics in defining consistency, a symmetric consistency notion which is checked once (and not twice in either direction), and explicit correspondences serving as traceability information. These aspects are not, or not sufficiently, addressed by QVT-R.

However, despite these advantages, progress in consistency checking with TGGs has been slow in more than twenty years since their introduction: The pioneer work concerning consistency checking with TGGs is [43] where *operational* rules are derived from TGG rules to perform consistency checking. These operational rules do not build all source, correspondence, and target models simultaneously as the TGG rules do. Instead, they require source and target models as context and create only correspondences in conformance to the TGG rules. How to apply these rules correctly on a given pair of models, however, remains an open issue and, if done naïvely, can lead to misleading results where, e.g., two consistent models can be

recognized as inconsistent. Arguably due to that reason, TGG tools generally do not support consistency checking but only consistency restoration (in one direction and only with available correspondences from former runs). The only exception we are aware of is HenshinTGG [46] whose consistency checking, however, fails if there exist decision points regarding possible applications of operational rules.

Our goal in the rest of this section, therefore, is to identify and clear the last obstacles concerning consistency checking with TGGs and to come up with a viable and practical solution that provides the aforementioned advantages. We first discuss in more detail, in particular on the level of model elements via our running example, what we expect from a consistency checking approach. Subsequently, we present an overall architecture of a TGG-based consistency checking approach and investigate its single components. We adopt operational rules for consistency checking as proposed in [43], illustrate the problems with regard to wrong choices of applying these rules, and finally, exploit *optimization techniques* to calculate the "best" choice of rule applications when detecting consistent portions of models.

In sum, our contribution is shaped by (i) formulating consistency checking with TGGs as a linear optimization problem, (ii) concluding (in-)consistency of two models via the outcome of linear optimization, and finally, (iii) a consistency checking procedure based on these results.

3.1 Examples of Consistency Checking

Using some exemplary Java and UML model pairs shown in Section 1 and 2, we first discuss a mental evaluation of their consistency with respect to our TGG before automating this task. Our goal is to demonstrate what shapes the outcome of a consistency checking run. We start with the four possible model pairs representing a Rectangle entity from Figure 2.9 and depict in Figure 3.1 (this time in abstract syntax) the desired outcome of consistency checking for these model pairs.

A very important notion and notation when discussing the consistency of two models throughout the rest of this thesis is that of *markings*. We use markings for individual elements (vertices and edges) of source and target graphs to indicate which elements could be created by a given set of TGG rules when started with the empty triple graph. That is, marked elements in a source and target graph represent their consistent portions while unmarked elements require actions for consistency restoration. Consequently, two models are consistent when they can entirely be marked. In Figure 3.1, and also in future figures, marked elements are denoted with a checked box (☑) and unmarked elements with an unchecked box (☐).

Considering the first model pair at the top of Figure 3.1, both models consist of a Rectangle class which clearly can be created by our TGG with ClassRule (r_1). The outcome of consistency checking is then the markings at both sides and the correspondence of type C2C (as it would be created by our TGG). The models are consistent as all source and target elements can be marked. Similarly, the second model pair can completely be created by ImplClassRule (r_8). The outcome is thus again a completely marked model pair, this time with an additional correspondence of type C2C* showing us that consistency is inferred due to our extension with Impl-classes. The third model pair can be constructed by three rule applications:

ClassRule (r_1) for classes, InterfaceRule (r_2) for interfaces, and ContractRule (r_3) for the inheritance relations from classes to interfaces. The fourth model pair, however, is not consistent. In contrast to what our TGG intends to describe, the UML model has an interface with an Impl-class and the Java model is the more abstract one representing this with one single Java class. None of our rules could create any portion of this model pair, leaving the models completely unmarked.

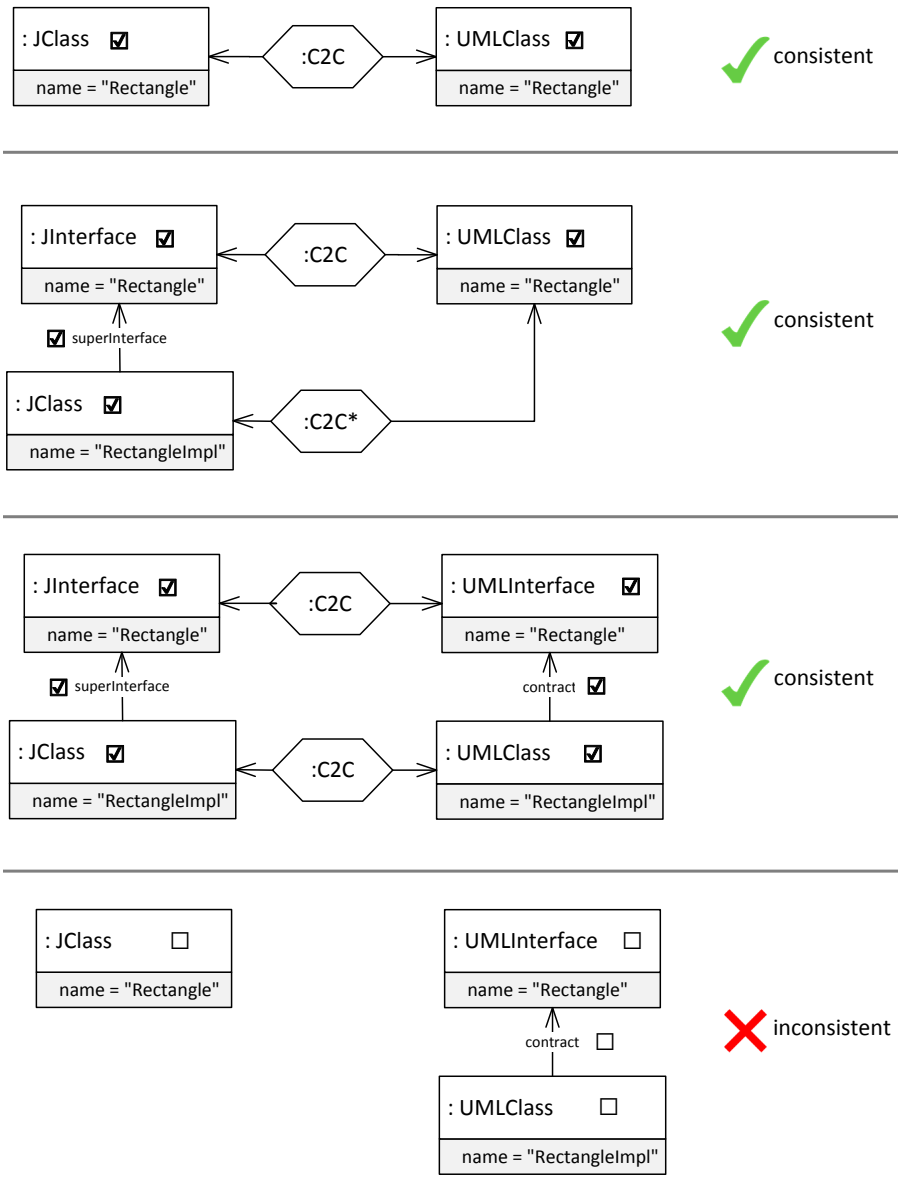


Figure 3.1: Expected results of consistency checking for the four model pairs from Figure 2.9

Overall, a mental or automated execution of consistency checking requires determining which rules could create the given model pair. While these decisions are obvious for the model pairs in Figure 3.1, "more careful" decision making is required when model elements seem to be creatable in different ways at a first glance. We illustrate such a situation in Figure 3.2 using model pairs that exhibit *method overloading* (these models originally stem from Figure 1.5).

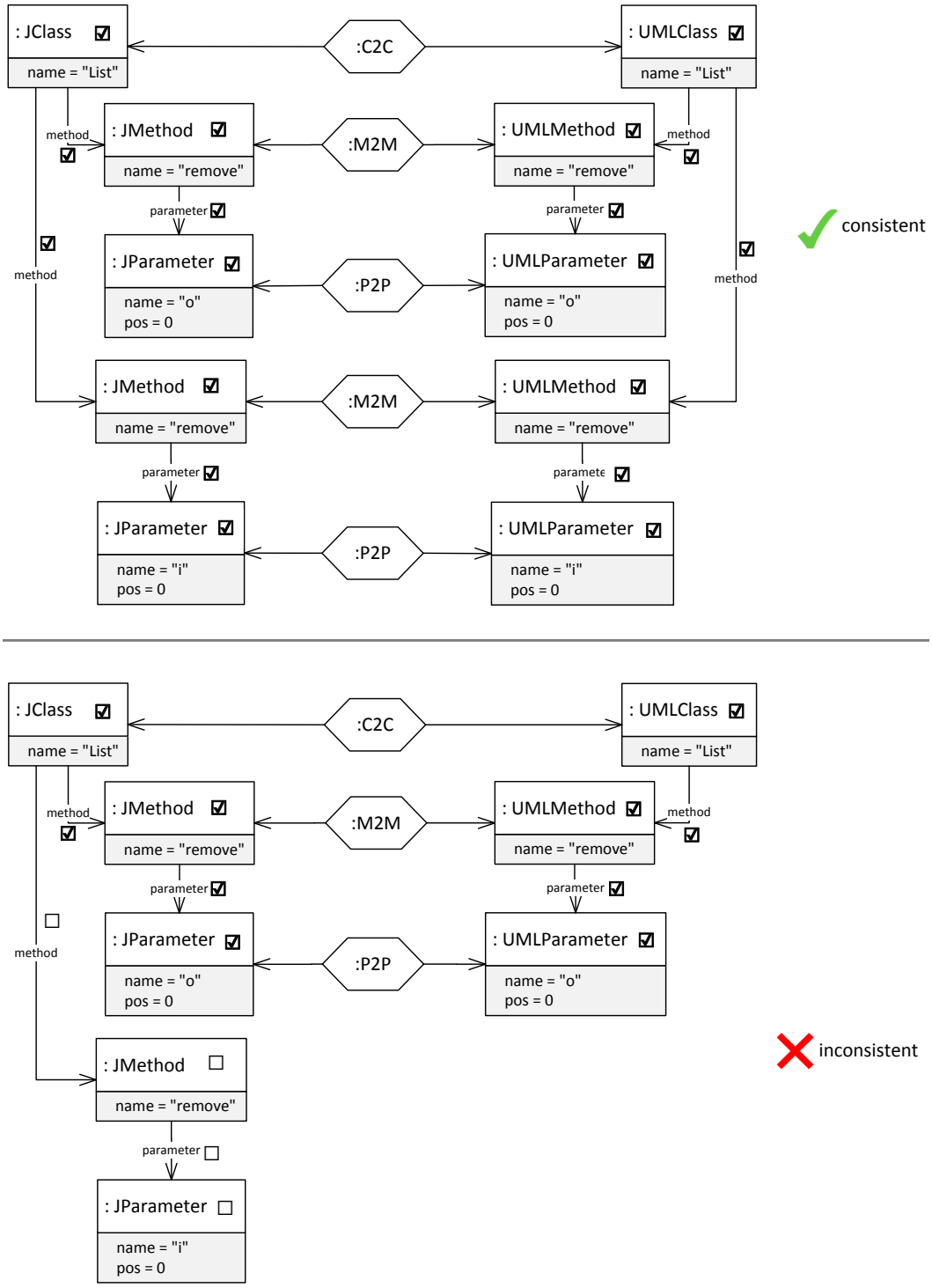


Figure 3.2: A consistent (top) and inconsistent (bottom) model pair with List classes

In the first model pair in the upper part of Figure 3.2, both models have a List class with two remove methods, one with an i parameter and one with an o parameter. Remember that we omit typing of parameters and methods in our running example for brevity and require only name equality for consistent pairs of elements (additionally position equality for parameters). At a first glance, having two remove methods on both sides, there seem to be two possible ways of how these methods could be created together by our TGG with the rule MethodRule (r_6). On the parameter level, however, it becomes clear which pairs of remove methods indeed belong together with respect to our TGG (namely those which have the same parameters).

Similar decisions are also due to be made when models are inconsistent, i.e., when there actually does not exist a way to create a given model pair entirely with a TGG. This is the case in the lower part of Figure 3.2 where the Java model again has the two remove methods but the UML model has only one remove method. No matter which Java method is chosen to be marked with the UML method, the other one remains unmarked. Hence, the model pair will be detected as inconsistent in all cases (which is correct). Again on the parameter level, however, it turns out to be the case that one of the choices is "better" as it at least leads to markings of some parameters. The o parameters can indeed be marked in Figure 3.2 and the only inconsistency is the second remove method in Java with the i parameter.

To sum up, the result of a consistency checking is not only a yes or no answer in our context but also a set of markings (indicating which model elements are part of consistency) and a correspondence graph. This process generally requires a decision making to determine which rules from a TGG could create a given model pair or at least its portions.

3.2 Consistency Rules

After discussing in the previous subsection, actually mainly via human intuition, how to detect whether (or to what extent) two models can be created by a TGG, we are now ready to formalize and automate this. The overall architecture of a consistency checking approach based on TGGs is depicted in Figure 3.3 consisting of two basic tasks: First, TGG rules are *operationalized* to consistency checking rules, short *consistency rules*, that do not create source and target elements but process these. Second, a *TGG engine* applies these consistency rules to a given model pair and produces, as discussed before, a correspondence model and indicates the inconsistent parts of the input models. Consistency rules as well as their applications, therefore, are central in the upcoming definitions and discussions.

We furthermore discuss *marking elements* of a consistency rule. Intuitively, marking elements of a consistency rule result from created elements on the source and target side in the original TGG rule. Our goal in introducing these marking elements is to "simulate" via consistency rules how a given pair of source and target graphs could be created by the original TGG rules.

The operationalization of a rule to a consistency rule can be captured as a pushout construction in **TripleGraphs**. Having morphisms in **TripleGraphs** as natural transformations consisting of three morphisms in **Graphs** (cf. Definition 5 for natural

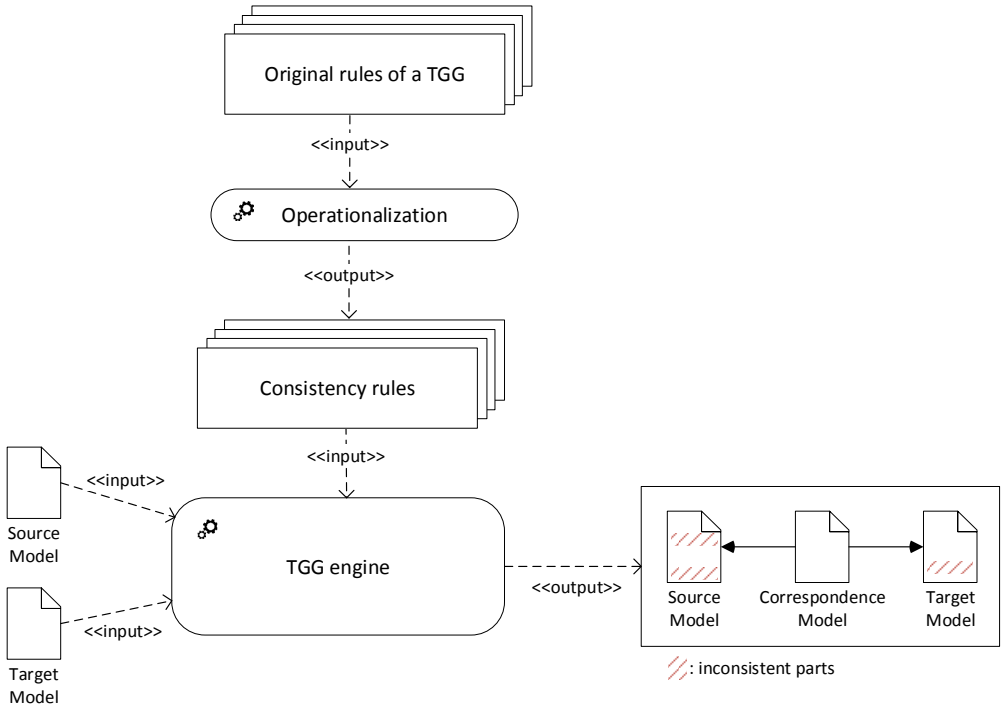
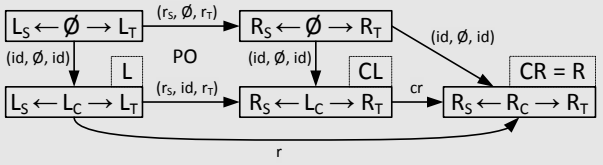


Figure 3.3: Overall architecture of consistency checking with TGGs

transformations and Definition 7 for **TripleGraphs**), we consider morphisms in **TripleGraphs** componentwise in the following definition when constructing a consistency rule from an original TGG rule. For example, (r_s, id, r_t) refers to a morphism in **TripleGraphs** consisting of the three morphisms r_s, id, and r_t in **Graphs**.

Definition 15 (Consistency Rule)**.**

Given a rule $r : L \to R$ with $L = L_S \leftarrow L_C \to L_T$ and $R = R_S \leftarrow R_C \to R_T$, the respective *consistency checking rule*, or short *consistency rule*,



$cr : CL \to CR$ is a rule constructed, as depicted in the diagram, such that CL is a pushout of L and $R_S \leftarrow \emptyset \to R_T$ over $L_S \leftarrow \emptyset \to L_T$, and $CR = R$. The morphism $cr : CL \to CR$ is induced as the universal property of the pushout. An element $e \in \mathsf{elements}(R_S) \cup \mathsf{elements}(R_T)$ is referred to as a *marking element* of cr if $\nexists\, e' \in \mathsf{elements}(L_S) \cup \mathsf{elements}(L_T)$ with $r_s(e') = e$ or $r_T(e') = e$.

When applying consistency rules to a given model pair, i.e., to a triple graph $G_S \leftarrow \emptyset \to G_T$ where the correspondence graph is missing and yet to be created, the goal is to determine whether a triple graph $G_S \leftarrow G_C \to G_T$ can be created by the original TGG rules. If not, a triple graph $G'_S \leftarrow G_C \to G'_T$ where G'_S and G'_T are subgraphs of G_S and G_T, respectively, must be explored which still can be created by the original rules. We exploit the notion of markings for this task in the following sense: The creation of individual source and target elements by the original rules is traced back to markings created by the respective consistency

rules. This is, in fact, similar to *parsing* in the classical theory of grammars but, additionally, correspondences are created on the way which direct the process to new matches and new rule applications.

Example 11. In Figure 3.4, the construction of a consistency rule for MethodRule (r_6) is exemplified. Basically, the left hand-side of a consistency rule cr for an original TGG rule r is constructed such that all source and target elements of r are required as context. The right hand-side of cr, furthermore, simply equals to the right hand-side of r. That is, cr creates only correspondences as created in r but no source or target elements. Moreover, cr marks exactly those source and target elements that are created by r.

At the bottom right of Figure 3.4, the result of this construction is depicted again with our compact syntax. The created elements (only correspondences in the case of a consistency rule) are again depicted with a (++)-markup while the marking elements are now denoted with a ☑-markup. From a purely cosmetic point of view, a consistency rule replaces the (++)-markups with ☑-markups on the source and target side.

Regarding the attribute parts of the graphs, a consistency rule requires exactly the same attribute constraints as the original TGG rule (e.g., name equality must hold for a given method pair to be marked together by the consistency rule in Figure 3.4).

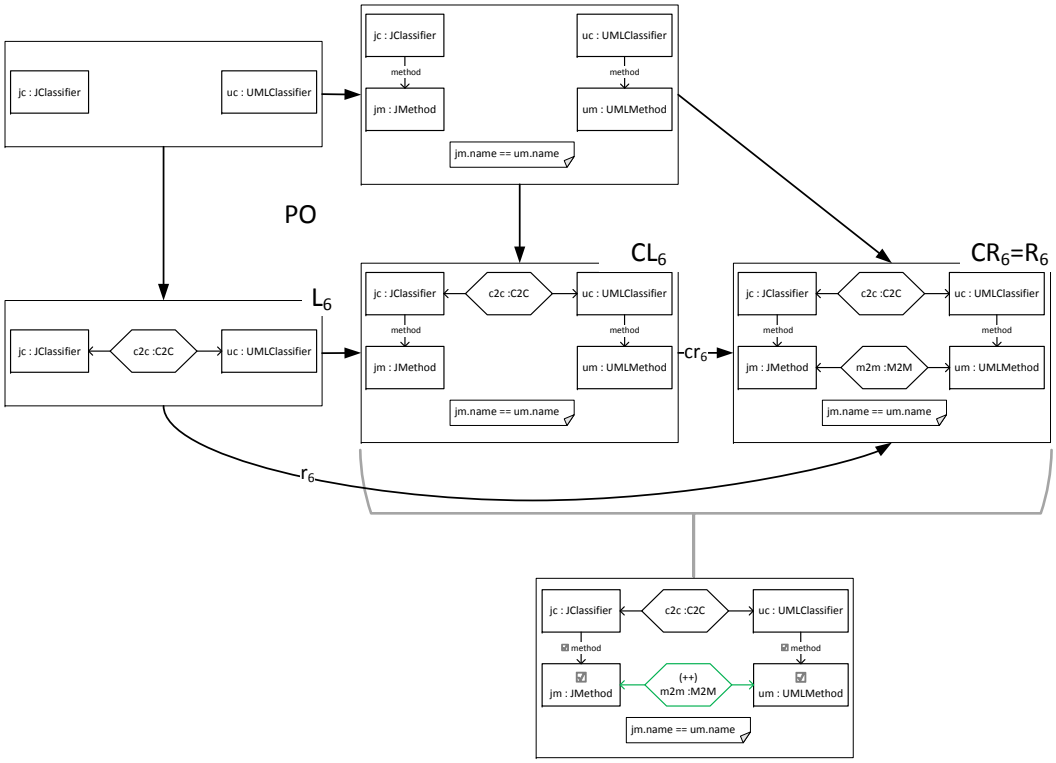


Figure 3.4: Example of a consistency rule construction

Finally, Figure 3.5 and 3.6 depict all consistency rules that are constructed for the original rules from Figure 2.7 and 2.12, respectively.

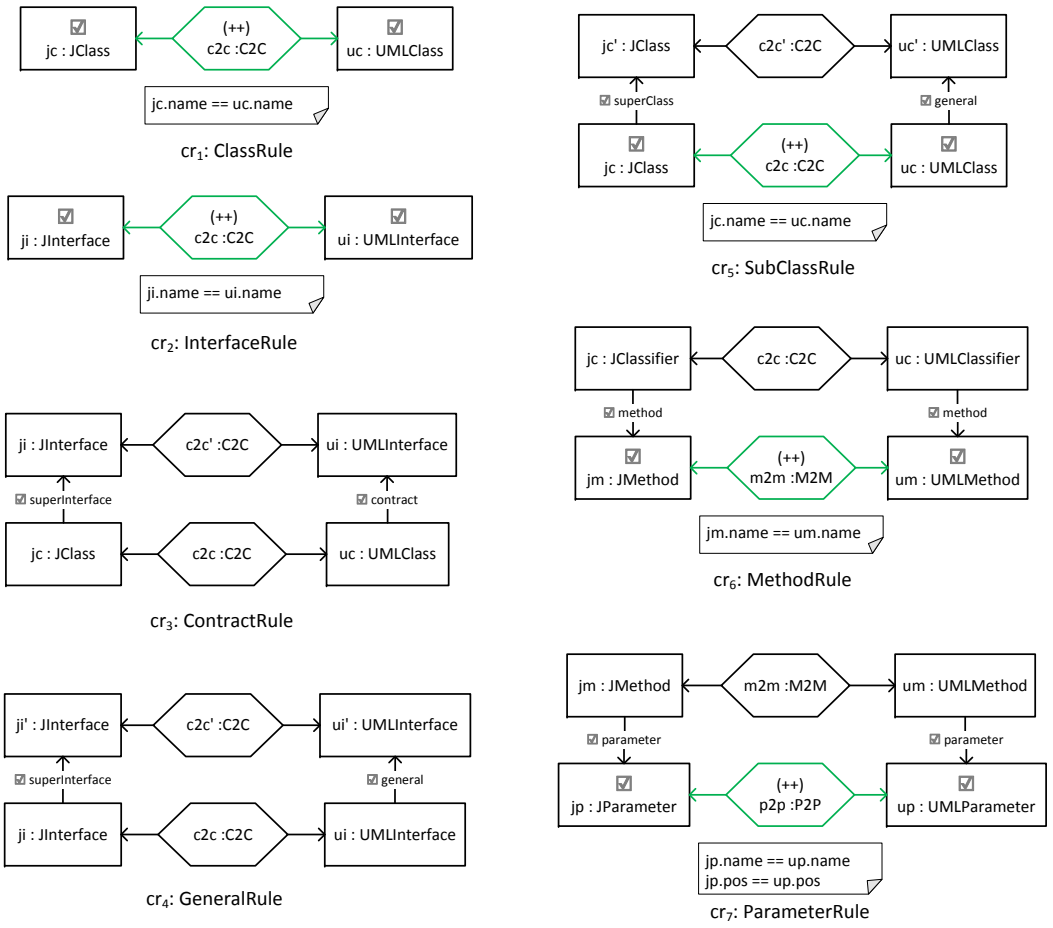


Figure 3.5: Consistency rules for the original TGG rules from Figure 2.7

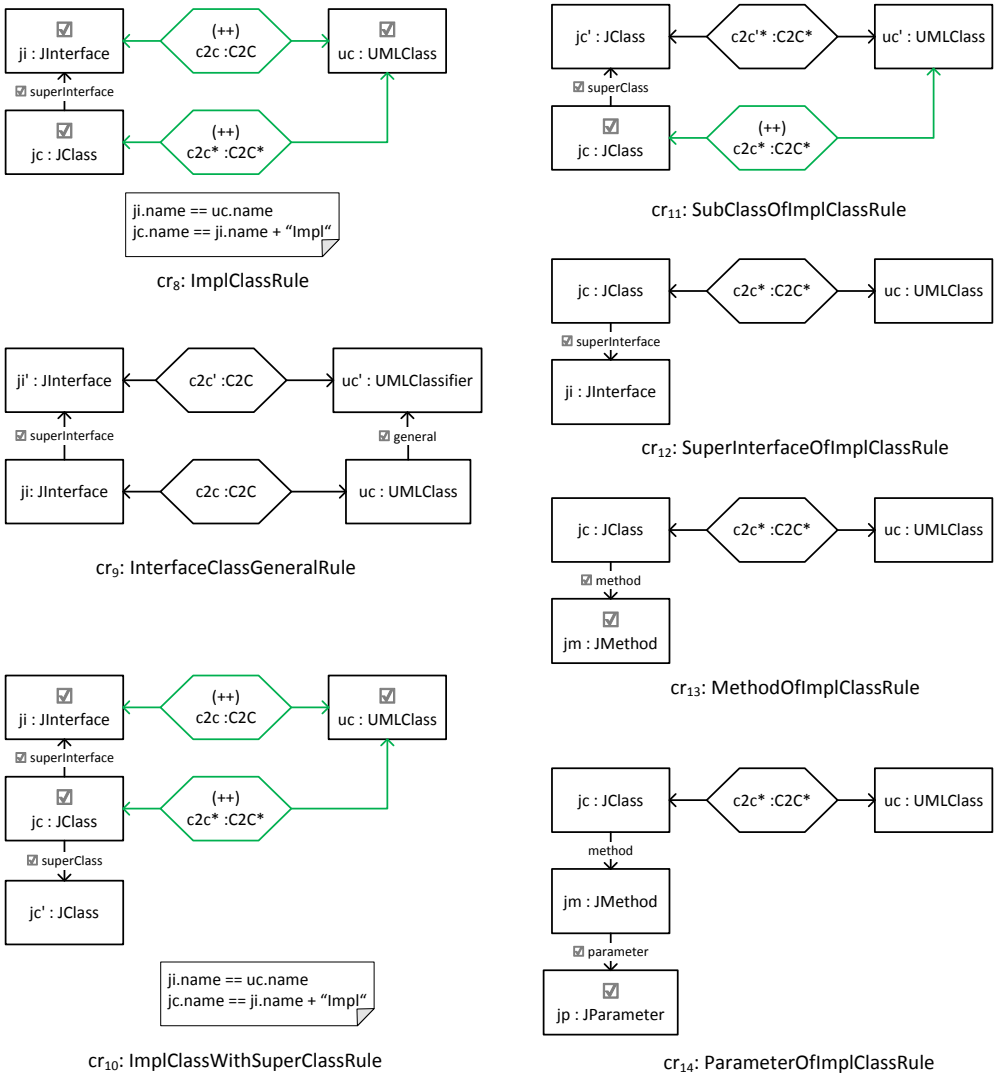


Figure 3.6: Consistency rules for the original TGG rules from Figure 2.12

In order to trace consistency rule applications to original TGG rule applications, we define sets of *marked*, *required*, and *created* elements of a consistency rule application. Marked elements of a consistency rule application are those source and target elements that are matched by the marking elements of the consistency rule, and created elements are created correspondences. Required elements of a consistency rule application, furthermore, are neither marked nor created but only matched.

Definition 16 (Marked, Required, and Created Elements)**.**
Let $cr : CL \rightarrow CR$ be a consistency rule with $CL = CL_S \leftarrow CL_C \rightarrow CL_T$, $CR = CR_S \leftarrow CR_C \rightarrow CR_T$, $CL_S = CR_S$, and $CL_T = CR_T$. For a rule application $\alpha : G \overset{cr@cm}{\Longrightarrow} G'$ with $G = G_S \leftarrow G_C \rightarrow G_T$ and $G' = G_S \leftarrow G'_C \rightarrow G_T$, we define the following sets:

- $\mathsf{marked}(\alpha) = \{e \in \mathsf{elements}(G_S) \cup \mathsf{elements}(G_T) \mid$
  $\exists\ e' \in \mathsf{elements}(CL_S) \cup \mathsf{elements}(CL_T)$ with $cm(e') = e$ where e' is a marking element of $cr\}$
- $\mathsf{requiredSrcTrg}(\alpha) = \{e \in \mathsf{elements}(G_S) \cup \mathsf{elements}(G_T) \mid$
  $\exists\ e' \in \mathsf{elements}(CL_S) \cup \mathsf{elements}(CL_T)$ with $cm(e') = e$ where e' is not a marking element of $cr\}$
- $\mathsf{requiredCorr}(\alpha) = \{e \in \mathsf{elements}(G_C) \mid$
  $\exists\ e' \in \mathsf{elements}(CL_C)$ with $cm(e') = e\}$
- $\mathsf{createdCorr}(\alpha) = \mathsf{elements}(G'_C) \setminus \mathsf{elements}(G_C)$.

Example 12. In Figure 3.7, the result of a derivation is depicted with the following three consistency rule applications in the given order: α_1 via the consistency rule cr_1 of ClassRule, α_2 via the consistency rule cr_6 of MethodRule, and α_3 via the consistency rule cr_7 of ParameterRule. For each marking, we explicitly state next to the respective checked box with which consistency rule application the marking takes place. For the three consistency rule applications, we get the following sets of marked, required, and created elements:

- For α_1, $\mathsf{requiredSrcTrg}(\alpha_1)$ and $\mathsf{requiredCorr}(\alpha_1)$ are empty, $\mathsf{marked}(\alpha_1)$ consists of the Java class and the UML class, and $\mathsf{createdCorr}(\alpha_1)$ consists of the correspondence between these classes.
- For α_2, $\mathsf{requiredSrcTrg}(\alpha_2)$ consists of the Java class and the UML class, $\mathsf{requiredCorr}(\alpha_2)$ consists of the correspondence between these two classes, $\mathsf{marked}(\alpha_2)$ consists of the upper Java method and the UML method, and $\mathsf{createdCorr}(\alpha_2)$ consists of the correspondence between these methods.
- For α_3, finally, $\mathsf{requiredSrcTrg}(\alpha_3)$ consists of the upper Java method and the UML method, $\mathsf{requiredCorr}(\alpha_3)$ consists of the correspondence between these two methods, $\mathsf{marked}(\alpha_3)$ consists of the upper Java param-

eter and the UML parameter, and createdCorr(α_3) consists of the correspondence between these parameters.

The lower Java method and its parameter are neither marked nor matched by any of these consistency rule applications and, therefore, do not belong to any of the sets of marked or required elements.

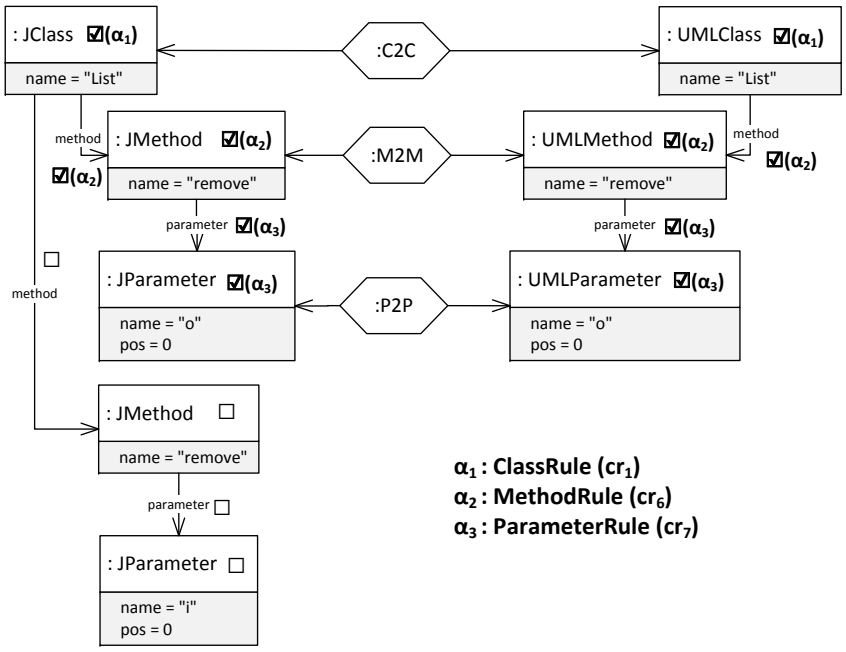


Figure 3.7: A derivation with consistency rules

A derivation with consistency rule applications must fulfill certain properties in order to conform to a derivation with original TGG rules:

- First, consistency rule applications should not mark an element in total more than once as the original rules can self-evidently create these elements only once. Derivations with consistency rule applications fulfilling this property are referred to as *creation preserving*.
- Second, the required source and target elements of a consistency rule application (denoted as requiredSrcTrg in Definition 16) must be marked by some preceding rule applications in the same derivation. We demand this as creations by the original TGG rules can only take place if context elements are already created (accordingly, markings should take place when required context elements are marked). We refer to derivations with consistency rule applications fulfilling this second property as *context preserving*. In a context preserving derivation, an edge, for example, is never marked "before" its connected vertices (in analogy to the fact that an edge can never be created before its vertices are created).

Definition 17 (Creation and Context Preserving Derivations)**.**
Let $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$ be a triple graph and $\mathcal{CR}$ a set of consistency rules. We refer to a derivation $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ with $cr_1, \ldots, cr_n \in \mathcal{CR}$ as

- *creation preserving*, if for all pairs of rule applications $\alpha_i : G_{i-1} \overset{cr_i@cm_i}{\Longrightarrow} G_i$ and $\alpha_j : G_{j-1} \overset{cr_j@cm_j}{\Longrightarrow} G_j$ in d with $1 \leq i, j \leq n$ and $i \neq j$, it holds that $\mathsf{marked}(\alpha_i) \cap \mathsf{marked}(\alpha_j) = \emptyset$
- *context preserving*, if for all rule applications $\alpha_i : G_{i-1} \overset{cr_i@cm_i}{\Longrightarrow} G_i$ in d with $1 \leq i \leq n$, it holds that $\forall e \in \mathsf{requiredSrcTrg}(\alpha_i)$, $\exists\, \alpha_j : G_{j-1} \overset{cr_j@cm_j}{\Longrightarrow} G_j$ in d with $1 \leq j < i$ such that $e \in \mathsf{marked}(\alpha_j)$.

Example 13. The derivation depicted in Figure 3.7 is creation preserving (as none of the Java or UML elements are marked by more than one consistency rule application) and context preserving (as all required source and target elements of a consistency rule application are marked by preceding ones). In particular, α_1 marks all required elements of α_2 and α_2 does the same for α_3.

While creation and context preserving derivations comply with the applications of original TGG rules, it is also of interest whether a derivation with consistency rules entirely marks a given pair of G_S and G_T.

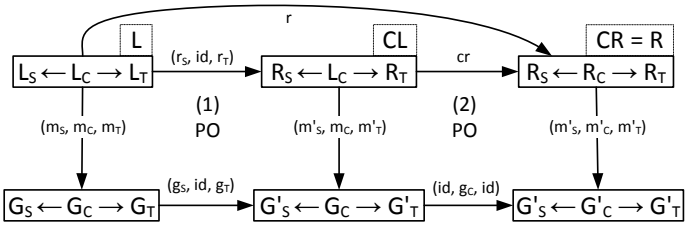
Definition 18 (Entirely Marking Derivations)**.**
Given a triple graph $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$ and a set $\mathcal{CR}$ of consistency rules, and a derivation $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ with $cr_1, \ldots, cr_n \in \mathcal{CR}$, let $\mathcal{D}$ be the set of all rule applications in d. We refer to d as *entirely marking*, if $\bigcup_{\alpha \in \mathcal{D}} \mathsf{marked}(\alpha) = \mathsf{elements}(G_S) \cup \mathsf{elements}(G_T)$.

We are now ready to state the most basic formal result that directly follows from consistency rule construction (Definition 15) and from the required properties for derivations with consistency rules (Definition 17 and 18): Consistent pairs of source and target graphs, and in fact only consistent ones, can entirely be marked via creation and context preserving derivations with consistency rules.

Lemma 1 (Marking Consistent Pairs)**.**
Given a TGG, let $\mathcal{CR}$ be the set of the respective consistency rules. For two graphs G_S and G_T, it holds that
$\exists\, (G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG) \iff \exists\, d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ where $cr_1, \ldots, cr_n \in \mathcal{CR}$, $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$, $G_n = G_S \leftarrow G_C \rightarrow G_T$, d is creation preserving, context preserving, and entirely marking.

Proof. As a result of pushout composition and decomposition (Fact 2.20 in [42]), each rule application $G \overset{r@m}{\Longrightarrow} G'$ via a rule $r \in TGG$ can uniquely be decomposed

into and accordingly composed from two pushouts depicted as (1) and (2) in the diagram below. The first pushout (1) creates only source and target elements while the second pushout (2) creates only correspondences. The second pushout, moreover, equals to a consistency rule application.



As a result of the consistency rule construction (Definition 15), the consistency rule application in (2) marks exactly those source and target elements which are created in (1). Applying the composition and decomposition of single rule applications with original rules to entire derivations, it follows that for each derivation d' with the original rules of a TGG, a derivation d exists consisting of the respective consistency rule applications in the same order. As markings in d bijectively correspond to creations in d', each source and target element is marked exactly once by d (i.e., d is creation preserving and entirely marking) and elements that are not marked by a rule application in d should have been previously marked (i.e., d is context preserving). Due to the bijection between creations and markings, this also applies in the reverse direction as well such that the existence of d' can be concluded from the existence of d. □

The equivalence relation ($\Longleftrightarrow$) stated in Lemma 1 captures two essential properties for TGGs as proposed in [129] (in fact, introduced only for model-to-model transformations with TGGs but analogously transferable to other operational scenarios including consistency checking). The $\Longleftarrow$-direction relates to *correctness*, i.e., that a triple graph constructed by the operational rules (consistency rules in our case) belongs to the language of the given TGG. The $\Longrightarrow$-direction, furthermore, relates to *completeness*, i.e., each triple graph in the language of the TGG can be constructed this way. In other words, for all triple graphs $G = G_S \leftarrow G_C \rightarrow G_T$ in the language of a TGG, the triple graph $G_S \leftarrow \emptyset \rightarrow G_T$ can evolve to G by a derivation d via consistency rules. Finding d, however, is an open challenge and defines our focus in the rest of this section (and shapes the core of our contribution).

3.3 Wrong Choices of Consistency Rule Applications

Lemma 1 provides a basic result to conclude (in-)consistency of a source graph G_S and a target graph G_T from the *existence* of derivations with consistency rules fulfilling certain properties. Fundamentally, a creation and context preserving derivation must be built to mark individual elements of G_S and G_T. When realizing this in a TGG engine, however, it is still open how to find a derivation with consistency rules which would entirely mark G_S and G_T if they are consistent.

As our discussions via examples in Section 3.1 already implied, the process of marking individual elements in G_S and G_T can require a decision making. Hence, if

consistency rules are applied naïvely, i.e., elements are marked step by step without comparing different possibilities of rule applications, consistency checking might end up with a misleading result where more markings would actually be possible. In fact, this also reflects the open challenges with regard to existing foundational work [43] and the respective tool support [46] of consistency checking with TGGs.

In Figure 3.8, a consistency checking run with our running example is depicted with an undesired outcome. The Java and UML models each containing a List class with two remove methods, which are indeed consistent as depicted previously in Figure 3.2, are indicated to be inconsistent as the "wrong" pairs of remove methods have been marked together (as they simply match for the consistency rule cr_6 of MethodRule). Parameters cannot be marked after this wrong choice as the name equality between parameters of corresponding methods, as required by the consistency rule cr_7 of ParameterRule, does not hold in this case.

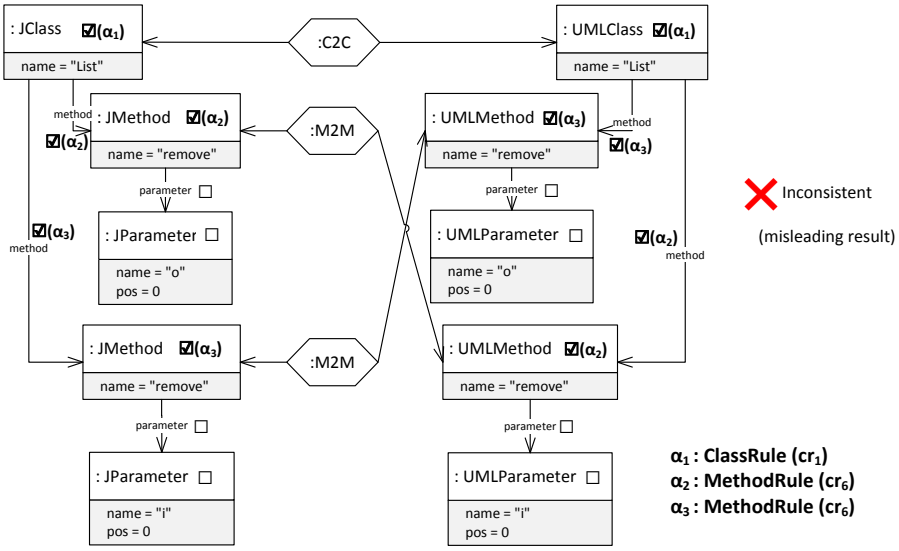


Figure 3.8: Undesired outcome of consistency checking with consistent models

A noteworthy aspect is that this example is only related to our first set of TGG rules (Figure 2.7) which actually describes nothing but a one-to-one mapping. While translating single Java elements to UML elements (or vice versa) one by one for consistency restoration purposes can be a straightforward task with these rules, setting given Java and UML elements in correspondence is apparently a more difficult task that is subject to a larger search space of possible rule applications.

The problem arising from possibly wrong choices of consistency rule applications does not only concern consistent models that are mistakenly indicated as inconsistent. Also in the case of inconsistent model pairs where an entirely marking derivation cannot be found anyway, decisions can still lead to a more appropriate or unsuitable set of markings. In Figure 3.9, a situation is depicted where the models are correctly indicated as inconsistent but actually marking more elements would be possible. The only remove method on the UML side with an o parameter is marked together with the remove method with the i parameter on the Java side. Hence, no more markings at the parameter level are possible. As already

depicted in Figure 3.2, however, setting the remove methods with o parameter in correspondence yields a better result where more elements are marked.

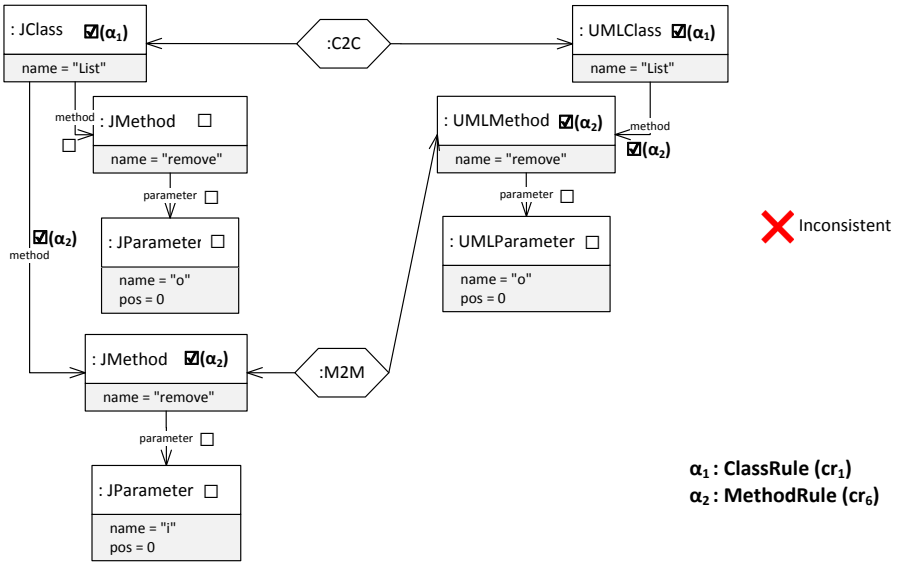


Figure 3.9: Undesired outcome of consistency checking with inconsistent models

Search space problems regarding different choices of rule applications are, in fact, not new in the broad field of graph grammars but in some way remarkably critical in the case of consistency checking (it should be noted that we demonstrate the problems at oversimplified examples whereas the situation is most likely aggravated in real-world consistency scenarios). In general, the following two measures are commonly practiced for eliminating wrong choices of rule applications:

- *backtracking*, i.e., revoking some rule applications back to a decision point after a wrong choice has been detected,
- *look-ahead*, i.e., exploring the input graph in advance to determine which rule application to choose at a decision point.

For our concrete examples with Java and UML models exhibiting method overloading, backtracking from the parameter level to the method level or look-ahead at the method level to the parameter level can yield satisfactory results. We, however, argue that a general solution with backtracking or look-ahead is not feasible as it requires in general arbitrary (and unknown) depth of backtracking or look-ahead. More crucially, if the models are inconsistent, it remains unclear whether unmarked elements are due to wrong choices of rule applications or just the consequence of inconsistency, i.e., exhaustive backtracking or look-ahead can be performed looking for a solution which is not there.

There are also static analysis techniques in graph grammars to check whether a set of rules fulfills certain properties, in particular *confluence* [40], such that their applications always lead to the same result irrespectively of what decisions are taken. If not, it is then up to the user (to the consistency tool developer in our concrete case) how the rules are modified to ensure this. While this additional

effort at specification time can eliminate the need for backtracking, look-ahead, or any other means to avoid wrong choices of rule applications at runtime, we would end up with a very restrictive solution even rejecting a one-to-one mapping as our first set of rules does. We are actually also not aware of any modification to our TGG such that it describes the same consistency but does not suffer from the search space involved in applying consistency rules.

If there are better or worse consistency rule applications, we propose to capture the situation as an *optimization problem*. Intuitively, we first consider different choices of consistency rule applications without explicitly taking a decision. Subsequently, we formulate constraints between these choices to calculate a proper subset such that creation and context preserving derivations can be built (and we can make use of Lemma 1). Finally, we pick a solution that satisfies the constraints and marks "as many elements as possible". This allows us to handle consistent and inconsistent models in a unified manner: If the global maximum in the sense of markings marks all model elements, consistency can be concluded. If not, the approach still gives its best with regard to the number of marked elements (and unmarked elements require action for consistency restoration).

Before discussing these ideas and stating our formal results, we make at this point a digression to the basics of our chosen optimization technique, namely *Integer Linear Programming* (ILP), with focus on our consistency checking goals.

3.4 Integer Linear Programming (ILP) Techniques

In recent years, there has been a growing interest in exploiting optimization techniques in model transformation. Optimization techniques, being already employed in many engineering and business fields, enjoy a mature foundation and practical tool support that is provided in a general way and can be used for custom purposes. While some approaches (e.g., [28, 81]) propose to formulate a model transformation task entirely as an optimization problem over the input model(s), some approaches (e.g., [17, 48, 49]) choose to use optimization techniques only as a means for decision making in rule-based model transformation. In the latter case, rules and their rationale (consistency in the case of TGGs) remain central and only decision making is outsourced to optimization techniques when different rule applications are possible. This strategy plays a key role in our statements in the rest of this section.

For consistency checking with TGGs, our formulation of an optimization problem mainly consists of the following two parts:

- *logical constraints* expressing implications and exclusions between rule applications such that creation and context preserving derivations can be built,
- an *objective* that maximizes the number of marked elements while satisfying these constraints.

Both aspects can be captured by ILP techniques. In this setting, logical constraints are formulated as linear equalities or inequalities between integer variables, and the objective is given by a linear function over these variables (which is then to be maximized or minimized). A special case of ILP (which is also easier to manage

from a practical point of view) is the so-called 0-1 ILP where all integer variables are restricted to integers between 0 and 1. As a result of its binary nature, the constraint part of a 0-1 ILP problem is comparable to a *Boolean Satisfiability Problem* (SAT) where 0 and 1 represent Boolean values false and true, respectively.

ILP techniques are commonly used as a computational means in management and economics to solve different resource allocation tasks such as capital budgeting, transportation and infrastructure planning. An artificial, though still well-fitting, analogy to resource allocation tasks can also be drawn for consistency checking with TGGs in the following sense: Source and target graph elements are resources which are consumed by consistency rule applications (in particular by their markings). The challenge is then to allocate as many elements as possible to their consumers such that creation and context preserving derivations can be constructed.

Understanding the art of formulating integer equalities and inequalities to solve certain tasks is also crucial to exploit ILP techniques. In [24], a collection of good practices is provided when modeling logical constraints as integer equalities and inequalities. Given n integer variables $x_1, \ldots, x_n$ where $0 \leq x_1, \ldots, x_n \leq 1$, we shall basically focus on the following three types of integer inequalities for our purposes. Note that each variable x_i represents a consistency rule application α_i in our case. When finding values for the variables, $x_i = 1$ indicates that α_i is to be chosen among the collected rule applications with consistency rules (and $x_i = 0$ accordingly indicates that α_i is eliminated).

- If at most one of $x_1, \ldots, x_n$ is allowed to be 1, i.e., at most one of $\alpha_i, \ldots, \alpha_n$ might be chosen as they compete for marking the same source and/or target elements, we state this as $x_1 + \ldots + x_n \leq 1$.

- If $x_i = 1$ implies that at least one of $x_1, \ldots, x_n$ is 1, i.e., choosing a consistency rule application α_i implies choosing at least one of the consistency rule applications $\alpha_1, \ldots, \alpha_n$ as these provide some markings required by α_i, we state this as $x_i \leq x_1 + \ldots + x_n$. A special case of such constraints is of the form $x_i \leq x_j$ where choosing α_i directly implies choosing α_j as α_j creates a correspondence required by α_i.

- If $x_1, \ldots, x_n$ cannot all be 1 at the same time, we state this as $x_1 + \ldots + x_n < n$. We use this type of constraint to avoid cyclic implications between rule applications. That is, if rule applications directly or transitively provide markings for each other in a cyclic manner, at least one of them must be eliminated to break the cycle and to sequence rule applications to a derivation.

Finally, objective functions are of the form $k_1 * x_1 + \ldots + k_n * x_n$ where each variable x_i is weighted with a constant k_i. In our concrete case, the weights are the numbers of marked source and target elements for each individual consistency rule application. While this is the most general (and arguably the most formally justified) way for determining the weights of consistency rule applications, case-specific weights can be assigned to consistency rule applications for a higher prioritization of certain correspondence types and/or markings. In all cases, the optimization task is then to find a solution that fulfills the constraints as introduced above and maximizes the value of the objective function.

3.5 Consistency Checking with TGGs and ILP

In a final step in this section, we formalize the integration of ILP techniques into consistency checking with TGGs to cope with the discussed search space problems. We consider derivations with consistency rules that possibly mark model elements multiple times and thus represent a superset of correct markings. While such a derivation is not necessarily creation and/or context preserving, we calculate a subset of the collected consistency rule applications which yields a creation and context preserving derivation and marks as many elements as possible.

In Figure 3.10, our ultimate goal is exemplified schematically with the consistent model pair containing two remove methods both on the Java and UML side. In the upper part, a derivation consisting of the seven rule applications $\alpha_1, \ldots, \alpha_7$ is depicted where all remove methods (and their incident edges from the List classes) are marked twice as there exist two possible ways to mark these elements. Besides markings, we explicitly show at each correspondence its creating rule application.

Of the four rule applications marking Java and UML methods, α_2 and α_3 lead to the undesired result previously depicted in Figure 3.8 while α_4 and α_5 lead to a desired result where parameters can also be marked. Performing ILP solving over these rule applications as we shall discuss in the following, the subset consisting of the five rule applications $\alpha_1, \alpha_4, \alpha_5, \alpha_6$, and α_7 is chosen to construct a creation and context preserving derivation. Finally, the models are indicated to be consistent as the found solution marks both models entirely. Further examples including inconsistent models shall be provided to exemplify our statements as well.

A *constraint* for a derivation d with consistency rules is a conjunction of integer inequalities (while ILP techniques generally support both equalities and inequalities, inequalities suffice for our purposes). These inequalities are stated over some integer variables each representing an individual consistency rule application. When choosing a subset of the consistency rule applications in d, the constraints must be satisfied by assigning 1 to the respective variables of the chosen consistency rule applications (and by assigning 0 to the respective variables of the eliminated consistency rule applications).

Definition 19 (Constraints for Derivations with Consistency Rules)**.**
Given a triple graph $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$, a set $\mathcal{CR}$ of consistency rules, and a derivation $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ with $cr_1, \ldots, cr_n \in \mathcal{CR}$, let $\mathcal{D}$ be the set of all rule applications in d. For each rule application $\alpha_1, \ldots, \alpha_n \in \mathcal{D}$, we define respective integer variables $x_1, \ldots, x_n$ with $0 \leq x_1, \ldots, x_n \leq 1$.
A *constraint* C for d is a conjunction of linear inequalities which involve $x_1, \ldots, x_n$.
A set $\mathcal{D}' \subseteq \mathcal{D}$ fulfills C, denoted as $\mathcal{D}' \vdash C$, if and only if C is satisfied for variable assignments $x_i = 1$ if $\alpha_i \in \mathcal{D}'$ and $x_i = 0$ if $\alpha_i \notin \mathcal{D}'$ with $1 \leq i \leq n$.

Given a model pair $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$ and a derivation d via some consistency rules that marks elements in G_0 (possibly multiple times as no explicit decisions are taken between alternative markings), our first constraint markedAtMostOnce(G_0) requires that each source and target element in G_0 be marked at most once by the chosen consistency rule applications. Consequently, an element can either remain unmarked (due to inconsistency) or it can be marked once. Having introduced

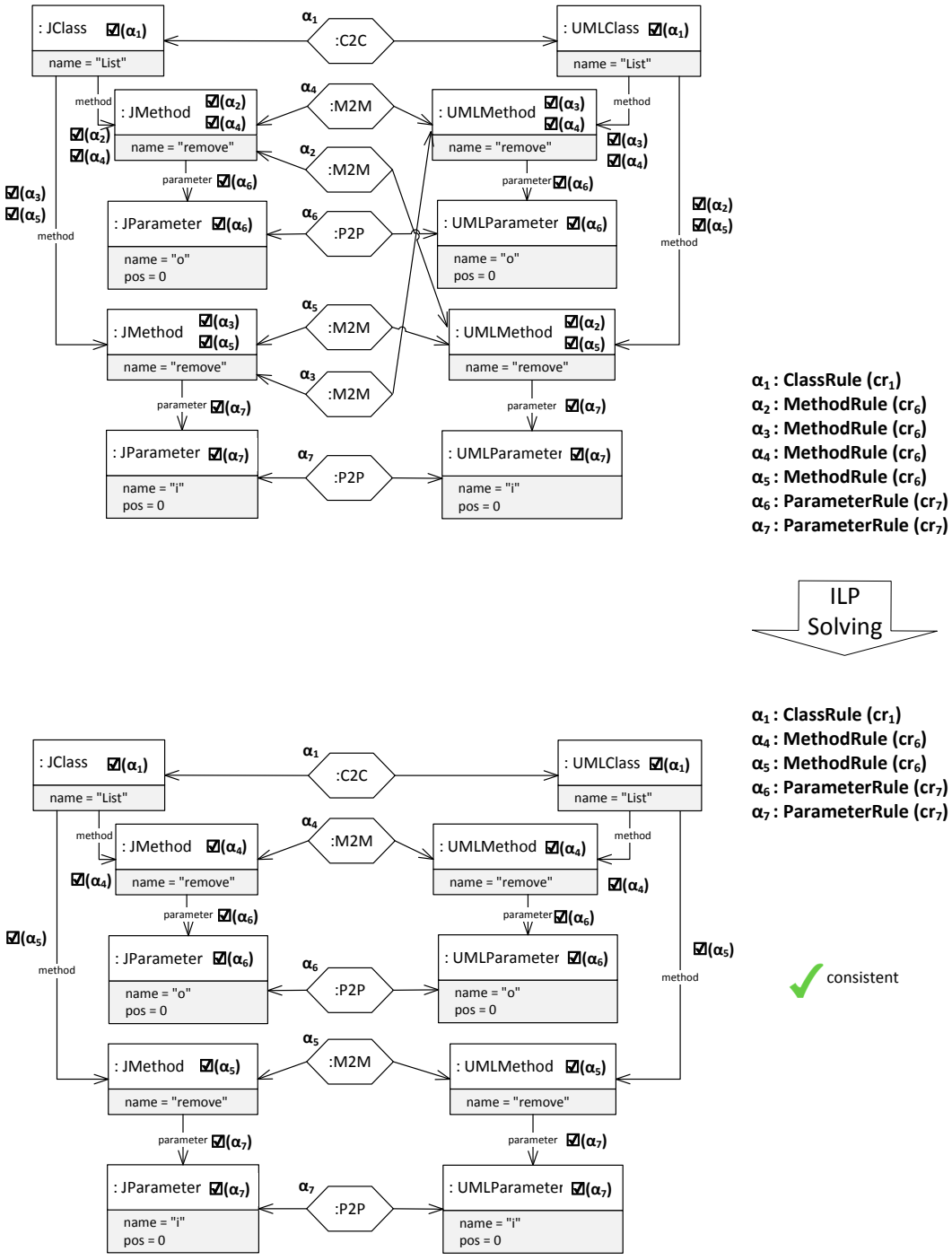


Figure 3.10: A schematic exemplification of our approach with consistent models

integer variables for each rule application in Definition 19, we first define the sum of alternative markings of the same element in Definition 20. Subsequently, this sum is restricted to 0-1 as a constraint in Definition 21.

Definition 20 (Sum of Alternative Markings for an Element)**.**
Given a triple graph $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$, a set $\mathcal{CR}$ of consistency rules, and a derivation $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ with $cr_1, \ldots, cr_n \in \mathcal{CR}$, let $\mathcal{D} = \{\alpha_i, \ldots, \alpha_n\}$ be the set of all rule applications in d where $\alpha_i : G_{i-1} \overset{cr_i@cm_i}{\Longrightarrow} G_i$, $1 \leq i \leq n$. For each element $e \in \mathsf{elements}(G_S) \cup \mathsf{elements}(G_T)$, we define an integer variable $\mathsf{markersSum}(e)$ that denotes the sum of integer variables for all rule applications that mark e, i.e., $\mathsf{markersSum}(e) = \sum_{i=1}^{n} k_i * x_i$ where $k_i = 1$ if $e \in \mathsf{marked}(\alpha_i)$ and $k_i = 0$ otherwise.

Definition 21 (Constraint 1: Marking Each Element At Most Once)**.**
Given a triple graph $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$, a set $\mathcal{CR}$ of consistency rules, and a derivation $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ with $cr_1, \ldots, cr_n \in \mathcal{CR}$, the constraint $\mathsf{markedAtMostOnce}(G_0)$ for d is defined as $\bigwedge_{\substack{e \in \mathsf{elements}(G_S) \cup \\ \mathsf{elements}(G_T)}} \mathsf{markersSum}(e) \leq 1$.

Example 14. For the derivation depicted in the upper part of Figure 3.10, we get the following inequalities for the markedAtMostOnce constraint. The inequalities are stated over the seven integer variables $x_1, \ldots, x_7$ representing the seven rule applications $\alpha_1, \ldots, \alpha_7$. In each inequality, the left hand-side of the $\leq$ sign represents the markersSum for the respective element(s) stated explicitly in brackets.

- $x_1 \leq 1$ (as the Java and UML classes are only marked by α_1)
- $x_2 + x_4 \leq 1$ (as the upper Java method is marked by α_2 and α_4)
- $x_3 + x_4 \leq 1$ (as the upper UML method is marked by α_3 and α_4)
- $x_3 + x_5 \leq 1$ (as the lower Java method is marked by α_3 and α_5)
- $x_2 + x_5 \leq 1$ (as the lower UML method is marked by α_2 and α_5)
- $x_6 \leq 1$ (as the upper Java and UML parameters are only marked by α_6)
- $x_7 \leq 1$ (as the lower Java and UML parameters are only marked by α_7).

Obviously, the goal of the markedAtMostOnce constraint is to make a choice of rule applications that lead to creation preserving derivations. The next constraint $\mathsf{context}(d)$ defines dependencies as implications between consistency rule applications in a derivation d due to their required context. A consistency rule application is either not chosen, or its required source and target elements (denoted as requiredSrcTrg in Definition 16) must be marked by some other chosen rule applications.

Similarly, the required correspondences (denoted as requiredCorr in Definition 16) of a chosen rule application must be created by some other chosen ones.

Definition 22 (Constraint 2: Providing Context for Markings)**.**
Given a triple graph $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$, a set $\mathcal{CR}$ of consistency rules, and a derivation $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ with $cr_1, \ldots, cr_n \in \mathcal{CR}$, for each rule application $\alpha_i : G_{i-1} \overset{cr_i@cm_i}{\Longrightarrow} G_i$ in d, we define the following constraints:

- $\mathsf{contextSrcTrg}(\alpha_i) = \bigwedge_{e \in \mathsf{requiredSrcTrg}(\alpha_i)} x_i \leq \mathsf{markersSum}(e)$,
- $\mathsf{contextCorr}(\alpha_i) = \bigwedge_{\substack{1 \leq j < i, \\ \mathsf{requiredCorr}(\alpha_i) \cap \mathsf{createdCorr}(\alpha_j) \neq \emptyset}} x_i \leq x_j$.

The constraint $\mathsf{context}(d)$ denotes $\bigwedge_{1 \leq i \leq n} \mathsf{contextSrcTrg}(\alpha_i) \wedge \mathsf{contextCorr}(\alpha_i)$.

Example 15. We state in the following the integer inequalities resulting from the context constraint for the derivation depicted in the upper part of Figure 3.10.

The following inequalities result from the consistency rule applications $\alpha_2, \alpha_3, \alpha_4$, and α_5 (each with cr_6 of MethodRule) where all of them require the marked elements and the created correspondence by α_1 (with cr_1 of ClassRule) as context. Each line below, therefore, represents $\mathsf{contextSrcTrg}(\alpha_i)$ and $\mathsf{contextCorr}(\alpha_i)$ at the same time where $2 \leq i \leq 5$:

- $x_2 \leq x_1$
- $x_3 \leq x_1$
- $x_4 \leq x_1$
- $x_5 \leq x_1$

Furthermore, the following inequalities result from the consistency rule applications α_6 and α_7 (with cr_7 of ParameterRule) requiring a correspondence created by α_4 and α_5, respectively. Hence, the lines below represent $\mathsf{contextCorr}(\alpha_6)$ and $\mathsf{contextCorr}(\alpha_7)$:

- $x_6 \leq x_4$
- $x_7 \leq x_5$

Finally, we discuss $\mathsf{contextSrcTrg}(\alpha_6)$ and $\mathsf{contextSrcTrg}(\alpha_7)$ which actually do not provide any logical significance to our set of inequalities. For example, the consistency rule application α_6 requires the upper Java method (which is marked by α_2 and α_4) and the upper UML method (which is marked by α_3 and α_4). Accordingly, we get $x_6 \leq x_2 + x_4$ and $x_6 \leq x_3 + x_4$ for $\mathsf{contextSrcTrg}(\alpha_6)$

whereas both of them are already implied by $x_6 \leq x_4$ above and can thus logically be omitted.

In general, it is most likely that contextCorr constraints already suffice to determine which consistency rule applications actually imply which other ones without considering contextSrcTrg constraints. It is, however, a case-specific matter depending on concrete rules and models whether such logical simplifications can be applied, while Definition 22 is formulated in the most general and conservative way (we also choose this conservative strategy in our current implementation for simplicity).

The constraint context(d) ensures that the context for each chosen consistency rule application is supplied but *cycles* must still be avoided. Intuitively, two selected rule applications may not provide correspondences and/or markings for each other (also not transitively) as such rule applications cannot be sequenced to a derivation in terms of the underlying TGG rules. In other words, a partial order between chosen consistency rule applications must exist with respect to their required elements.

Definition 23 (Cyclic Markings)**.**
Given a triple graph $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$, a set $\mathcal{CR}$ of consistency rules, and a derivation $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ with $cr_1, \ldots, cr_n \in \mathcal{CR}$, let $\mathcal{D}$ be the set of all rule applications in d. We define the relations $\lhd^C, \lhd^{ST}, \lhd \subseteq \mathcal{D} \times \mathcal{D}$ between two rule applications $\alpha_i, \alpha_j \in \mathcal{D}$ as follows:

- $\alpha_i \lhd^C \alpha_j$ if requiredCorr(α_i) $\cap$ createdCorr(α_j) $\neq \emptyset$,
- $\alpha_i \lhd^{ST} \alpha_j$ if requiredSrcTrg(α_i) $\cap$ marked(α_j) $\neq \emptyset$,
- $\alpha_i \lhd \alpha_j$ if $(\alpha_i \lhd^C \alpha_j) \vee (\alpha_i \lhd^{ST} \alpha_j)$.

We refer to a sequence $cy \subseteq \mathcal{D}$ of rule applications with $cy = \{\alpha_i, \ldots, \alpha_{i+k}\}$ as a *cycle* if $\alpha_i \lhd \ldots \lhd \alpha_{i+k} \lhd \alpha_i$.

Definition 24 (Constraint 3: Eliminating Cycles)**.**
Given a triple graph $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$, a set $\mathcal{CR}$ of consistency rules, and a derivation $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ with $cr_1, \ldots, cr_n \in \mathcal{CR}$, let $\mathcal{D}$ be the set of all rule applications in d and let $\mathcal{CY}$ be the set of all cycles $cy \subseteq \mathcal{D}$. We define a constraint acyclic(d) as follows:
acyclic(d) $= \bigwedge_{\substack{cy \in \mathcal{CY}, \\ cy = \{\alpha_i, \ldots, \alpha_{i+k}\}}} x_i + \ldots + x_{i+k} < |cy|$ where $|cy|$ is the cardinality of cy.

Example 16. The derivation in the upper part of Figure 3.10 does not contain any cycles. The $\lhd$ relation forms in this case a partial order between the consistency rule applications (in particular, we get $\alpha_2 \lhd \alpha_1$, $\alpha_3 \lhd \alpha_1$, $\alpha_4 \lhd \alpha_1$, $\alpha_5 \lhd \alpha_1$, $\alpha_6 \lhd \alpha_4$, and $\alpha_7 \lhd \alpha_5$). Hence, acyclic(d) is trivially satisfied.

In most cases, cycles over the $\lhd$ relation between chosen consistency rule applications are already avoided by satisfying markedAtMostOnce and context constraints. There exist, however, some corner cases where these constraints do not suffice. We, therefore, make an exception at this point by leaving our running example and resort to an abstract example demonstrating this.

In Figure 3.11, a TGG rule r is given that requires a source and target vertex of type S and T, respectively. Different than our exemplary TGG rules so far, no correspondence is required as context between these context vertices. Finally, r creates a source and target vertex (again of type S and T) both with an outgoing edge. A correspondence between the new vertices is created as well. The respective consistency rule cr accordingly marks these source and target vertices with their outgoing edges.

Considering now a model pair $G_0 : G_S \leftarrow \emptyset \rightarrow G_T$ where both G_S and G_T are cycle graphs with two vertices, a derivation d with cr can mark all vertices and edges (depicted at the bottom of Figure 3.11) with the two consistency rule applications α_1 and α_2. As α_1 and α_2 do not overlap in their marked elements, the markedAtMostOnce(G_0) constraint is satisfied by d. Moreover, the context(d) constraint is also satisfied as α_1 marks the required elements of α_2 and, reversely, α_2 marks the required elements of α_1.

While it can trivially be concluded that r alone cannot create any triple graph starting with the empty triple graph, the given model pair seems to be entirely marked using cr. The two rule applications α_1 and α_2, however, mark the required elements for each other (we get $\alpha_1 \lhd^{ST} \alpha_2$ and $\alpha_2 \lhd^{ST} \alpha_1$ which consequently leads to $\alpha_1 \lhd \alpha_2$ and $\alpha_2 \lhd \alpha_1$). That is, they actually cannot be sequenced to a derivation in the sense of the original TGG rule r. Accordingly, α_1 and α_2 cannot be chosen at the same time, i.e., we get $x_1 + x_2 < 2$ as the acyclic(d) constraint.

Satisfying the acyclic(d) constraint in this concrete case means that, if one of α_1 and α_2 is chosen, there must be some other consistency rule applications (obviously with some consistency rules other than cr) which mark the required source and target elements.

Summarizing our constraints so far, we enforce the following conditions:

- each source and target element in a model pair is marked at most once (markedAtMostOnce in Definition 21),
- each chosen consistency rule application completely satisfies its context, i.e., markings of required source and target elements as well as creations of required correspondences are provided by some other chosen consistency rule applications (context in Definition 22),
- chosen consistency rule applications do not provide markings and/or correspondences for each other in a cyclic manner (acyclic in Definition 24).

Given a set $\mathcal{D}$ of consistency rule applications (which do not necessarily satisfy these constraints), we refer to each subset $\mathcal{D}' \subseteq \mathcal{D}$ that satisfies the constraints as a *proper subset* of $\mathcal{D}$.

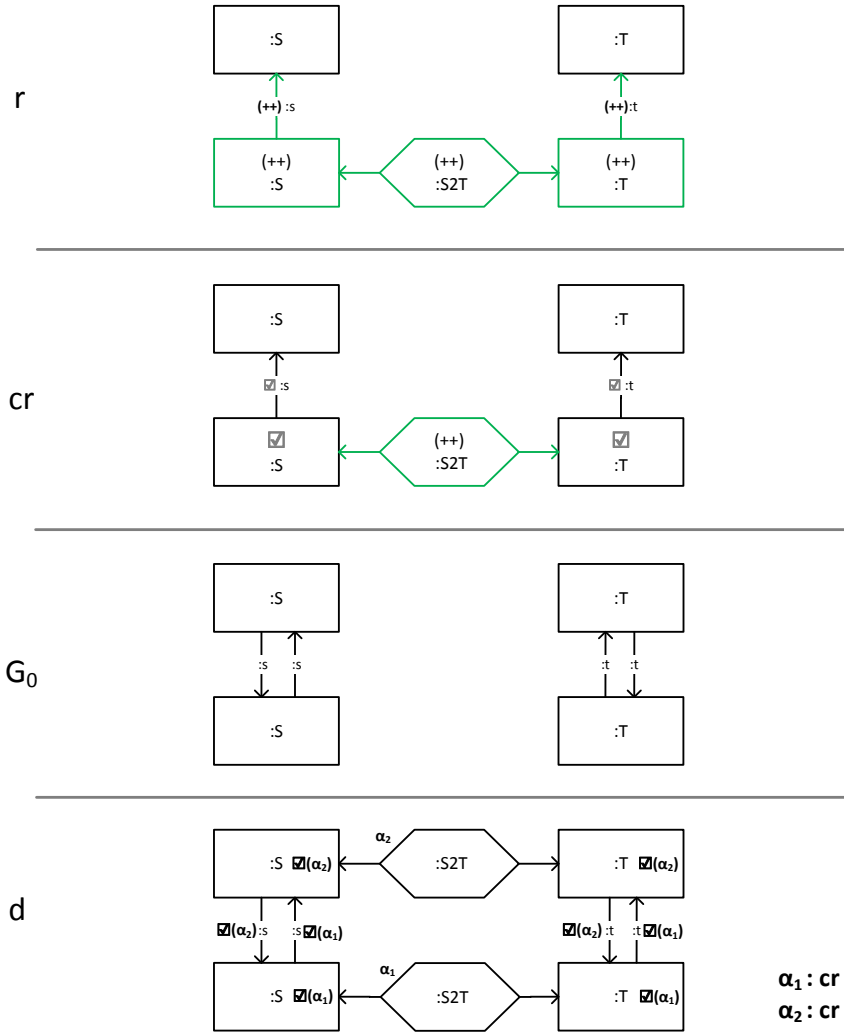


Figure 3.11: From top to bottom; a TGG rule r, the respective consistency rule cr, a model pair G_0, and a derivation d with cr where $\alpha_1 \triangleleft \alpha_2$ and $\alpha_2 \triangleleft \alpha_1$

Definition 25 (Proper Subset of Consistency Rule Applications)**.**
Given a TGG with the set $\mathcal{CR}$ of the respective consistency rules, a triple graph $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$, and a derivation $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ with $cr_1, \ldots, cr_n \in \mathcal{CR}$, let $\mathcal{D}$ be the set of all rule applications in d. We refer to any subset $\mathcal{D}' \subseteq \mathcal{D}$ with $\mathcal{D}' \vdash \mathsf{markedAtMostOnce}(G_0) \wedge \mathsf{context}(d) \wedge \mathsf{acyclic}(d)$ as a *proper subset* of $\mathcal{D}$.

Lemma 2 in the following states that, given a model pair $G_0 : G_S \leftarrow \emptyset \rightarrow G_T$ and a derivation d starting from G_0 with some consistency rules of a TGG, each proper subset of consistency rule applications in d leads to a triple graph representing a consistent portion of G_S and G_T. This consistent portion consists of source and target elements that are marked by the chosen rule applications in the proper subset. Inversely, for each consistent portion of G_S and G_T, a proper subset of consistency rule applications exists marking this consistent portion.

Lemma 2 (Consistent Portions of Source and Target Graphs)**.**
Given a TGG with the set $\mathcal{CR}$ of the respective consistency rules, a triple graph $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$, and a derivation $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ with $cr_1, \ldots, cr_n \in \mathcal{CR}$, let $\mathcal{D}$ be the set of all rule applications in d.
$\exists\ \mathcal{D}' \subseteq \mathcal{D} \iff \exists\ G' \in \mathcal{L}(TGG)$ with $G' = G'_S \leftarrow G_C \rightarrow G'_T$ fulfilling the following properties:

- $\mathcal{D}'$ is a proper subset of $\mathcal{D}$,
- $\mathsf{elements}(G'_S) \subseteq \mathsf{elements}(G_S)$ and $\mathsf{elements}(G'_T) \subseteq \mathsf{elements}(G_T)$,
- $\mathsf{elements}(G'_S) \cup \mathsf{elements}(G'_T) = \bigcup_{\alpha \in \mathcal{D}'} \mathsf{marked}(\alpha)$,
- $\mathsf{elements}(G_C) = \bigcup_{\alpha \in \mathcal{D}'} \mathsf{createdCorr}(\alpha)$.

Proof. Existence of a proper subset $\mathcal{D}'$ of $\mathcal{D}$ is equivalent to the existence of a derivation d' consisting of the consistency rule applications in $\mathcal{D}'$ (sequenced over the $\lhd$ relation). Our constraints, moreover, logically describe the properties of creation and context preserving derivations. In particular, we get:

- $\mathcal{D}' \vdash \mathsf{markedAtMostOnce}(G_0) \iff d'$ is creation-preserving,
- $\mathcal{D}' \vdash \mathsf{context}(d) \wedge \mathsf{acyclic}(d) \iff d'$ is context-preserving.

According to Lemma 1, the existence of d' is equivalent to the existence of a triple graph $G' = G'_S \leftarrow G_C \rightarrow G'_T$ with $G' \in \mathcal{L}(TGG)$ which consists of the elements marked and created by d'. Finally, $\mathsf{elements}(G'_S) \subseteq \mathsf{elements}(G_S)$ and $\mathsf{elements}(G'_T) \subseteq \mathsf{elements}(G_T)$ hold as consistency rule applications in $\mathcal{D}'$ and consequently in d' solely mark source and target elements in $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$. □

With Lemma 2, we have captured the constraint part of our optimization problem such that a creation and context preserving derivation can be built via a proper subset of all collected consistency rule applications. This, however, still does not cover our optimization goal with respect to detecting the *maximal* consistent portion of two models. In an extreme case, choosing none of the collected consistency rule applications is also a valid instantiation of Lemma 2 resulting in the empty triple graph as the detected consistent portion of a model pair.

Our next step, therefore, is to define the quality of proper subsets of consistency rule applications, referred to as *fitness* in the following definition. To calculate the fitness value of a proper subset of consistency rule applications, we consider each chosen consistency rule application α and sum up the cardinalities of $\mathsf{marked}(\alpha)$ (denoted as $|\mathsf{marked}(\alpha)|$ in the definition). We furthermore define proper subsets with the maximal fitness value among all proper subsets. Note that more than one proper subset might exist with the maximal fitness value. Our upcoming statements hold irrespectively of which one is selected by linear optimization.

Definition 26 (Maximal and Proper Subset of Consistency Rule Applications)**.** Given a TGG with the set $\mathcal{CR}$ of the respective consistency rules, a triple graph $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$, and a derivation $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ with $cr_1, \ldots, cr_n \in \mathcal{CR}$, let $\mathcal{D} = \{\alpha_1, \ldots, \alpha_n\}$ be the set of all rule applications in d. For each proper subset $\mathcal{D}'$, we define an integer value $\mathsf{fitness}(\mathcal{D}') = \sum_{i=1}^{n} k_i * x_i$ such that $k_i = |\mathsf{marked}(\alpha_i)|$, $x_i = 1$ if $\alpha_i \in \mathcal{D}'$, and $x_i = 0$ otherwise. We refer to $\mathcal{D}'$ as a *maximal and proper* subset of $\mathcal{D}$ if there does not exist any proper subset $\mathcal{D}''$ of $\mathcal{D}$ such that $\mathsf{fitness}(\mathcal{D}'') > \mathsf{fitness}(\mathcal{D}')$.

Example 17. Considering the derivation in the upper part of Figure 3.10, the integer value $\mathsf{fitness}(\mathcal{D}')$ for each proper subset $\mathcal{D}'$ of $\mathcal{D}$ is given by

$$2 * x_1 + 4 * x_2 + 4 * x_3 + 4 * x_4 + 4 * x_5 + 4 * x_6 + 4 * x_7$$

where $x_i = 0$ or $x_i = 1$ depending on whether α_i is contained in $\mathcal{D}'$. The weights (2 for x_1 and 4 for all other integer variables) result from the number of marked elements for each individual consistency rule application (we handle vertices and edges equally). The proper subset depicted in the lower part of Figure 3.10 marks all source and target elements (overall 18 elements). The proper subset, therefore, is maximal in the sense of markings.

When a proper subset $\mathcal{D}'$ of consistency rule applications is maximal in the sense of the number of marked elements as required in Definition 26, the triple graph yielded from $\mathcal{D}'$ according to Lemma 2 is of special interest.

Definition 27 (Maximally Marked Triple Graph)**.**
Given a TGG with the set $\mathcal{CR}$ of the respective consistency rules, a triple graph $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$, and a derivation $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ with $cr_1, \ldots, cr_n \in \mathcal{CR}$, let $\mathcal{D}$ be the set of all rule applications in d and let $\mathcal{D}'$ be a maximal and proper subset of $\mathcal{D}$. We refer to the triple graph $G' : G'_S \leftarrow G_C \rightarrow G'_T$ yielded from $\mathcal{D}'$ according to Lemma 2 as a *maximally marked triple graph* with respect to d.

We are now ready to state one of the main formal results of this thesis, namely *a sufficient condition for consistency* of two models with respect to a given TGG.

Theorem 1 (A Sufficient Condition for Consistency)**.**
Given a TGG with the set $\mathcal{CR}$ of the respective consistency rules, a triple graph $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$, and a derivation $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ with $cr_1, \ldots, cr_n \in \mathcal{CR}$, let $G'_S \leftarrow G_C \rightarrow G'_T$ be a maximally marked triple graph with respect to d. G_S and G_T are consistent if $G_S = G'_S$ and $G_T = G'_T$.

Proof. This is an instantiation of Lemma 2 where the proper subset $\mathcal{D}'$ is maximal in the number of marked elements (Definition 26). As $G' \in \mathcal{L}(TGG)$ (Lemma 2), we can conclude the consistency of G_S and G_T if $G'_S = G_S$ and $G'_T = G_T$. □

Example 18. Considering the seven consistency rule applications depicted in the upper part of Figure 3.10, the chosen five rule applications in the lower part form a maximal and proper subset. The chosen consistency rule applications mark the models entirely, hence consistency is concluded.

Theorem 1 lets us conclude consistency from *arbitrarily collected* consistency rule applications in a derivation d. Note that we do not yet state or require any properties for d (but rather for a chosen subset of consistency rule applications in d). While this makes Theorem 1 independent of any strategy for applying consistency rules, our condition is *not* sufficient *and necessary* due to a missing characterization of d. If consistency cannot be concluded from d according to Theorem 1, it is unclear if the models are really inconsistent or if there are some further consistency rule applications that were not collected in d. That is, inconsistency cannot be concluded.

Although a sufficient condition for consistency stated over incompletely applied consistency rules is useful in most cases, lifting the result to a sufficient and necessary condition is a desirable supplement from a formal point of view. First and foremost, "all possible" consistency rule applications that potentially can be a part of a creation and context preserving derivation must be collected for being able to conclude inconsistency via an optimization problem. We, therefore, restrict ourselves to TGGs where applying a consistency rule over the "same" match multiple times does not enable new markings. In particular, we forbid TGG rules of the form depicted in Figure 3.12 which do not create any source and target elements but only correspondences. In such a case, the respective consistency rule looks the same and does not contribute any markings but only correspondences. Applying consistency rules of this form multiple times over the same match does not violate any of our constraints but can yield some new matches for other consistency rules after arbitrarily many repetitions. This is exactly what we want to avoid!

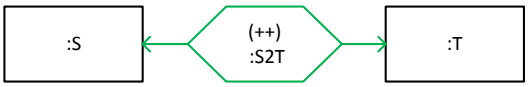


Figure 3.12: A TGG rule that does not create any source or target elements (the respective consistency rule looks the same)

Thus, the process of applying consistency rule applications must be *progressive* such that each distinct consistency rule should mark at least one element (hence the word choice in the following definition).

Definition 28 (Progressive TGGs)**.**
We refer to a TGG with the set $\mathcal{CR}$ of consistency rules as *progressive* if each consistency rule $cr \in \mathcal{CR}$ has at least one marking element.

In a progressive TGG, repeating consistency rule applications over the same match with the same consistency rule does not bring any significant contribution to the optimization problem stated over the collected consistency rule applications. Hence, we consider derivations that contain each possible consistency rule application over a distinct match only once.

A critical problem of technical nature, however, arises when the process applying consistency rules over distinct matches does not terminate. In a consistency checking run, the correspondence graph is the only one evolving after consistency rule applications. When consistency rules, however, continuously produce new correspondences for each other (or for themselves) in a cyclic manner, there exists always a distinct match which has not yet been considered (and actually does not need to be considered in a progressive TGG due to repeated markings). Such a situation is depicted in Figure 3.13 based on our running example.

Both the Java and the UML model in Figure 3.13 have two classes (List and Queue) inheriting from each other via a superClass reference. These models are neither valid in Java and UML nor producible with our TGG (but assume that our goal is to find out the latter). The derivation depicted in Figure 3.13 first marks the List classes via the consistency rule cr_1 of ClassRule. The Queue classes are then marked as subclasses via the consistency rule cr_5 of SubclassRule. Due to the created correspondence between the Queue classes, however, this leads to a new match for marking List classes this time as subclasses via the same rule. This again leads to a new match for marking Queue classes as subclasses (this time under a new correspondence). Hence, each new correspondence leads to a new match for cr_5 and applying consistency rules does not terminate if the process naïvely reacts to each distinct match (occurring due to a new correspondence).

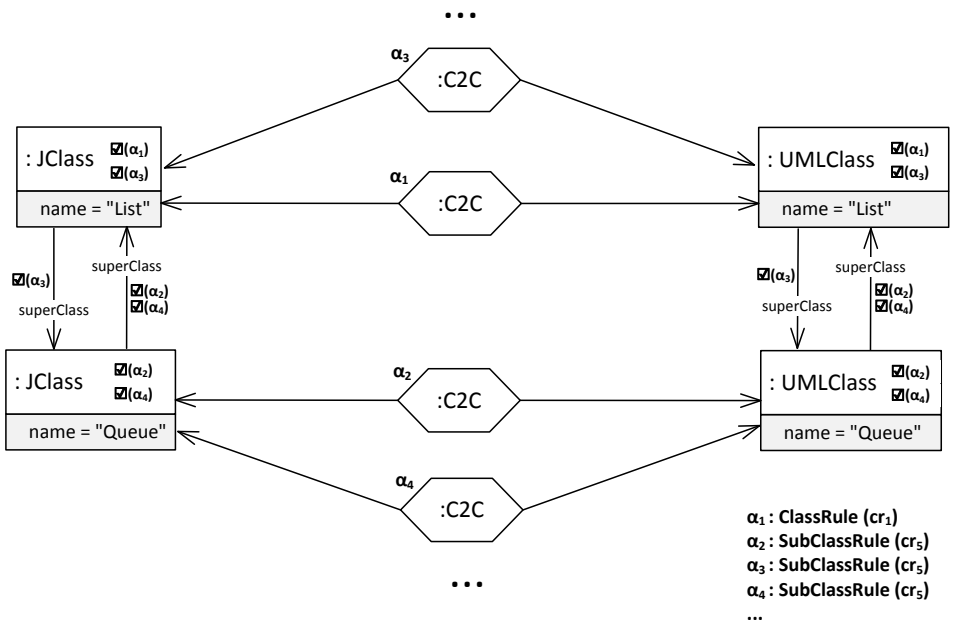


Figure 3.13: A derivation with consistency rules where the search for all possible rule applications does not terminate

The problem, in fact, constitutes a special case of what we have defined as a cycle in Definition 23: A consistency rule application matches the created correspondence of another consistency rule application but marks the required source/target elements thereof. Eliminating repeated and superfluous markings in Figure 3.13 would not be a challenge from a logical point of view as our markedAtMostOnce and context constraints already suffice (even without the acyclic constraint). From

a graph grammar point of view, however, the search for possible rule applications must be adjusted to avoid a non-terminating rule application process.

In order to avoid termination problems, we need to restrict which consistency rule applications are really to be performed. In the following definition, therefore, we distinguish between *essential* and *superfluous* consistency rule applications. Intuitively, a consistency rule application α is essential for a derivation d if

- there does not already exist a consistency rule application in d over the same match with the same consistency rule,
- α does not imply other consistency rule applications due to correspondences while excluding them at the same time.

In Figure 3.13, for example, α_3 violates the second condition as it implies α_1 (we get $\alpha_3 \leq \alpha_2 \leq \alpha_1$ due to the contextCorr constraints) while marking the same elements as α_1 (we get $\alpha_1 + \alpha_3 \leq 1$ due to the markedAtMostOnce constraints). Overall, no proper subset of these consistency rule applications can contain α_3, i.e., α_3 is not essential but superfluous.

Definition 29 (Essential and Superfluous Consistency Rule Applications)**.** Given a triple graph $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$, a set $\mathcal{CR}$ of consistency rules, and a derivation $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ with $cr_1, \ldots, cr_n \in \mathcal{CR}$, let $\mathcal{D}$ be the set of all rule applications in d and let $\lhd^C_* \subseteq \mathcal{D} \times \mathcal{D}$ be the transitive closure of the $\lhd^C$-relation in Definition 23.
We refer to a consistency rule application $\alpha_{n+1} : G_n \overset{cr_{n+1}@cm_{n+1}}{\Longrightarrow} G_{n+1}$ with $cr_{n+1} \in \mathcal{CR}$ as *essential* for d if:

- $\nexists\, \alpha_i \in \mathcal{D}$, $\alpha_i : G_{i-1} \overset{cr_i@cm_i}{\Longrightarrow} G_i$, such that $cr_{n+1} = cr_i$ and $cm_{n+1}(G_n) = cm_i(G_{i-1})$, and
- $\mathsf{marked}(\alpha_{n+1}) \cap \bigcup_{\substack{\alpha \in \mathcal{D},\\ \alpha_{n+1} \lhd^C_* \alpha}} \mathsf{marked}(\alpha) = \emptyset$.

We refer to α_{n+1} as superfluous otherwise.

Example 19. The consistency rule applications α_1 and α_2 in Figure 3.13 are essential while α_3 and α_4 are superfluous due to the second condition in Definition 29. In the case of α_3, for example, we get $\alpha_3 \lhd^C \alpha_2$ and $\alpha_2 \lhd^C \alpha_1$ leading to $\alpha_3 \lhd^C_* \alpha_1$. However, α_3 marks the same Java and UML classes as α_1, i.e., α_3 excludes α_1 while implying it at the same time and can never be chosen to satisfy our constraints.

Superfluous consistency rule applications constitute a "stopping criterion" when applying consistency rules. We refer to a derivation d via consistency rules as *final* if any further consistency rule application that can be added to d is superfluous.

Definition 30 (Final Derivations with Consistency Rules)**.**
Given a triple graph $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$, a set $\mathcal{CR}$ of consistency rules, let $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ be a derivation with $cr_1, \ldots, cr_n \in \mathcal{CR}$.
We refer to d as *final* if $\forall\ \alpha_{n+1} : G_n \overset{cr_{n+1}@cm_{n+1}}{\Longrightarrow} G_{n+1}$ with $cr_{n+1} \in \mathcal{CR}$ it holds that α_{n+1} is superfluous for d.

Before lifting Theorem 1 to a sufficient and necessary condition given a final derivation, we first state in the following lemma that final derivations exist for all model pairs $G_S \leftarrow \emptyset \rightarrow G_T$ in a progressive TGG. Practically, existence of a final derivation leads us to the notion of termination when applying essential consistency rules over distinct matches.

Lemma 3 (Existence of a Final Derivation)**.**
Given a *progressive TGG* with the set $\mathcal{CR}$ of the respective consistency rules, for all triple graphs $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$ a *final* derivation $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ with $cr_1, \ldots, cr_n \in \mathcal{CR}$ exists.

Proof. As the correspondence graph is the only one that evolves when applying consistency rules, we must show that the number of essential consistency rule applications that require a correspondence is always limited. With regard to their required correspondences, essential consistency rule applications form sequences of the form $\alpha_i \lhd^C \ldots \lhd^C \alpha_j$ over the $\lhd^C$ relation. In the first part of the proof, we show that the number of such sequences with a particular length is always finite whereas the length itself is not necessarily bounded (and can go from 1 to infinite). In the second part, we show that a sequence of infinite length cannot be constructed with essential consistency rule applications.

Given that G_0 and $\mathcal{CR}$ have finite sizes, it can be proven by induction that the number of sequences $\alpha_i \lhd^C \ldots \lhd^C \alpha_j$ of length ℓ is limited for all $\ell \in [1, \infty)$.

> **Base case** ($\ell = 1$): A sequence α_i of length 1 can only be constructed by consistency rule applications that do not match any correspondences (but only elements from G_0). The number of consistency rules in $\mathcal{CR}$ that do not match any correspondence and the number of elements in G_0 are finite. We, moreover, consider only distinct matches in the case of essential consistency rule applications. The number of sequences α_i of length 1, therefore, is finite.
>
> **Induction step** (from ℓ to $\ell + 1$): Let $\mathcal{A}_\ell$ be the set of all consistency rule applications whose position in a sequence $\alpha_i \lhd^C \ldots \lhd^C \alpha_j$ is never greater than ℓ. If the number of sequences of lengths between 1 and ℓ is finite, $\mathcal{A}_\ell$ is finite. A sequence $\alpha_i \lhd^C \ldots \lhd^C \alpha_j \lhd^C \alpha_k$ of length $\ell + 1$ can only be constructed with a consistency rule application α_k that requires correspondences created by other consistency rule applications in $\mathcal{A}_\ell$. If $\mathcal{A}_\ell$ is finite, the number of these correspondences is finite. Having only distinct matches of consistency rule applications, therefore, the number of possible sequences $\alpha_i \lhd^C \ldots \lhd^C \alpha_j \lhd^C \alpha_k$ of length $\ell + 1$ is finite.

In the remaining part of the proof, we show that a sequence over the $\lhd^C$ relation has always a finite length for essential consistency rule applications. For

each sequence $\alpha_i \lhd^C \ldots \lhd^C \alpha_j$ of essential consistency rule applications, we define a sequence $M_i, \ldots, M_j$ of natural numbers calculated cumulatively via the sets of marked elements such that:

- $M_i = |\mathsf{marked}(\alpha_i)|$,
- $M_j = |\mathsf{marked}(\alpha_i) \cup \ldots \cup \mathsf{marked}(\alpha_j)|$

The sets $\mathsf{marked}(\alpha_i), \ldots, \mathsf{marked}(\alpha_j)$ are non-empty (as the TGG is progressive), while their pairwise intersections are empty (cf. the second condition in Definition 29). Hence, we get $M_i < \ldots < M_j$. While always getting greater, however, the sequence $M_i < \ldots < M_j$ is subject to an upper bound given by the overall number of source and target elements in G_0 (i.e., $M_j \leq |\mathsf{elements}(G_S) \cup \mathsf{elements}(G_T)|$). That is, not overlapping in the marked elements, the sequence $\alpha_i \lhd^C \ldots \lhd^C \alpha_j$ together cannot mark more than what is available in G_S and G_T. Consequently, the length of all sequences over the $\lhd^C$ relation is finite for essential consistency rule applications.

To sum up, essential consistency rule applications form a finite number of sequences over the $\lhd^C$ relation where the lengths of these sequences are finite as well. Hence, a final derivation always exists. □

We are now ready to lift our sufficient condition for consistency in Theorem 1 to a sufficient and necessary condition. Additional statements in Theorem 2 as compared to Theorem 1 are italicized.

Theorem 2 (A Sufficient and Necessary Condition for Consistency)**.**
Given a *progressive* TGG with the set $\mathcal{CR}$ of the respective consistency rules, a triple graph $G_0 = G_S \leftarrow \emptyset \rightarrow G_T$, and a *final* derivation $d : G_0 \overset{cr_1@cm_1}{\Longrightarrow} G_1 \ldots \overset{cr_n@cm_n}{\Longrightarrow} G_n$ with $cr_1, \ldots, cr_n \in \mathcal{CR}$, let $G'_S \leftarrow G_C \rightarrow G'_T$ be a maximally marked triple graph with respect to d. G_S and G_T are consistent *if and only if* $G_S = G'_T$ and $G_T = G'_T$.

Proof. If $G_S = G'_S$ and $G_T = G'_T$, consistency of G_S and G_T can be concluded using exactly the same arguments as in Theorem 1. We, therefore, must only show that inconsistency can be concluded if $G_S \neq G'_S$ or $G_T \neq G'_T$.

TGG is progressive and d is final. Any further consistency rule application $\alpha_{n+1} : G_n \overset{cr_{n+1}@cm_{n+1}}{\Longrightarrow} G_{n+1}$, therefore, is superfluous where at least one of the following arguments holds for not choosing α_{n+1} (due to the two conditions stated in Definition 29).

- α_{n+1} violates the first condition Definition 29: There already exists a consistency rule application α_i that contributes the same markings over the same match. In particular, we get $x_i + x_{n+1} \leq 1$ due to the markedAtMostOnce constraints, i.e., α_i and α_{n+1} cannot be chosen at the same time. Considering the markings of α_i and α_{n+1} required by another consistency rule application α_j, furthermore, all respective contextSrcTrg constraints are of the form $x_j \leq x_i + \ldots + x_{n+1}$, i.e., choosing α_{n+1} satisfies exactly the same contextSrcTrg constraints as choosing α_i. Moreover, if any subsequent consistency rule

application α_{n+2} requires a correspondence created by α_{n+1}, i.e., $x_{n+2} \leq x_{n+1}$, there already exists a consistency rule application α_k that requires a correspondence created by α_i and differs from α_{n+2} only in its required correspondences, i.e., $\mathsf{marked}(\alpha_k) = \mathsf{marked}(\alpha_{n+2})$ and $x_k \leq x_i$. Hence, choosing α_{n+2} and consequently α_{n+1} due to the contextCorr constraints marks exactly the same elements as choosing α_k and consequently α_i.

- α_{n+1} violates the second condition Definition 29: There exists a consistency rule application α_i implied as well as excluded by α_{n+1}. In particular, α_{n+1} marks some elements that are marked by α_i as well but relies on some correspondences created by α_i (i.e., $\alpha_{n+1} \lhd^C_* \alpha_i$). As a result, we get $x_i + x_{n+1} \leq 1$ due to the markedAtMostOnce constraints and $x_{n+1} \leq \ldots \leq x_k \leq \ldots \leq x_i$ (or directly $x_{n+1} \leq x_i$ if $\alpha_{n+1} \lhd^C \alpha_i$) due to the contextCorr constraints. These inequalities, however, cannot be satisfied with $x_{n+1} = 1$. Hence, α_{n+1} cannot be chosen at all.

Consequently, choosing α_{n+1} cannot increase the number of marked elements while satisfying our constraints at the same time. Hence, all possible consistency rule applications that can be part of a creation and context preserving derivation are exhaustively contained in d. If $G_S \neq G'_S$ or $G_T \neq G'_T$ while $G'_S \leftarrow G_C \rightarrow G'_T$ is maximally marked, therefore, the absence of a proper subset $\mathcal{D}'$ of $\mathcal{D}$ that entirely marks G_S and G_T can be concluded (Lemma 2). This, finally, leads to the absence of a derivation via original TGG rules that would create G_S and G_T together as markings via consistency rules and creations via TGG rules bijectively exist (Lemma 1).

The arguments apply to all triple graphs $G_S \leftarrow \emptyset \rightarrow G_T$ as a final derivation exists in all cases (Lemma 3). □

Example 20. The derivation depicted in Figure 3.10 is final and exemplifies not only Theorem 1 but also Theroem 2 when concluding consistency. We show a further example in Figure 3.14 where this time inconsistency is concluded. In the example, the UML model has only one remove method, while the Java model has two remove methods (the question then which one of the two is a better choice to be related to the UML method).

In the upper part of Figure 3.14, a derivation with four consistency rule applications is depicted which, again, is final (i.e., any further consistency rule application would only repeat existing ones). We also explicitly show the integer inequalities (our constraints) and the objective function (the fitness of a proper subset) that is to be maximized. These form the input for the ILP solving step whose outcome is a proper subset consisting of three rule applications. The outcome of the optimization reveals that choosing the upper Java method is a beter choice leading to a greater number of marked elements. Having unmarked elements after the optimization step, however, models are indicated to be inconsistent. The marked portions, nevertheless, represent a maximally marked triple graph for this model pair (Definition 27) which is a consistent triple graph (Lemma 2). The unmarked elements, finally, require action for consistency restoration (the focus of our next contribution).

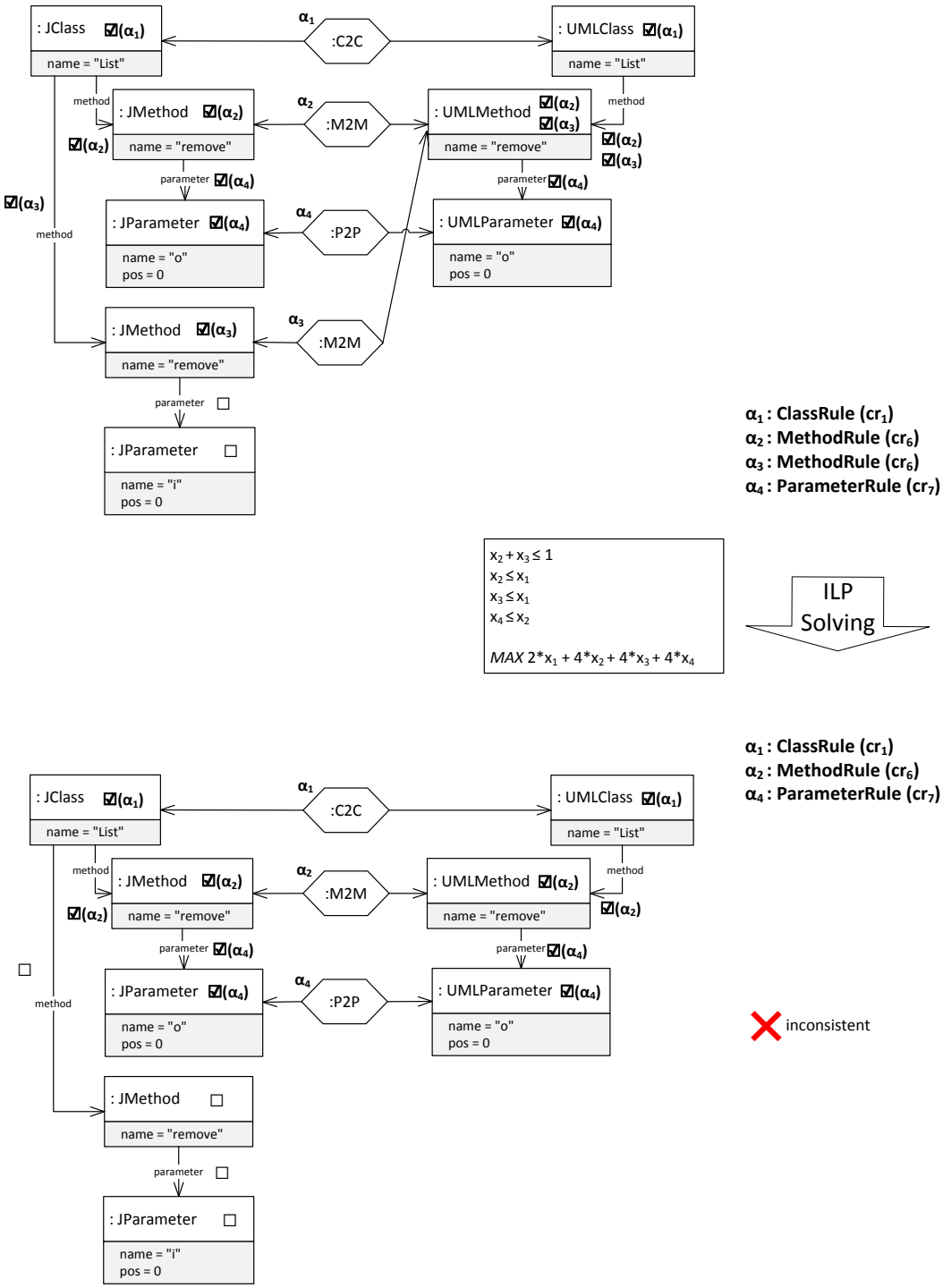


Figure 3.14: A schematic exemplification of our approach with inconsistent models

Next, we provide a consistency checking procedure in Algorithm 1 that basically summarizes our formal results in a procedural overview. The inputs are the consistency rules of a progressive TGG and a triple graph of the form $G_S \leftarrow \emptyset \rightarrow G_T$ (and the goal is a consistency checking between G_S and G_T). The outputs are threefold: A boolean value stating whether G_S and G_T are consistent, a triple graph $G'_S \leftarrow G_C \rightarrow G'_T$ representing the maximal consistent portions of G_S and G_T together with correspondences, and also a derivation d' stating how these consistent portions can be created by the TGG. The latter represents a "consistency history" stating how the (possibly partial) consistency between G_S and G_T is induced (this information will be later used for consistency restoration purposes in the next section).

Algorithm 1 makes use of Theorem 2 and has basically the following two phases:

- **Phase 1** (Line 3-10): Consistency rules are applied to the input triple graph until no more essential consistency rule application exists. Hence, the resulting derivation d is final (in the sense of Definition 30).
- **Phase 2** (Line 12-20): Making choices among consistency rule applications involved in d is captured in this phase. In particular, Line 13 represents the linear optimization step where a maximal and proper subset of the collected consistency rule applications are calculated (Definition 26). This yields the individual components of the output including a derivation d' and the triple graph $G'_S \leftarrow G_C \rightarrow G'_T$ (induced according to Lemma 2). Finally, it is examined if G'_S and G'_T entirely covers G_S and G_T, respectively.

Note that the correctness of Algorithm 1 and its termination are direct consequences of Theorem 2 and Lemma 3, respectively.

In particular, Theorem 2 (and consequently Algorithm 1) relies on an exhaustive search of consistency rule applications. A strong distinction must be made, however, between our approach and a *brute force* approach that enumerates all possible derivations with consistency rules. We only require an exhaustive search of distinct consistency rule applications but not derivations. Consequently, applying consistency rules repeatedly for obtaining (probably only slightly) different derivations does not represent our strategy. Instead, we build one (possibly large) final derivation by applying consistency rules in a progressive manner and calculate a subset of collected consistency rule applications based on linear optimization.

We shall discuss and evaluate the applicability of combining TGGs and ILP based on a concrete implementation as part of our practical contribution. Nevertheless, some expectations and observations in terms of computability and complexity theory still merit a closer look on the conceptual basis.

For indicating (in-)consistency of two graphs, we restrict ourselves to progressive TGGs that always mark some source and / or target elements. Our motivation behind introducing progressive TGGs is to avoid a potentially infinite evolvement of the correspondence graph when marking a given pair of source and target graphs (in other words, we want to avoid the need for anticipating and performing arbitrary computations in the correspondence graph). This is, in fact, a classical example where Turing-completeness of a rule-based mechanism is sacrificed for a viable rule application process. The reward we get for this is the notion of essential consistency

Algorithm 1 Consistency Checking

Require:

TGG : A progressive TGG

$\mathcal{CR}$: Consistency rules of TGG

$G_0 : G_S \leftarrow \emptyset \rightarrow G_T$

1: **procedure** CheckConsistency(G, $\mathcal{CR}$)
2:
3: $d \leftarrow$ empty derivation ▷ **Phase 1**
4: $G \leftarrow G_0$
5:
6: **while** $\exists\, \alpha : G \overset{cr@cm}{\Longrightarrow} G'$ with $cr \in \mathcal{CR}$ and α is essential for d **do**
7: perform α
8: $d \leftarrow$ add α to d
9: $G \leftarrow G'$
10: **end while**
11:
12: $\mathcal{D} \leftarrow$ all consistency rule applications in d ▷ **Phase 2**
13: $\mathcal{D}' \leftarrow$ maximal and proper subset of $\mathcal{D}$
14: $d' \leftarrow$ derivation consisting of consistency rule applications in $\mathcal{D}'$
15: $(G'_S \leftarrow G_C \rightarrow G'_T) \leftarrow$ triple graph consisting of marked elements and created correspondences in d'
16:
17: **bool** consistency
18: **if** $G'_S = G_S$ and $G'_T = G_T$ **then** consistency $\leftarrow$ **true**
19: **else** consistency $\leftarrow$ **false**
20: **end if**
21:
22: **return** (consistency, $(G'_S \leftarrow G_C \rightarrow G'_T)$, d')
23:
24: **end procedure**

rule applications (Definition 29) and consequently final derivations (Definition 30). Having shown that a final derivation exists in all cases, the rule application part of our approach always exits after performing a finite number of consistency rule applications (without any repetitions). Linear optimization via 0-1 ILP, moreover, forms the second subtask that is effectively solvable. Hence, consistency checking with TGGs is a decidable problem.

Before discussing the complexity of our solution, we first must understand the hardness of the formulated problem (in terms of complexity theory). First of all, consistency checking with a progressive TGG belongs to the complexity class NP as a proof of consistency (e.g., a derivation building up a given graph pair) is of polynomial length. In fact, the size of the derivation never exceeds the size of the input graph pair as each rule creates at least one element. We, however, have an NP-complete problem (for which there is no known way to locate a solution efficiently). An indicative argument for this can be inferred from reducing subgraph isomorphism check (which is known to be NP-complete) to consistency checking. Assume a TGG describing consistency as subgraph isomorphism, e.g., some rules create isomorphic vertices and edges on both sides and some rules create additional elements only on the target side. Given a source graph G_S and a target graph G_T in such a case, G_S is a subgraph of G_T if and only if G_S and G_T are consistent.

Having inherently an NP-complete problem, our approach is not an attempt to come up with polynomial time complexity. First, the number of collected consistency rule applications can grow exponentially or even factorially in the size of the input graph pair (a factorial growth occurs, for example, when a TGG constructs sequences of vertices and consistency checking permutes all possible sequences in a given complete graph). Second, preparing integer inequalities and an objective function is polynomial in the size of a final derivation (we solely require the knowledge on which consistency rule application marks/creates which elements). Third, moreover, ILP solving is not a decision problem but an optimization problem and is NP-hard (going beyond NP-completeness). Nevertheless, our special case of linear optimization has an upper bound (given by the overall number of graph elements) and can thus be solved purely with integer inequalities. Given a graph pair with n elements, the sum of markings can be equated with n and, if not satisfiable, then with $n-1$, $n-2$, and so on until a feasible solution is found. This way, the problem can be solved with 0-1 ILP without any objective function which, again, is known to be NP-complete [80].

Reformulating an NP-complete problem as an optimization problem brings the following practical advantage in our case: We do not only answer consistency checking with a yes or no but also detect what is at least partially consistent in the case of no. As a final remark, therefore, it should be noted that we have pragmatic understanding of scalability and define it, as already mentioned in the introduction, as the capability of dealing with industry-sized consistency scenarios. The applicability of our approach relies on optimized tool support for graph transformation and ILP solving which shall be experimentally evaluated in the next section but one.

3.6 Related Work

We have already mentioned some important related work to motivate our consistency checking approach. Some lines of research, nevertheless, should be discussed in more detail to understand the novelty as well as the inspiration of our approach.

We first give an overview of consistency checking in the context of BX languages, in particular based on TGGs and QVT-R. Subsequently, custom solutions to traceability management and model differencing, which pursue similar goals to ours, are discussed. Finally, having combined linear optimization techniques with graph grammars in our formalization, similar combinations from the broader field of MDE merit mentioning (even if they are not necessarily meant for consistency checking purposes).

TGGs: Most closely, we consider our consistency checking approach as an extension to [43]. At a time when most of the research on TGGs had been conducted for consistency restoration, the new type of operational rules is introduced in [43] which, for the first time, takes two models as input and creates correspondences. More importantly, it has been shown that applying these rules is equivalent to applying original TGG rules, shaping the foundation for our intermediate results (in particular for our lemmas before incorporating optimization techniques). The search space problems in applying the operational rules, however, remain open and, in fact, go beyond the scope of [43]. Our work demonstrates these open issues and exploits linear optimization techniques to overcome them.

In addition to [43], yet another new type of operational rules is proposed in [68], referred to as *consistency creating rules*. In this case, all of the three models (i.e., not only the source and target but also the correspondence model) are taken as input and marked. Furthermore, partial derivations with these rules (partial markings when transferred to our setting) are discussed in [44]. Overall, these extensions intend to reuse existing correspondences in consistency checking (which might, e.g., have been created in previous runs). Reusing existing correspondences can potentially mitigate search space problems in consistency checking but do not clear them entirely, i.e., wrong decisions can still be taken when creating the remaining correspondences (and markings). Nevertheless, a combination of our approach and [44, 68] is worthwhile to consider as it can reduce rule application and optimization efforts. This can be achieved by incorporating applications of consistency creating rules in the sense of [44, 68] into our constraints and objectives.

QVT-R: The checkonly mode of QVT-R [120] is the only available standard as previously mentioned at the beginning of this section. The central elements of a QVT-R specification are *relations* consisting of a set of domain patterns (source and target patterns) describing what should hold given the occurrences of these patterns in a model pair. As a means for modularity, a relation can make use of and invoke other relations as pre- and post-conditions, referred to as *when* and *where* clauses, respectively. Similar to TGG rules, relations in QVT-R form nothing but a consistency description in the first place, though not a grammatical one but rather based on predicates (evaluating to true or false for a set of elements). Again similar to TGGs, therefore, an operationalization step is needed to perform consistency

checking. The standard proposes to translate QVT-R to *QVT Core*, a language that is intended to be "simpler" and to support explicit correspondences between domains (while QVT-R specifications are independent of any correspondence notion).

The translation from QVT-R to QVT Core is the least understood and most criticized part of QVT-R, and even its semantics preservation has been justifiably challenged due to discrepancies between the two languages [133]. Briefly summarizing the observations stated in [133], the most crucial discrepancy is due to how *bindings* are handled to satisfy relations. While QVT-R seems to be more relaxed specifying consistency over existence quantifiers (e.g., for all source patterns there exists a target pattern), mappings in QVT Core restrictively demand unique bindings (e.g., for all source patterns there exists a unique target pattern).

Having identified operationalization issues, the author in [133] furthermore proposes a game-theoretic approach for formalizing consistency checking based on a QVT-R specification. Consistency checking is a game between two players whose interests are either to verify or to refute consistency by choosing bindings for individual relations. Consistency is concluded if the verifier has a strategy with which they must necessarily win. While only non-recursive invocations among relations (over the when and where clauses) are considered in [133], the approach is later extended by capturing recursive invocations in [22].

At least two further approaches to consistency checking with QVT-R strive to bypass the translation to QVT Core and represent their own formalisms. In [103], QVT-R relations are translated to logical constraints. Accordingly, consistency checking (besides consistency restoration) is realized as constraint solving. Although we also make use of constraints in the form of integer inequalities, a distinction must be made: Our approach is rule-based in the first place and constraints are only formulated over rule applications to choose between alternatives. In [103], on the contrary, constraint solving is the main mechanism operating on individual model elements. This distinction practically has an influence on the processable model sizes (e.g., tens of elements versus thousands of elements). In [62], furthermore, QVT-R relations are translated to graph patterns which resemble our consistency checking rules but without correspondences. Satisfying the relations is then traced back to existence of morphisms (from graph patterns to a given model pair).

While these formalizations seminally cope with the ambiguities in the standard, we observe the further two general drawbacks of QVT-R (and advantages of TGGs) for consistency checking purposes:

- In QVT-R, consistency of two models is designed and checked separately in two directions (in our running example, this would mean that consistency from Java to UML is not necessarily the same matter as consistency from UML to Java). The main reason for this is that the checkonly mode of QVT-R is only considered as a special case (or an integral subtask) of model-to-model transformation which, however, has a direction. This is also reflected in the aforementioned formalizations of consistency checking. That is, matching the graph patterns in [62], solving the constraints in [103], or playing the game in [133] must be done in both directions if a symmetric conclusion of consistency is desired. Related to this point, TGGs provide a direction-agnostic notion of consistency which is checked once. We argue that this is

advantageous and easier to manage for all stakeholders in a BX landscape (Figure 1.7) including meta-tool developers, consistency tool developers, and finally, model owners.

- A QVT-R specification is independent of an explicit trace model. A concrete implementation, nevertheless, can introduce some internal trace information depending on the chosen formalism to realize consistency checking. The traces, however, cannot be read symmetrically in either direction as consistency checking itself is not symmetric. In TGGs, correspondences are vital for the symmetric consistency notion and serve as explicit traceability information between two models. Hence, further tools for traceability purposes (e.g., traceability matrices or change impact analyzers) can be built operating upon correspondences resulting from a consistency checking run.

Having TGGs (including previous works [43, 44, 68]) and QVT-R as the two main fronts in consistency checking in MDE, we conclude this discussion in Table 3.1 with our estimation of how our approach is seated in this landscape (before coming to the wider areas of related work).

	Reliability	**Traceability**	**Symmetry**	**Reuse of Traceability**
TGGs à la [43, 44, 68]	-	+	+	+
QVT-R à la [62, 103, 133]	+	-	-	-
this work	+	+	+	-

Table 3.1: Comparing our approach to related work based on TGGs and QVT-R

The individual columns in Table 3.1 represent different aspects of consistency checking ordered from left to right according to their importance in our belief. With regard to providing correct results for consistency checking, we adopt the term *reliability* from software engineering referring there to "continuity of correct service". Having addressed search space problems, the strength of our extensions to TGGs lies in reliable consistency checking. Accordingly, reliability of consistency checking is not sufficiently addressed in former TGG approaches [43, 44, 68] as the process might end up with wrong results due to the involved search space. Symmetric consistency checking with explicit traceability support forms the main advantages against QVT-R. Reusing traceability information from former runs as proposed in [44, 68], finally, seems to be the next logical step in further research.

Traceability Management: Notwithstanding that we name our ultimate goal as consistency checking, it is at the same time a means for managing traceability between two models by creating correspondences on the way. Reversely, we observe that custom traceability management approaches pursue similar goals to ours even if a consistency notion is not the centre of attention.

Traceability is usually associated with requirements engineering (with prominent tool support such as DOORS [38], Reqtify [125], or YAKINDU [152]), in particular for reasoning the existence of an element (be it a document, a product, or a component) according to requirements. Going beyond requirements engineering, there is an increasing awareness of the need for traceability management in any collaborative engineering environment. Consequently, there is a vast literature addressing

traceability practices mostly dedicated to a particular context such as, for example, software engineering or mechatronics.

For comprehensive surveys of traceability practices (that manage to remain general), we exemplarily refer to [1, 53, 145, 89, 147]. These surveys define different aspects to classify traceability approaches and also to identify relevant research directions. Some of the directions are indeed very helpful to locate consistency checking (or correspondence creation in this particular case) with TGGs in the context of traceability management. First and foremost, all considered surveys identify *creating traceability information* as an important challenge. Common practices to traceability usually rely on manual extraction of traceability links (by model owners) or strive to make suggestions to mitigate this substantial effort. In the case of our approach, we propose a fully automated mechanism that extracts correspondences via a combination of graph grammars and linear optimization. Development effort, of course, still exists and is shifted to consistency tool developers who should specify consistency in the form of TGG rules. This effort, nevertheless, is nonrecurring as soon as, or as long as, a TGG is finalized and in operation.

With the establishment of MDE, furthermore, new chances are expected to arise for capturing traceability information as models [1]. Indeed, providing a *traceability scheme* (which can be understood as a meta-model for traceability information in our setting) seems to be a desirable practice to semantically distinguish between trace links [1, 53, 89, 147]. This is similar to the idea of specifying a correspondence meta-model in TGGs to capture different mapping possibilities even for the same types of elements (for example, remember how we have used the two correspondence types C2C and C2C* to distinguish between two different ways of relating Java and UML classifiers). Again coupled with MDE, a distinction between horizontal traceability (within the PSM level or the PIM level) and vertical traceability (between PSMs and PIMs) is made in [145]. This relates to our vision for addressing horizontal and vertical consistency uniformly as motivated in Figure 1.1.

While TGGs can increase the level of automation in traceability management, existing traceability solutions reversely can inspire TGGs for user involvement. A good example is the ModelTracer approach [135] which is rule-based similar to TGGs but allows for manually created trace links. The proposed workflow strives to retain manually created trace links but can also discard these when contradicting the rules. Furthermore, *incremental* traceability management as illustrated by, e.g., [76, 102, 104] offers an appealing approach to updating traceability information. In general, however, we observe that traceability approaches usually lack an underlying formal foundation (focusing directly on concrete tool architectures and their features). A good exception, finally, is made in [58, 59] introducing algorithms in detail to map the vertices of two models pairwise using similarity functions. The formalization of models and their mappings, indeed, is based on the notion of graphs. While the approach considers only two vertices in each mapping step (one from the source and one from the target model), a TGG rule can be considered as a generalization of this going beyond two vertices.

Model Differencing: A relevant task in MDE is to compare two different versions of a model and to calculate what is changed from a version to another, referred to as model differencing. In fact, this can be considered as a special case of what

we call consistency checking assuming that consistency is defined as isomorphism between the source and target model (conforming to the same meta-model).

For surveys of model differencing approaches, we refer to [88, 131] both identifying four groups of model differencing approaches: *Identity-based* approaches [2, 91] assume a non-volatile and unique identifier feature for each element to govern the process of relating individual element pairs. *Signature-based* approaches [122, 130, 137] operate similarly but the identity is calculated by custom (user-defined) functions over possibly volatile values. *Similarity-based* approaches [45] rely on "weights" for features to calculate similarities between individual element pairs. Finally, *language-specific* approaches provide specific matching algorithms dedicated to a particular meta-model, e.g., for the UML meta-model as is the case in [112, 148].

Overall, the strategies used in model differencing approaches can also facilitate consistency checking with TGGs. First, static or dynamically calculated identifiers, if available, can help reducing the search space involved in applying consistency rules. Second, custom similarity notions can inspire new types of objective functions in our linear optimization, e.g., by maximizing the achieved similarity instead of maximizing the number of marked elements as we do. Reversely, our consistency checking approach can provide an alternative way of model differencing (where unmarked elements represent the differences between two models). Of course, we cannot claim to have come up with a tailored solution for model differencing but can always guarantee a maximal matching in the terms of model elements (while especially similarity-based approaches rely on fine-tuning and trial-and-error processes for accurate results). In our practical contribution (in the next section but one), we shall experimentally evaluate how costly our combination of graph grammars and linear optimization operates for model differencing purposes.

Optimization Techniques in MDE: Our utilization of optimization techniques may be unique in consistency checking but not in the general field of model transformations. Recently, *genetic algorithms* have been investigated to govern rule application process in model transformation [48, 49]. This setting gets its inspiration from biology by considering individual rule applications as genomes and derivations as individuals. Generations of individuals are calculated by mutation and crossover operations. Having an objective in mind as we do, individuals have a fitness value calculated according to this objective, and these fitness values determine the probability of passing on genes from one generation to the next. The technique is demonstrated in [105] for object-oriented refactoring where refactorings are captured as rules and fitness values are given by classical metrics such as coupling and cohesion, i.e., multiple objectives are considered different than our case.

The type of our constraints and objective justifies our choice for ILP. Nevertheless, genetic algorithms can provide an interesting alternative to cope with the search space problems. While our approach possibly puts a lot of effort into rule application (in finding final derivations) and solves rather a simple optimization problem in retrospect, an approach like [48, 49] can reverse the situation by skipping some rule applications but performing a continuous and more complex optimization. Again reduced to the biological analogy, however, this can also mean that some individuals possibly cannot pass on their genes although they would eventually lead to a correct result. Hence, we consider genetic algorithms as an alternative that

optimizes well for multiple objectives but possibly compromises reliability. ILP techniques, finally, are used in [144], similar to our utilization, to address the large search space of a graph grammar, but different than our purposes, for a guided traversal of the search space.

3.7 Summary and Future Work

In this section, we have

- identified the search space problems that prevented practical solutions for consistency checking with TGGs,
- formulated decision making in the search space of consistency rule applications as a linear optimization problem,
- stated sufficient as well as sufficient and necessary conditions for consistency with formal proofs based on the outcome of the optimization problem,
- provided a procedure for consistency checking based on these conditions,
- discussed the complexity of our procedure in terms of complexity theory,
- given an overview of related work that pursues the same or comparable goals as we do with consistency checking.

There are at least two important lines of tasks for future work in our belief:

- First and foremost, *incremental* consistency checking based on our results is the next logical step. That is, given that a consistency checking run has already been performed between two models and the models are then changed, the results from the former run (i.e., markings and correspondences) should be reused instead of a computation from scratch. An important decision when addressing incremental consistency checking is to define what to do with eliminated consistency rule applications. When models change, eliminated consistency rule applications indeed might become "choosable" or even better choices than formerly chosen ones. An incremental approach, therefore, should either maintain eliminated consistency rule applications or be capable of "complementing" missing rule applications upon model changes. Incrementality of the linear optimization part defines the second dimension of research in this regard. While incremental ILP solving is an open issue (with regard to the state-of-the art solvers), incrementality can be provided at the client side by formulating the optimization problem for a subset of consistency rule applications (and not for all of them as we currently do).
- Furthermore, incorporating *user-defined preferences* into the optimization problem would practically provide an added value to our approach with regard to its use cases. Given that, e.g., model owners prefer certain types of correspondences or have additional requirements on the mapping of two models, new types of logical constraints and objective functions are to be formulated to

tackle a broader class of mapping problems. A current example is already being investigated in the network domain where our implementation is used to map the resources of a computer network to the requirements of a data centre. Besides mapping the models in a graph-grammatical sense as we discussed so far, further bandwidth and CPU constraints form the additional part of the optimization problem. While it is a case-specific matter to choose how to enrich the optimization problem, it is important to come up with formal arguments for respecting consistency in the sense of a TGG while supporting additional requirements. To this end, even multi-objective optimization is worthwhile to consider to address correctness of consistency checking on the one hand and optimizing for user-defined preferences on the other hand.

4

CONSISTENCY RESTORATION WITH TGGS

This section presents our second contribution based on [99], namely consistency restoration between two models. Given a consistent pair of source and target models together with correspondences, consistency restoration is the process of propagating a set of changes, referred to as a *delta*, in one of the models to the other.

A delta changing a graph (e.g., a source graph) is given by added and deleted elements (vertices and edges) in the graph.[1] Generally, deltas in the source and target graph of a consistent triple are induced from the activities of model owners. In our context, furthermore, after executing consistency checking as discussed in the previous section, source and target elements beyond the consistent portions (or a selection of them depending on the preferences of model owners) can also be considered as deltas in the form of added elements (i.e., as additions to the consistent portions). Hence, our consistency restoration can operate complementarily to a preceding consistency checking run. This completes the picture of the BX vision depicted in Figure 1.3 and makes our overall approach to consistency management with TGGs unique in terms of a generalized support. While deltas and their propagation are at the center of attention in the following discussions, we handle deltas uniformly no matter whether they result from modifications by model owners or from a consistency checking run (or from a combination of both).

Given a TGG and a consistent triple graph $G_S \leftarrow G_C \rightarrow G_T$, the task of a consistency restoration in the *forward* direction (i.e., from source to target) is to propagate a source delta δ_S that changes G_S to G'_S and to create again a consistent triple graph $G'_S \leftarrow G'_C \rightarrow G'_T$ if there exists one. This applies analogously to the *backward* direction when propagating a target delta δ_T. Due to the symmetric nature of TGGs and the analogy between the forward and backward directions, we provide our statements only in the forward direction throughout this section (and switch back to the backward direction only for illustrating some interesting situations based on our running example).

One of the most important challenges (for TGGs as well as for BX in general) when restoring consistency is to realize delta propagation as an *incremental update*. That is, the previously consistent state of the model pair (and correspondences in the case of TGGs) must be taken into account and only those parts of the models must be transformed whose consistency may be affected by a given delta.

[1] A third type of change, namely attribute changes, and their special treatment are out of scope in this thesis. Although not always ideal from a practical point of view, nevertheless, such changes can be captured as a combination of deletions and additions (i.e., deleting a vertex and adding a new one with new attribute values).

There exist at least two main reasons to prefer incremental updates over transforming a changed model from scratch: The first and probably more obvious reason is the *efficiency* of consistency restoration. Efficiency is especially crucial if consistency restoration operates frequently with rather tiny deltas and is subject to performance requirements. Second, a source and/or target model can contain information which is not producible from the other one (considering consistency between Java and UML models, for example, language-specific programming practices and method implementations in a Java model cannot be produced from a more abstract UML model). Hence, incremental updates are necessary for *information preservation* in the already consistent parts of two models.

Similar to consistency checking as discussed in the previous section, consistency restoration with TGGs also relies on operational rules, e.g., *forward rules* for consistency restoration in the forward direction. Applications of operational rules, again, are traced back to applications of original TGG rules to argue that a consistent triple is created at the end of the process.

For incremental updates, consistency restoration with TGGs furthermore relies on a given derivation representing the consistency history of two models, i.e., a derivation that describes how the previously consistent state of the two models has been constructed. Such a derivation can either be inferred from consistency checking or from former consistency restoration runs, in both cases from operational rule applications that are traced back to TGG rule applications. Accordingly, the main runtime tasks of consistency restoration in the case of TGGs are:

- to revoke some rule applications in the consistency history, namely those that are no more available for the changed source graph,
- to perform new rule applications that are available for the changed source graph.

These tasks have been the focus of research on TGGs in the last two decades and investigated with diverse techniques, formalisms, and implementations. All state-of-the-art TGG-based consistency restoration frameworks we are aware of (in particular [5, 60, 66, 70, 86, 93, 116]) address these runtime steps either (i) in a simple but non-scalable manner, calculating a valid consistency history each time from scratch over the entire models [66], or (ii) by performing auxiliary dependency analyses over the source and target model elements [93], correspondences [70], or operational rule matches [5, 116] to calculate the consequences of a delta, or (iii) by exploiting practically justified but as yet informal heuristics without general proofs of consistency [60, 86].

There is no doubt that this diversity is advantageous from a scientific community point of view. Our observation, however, is that the complexity in addressing the runtime steps of consistency restoration is accidental and is caused by entangling high-level delta propagation strategies with low-level details of how deltas and their consequences must be handled efficiently and correctly. Due to this entanglement, it is difficult to exchange ideas amongst the different TGG approaches, even though they all essentially share the same basic goal and the same formal foundation.

Considering the state-of-the-art tool support in the general field of graph grammars, moreover, our second observation is that *incremental pattern matching* tech-

niques (e.g., [50]) naturally address the requirements for incremental updates (as a matter of fact, the shared term "incremental" here is not a coincidence and refers to a change-driven operation mode in both cases). An incremental pattern matcher monitors all (potential) matches of a given set of patterns in a (possibly changing) host graph and thus can report consequences of deltas. For consistency restoration purposes, this can eliminate the need for additional dependency analyses, heuristics, or calculations from scratch when determining rule applications that become invalidated or available as a result of a given delta.

Our main contribution in this section, therefore, is to integrate incremental pattern matching into consistency restoration based on TGGs. Our goal is to provide a solid yet simplified foundation for a new generation of TGG tools that can now leverage available incremental graph pattern matching frameworks [139, 142]. We reduce consistency restoration with TGGs to a relatively straightforward component that reacts to invalidated or available rule applications reported by its underlying incremental pattern matcher. Our formal results consist of Algorithm 2 representing this simplified concept as a procedure, and the formal proofs for its termination (Theorem 3) and correctness as well as completeness (Theorem 4).

On the way of stating these results, we again identify search space problems in consistency restoration. Different than consistency checking where we exploited linear optimization techniques, we use *application conditions*, a well-known technique from graph grammars to prevent undesired rule applications, to overcome search space problems in consistency restoration. This choice has practical reasons in the first place as we shall discuss (in particular, differences between consistency checking and restoration lead to different solutions for the two use cases).

Outsourcing technical aspects of consistency restoration to incremental pattern matching techniques, the added value of consistency restoration as proposed in this thesis lies in its simplification and viability from an implementation point of view. The contribution, therefore, is addressed to meta-tool developers in the first place. We also believe that the simplified conception furthermore brings a didactic advantage for consistency tool developers and model owners to understand how consistency restoration works.

In the rest of this section, we first discuss examples of consistency restoration (again based on the consistency between Java and UML models). Subsequently, we formally present operational rules for consistency restoration, identify challenges with respect to their applications, and finally propose an algorithmic approach for consistency restoration that makes use of incremental pattern matching techniques to calculate consequences of deltas.

4.1 Examples of Consistency Restoration

As previously done for consistency checking, we exemplarily discuss a mental execution of consistency restoration before automating this task. Starting with consistent triple graphs (and with available consistency history showing us with which TGG rule applications the triple graph has been constructed), our goal is to demonstrate how a delta is formed and what its consequences are.

If we consider consistency restoration in the forward direction as processing the elements of a (changed) source model, the notation of markings (☑) again helps us to indicate which part of the source model is already consistent. Unmarked elements (□), accordingly, indicate which part of the source model is beyond consistency.

At the top of Figure 4.1, a model triple is given consisting of a Java and a UML interface Shape with a correspondence in between. Furthermore, a consistency history is provided showing how these models are created with our TGG rules. The only rule application in the consistency history is α_1 via InterfaceRule (r_2). We explicitly annotate at each model element (be it from the source, correspondence, or the target model) by which rule application it has been created, and additionally provide markings on the source model. The source model at the top of Figure 4.1 is entirely marked, i.e., there is not yet a need for any action for consistency restoration. Below this consistent model pair, a source delta is given adding some new (and consequently unmarked) Java elements highlighted in a green and transparent rectangle with an «add» markup. The source delta consists of the following added elements: A Java interface Rectangle inheriting from Shape and a Java class RectangleImpl inheriting from Rectangle.

The source delta in Figure 4.1 does not invalidate any rule application from the consistency history, i.e., the consistency of the previously existing Shape interfaces is not affected by the added elements. Therefore, it remains to determine with which rule applications the added source elements can be created (and accordingly the missing correspondence and target elements are yet to be complemented).

There exist two possibilities to create the added Java elements in Figure 4.1: First, the Rectangle interface and the RectangleImpl class can be created one by one using our first set of rules in Figure 2.7, in particular by applying InterfaceRule (r_2) and ClassRule (r_1). Second, we can create both together using our Impl-class strategy provided by our extended set of rules in Figure 2.12, in particular by applying ImplClassRule (r_8). We choose the latter at the bottom of Figure 4.1 and restore consistency by creating a UML class Rectangle. Furthermore, the inheritance relation between the Rectangle interface and the Shape interface on the Java side is reflected on the UML side by applying InterfaceClassGeneralRule (r_9).

The consistency history is now extended to three rule applications incrementally, i.e., α_1 is retained and α_2 as well as α_3 are added. Having found rule applications to create all added source elements, the model triple is again in a consistent state where all source model elements are marked (and the missing correspondence and target elements are complemented).

Starting with this consistent state, a further example of consistency restoration is shown in Figure 4.2. This time, some Java elements are deleted, namely the RectangleImpl class together with its inheritance relation, highlighted in a red and transparent rectangle with a «del» markup. This clearly invalidates the rule application α_2 via ImplClassRule (r_8) in the consistency history, i.e., the Impl-class strategy is not applicable anymore after deleting the RectangleImpl class. As a result, α_2 must be revoked such that the obsolete correspondence and target elements must be deleted (and α_2 is removed from the consistency history). Consequently, the rule application α_3 is also invalidated and revoked as its context elements (e.g., the UML class Rectangle) are deleted.

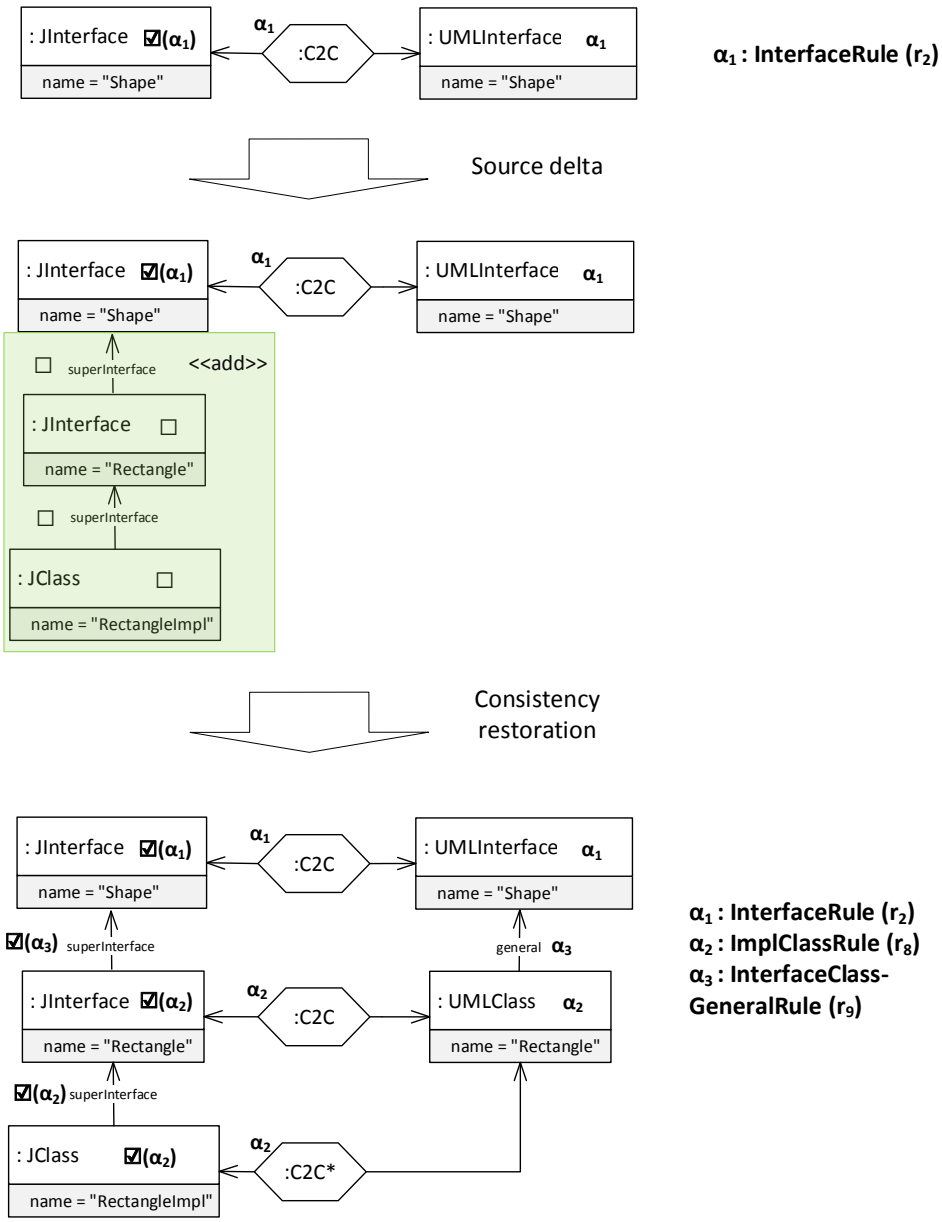


Figure 4.1: From top to bottom; a consistent model pair, a source delta adding some new elements to the source model, and expected result after consistency restoration

In a final step, it must be determined how the remaining Rectangle interface and its inheritance relation on the Java side can now be created again (note that these elements are not deleted but only their consistency is affected by the delta). The only possibility for this is to create the Rectangle interface with InterfaceRule (r_2) and subsequently to create the inheritance relation with GeneralRule (r_4). The result of these rule applications is depicted at the bottom of Figure 4.2 showing the models again in a consistent state. Overall, two new rule applications (α_4 and α_5) have been added to the consistency history representing these actions, two have been removed (α_2 and α_3) due to being invalidated, and α_1, finally, is just retained.

Regarding the two examples of consistency restoration discussed so far, it can intuitively be observed that added elements in a delta lead to new rule applications and deleted elements lead to invalidated rule applications. As illustrated in Figure 4.2, however, deleted elements can also lead to new rule applications as deletions can affect the consistency of some remaining (non-deleted) elements. Even more counter-intuitively, added elements can lead to invalidated rule applications. Such a situation is depicted in Figure 4.3.

The consistent state depicted at the top of Figure 4.3 represents a pair of Rectangle and a pair of Square classes where the consistency history consists of the rule applications α_1 and α_2 with ClassRule (r_1). The source delta solely adds an inheritance relation (a superClass edge) from the Square class to the Rectangle class on the Java side. If we retain α_1 and α_2 in the consistency history (claiming that an added element would not affect the consistency of previously existing elements), however, there does not exist any rule in our consistency specification that would only create the added superClass edge (remember that our consistency specification creates superClass edges always together with a new Java class to avoid multiple inheritance). A rule application with SubClassRule (r_3) is the only way to create the Square class with its superClass edge. Hence, due to the given delta, α_2 with ClassRule turns out to be a "wrong" choice for the Square class and is thus invalidated. Restoring consistency at the bottom of Figure 4.3, α_2 is removed from the consistency history and a new rule application α_3 with SubClassRule (r_3) is added.

To sum up, the main task of consistency restoration in a rule-based approach such as TGGs is to determine which rule applications from former runs are invalidated and which ones become available due to a given delta. In the forward direction where the source model is changed, the question is how the changed source model can be created using TGG rules (and the missing correspondence and target elements are created accordingly). We incorporate a consistency history in the process and retain former rule applications that are still valid after a delta.

As a final remark, though not illustrated explicitly, transforming an entire source model to a target model (as there does not exist a corresponding target model yet) is a special case of what we call consistency restoration. In this case, the entire source model can be regarded as a delta where the initially consistent state is the empty triple graph (and the consistency history does not contain any rule applications).

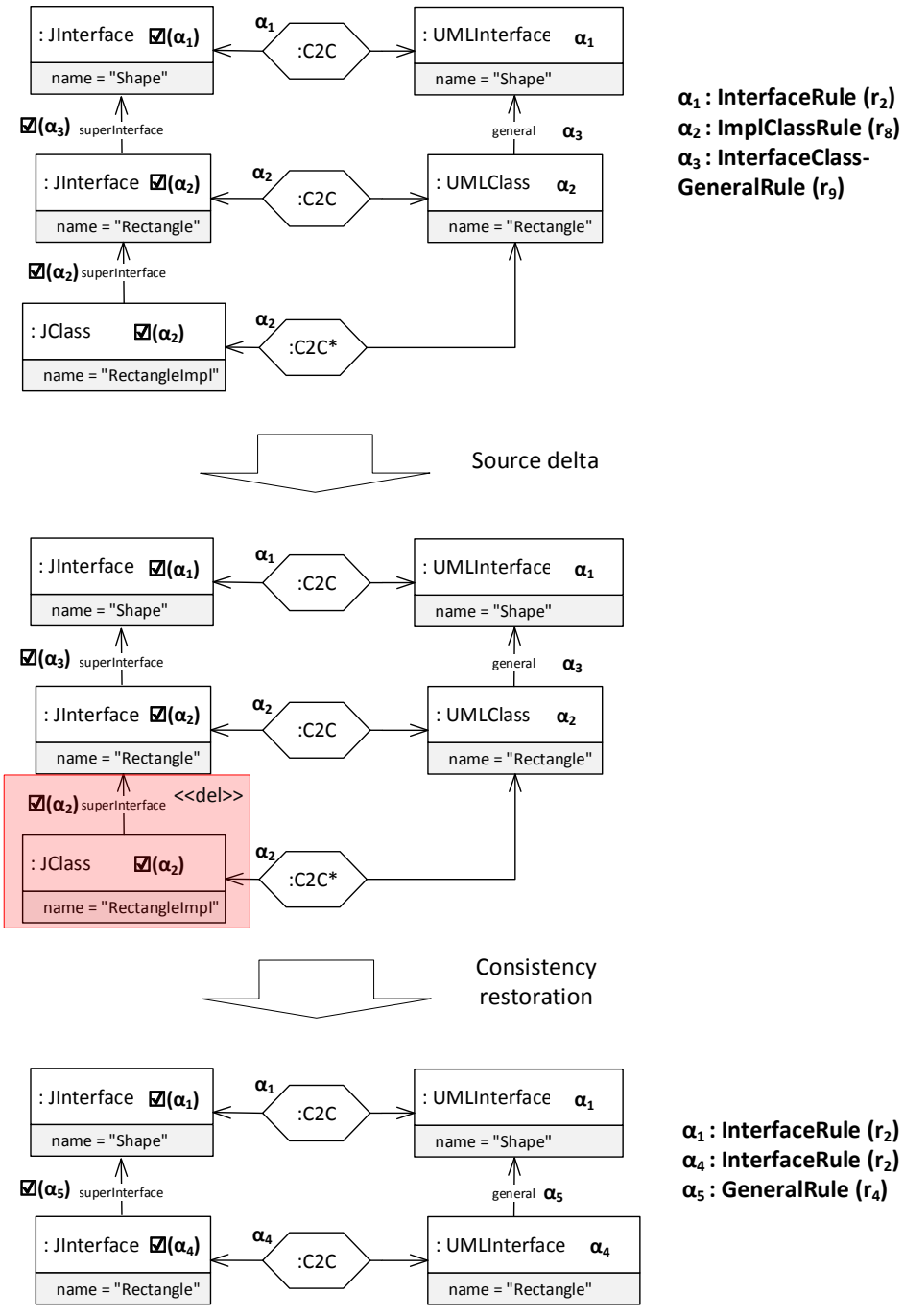


Figure 4.2: From top to bottom; a consistent model pair, a source delta deleting some elements in the source model, and expected result after consistency restoration

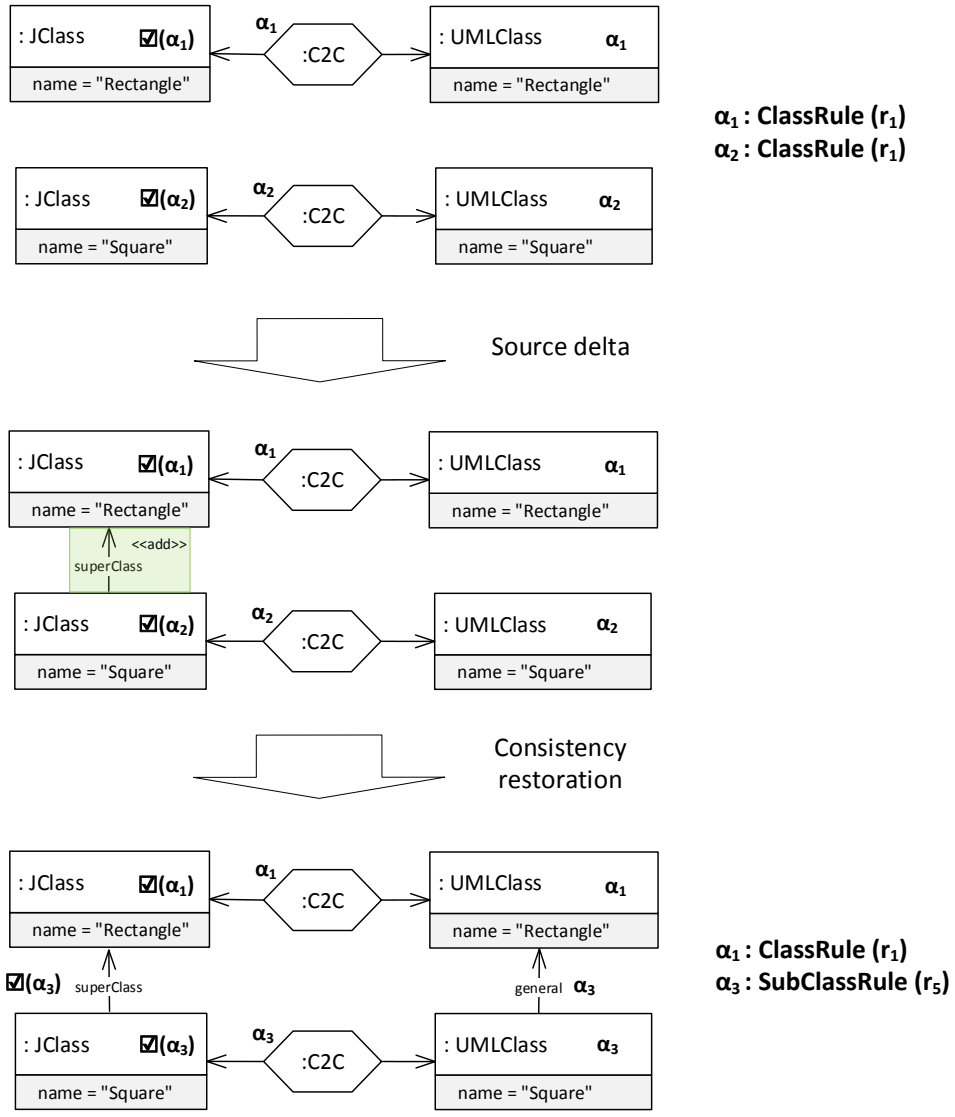


Figure 4.3: A further example of consistency restoration where an added element leads to invalidated as well as new rule applications

4.2 Forward Rules

In line with consistency checking, consistency restoration with TGGs also operates upon operational rules and their applications. In Figure 4.4, the overall architecture of consistency restoration in the forward direction is depicted.

First, TGG rules are operationalized to forward rules. A forward rule fr differs from its original TGG rule r in the following sense: fr does not create any source elements but marks those source elements which would be created by r. Furthermore, fr creates correspondence as well as target elements in accordance to r.

Second, given a consistent triple graph (in its present version indicated as V.1 in Figure 4.4) with a consistency history showing how the triple graph is constructed and a source delta, a TGG engine (i) revokes some invalidated rule applications, i.e., removes obsolete correspondence and target elements that have been created by the invalidated rule applications and (ii) performs new rule applications to mark remaining source elements. The result is a consistent state of the triple graph representing the updated versions (indicated as V.2) of the source, correspondence, and target graphs, and additionally, the updated consistency history. Note that consistency restoration in the forward direction changes only the correspondence and target graph but not the source graph (i.e., V.2 on the source side of the output is solely the result of the source delta).

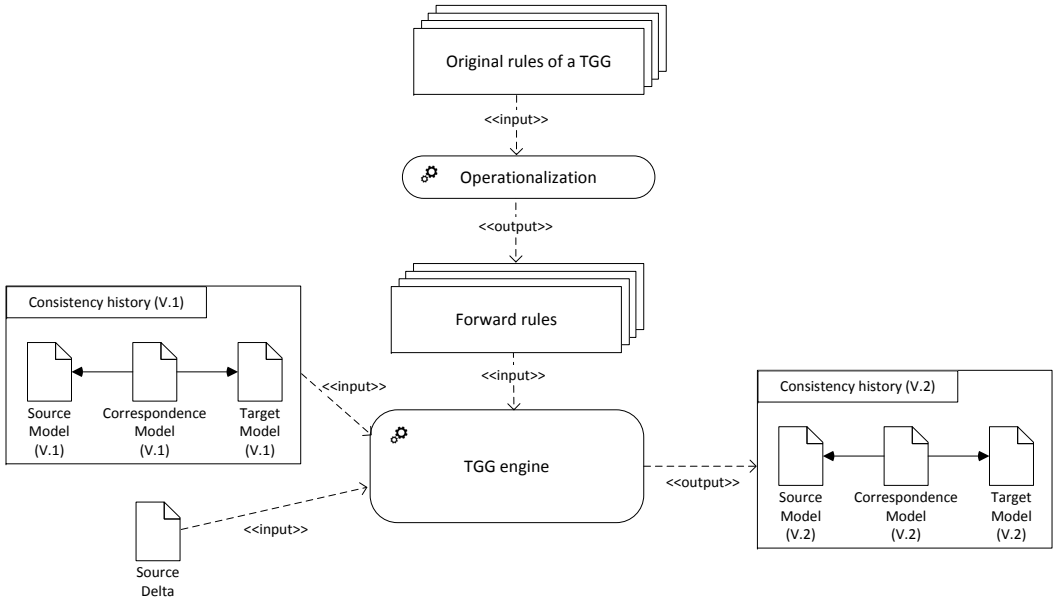
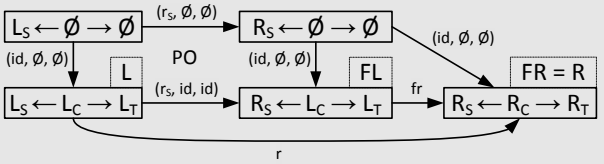


Figure 4.4: Overall architecture of consistency restoration with TGGs

While our contribution mainly concerns the TGG engine part of this architecture (discussing how to revoke rule applications and to perform new ones), we start with defining how a forward rule is constructed from a TGG rule. In analogy to consistency rules in the previous section, we define the *marking* elements of a forward rule (in fact, only the source marking elements as the goal of consistency restoration in the forward direction is to process a source model).

Remark 4. Definitions and statements in this subsection form the consistency restoration counterpart of some former definitions and statements for consistency checking. In particular, Definition 31, 32, 33, 34 and Lemma 4 in the following are adopted from Definition 15, 16, 17, 18 and Lemma 1, respectively. Treatment of wrong choices of forward rules and the incrementality aspect, however, distinguish our consistency restoration approach from consistency checking as we shall discuss after this subsection.

Definition 31 (Forward Rule)**.**
Given a rule $r : L \to R$ with $L = L_S \leftarrow L_C \to L_T$ and $R = R_S \leftarrow R_C \to R_T$, the respective *forward rule* $fr : FL \to FR$ is a rule constructed, as depicted in the diagram, such that FL is a pushout of L and $R_S \leftarrow \emptyset \to \emptyset$ over $L_S \leftarrow \emptyset \to \emptyset$, and $FR = R$. The morphism $fr : FL \to FR$ is induced as the universal property of the pushout.



An element $e \in \mathsf{elements}(R_S)$ is referred to as a *source marking element* of fr if $\nexists\, e' \in \mathsf{elements}(L_S)$ with $r_s(e') = e$.

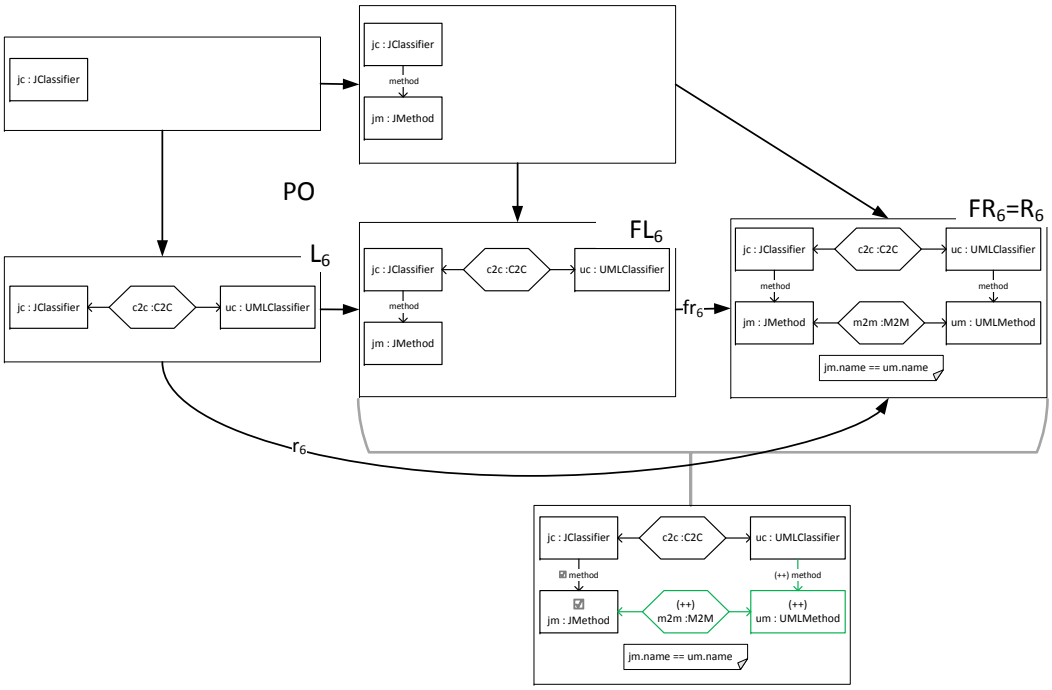


Figure 4.5: Example of a forward rule construction

Example 21. The construction of the forward rule for MethodRule (r_6) is exemplified in Figure 4.5. The left hand-side of a forward rule fr for an original TGG rule r is constructed such that all source elements of r are required as context.

The right hand-side of fr, furthermore, simply equals to the right hand-side of r. That is, fr creates correspondence as well as target elements as created in r but no source elements. Moreover, fr marks exactly those source elements that are created by r. The result of this construction is depicted again with our compact syntax at the bottom right of Figure 4.5. While created correspondence and target elements are denoted with a (++)-markup, the source marking elements are now denoted with a ☑-markup.

A forward rule, furthermore, requires exactly the same attribute constraints as its respective TGG rule (e.g., name equality must hold when marking a Java method and creating a corresponding UML method in Figure 4.5).

Finally, Figure 4.6 and 4.7 depict all forward rules that are constructed from the original rules from Figure 2.7 and 2.12, respectively. A noteworthy aspect is that some forward rules in Figure 4.7 have only source marking elements but do not create any correspondence or target elements. Hence, these rules contribute to a consistency restoration process only with their markings (i.e., by processing source elements) without changing the triple graph.

Our next goal is to discuss how to apply forward rules *correctly*, i.e., that a consistent triple graph is created at the end of a consistency restoration process. Given a consistent triple graph $G_S \leftarrow G_C \rightarrow G_T$ and a source delta δ_S changing G_S to G'_S, our ultimate strategy for correctness is as follows:

- We take a derivation d via forward rule applications that entirely marks the triple graph $G_S \leftarrow \emptyset \rightarrow \emptyset$ (and creates $G_S \leftarrow G_C \rightarrow G_T$) as the starting point.
- Reacting to the consequences of δ_S, we change d to a derivation d' that entirely marks the triple graph $G'_S \leftarrow \emptyset \rightarrow \emptyset$ (and creates a triple graph $G'_S \leftarrow G'_C \rightarrow G'_T$). Forward rule applications in d are retained in d' when possible.

In order to argue that a derivation via forward rules (starting from $G'_S \leftarrow \emptyset \rightarrow \emptyset$ and ending with $G'_S \leftarrow G'_C \rightarrow G'_T$) amounts to a derivation via original TGG rules (starting from $\emptyset \leftarrow \emptyset \rightarrow \emptyset$ and ending with $G'_S \leftarrow G'_C \rightarrow G'_T$), forward rule applications must fulfill certain properties with regard to their markings G'_S.

Definition 32 (Marked and Required Source Elements)**.**
Let $fr : FL \rightarrow FR$ be a forward rule with $FL = FL_S \leftarrow FL_C \rightarrow FL_T$ and $FR = FR_S \leftarrow FR_C \rightarrow FR_T$. For a rule application $\alpha : G \overset{fr@fm}{\Longrightarrow} G'$ with $G = G_S \leftarrow G_C \rightarrow G_T$, we define the following sets:

- $\mathsf{marked}(\alpha) = \{e \in \mathsf{elements}(G_S) \mid$
  $\exists\ e' \in \mathsf{elements}(FL_S)$ with $fm(e') = e$ where e' is a source marking element of $fr\}$
- $\mathsf{required}(\alpha) = \{e \in \mathsf{elements}(G_S) \mid$
  $\exists\ e' \in \mathsf{elements}(FL_S)$ with $fm(e') = e$ where e' is not a source marking element of $fr\}$

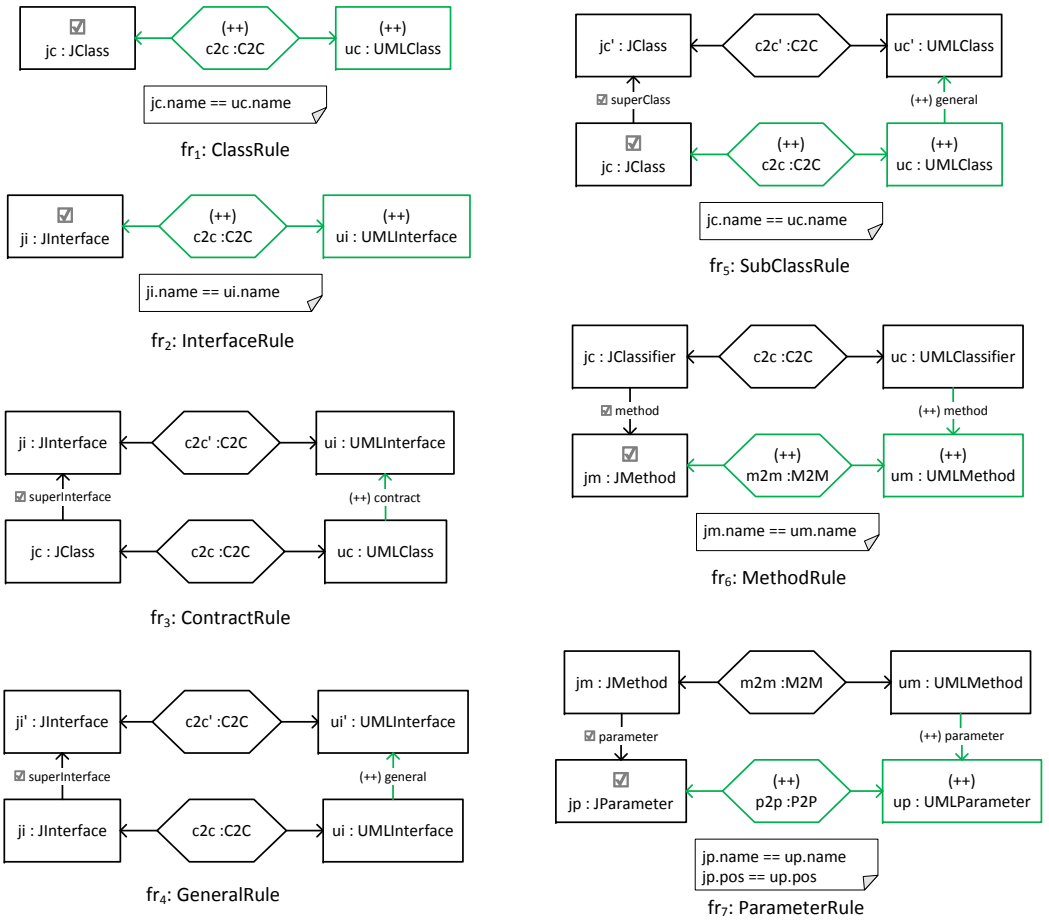


Figure 4.6: Forward rules for the original TGG rules from Figure 2.7

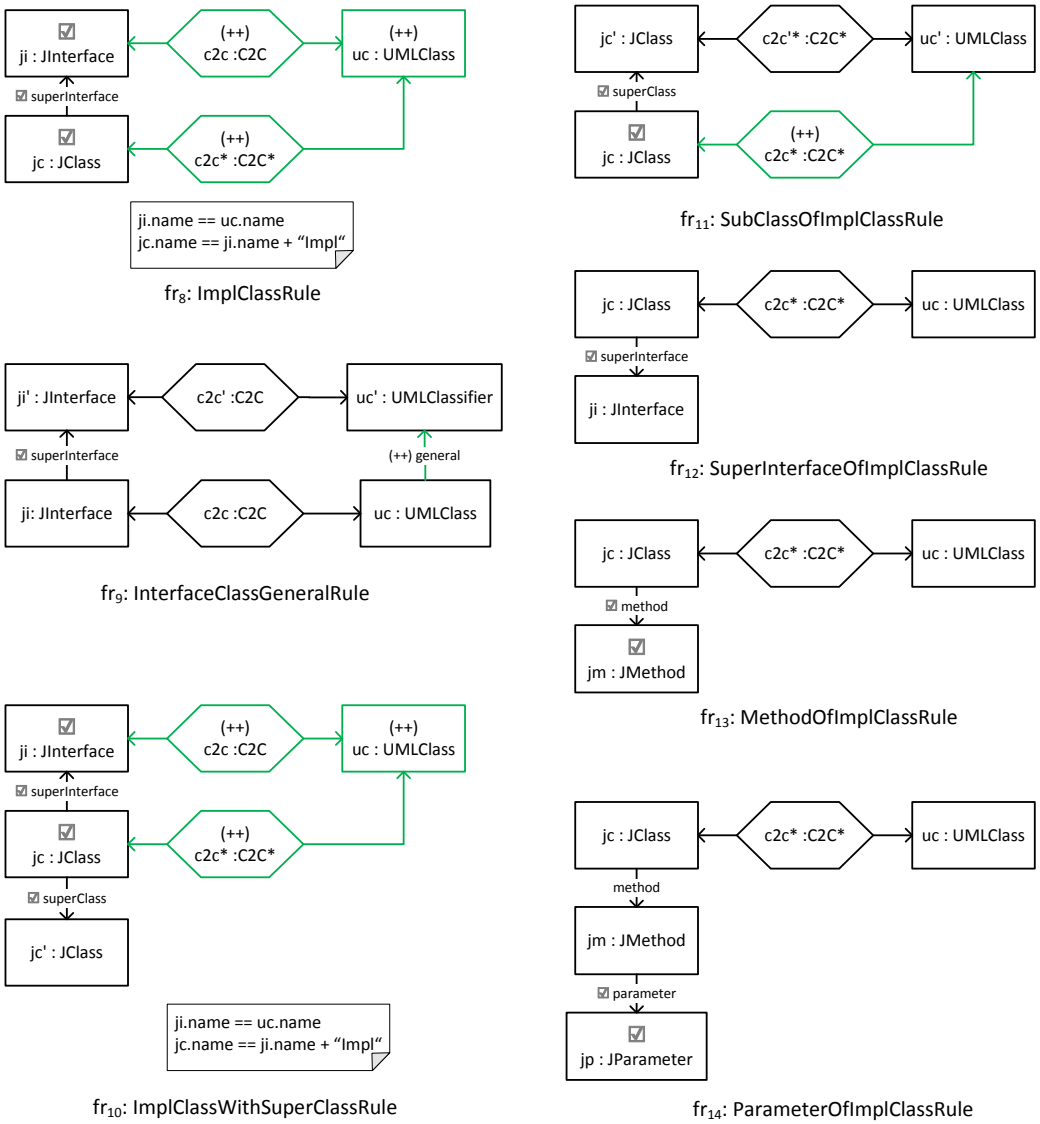


Figure 4.7: Forward rules for the original TGG rules from Figure 2.12

Definition 33 (Creation and Context Preserving Derivations)**.**
Let $G_0 = G_S \leftarrow \emptyset \rightarrow \emptyset$ be a triple graph and $\mathcal{FR}$ a set of forward rules. We refer to a derivation $d : G_0 \overset{fr_1@fm_1}{\Longrightarrow} G_1 \ldots \overset{fr_n@fm_n}{\Longrightarrow} G_n$ with $fr_1, \ldots, fr_n \in \mathcal{FR}$ as

- *creation preserving*, if for all pairs $\alpha_i : G_{i-1} \overset{fr_i@fm_i}{\Longrightarrow} G_i$ and $\alpha_j : G_{j-1} \overset{fr_j@fm_j}{\Longrightarrow} G_j$ of forward rule applications in d with $1 \leq i, j \leq n$ and $i \neq j$, it holds that $\mathsf{marked}(\alpha_i) \cap \mathsf{marked}(\alpha_j) = \emptyset$
- *context preserving*, if for all forward rule applications $\alpha_i : G_{i-1} \overset{fr_i@fm_i}{\Longrightarrow} G_i$ in d with $1 \leq i \leq n$, it holds that $\forall e \in \mathsf{required}(\alpha_i)$, $\exists\, \alpha_j : G_{j-1} \overset{fr_j@fm_j}{\Longrightarrow} G_j$ in d with $1 \leq j < i$ such that $e \in \mathsf{marked}(\alpha_j)$.

Remark 5. The conditions for creation and context preserving derivations with forward rules can inherently be satisfied by encoding markings into graphs and forward rules. In such a setting, forward rules can only have a match if the marking conditions in Definition 33 are met. To this end, *translation attributes* in [65] or *markers* in [99] are two categorical attempts both enriching the category **Graphs** (and accordingly **TripleGraphs**) with marking information. For brevity and readability, we choose to remain on a set-theoretical basis regarding markings (as was the case for consistency checking in the previous section) and only state what conditions a forward rule application needs to satisfy over these sets. In practice, marking information can indeed be encoded as additional parts of the graph structures or can be provided as an additional service by the underlying pattern matcher.

Definition 34 (Entirely Marking Derivations)**.**
Given a triple graph $G_0 = G_S \leftarrow \emptyset \rightarrow \emptyset$ and a set $\mathcal{FR}$ of forward rules, and a derivation $d : G_0 \overset{fr_1@fm_1}{\Longrightarrow} G_1 \ldots \overset{fr_n@fm_n}{\Longrightarrow} G_n$ with $fr_1, \ldots, fr_n \in \mathcal{FR}$, let $\mathcal{D}$ be the set of all forward rule applications in d. We refer to d as *entirely marking*, if $\bigcup_{\alpha \in \mathcal{D}} \mathsf{marked}(\alpha) = \mathsf{elements}(G_S)$.

Example 22. Figure 4.8 depicts a derivation consisting of the two forward rule applications α_1 and α_2. While $\mathsf{required}(\alpha_1)$ is empty, $\mathsf{marked}(\alpha_1)$ as well as $\mathsf{required}(\alpha_2)$ consists of the Java class Rectangle and $\mathsf{marked}(\alpha_2)$ consists of the Java class Square with its outgoing superClass edge. Overall, the two rule applications do not overlap in their marked elements, and their required elements are marked by some previous forward rule applications (this especially concerns α_2 whose required elements are marked by α_1). Hence, the depicted derivation is creation and context preserving. As all source elements are marked with the two rule applications, furthermore, the derivation is entirely marking.

As previously done for Lemma 1 for consistency checking, we again exploit creation and context preservation of operational rule applications to argue that

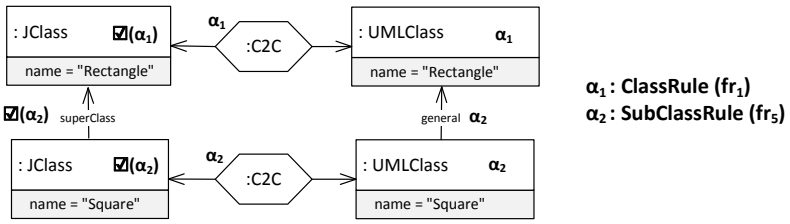
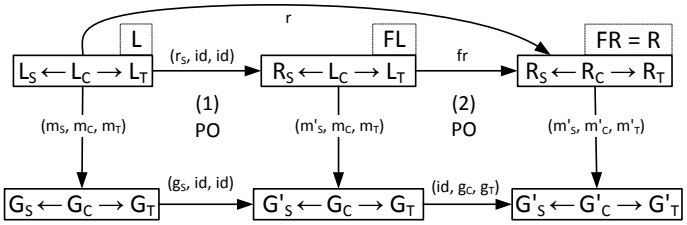


Figure 4.8: A creation and context preserving derivation with forward rules

an equivalent derivation via the original TGG rules exists. In the case of forward rules, creation preservation ensures that source elements are marked exactly once (as they would be created once by the original TGG rules). Context preservation, furthermore, ensures that a forward rule application can be performed only if the required source elements are previously marked (as they would be previously created when applying the original TGG rule).

Lemma 4 (Marking a Source Graph)**.**
Given a TGG, let $\mathcal{FR}$ be the set of the respective forward rules.
$\exists\,(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG) \iff \exists\, d : G_0 \overset{fr_1@fm_1}{\Longrightarrow} G_1 \ldots \overset{fr_n@fm_n}{\Longrightarrow} G_n$ where $fr_1, \ldots, fr_n \in \mathcal{FR}$, $G_0 = G_S \leftarrow \emptyset \rightarrow \emptyset$, $G_n = G_S \leftarrow G_C \rightarrow G_T$, d is creation preserving, context preserving, and entirely marking.

Proof. The proof is analogous to that of Lemma 1 and follows from pushout composition and decomposition. This time, as depicted in the following diagram, each rule application $G \overset{r@m}{\Longrightarrow} G'$ via a rule $r \in TGG$ can uniquely be decomposed into and accordingly composed from two pushouts. The first pushout (1) creates only source elements while the second pushout (2) creates only correspondence and target elements. The second pushout equals to a forward rule application and, due to forward rule construction (Definition 31), marks exactly those source and target elements that are created in the first pushout.



If $(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$, there exists a derivation $d' : (\emptyset \leftarrow \emptyset \rightarrow \emptyset) \overset{r_1@m_1}{\Longrightarrow} \ldots \overset{r_n@m_n}{\Longrightarrow} (G_S \leftarrow G_C \rightarrow G_T)$ where $r_1, \ldots, r_n$ are the rules of the TGG. Transferring the idea of composition and decomposition depicted above to d', the existence of d' is equivalent to the existence of d stated in the lemma where $fr_1, \ldots, fr_n$ are the respective forward rules of $r_1, \ldots, r_n$. □

4.3 Wrong Choices of Forward Rule Applications

Similar to consistency checking, consistency restoration with a TGG is also subject to a search space involved in applying the respective forward rules. A forward rule application process can lead to a result where a source graph G_S is not entirely marked even though a triple graph $G_S \leftarrow G_C \rightarrow G_T$ exists in the language of the TGG. The challenges mainly arise from forward rule applications that overlap in their marked elements and thus constitute alternatives to each other.

While we have depicted in Figure 4.8 a creation and context preserving derivation that marks a Java class and its subclass entirely, things can also go wrong when marking this model as depicted in Figure 4.9. This time, both classes have been marked using the forward rule fr_1 of ClassRule leaving the superClass edge unmarked as none of our forward rules (in Figure 4.6 and 4.7) would mark this edge alone. Obviously, even though a match might exist, the forward rule fr_1 of ClassRule should not be applied to mark subclasses.

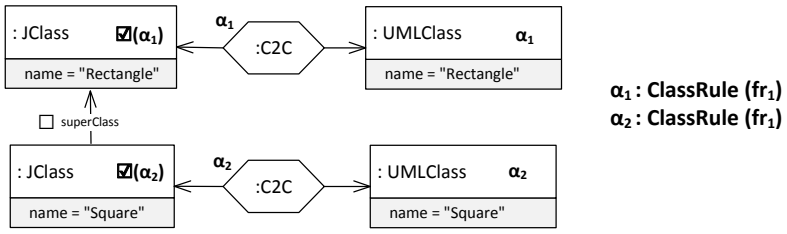


Figure 4.9: An undesired state in applying forward rules

Considering the incrementality aspect of consistency restoration, moreover, search space problems do not only arise from choosing the wrong one among alternative forward rule applications but also from deltas that make a forward rule application wrong "in retrospect". As previously exemplified in Figure 4.3, marking two Java classes with the forward rule fr_1 of ClassRule might be a correct (and in fact the only possible) choice. A source delta adding a superClass edge between the two classes, however, requires a reconsideration of this choice.

A similar and arguably more challenging, though less obvious, problem exists in the backward direction when applying backward rules to UML models exhibiting multiple inheritance among UML classes. Such a situation is depicted in Figure 4.10. As backward rules have been omitted so far, we first show at the top of Figure 4.10 the four backward rules that mark UML classes and their general edges (these backward rules suffice for the current discussion while we further on mainly focus on the forward direction in the rest of this section).

Below the backward rules, an undesired outcome of applying them to an UML model is depicted. Concretely, the Button class inherits from the Rectangle and the Clickable class on the UML side. When these UML classes are marked by the respective backward rules of ClassRule and SubClassrule, one of the two general edges of the Button class remains unmarked. Apparently, as long as UML classes exhibit multiple inheritance, the Impl-class strategy, i.e., marking the UML classes with the respective backward rule of ImplClassRule, is the only correct choice in the backward direction. The desired outcome (respecting this condition when marking

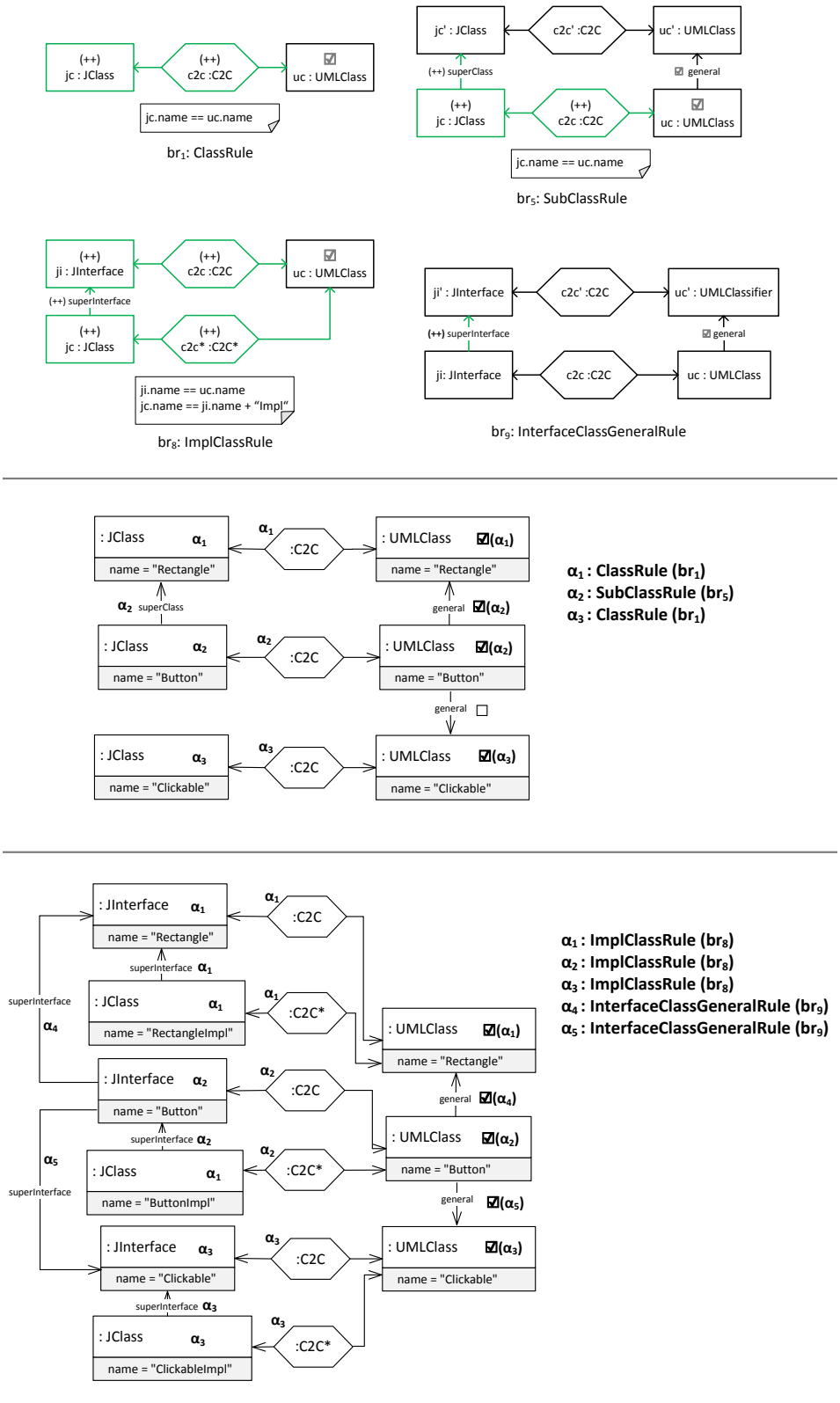


Figure 4.10: From top to down; four backward rules from our running example, an undesired outcome of applying these backward rules for marking UML classes with multiple inheritance, and the desired outcome

the UML model) is depicted at the bottom of Figure 4.10. Again, even more critically, if multiple inheritance among UML classes occurs first due to a target delta, this can concern the consistency of an entire UML class hierarchy (assuming that the UML classes are not marked via the Impl-class strategy in the consistency history).

To sum up, rule applications in a consistency restoration process may be subject to different alternatives when marking elements. Some of the alternatives, however, are wrong choices or become wrong choices in retrospect due to a given delta. Such cases indeed act as an indicator to determine the capabilities of different consistency restoration approaches with TGGs. While some approaches [60, 70] simply do not consider (wrong) choices of forward rule applications and exclude TGGs with these search space problems, some of them [5, 93, 116] resort to additional dependency analyses, partly with user involvement [116], to detect invalidated alternatives.

A reasonable suggestion to tackle such search space problems in consistency restoration is to capture decisions again as an optimization problem as done for consistency checking in the previous section. Though unified with consistency checking and formally justified, we argue against performing multiple alternatives of forward rule applications due to two practical reasons:

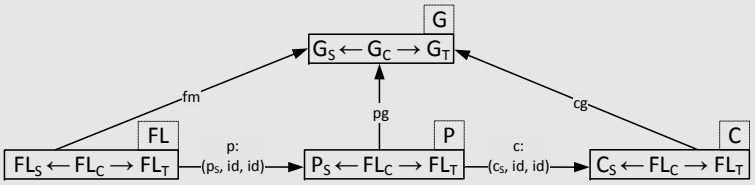
- First, it contradicts the performance-related motivations of incremental updates. We have made this compromise for consistency checking which is in our opinion a more difficult task as correspondences and a consistency history miss initially. For consistency restoration, however, exploring and maintaining alternative decisions can lead to high costs for propagating expectedly tiny deltas.
- Second, and more crucially, performing alternative forward rule applications means that superfluous elements (that are to be removed in retrospect after the optimization step) are not only created in the correspondence graph but also in the target graph. Though rather harmless in the correspondence graph, multiplicity constraints for a target graph can technically prevent creating superfluous target elements. We do not consider leaving the technical space of MDE and resorting to a hand-crafted model infrastructure (with relaxed multiplicity constraints) as a satisfactory solution.

We, therefore, argue that a consistency restoration process must be "governed" at rule application time and propose an algorithmic approach to consistency checking (and not an optimization-based approach as done for consistency checking). Before stating an algorithm for governing forward rule applications, we first discuss *application conditions* to "block" some wrong choices of forward rule applications.

4.4 Application Conditions

Application conditions are a common technique in graph grammars to forbid a rule application depending on the existence or absence of additional morphisms. The following definition is adopted from [42] with focus on forward rules. Moreover, application conditions in our context only relate to the source side of a triple graph, i.e., additional morphisms are only defined for the source graph while only *id* morphisms are used for the correspondence and target graphs.

Definition 35 (Application Conditions)**.**
Given a forward rule $fr : FL \rightarrow FR$ with $FL = FL_S \leftarrow FL_C \rightarrow FL_T$, an *application condition* $(p, \mathcal{C})$ for fr is given by a monomorphism $p : FL \rightarrow P$, called *premise*, with $P = P_S \leftarrow FL_C \rightarrow FL_T$ and $p = (p_S, id, id)$ and a set $\mathcal{C}$ of monomorphisms, called *conclusions*, with $\forall c \in \mathcal{C}, c : P \rightarrow C$ with $C = C_S \leftarrow FL_C \rightarrow FL_T$ and $c = (c_S, id, id)$. We refer to an application condition $(p, \mathcal{C})$ as a *negative application condition* if $\mathcal{C}$ is empty.
A forward rule application $G \overset{fr@fm}{\Longrightarrow} G'$ fulfills an application condition $(p, \mathcal{C})$ for fr if for each monomorphism $pg : P \rightarrow G$ with $pg \circ p = fm$ the diagram below commutes, i.e., there exist a conclusion $c : P \rightarrow C$ in $\mathcal{C}$ and a monomorphism $cg : C \rightarrow G$ with $cg \circ c = pg$.



It is important to note that a source graph does not evolve after a forward rule application while our application conditions introduce additional morphisms only on the source side. Hence, if a forward rule application fulfills its application conditions, it keeps doing so as long as the source graph is not changed by a delta (and, in fact, consequences of deltas concerning application conditions shall constitute one of the matters to be observed via incremental pattern matching).

Besides the general definition of application conditions, the special case of negative application conditions (NACs) is of particular interest in our running example. A NAC simply forbids the occurrence of its premise commuting with the match of the forward rule. As the set of conclusions is empty in a NAC, the requirement in Definition 35 is trivially unsatisfied when the premise occurs at all.

Example 23. In Figure 4.11, our forward rule fr_1 of ClassRule is extended with a NAC where the additional elements in P as compared to FL are depicted in a red rectangle with a «NAC»-markup (we do not have any conclusion in the case of a NAC). When marking a Java class with fr_1, this NAC can only be fulfilled if the Java class does not have any super class. Accordingly, assuming that a Java class has been marked with fr_1, the NAC can get violated in retrospect if the Java class gets a new super class due to a source delta.

A NAC as depicted in Figure 4.11 can automatically be generated using the state-of-the-art operationalization techniques for TGGs. Apparently developed independently from each other but pursuing the same goals, *filter NACs* in [64] and *Dangling Edge Conditions* (DECs) in [85] propose a technique to analyze forward rules in the following sense: If a forward rule marks a vertex but not all of its incident edges, there should exist further "candidates" of forward rules that would later mark these edges. Otherwise, a NAC is generated to forbid such edges, i.e., a forward rule application is blocked via NACs if some edges would remain unmarked otherwise. Recently, a more generalized analysis for incident edges of vertices have

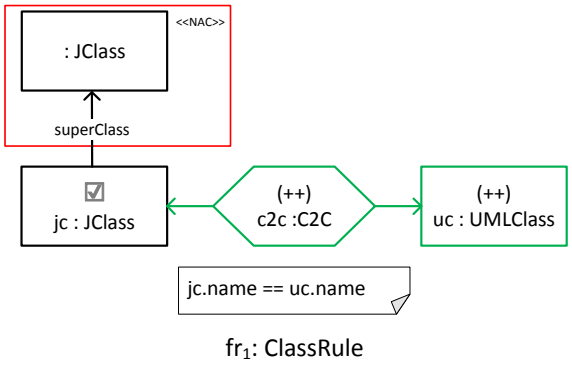


Figure 4.11: A NAC for the forward rule fr_1 of ClassRule

been proposed in [52]. In addition to [64, 85], the approach does not only generate NACs but also normal application conditions (with at least one conclusion) to check whether incident edges can eventually have a match for the candidate forward rules.

For the backward direction in our running example, however, the problems discussed in the previous subsection go beyond edges of single vertices and, therefore, beyond the capabilities of these techniques. More sophisticated and possibly manual analyses are required in this case to detect that Impl-class strategy must consequently be chosen to mark UML classes if multiple inheritance exists somewhere in an inheritance hierarchy.

4.5 Marking-Complete Forward Rules

Having introduced application conditions as an additional means for forbidding rule applications in certain cases, the next question is whether a set of forward rules can indeed be "arbitrarily" applied for an entire marking of a source graph G_S (consistency restoration might end up with a misleading result otherwise). We refer to forward rules exhibiting this property as *marking-complete*. Marking-completeness here should imply that we only deal with markings of the source elements and not with the correspondence and target graphs whose evolvement can still depend on the choices of forward rule applications.

Note that, as from now on, we only assume that some application conditions are given for the forward rules and state our results orthogonally to how they are constructed (be it by exploiting [52, 64, 85] or any future construction technique for application conditions). Our focus is rather what a derivation via forward rules must fulfill given these application conditions.

In a first step, we restrict ourselves to *source-progressive* TGGs where each forward rule application marks at least one source element (so that repeating a forward rule application over the same match arbitrarily many times is never needed).

Definition 36 (Source-Progressive TGGs)**.**
We refer to a TGG with the set $\mathcal{FR}$ of forward rules as *source-progressive* if each forward rule $fr \in \mathcal{FR}$ has at least one source marking element.

Example 24. The TGG for our running example is source-progressive as each forward rule (Figure 4.6 and 4.7) marks at least one source element. The analogous property for the backward direction, however, does not hold, i.e., the TGG is not *target-progressive* as some TGG rules do not create any target elements (and thus their respective backward rules do not mark any target elements).

Remark 6. If a TGG is not source-progressive (or target-progressive), it requires case-specific arguments whether the forward (or backward) rules violating the condition in Definition 36 can be "ignored" in a consistency restoration process. Regarding the backward direction in our running example, considering only those backward rules that mark at least one target element suffices to entirely mark a UML model (while the remaining rules would only create some additional Java elements without any contribution to the marking process).

What we want to avoid here is repeating some forward (or backward) rule applications in the hope that they would eventually lead to some new matches for other effectively marking forward (or backward) rules. This is not the case in our running example where we can simply restore consistency with a subset of the backward rules (that have at least one target marking element).

Next, we consider only those forward rule applications that (i) construct creation and context preserving derivations and (ii) satisfy their application conditions. Derivations with such forward rule applications are referred to as *final* if there does not exist any further forward rule application satisfying these properties.

Definition 37 (Valid and Final Derivations with Forward Rules)**.**
Given a source-progressive TGG with the set $\mathcal{FR}$ of the respective forward rules where each $fr \in \mathcal{FR}$ is provided with a set $\mathcal{AC}$ of application conditions, let $d : G_0 \overset{fr_1@fm_1}{\Longrightarrow} G_1 \ldots \overset{fr_n@fm_n}{\Longrightarrow} G_n$ be a derivation with $fr_1, \ldots, fr_n \in \mathcal{FR}$. We refer to d as a *valid* derivation if

- d is creation and context preserving,
- each forward rule application $\alpha_i : G_{i-1} \overset{fr_i@fm_i}{\Longrightarrow} G_i$ in d fulfills the respective set $\mathcal{AC}_i$ of application conditions.

Given d, a forward rule application $\alpha_{n+1} : G_n \overset{fr_{n+1}@fm_{n+1}}{\Longrightarrow} G_{n+1}$ with $fr_{n+1} \in \mathcal{FR}$ is *valid* if the derivation $d' : G_0 \overset{fr_1@fm_1}{\Longrightarrow} G_1 \ldots \overset{fr_n@fm_n}{\Longrightarrow} G_n \overset{fr_{n+1}@fm_{n+1}}{\Longrightarrow} G_{n+1}$ is valid. We refer to d as *final* if there does not exist any valid forward rule application α_{n+1}.

Example 25. Assuming that the forward rule fr_5 of SubClassRule is provided with the NAC depicted in Figure 4.11, the derivation depicted in Figure 4.8 is

valid and final fulfilling this NAC while the derivation in Figure 4.9 is not valid (violating the NAC).

In the case of a marking-complete set of forward rules as stated in the following definition, available forward rule applications can be arbitrarily chosen (either randomly or by asking model owners) to entirely mark a source graph G_S.

Definition 38 (Marking-Complete Forward Rules)**.**
Given a source-progressive TGG, let $\mathcal{FR}$ be the set of the respective forward rules where each $fr \in \mathcal{FR}$ is provided with a set $\mathcal{AC}$ of application conditions. We refer to $\mathcal{FR}$ as *marking-complete* if, for each $G_S \in \mathcal{L}_S(TGG)$, all valid and final derivations $d : G_0 \overset{fr_1@fm_1}{\Longrightarrow} G_1 \ldots \overset{fr_n@fm_n}{\Longrightarrow} G_n$ with $G_0 = G_S \leftarrow \emptyset \rightarrow \emptyset$ and $fr_1, \ldots, fr_n \in \mathcal{FR}$ are entirely marking (Definition 34).

While Definition 38 states what we require in an abstract manner, the following points intuitively add up to marking-completeness (or constitute a threat to it):

- The evolvement of the correspondence and target graph must not lead to a state where no more forward rule applications can be found while G_S is still not entirely marked,
- The markings by forward rules must not lead to a state where any further marking would necessarily violate creation and context preservation while G_S is still not entirely marked,
- Application conditions must block forward rule applications to prevent that one of the previous two conditions is violated.

Remark 7. Our definition of marking-complete forward rules represent the minimum requirement for a "successful" consistency restoration run. In particular, application conditions are required to block wrong choices of forward rule applications but we do not make any explicit statement on blocking some correct choices. We indeed do not forbid this as long as there exists for each source graph at least one entirely marking (and valid) derivation. In summary it can be said, therefore, that we conservatively allow application conditions to be more restrictive than necessary.

The backward direction of our running example helps us construct a situation justifying this choice. As previously discussed (mainly based on Figure 4.10), the Impl-class strategy must be used to mark a UML class in the backward direction as long as multiple inheritance exists in the inheritance hierarchy (of the marked UML class). Application conditions navigating through a hierarchy of arbitrary depth, however, are not possible in our case as we only resort to the classical theory of graph grammars consisting of single, and not recursively nested, graph patterns. Given this expressiveness issue, enforcing the Impl-class strategy if multiple inheritance exists "somewhere" in the UML model can be a good compromise for the sake of guaranteeing successful consistency restoration. In Figure 4.12, the backward rule br_1 of ClassRule is enriched with three

NACs forbidding multiple inheritance globally for applying this backward rule. NAC1 and NAC2 relate to the neighborhood of the UML class, while NAC3 relates to any other location (NAC3 does not cover the neighborhood of the class as matches are injective in our formalization). These NACs, in particular NAC3, can possibly block harmless one-to-one mappings of UML classes to Java classes but at least the Impl-class strategy still remains a possibility for marking all UML models entirely. Nevertheless, if possible, it is more ideal from a practical point of view that application conditions only block wrong choices.

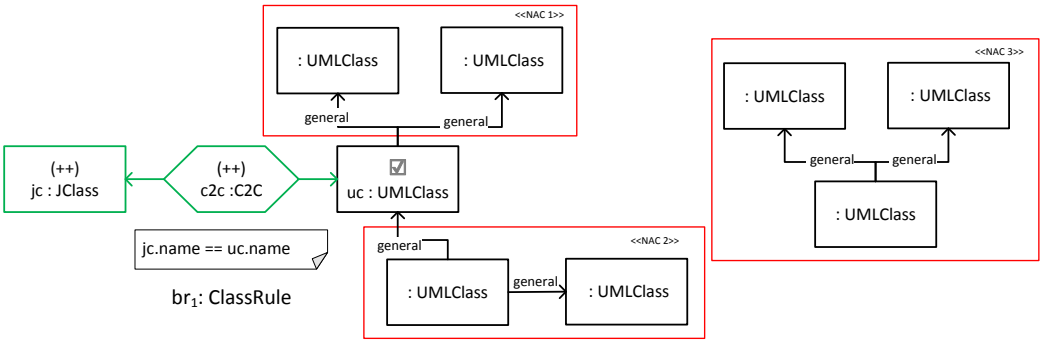


Figure 4.12: Three conservative NACs for the backward rule br_1 of ClassRule forbidding multiple inheritance globally (the Impl-class strategy remains the only way for marking UML classes in the case of multiple inheritance)

A further relevant question in the current research on TGGs is how to check whether a set of forward rules with application conditions is marking-complete. Currently, there exist two different static analysis techniques (both adopted from the general field of graph grammars) that state sufficient conditions for applying forward rules arbitrarily (and getting a consistent triple graph at the end):

- In [65], confluence is required among forward rules such that each derivation starting from the same source graph results in the same triple graph (i.e., determinism is not only required for marking the source side but also for the evolvement of the entire triple).

- In [9], furthermore, the so-called *local-completeness* property and a static analysis to check this property have been proposed. The approach makes use of *source rules* (not required in our setting) that only mark a source graph without requiring or creating any correspondence and target elements. Using the source rules, an ordering is calculated for how to mark the individual elements of a source graph G_S (an intermediate condition is that G_S can always be entirely marked by these source rules). Accordingly, local-completeness means that the forward rules can indeed mark G_S in this calculated order. This way, the approach strives to govern forward rule applications solely based on analyzing the source side. While the order of source elements to be marked is fixed, the choice of forward rules is not fixed (in the case of alternatives) differentiating this technique from confluence.

Our forward rules, for example, are not confluent in the forward direction as we allow different UML models for the same Java model depending on whether (or to what extent) the Impl-class strategy is used. Nevertheless, the local-completeness property is satisfied as our rules are rather of symmetric nature. That is, a match for the source side implies a match for the entire forward rule.

Unfortunately, or perhaps fortunately from a restrictiveness point of view, none of the confluence and local-completeness requirements proves to be a generalization of the other. That is, a set of forward rules can be confluent but not local-complete, or the other way around, but is in both cases marking-complete. Nevertheless, it should be mentioned that the motivation of local-completeness is to increase the industrial applicability of TGGs while confluence enforces TGGs to be a function from source to target graphs.

Besides these techniques (and their extension/combination which is the focus of an ongoing research project), it is also worthwhile to consider *state space exploration* (e.g., as practically supported by [55]) as a third option to check whether a set of forward rules always entirely marks source graphs (or at least a representative group of source graphs). Though sounding inefficient, it should be noted that such an analysis is to be performed at specification time and not at runtime. The rest of this section (and the core of our contribution) rather focuses on runtime.

4.6 A Consistency Restoration Procedure

We discuss in the following a consistency restoration procedure that "reacts" to the consequences of a delta applied to the source graph of a consistent triple graph. We define some concepts that are relevant to this procedure, provide subsequently pseudo code for consistency restoration in the forward direction, demonstrate its intermediate and end results based on the three examples previously discussed via human intuition at the beginning of this section, and, finally, conclude with stating the formal guarantees of consistency restoration if certain conditions are met.

In a first step, we formally define a delta changing a graph G to G' as a span of two monomorphisms $G \leftarrow G^- \rightarrow G'$ (not to be confused with a triple graph). The first monomorphism $G \leftarrow G^-$ indicates which elements are deleted in G (namely those that are not involved in G^-). The second monomorphism $G^- \rightarrow G'$, furthermore, indicates which elements are added (namely those that are in G' but not in G^-).

Definition 39 (Delta)**.**
A *delta* $\delta : G \leftarrow G^- \rightarrow G'$ for a graph G is given by two monomorphisms $\delta^- : G \leftarrow G^-$ and $\delta^+ : G^- \rightarrow G'$ in **Graphs**.
Given a delta $\delta : G \leftarrow G^- \rightarrow G'$, an element $e \in \mathsf{elements}(G)$ with $\nexists\, e' \in \mathsf{elements}(G^-)$ such that $\delta^-(e') = e$ is referred to as *deleted* by δ. Accordingly, an element $e \in \mathsf{elements}(G')$ with $\nexists\, e' \in \mathsf{elements}(G^-)$ such that $\delta^+(e') = e$ is referred to as *created* by δ.

As the forward direction is the focus in our statements, we consider *source deltas* of the form $\delta_S : G_S \leftarrow G_S^- \rightarrow G_S'$ changing a source graph G_S to G_S'.

Example 26. Changes on the Java models depicted in Figure 4.1, 4.2, and 4.3 are exemplary source deltas.

Given a derivation d of forward rule applications, a source delta δ_S does not only lead to new forward rule applications but also invalidates some existing ones in d. This must be handled by *revoking* the respective forward rule applications. Revoking a forward rule application α means that (i) the created correspondence and target elements are deleted (but no source element is deleted as the source graph is only to be changed by δ_S) and (ii) the source elements marked by α become *unmarked*.

Definition 40 (Revoking a Forward Rule Application)**.**
Let $\alpha : G \overset{fr@fm}{\Longrightarrow} G'$ be a forward rule application with $G = G_S \leftarrow G_C \rightarrow G_T$ and $G' = G'_S \leftarrow G'_C \rightarrow G'_T$. *Revoking* α refers to deleting all elements in $\mathsf{elements}(G'_C) \setminus \mathsf{elements}(G_C)$ and in $\mathsf{elements}(G'_T) \setminus \mathsf{elements}(G_T)$. All source elements in $\mathsf{marked}(\alpha)$, furthermore, are said to be *unmarked* when revoking α.

Remark 8. When deleting graph elements for revoking a forward rule application as stated in Definition 40, "dangling edges" can occur which are not deleted (for the moment) but their vertices are. As we shall discuss in the next statements, nevertheless, revoking a forward rule application triggers further revocations such that the dangling edges are also eventually deleted. In practice as well as in our examples, deleting a vertex implicitly deletes its incident edges (i.e., some edges are cleaned up in advance whereas upcoming revocations would delete them as well).

Besides forward rule applications, deltas and revoking forward rule applications constitute now two further concepts that change a triple graph during a consistency restoration process. Our strategy for consistency restoration is mainly concerned with identifying the consequences of these changes. We, therefore, discuss in the following what is to be "observed and reported" when restoring consistency.

Obviously, a consistency restoration procedure must perform valid forward rule applications (Definition 37). Before this step, however, *invalidated* forward rule applications must be detected and revoked. A forward rule application α might be invalidated due to three different reasons: First, some elements matched by α might have been deleted (by a source delta or by revoking some other forward rule applications). Second, some source elements required by α ($\mathsf{required}(\alpha)$) might have been unmarked (by revoking some other forward rule applications). Finally, the application conditions stated for α might be no more fulfilled (e.g., due to the source delta adding some forbidden elements).

Definition 41 (Invalidated Forward Rule Applications)**.**
Given a TGG with the set $\mathcal{FR}$ of the respective forward rules where each $fr \in \mathcal{FR}$ is provided with a set $\mathcal{AC}$ of application conditions, let $d : G_0 \overset{fr_1@fm_1}{\Longrightarrow} G_1 \ldots \overset{fr_n@fm_n}{\Longrightarrow} G_n$ be a valid derivation with $fr_1, \ldots, fr_n \in \mathcal{FR}$.

When changing G_n by a source delta or by revoking some forward rule applications in d, we refer to a forward rule application $\alpha_i : G_{i-1} \overset{fr_i@fm_i}{\Longrightarrow} G_i$ in d as *invalidated* if at least one of the following conditions occurs:

- some elements matched by α_i, i.e., $fm_i(G_{i-1})$, are deleted,
- some elements in $\mathsf{required}(\alpha_i)$ are unmarked,
- α_i does not fulfill the set $\mathcal{AC}_i$ of application conditions for fr_i.

Before introducing our consistency restoration procedure, we briefly and informally discuss its underlying technical component, namely an incremental pattern matcher. While incremental pattern matchers have the general purpose of observing appearances and disappearances of predefined graph patterns in a host graph, we exploit this concept to detect valid and invalidated forward rule applications (stated in Definition 37 and 41, respectively). That is, the left- and right-hand sides of our forward rules together with application conditions and marking information constitute what we observe via incremental pattern matching. In Figure 4.13, these patterns are exemplified for the forward rules fr_1 and fr_5.

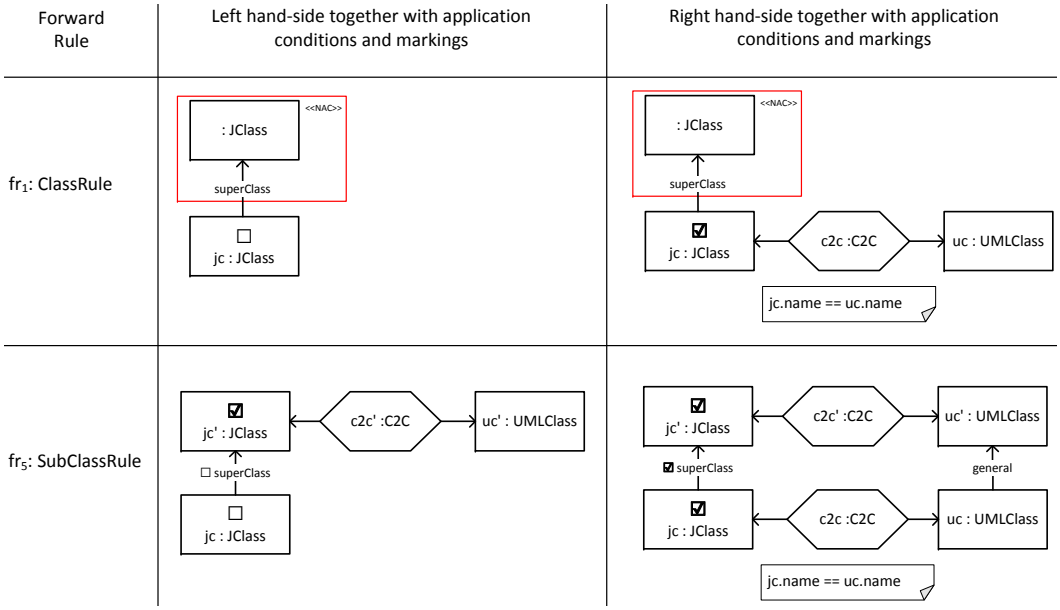


Figure 4.13: Patterns to be observed for valid and invalidated applications of fr_1 and fr_5

In the case of fr_1, for example, assuming furthermore the NAC depicted in Figure 4.11, unmarked Java classes without any super class are observed for the left hand-side. Each appearance of such a pattern reported by the incremental pattern matcher points at an available valid application of fr_1. After applying fr_1, obviously, a match for such a pattern disappears (as the marking state changes). For the right hand-side of fr_1, furthermore, marked Java classes without a super class but with a corresponding UML class are observed. Each disappearance of such a pattern reported by the incremental pattern matcher points at an invalidated application of fr_1. This holds analogously for the left and right-hand side of fr_5 (depicted at the bottom of Figure 4.13), this time without any application condition but with

some context elements whose existence and markings are observed as well. To sum up, appearance of the patterns shown in the left column of Figure 4.13 and disappearance of the patterns shown in the right column constitute the events which are to be reported by an incremental pattern matcher and to which consistency restoration needs to react.

Considering the state-of-the-art support [139, 142] for incremental pattern matching, the *Rete algorithm* [50] seems to have gained acceptance as the underlying technique. The Rete algorithm stores partial matches of patterns as data tuples and joins them to complete matches. Its strength lies in eliminating redundancy of matched structures depending on how the data tuples are chosen for partial matches, i.e., a data tuple is maintained once but can be used as a part of multiple matches for possibly different patterns. Given that some changes are made in the host graph, individual data tuples containing the changed elements are updated (instead of a complete re-evaluation of the patterns), hence the qualification as "incremental". Storing partial matches, of course, can be regarded as sacrificing memory for faster evaluations as compared to conventional pattern matching.

While we strongly recommend the interested reader to consult [50, 142] for the insights of the Rete algorithm, it is transparent (invisible) to our upcoming consistency restoration procedure how an incremental pattern matcher internally works. We, therefore, only define the interaction with such a component (this interaction basically must be implemented as an interface between a TGG engine and a general-purpose incremental pattern matcher).

Definition 42 (Incremental Pattern Matcher)**.**
Given a TGG with the set $\mathcal{FR}$ of the respective forward rules with a set $\mathcal{AC}$ of application conditions, let $d : G_0 \overset{fr_1@fm_1}{\Longrightarrow} G_1 \ldots \overset{fr_n@fm_n}{\Longrightarrow} G_n$ be a derivation with $fr_1, \ldots, fr_n \in \mathcal{FR}$. An *incremental pattern matcher* $\mathsf{ipm}(\mathcal{FR}, \mathcal{AC}, d)$ refers to a component with the following operations:

- ipm.getValid returns all possible valid applications of forward rules in $\mathcal{FR}$ that can be added to d,
- ipm.getInvalidated returns all invalidated forward rule applications in d that are not revoked.

Finally, Algorithm 2 presents our procedure in pseudo code which requires, besides the incremental pattern matcher, the following inputs: A source-progressive TGG whose forward rules are marking-complete (possibly provided with additional application conditions), a consistent triple graph $G_S \leftarrow G_C \rightarrow G_T$ whose creation is traced to a final derivation d via the forward rules, and, finally, a source delta δ_S that changes G_S to G'_S.

Overall, the incremental pattern matcher operates on the input derivation d. The changed derivation after revoking and performing forward rule applications is stored in d'. The evolvement of G, furthermore, is stored in G'. Note that both G' and d' together constitute the output of Algorithm 2 and serve as input for further runs. The consistency restoration procedure consists of the following three phases to produce G' and d':

Algorithm 2 Consistency Restoration

Require:

TGG : A source-progressive TGG

$\mathcal{FR}$: A marking-complete set of forward rules for TGG with a set $\mathcal{AC}$ of application conditions

G : A triple graph $(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$

d : A valid, final, and entirely marking derivation $G_0 \overset{fr_1@fm_1}{\Longrightarrow} G_1 \ldots \overset{fr_n@fm_n}{\Longrightarrow} G_n$ with $fr_1, \ldots, fr_n \in \mathcal{FR}$, $G_0 = G_S \leftarrow \emptyset \rightarrow \emptyset$, and $G_n = G$

δ_S : A source delta $G_S \leftarrow G_S^- \rightarrow G_S'$

ipm : an incremental pattern matcher ipm$(\mathcal{FR}, \mathcal{AC}, d)$

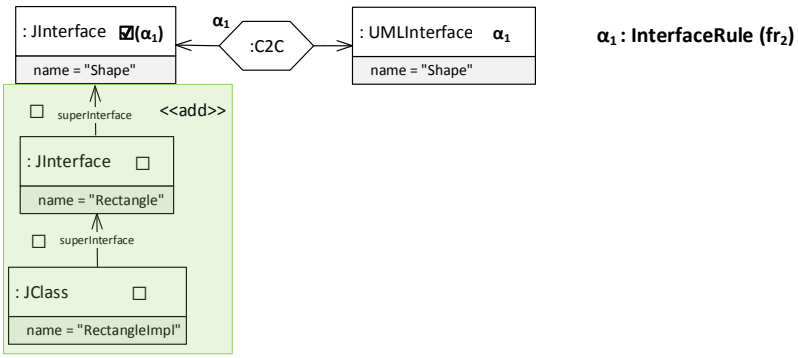
1: **procedure** RestoreConsistencyInForward(G, d, δ_S, ipm)
2:
3: $G' \leftarrow$ apply δ_S to G ▷ **Phase 1**
4: $d' \leftarrow d$
5:
6: **while** ipm.getInvalidated is not empty **do** ▷ **Phase 2**
7: $\alpha \leftarrow$ choose from ipm.getInvalidated
8: $(G', d') \leftarrow$ revoke α
9: **end while**
10:
11: **while** ipm.getValid is not empty **do** ▷ **Phase 3**
12: $\alpha' \leftarrow$ choose from ipm.getValid
13: $(G', d') \leftarrow$ perform α'
14: **end while**
15:
16: **if** d' is entirely marking **then return** (G', d')
17: **else** Error: $\nexists\, (G_S' \leftarrow G_C' \rightarrow G_T') \in \mathcal{L}(TGG)$
18: **end if**
19:
20: **end procedure**

- **Phase 1** (Line 3-4): δ_S is applied to the source graph G_S in this initial phase. This step can lead to valid and/or invalidated forward rule applications (ipm will be inquired about these in the next phases).
- **Phase 2** (Line 6-9): Invalidated forward rule applications in d are revoked in this intermediate phase. First and foremost, deletions in δ_S lead to the disappearance of some matches of forward rule applications where additions as well as deletions in δ_S can possibly lead to violation of some application conditions in d. Revoking a forward rule application, furthermore, can lead to further invalidated forward rule applications (as correspondences and target elements are deleted and source elements become unmarked). This phase exits after no more invalidated forward rule application is reported by ipm.
- **Phase 3** (Line 11-14): Valid forward rule applications are performed in this final phase. Valid rule applications can occur due to additions in δ_S (where added elements are inherently unmarked) and also due to revoking steps in the previous phase (where some source elements become unmarked but not deleted by δ_S). In each iteration, a valid forward rule application is chosen and performed. This, in turn, might give rise to new valid forward rule applications as the correspondence and the target graph as well as the markings at the source side evolve. Note that the choice of performed forward rule application in each iteration (Line 12) can either be made arbitrarily or by asking the model owner but does not constitute a threat to marking G'_S entirely if the forward rules are marking-complete. This phase exits after no more valid forward rule application is reported by ipm.

The resulting G' is a triple graph $G'_S \leftarrow G'_C \rightarrow G'_T$ produced from $G_S \leftarrow G_C \rightarrow G_T$ after applying δ_S, revoking invalidated rule applications, and performing valid rule applications. The changed derivation d' contains those forward rule applications that are either retained (not revoked) in Phase 2 or performed in Phase 3. If d' is not entirely marking, however, the procedure terminates with an error stating that no $G'_S \leftarrow G'_C \rightarrow G'_T$ exists. We will conclude this subsection with a theorem showing that this error state is never reached if $G'_S \in \mathcal{L}_S(TGG)$.

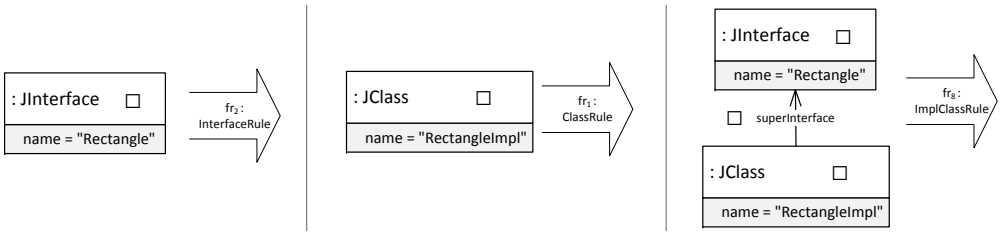
Before discussing the formal guarantees of Algorithm 2, however, we first demonstrate how it operates using the three examples of consistency restoration discussed with human intuition at the beginning of this section.

Example 27. We start with the example previously depicted in Figure 4.1. As repeated in the following diagram, the consistent triple graph contains a pair of Java and UML interfaces (named Shape) and the source delta adds a new Java interface Rectangle (inheriting from Shape) as well as a new Java class RectangleImpl (inheriting from Rectangle). The derivation marking the initial state of the source graph (required as d in Algorithm 2) consists of only one forward rule application (α_1 via the forward rule fr_2 of InterfaceRule).



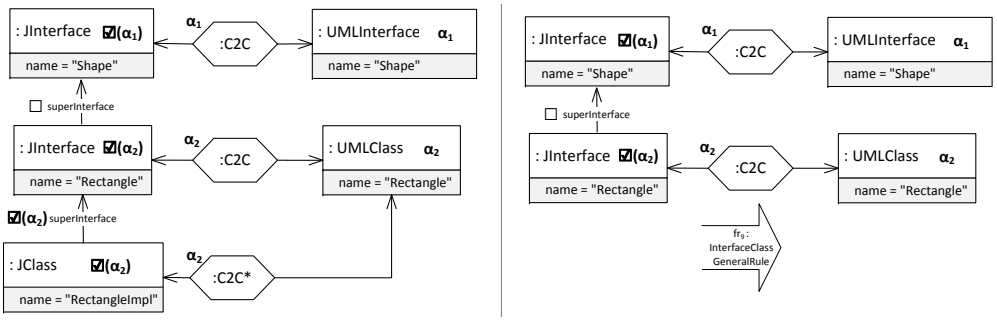
Given these inputs, we discuss the intermediate and end results of the three phases in Algorithm 2.

Phase 1: The source delta does not invalidate α_1. That is, the source delta neither deletes an element in the match of α_1 nor violates its application conditions (in fact, we do not have any application condition stated for fr_2). The source delta, however, gives rise to matches for three valid forward rule applications depicted below (the arrows explicitly indicate with which forward rules). Note that the available forward rule applications partly overlap in their marked source elements (i.e., we have alternatives to mark the added Java elements).

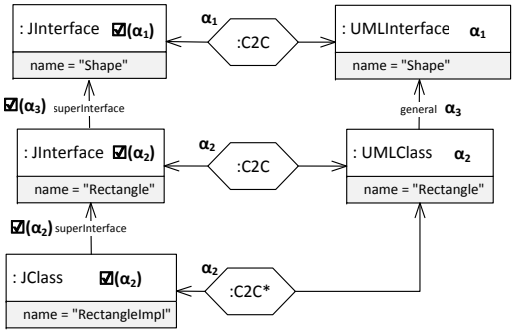


Phase 2: As the source delta does not invalidate any forward rule application from former runs, this phase exits without any iteration.

Phase 3: Having the three valid forward rule applications induced due to the source delta, one of them is to be chosen in the first iteration of this phase. Depending on what a concrete implementation supports, this choice can either be made arbitrarily or some user decisions can be incorporated (be it over user interaction or some predefined configuration). We assume that user decisions are supported and choose to apply the forward rule fr_8 of ImplClassRule (i.e., we prefer the Impl-class strategy to a one-to-one mapping of interfaces and classes). The result after applying the chosen forward rule is depicted to the left in the following diagram. While the other two forward rule applications induced due to the source delta are no more valid (as they mark the same Java elements), a new valid forward rule application becomes available to mark the superInterface edge between the Rectangle and the Shape interface (depicted to the right). Hence, a second iteration is required in this phase where there is only one valid forward rule application to be chosen.

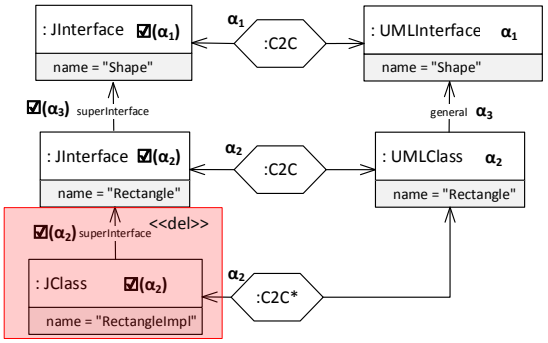


After performing this valid forward rule application in the second iteration, all source elements are marked and this phase accordingly exits. The end result consists of, as depicted below, the extended triple graph and the derivation containing now three forward rule applications.



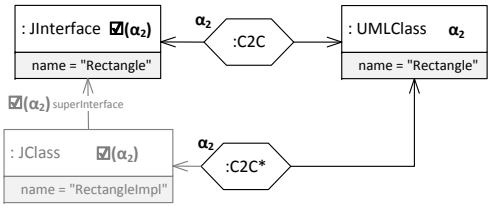
α1 : InterfaceRule (fr2)
α2 : ImplClassRule (fr8)
α3 : InterfaceClass-GeneralRule (fr9)

Example 28. We start with the end result of the previous example and delete the RectangleImpl class in the Java model as depicted below.

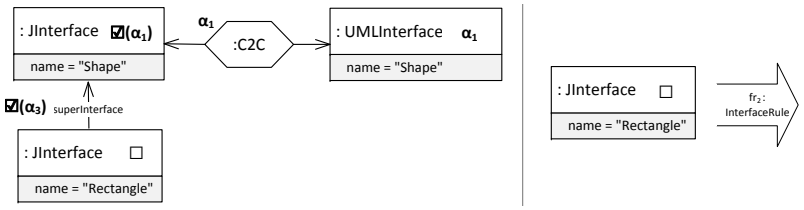


α1 : InterfaceRule (fr2)
α2 : ImplClassRule (fr8)
α3 : InterfaceClass-GeneralRule (fr9)

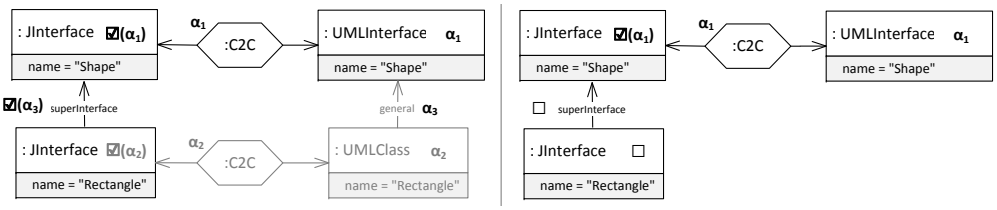
Phase 1: The source delta invalidates α_2 (and for the moment only α_2) as some of its matched source elements are deleted. The following diagram depicts α_2 while the elements that are now "missed" by α_2 are grayed out.



Phase 2: In the first iteration of this phase, α_2 is to be revoked which means that (i) the Rectangle interface in the Java model is now unmarked and (ii) the Rectangle class in the UML model (which was created by α_2) is deleted together with its outgoing edge. The correspondences created by α_2 are deleted as well. The state of the triple graph after revoking α_2 is depicted below to the left. Unmarking the Rectangle interface in the Java model, moreover, gives rise to a new valid forward rule application as depicted to the right.



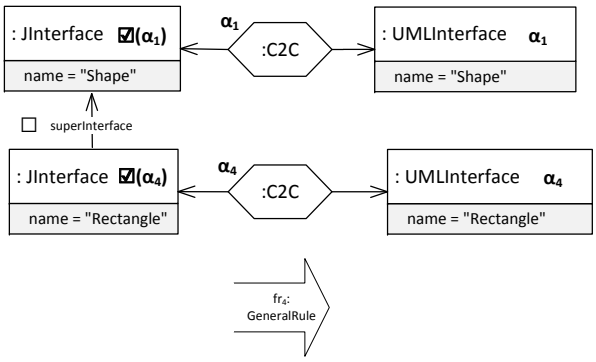
Performing the new forward rule application, however, is the task of the next phase while we are not yet finished in this phase. Revoking α_2 invalidates α_3. As depicted below to the left, α_3 now misses some of its required UML elements and correspondences as well as the marking of the Java interface Rectangle (all grayed out). Hence, a second iteration is needed to revoke α_3. The state after revoking α_3 is depicted below to the right where the superInterface edge in the Java model is now unmarked. This phase finally exits as α_1 (being the only remaining forward rule application from the former runs) is not invalidated.



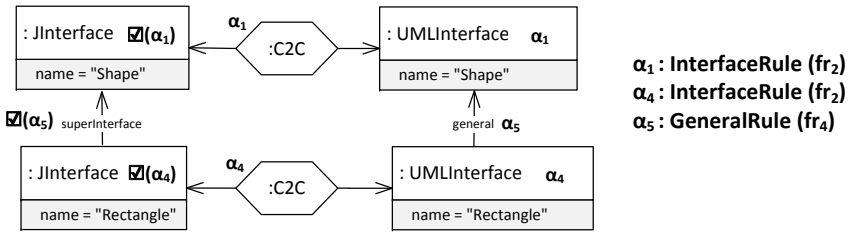
Most importantly, the end result of this phase consists of only valid forward rule applications and the source graph is partially marked. The remaining markings are to be complemented in the next phase.

Phase 3: In the first iteration, the only valid forward rule application that became available in the previous phase is performed (the Rectangle interface in the Java model is marked and a corresponding Rectangle interface in the UML model is created). The state of the triple graph is depicted in the following diagram. The created elements and markings now give rise to a new valid

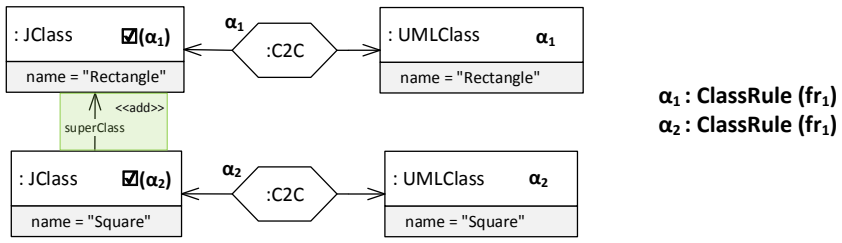
forward rule application via the forward rule fr_4 of GeneralRule (the entire triple graph below constitutes the match for this forward rule application).



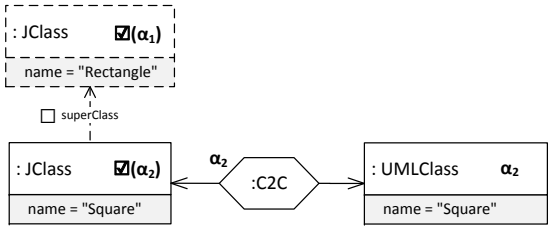
Performing this valid forward rule application in the second iteration of this phase, the superInterface edge in the Java model is marked and a general edge in the UML model is created. Having all source elements marked, this phase exits with the result depicted below. Overall, α_1 is retained from former runs, α_2 and α_3 have been revoked, and finally, α_4 and α_5 have been performed.



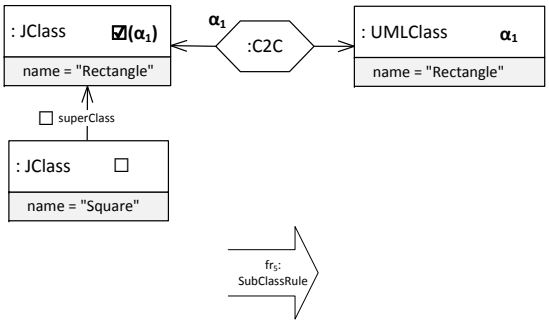
Example 29. In a last example, we demonstrate how an added element invalidates a forward rule application, previously shown in Figure 4.3. As depicted below, the initially consistent state consists of two pairs of Java and UML classes. The Java classes are marked by two forward rule applications α_1 and α_2 via the forward rule fr_1 of ClassRule. The source delta adds an inheritance link.



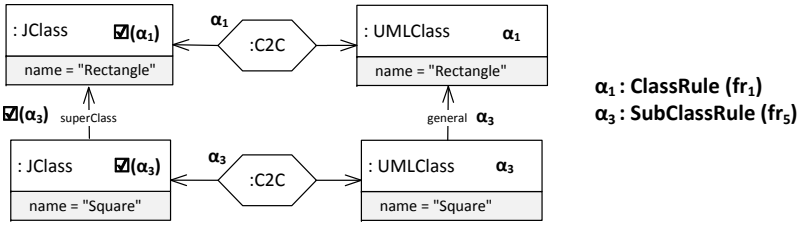
Phase 1: Applying the source delta to the Java model and assuming that the forward rule fr_1 of ClassRule is provided with the NAC depicted in Figure 4.11, the forward rule application α_2 is invalidated. In the following diagram, the new Java elements that violate the NAC are depicted via dashed lines.



Phase 2: In the first (and only) iteration, α_2 is revoked. The result is depicted below. This, moreover, gives rise to a new valid rule application via the forward rule fr_5 of SubClassRule (where the entire triple graph constitutes the match). As α_1 is not invalidated, this phase exits after one iteration.



Phase 3: Performing the only available valid forward rule application, the Java model is again entirely marked. The end result, as depicted below, is a triple graph having now inheritance links at both sides. Overall, α_1 is retained from former runs, α_2 is revoked due to the added elements, and α_3 has been performed to complement the missing markings.



A noteworthy aspect of this example is that some created UML elements are identical to some deleted UML elements. In particular, the UML class Square is deleted when revoking α_2 but a UML class Square is again created when performing α_3. While deleting something just in order to create it again is not always ideal especially with regard to information preservation, possible measures (and extensions) addressing such cases shall be discussed in the next subsection.

We conclude the contribution part of this section by discussing the formal guarantees of Algorithm 2. First, termination of Algorithm 2 is shown, i.e., both loops (in Phase 2 and 3) exit after a finite number of iterations.

Theorem 3 (Termination of Algorithm 2)**.**
Given its required inputs, Algorithm 2 always terminates.

Proof. Termination of the individual loops in Phase 2 and Phase 3 must be shown for the proof. Given the input derivation d, Phase 2 only revokes some forward rule applications in d. Phase 2 terminates as d is finite (in the worst case all forward rule applications are revoked). Considering Phase 3, each iteration marks some elements of G'_S as the TGG is source-progressive, while valid forward rule applications never overlap in their marked elements (Definition 37). As $\mathsf{elements}(G'_S)$ is finite, the number of iterations in Phase 3 is finite. Phase 3, therefore, terminates as well. □

Finally, we relate Algorithm 2 to the research challenges stated in [129] for consistency restoration with TGGs. In particular, [129] identifies correctness, i.e., that the returned triple graph is consistent, and completeness, i.e., that a triple graph is returned for all $G'_S \in \mathcal{L}_S(TGG)$, as two desired properties. Of course, source-progressive TGGs and marking-complete forward rules define our required conditions to fulfill these properties with Algorithm 2. The following theorem is stated for all $G'_S \in \mathcal{L}_S(TGG)$ and thus captures both correctness and completeness at the same time. Hence, the error state in Algorithm 2 can only be reached if $G'_S \notin \mathcal{L}_S(TGG)$.

Theorem 4 (Correctness and Completeness of Algorithm 2)**.**
Given its required inputs and provided that $G'_S \in \mathcal{L}_S(TGG)$, Algorithm 2 returns a triple graph G' such that $G' \in \mathcal{L}(TGG)$.

Proof. We consider the three phases of Algorithm 2 with the following inputs required as a starting point:

- A source-progressive TGG,
- A marking-complete set $\mathcal{FR}$ of forward rules for TGG,
- A triple graph $G = G_S \leftarrow G_C \rightarrow G_T$ with $G \in \mathcal{L}(TGG)$,
- A valid, final, and entirely marking derivation $d : G_0 \overset{fr_1@fm_1}{\Longrightarrow} G_1 \ldots \overset{fr_n@fm_n}{\Longrightarrow} G_n$ with $fr_1, \ldots, fr_n \in \mathcal{FR}$, $G_0 = G_S \leftarrow \emptyset \rightarrow \emptyset$, and $G_n = G$,
- A source delta $\delta_S : G_S \leftarrow G_S^- \rightarrow G'_S$.

Phase 1: G_C and G_T are not changed in this phase but only G_S is changed to G'_S.
Phase 2: This phase directly exits if applying δ_S in the previous phase did not invalidate any forward rule application in d. Otherwise, invalidated forward rule applications are revoked. Revoking a forward rule application, however, can invalidate further forward rule applications in d which are also revoked. Each individual revoking clears the markings and created elements of the respective forward rule application. Provided that $\mathsf{ipm.getInvalidated}$ returns all invalidated forward rule applications as required in Definition 42, the remaining (non-revoked) forward rule applications are valid, i.e., they still satisfy their required context, application conditions, and marking states. Hence, the resulting derivation d' at the end of Phase 2 represents a valid (but possibly not final) derivation starting from $G'_S \leftarrow \emptyset \rightarrow \emptyset$.

Phase 3: Valid forward rule applications are performed in this phase until d' is final. A forward rule application α' performed in this phase never invalidates previously performed ones. In particular, it holds that:

- α' does not delete any element. Hence, all forward rule applications in d' further on satisfy their context.
- α' does not change G'_S but only creates new markings of G'_S. Hence, application conditions of all forward rule applications in d' are further on fulfilled as these solely relate to G'_S but not to its markings.
- α' neither deletes markings nor overlaps with other forward rule applications in its markings. Hence, d' further on remains creation and context preserving.

Provided that ipm.getValid returns all possible valid forward rule applications in each iteration, the derivation d' is final at the end of this phase. If $G'_S \in \mathcal{L}_S(TGG)$, moreover, d' is entirely marking when final as $\mathcal{FR}$ is marking-complete (Definition 38). Finally, being creation and context preserving, d' can be traced back to a derivation via the original TGG rules (Lemma 4). Hence, $G' \in \mathcal{L}(TGG)$. □

Finally, the complexity-related consequences of using an incremental pattern matcher merit mentioning as compared to conventional pattern matching strategies (i.e., pattern matching without any internal data tuples for maintaining partial matches). In particular, using a conventional pattern matching strategy, the runtime complexity of consistency restoration with TGGs is estimated to be $O(n^k)$ in [94] where n is the size of the input model and k is the size of a rule (size refers to the number of vertices and edges in graphs). This estimation is based on the runtime complexity of conventional pattern matching in general. The runtime complexity of state-of-the-art incremental pattern matchers based on Rete, however, is $O(n^{2k-1})$ [50] as partial matches are maintained as well, i.e., almost a quadratic growth of complexity can be expected as compared to conventional solutions. In the incremental case, nevertheless, n does not refer to size of entire graphs but to the number of elements operated on (e.g., new elements that are added by a delta). Hence, it can be expected that incremental pattern matching brings an overhead if the delta size is large (e.g., in the case of an initial transformation) but scales well for small deltas. Our experimental evaluation in the upcoming section also backs up this expected overhead (and, in fact, custom optimizations for conventional pattern matching even enlarge the difference in the current tooling). While improvements in the sense of complexity and scalability rely on progress in research on incremental pattern matching, simplifying consistency restoration and providing a viable basis for developing meta-tools remain the main advantages of our concept.

4.7 Related Work

Although bidirectionality does not necessarily mean incrementality, consistency restoration in the sense of incremental updates has mostly been investigated within the BX field. We, therefore, first discuss different BX approaches to consistency restoration in the following and, subsequently, consider some *unidirectional* approaches that pursue the same goals.

TGGs: We have already mentioned between the lines that consistency restoration has been the main focus of research on TGGs since their introduction.

In particular, the *precedence-driven* approaches [5, 93, 116] shape the most representative group formalizing consistency restoration over precedences to decide what is to be done first and next. In [93], TGG rules are analyzed and precedences are calculated over model elements inducing a partial order for markings. Transferred to our running example, this would mean that marking a Java class "precedes" marking its subclasses and methods (and marking a method precedes marking its parameters). Given a delta changing the source model, consistency restoration first updates the precedences and concludes which elements are affected by the delta, namely those "losing" their predecessors (due to deletions) or getting new predecessors (due to additions). As the calculation of precedences is governed by static information gathered from TGG rules, however, discrepancies are possible between calculated and actual precedences for a concrete model. That is, a precedence relation between two model elements in the sense of [93] does not necessarily mean that they must be marked in this precedence order. This over-approximation of precedences, of course, can mislead the process of applying or revoking operational rules in consistency restoration.

Refining the precedence idea of [93], therefore, calculating the precedences over model elements is detached in [116] from static information (and only actual precedences are considered). Though avoiding over-approximation, the approach suffers from under-approximation especially when handling additions. If an added element changes the predecessors of an already existing element (e.g., when an existing class gets a new super class as we have exemplified), the proposed algorithm resorts to fallback procedures that are not further specified (user involvement, heuristics, or backtracking are mentioned as possibilities).

Addressing the over- and under-approximation issues, precedences are calculated in [5] directly over the source matches of forward rules. The approach performs auxiliary procedures dedicated to updating precedences for a given delta, and consistency restoration subsequently operates upon the results of these procedures. The proposed algorithm governs consistency restoration solely over the precedences of source matches. However, it can terminate with an error if the correspondence and target graphs have indeed an impact on which forward rule to apply next. In other words, a source match whose turn has come according to precedences must necessarily guarantee a successful forward rule application irrespectively of how the correspondence and target graphs have evolved (the aforementioned local-completeness property refers to this requirement). We relinquish the decomposition into source matches and govern consistency restoration over the entire matches of forward rules. Calculating the individual steps, moreover, is outsourced to incremental pattern matching instead of auxiliary procedures which, in our belief, is advantageous from a practical and didactic point of view.

Different than precedences, a special type of correspondences as introduced in [70] is another means to govern consistency restoration. While a correspondence in the sense of [70] has connections to all source and target elements within a rule application, the proposed algorithm navigates from changed elements to their correspondences, and from there to all other affected elements. Besides the existence

of at least one correspondence in each TGG rule, a further assumption is confluence, i.e., the evolvement of the triple graph at the end of consistency restoration is deterministic. The focus of the approach lies in performant consistency restoration. To this end, repairing forward rule applications is discussed instead of revoking them. This increases information preservation capabilities and reduces the effort in consistency restoration, especially when revoking a forward rule application has a domino effect invalidating some further ones. Repair rules, unfortunately, are not specified or formalized in detail. Our reactive concept of consistency restoration indeed can offer new chances for this task.

Although simplicity is one of our main arguments for Algorithm 2, an even simpler approach is introduced in [66]. The proposed algorithm marks in a first step the changed source graph G'_S from scratch but reuses the given correspondence and target elements. All correspondence and target elements that are not reused in this first step are deleted. In a second step, the remaining (unmarked) parts of G'_S are marked by applying forward rules (and creating new correspondence and target elements). Overall, incrementality is not addressed in the terms of computational effort as G'_S is entirely analyzed. Similar to [70], moreover, confluence is required for successful consistency restoration. Our observation is that the approach enjoys a very solid formal foundation but compromises practicality for this.

The opposite extreme compromising formal foundation for practicality can be observed in [60]. Given that the source graph is changed in a consistent triple, the proposed algorithm first iterates over the forward rule applications from former runs. If a forward rule application is invalidated but some of its marked elements still remain in the source graph, a repair is attempted by applying the same or another forward rule. In the case of a repair, the correspondence and target elements from the invalidated forward rule application are reused "as much as possible". Required conditions for repairs, governing how to reuse elements, and the consequences in the case of an impossible direct repair, however, remain open.

Table 4.1 summarizes our estimation for comparing different consistency restoration approaches based on TGGs. Our approach, similar to the precedence-driven approaches [5, 93, 116], strives to provide a formal foundation for incremental updates while supporting a large class of TGGs that go beyond confluence. A straightforward algorithm that eliminates the need for auxiliary precedence analyses makes our approach unique in this regard. On the other side, we make consistency restoration with TGGs "technology-dependent" where technology here refers to incremental pattern matching but not to a specific tool or algorithm thereof.

	formal foundation	incremental computation	straightforward algorithm	non-confluent TGGs	quality heuristics
Anjorin et al. [5]	+	+	-	+	-
Greenyer et al. [60]	-	-	+	+	+
Hermann et al. [66]	+	-	+	-	-
Hildebrandt et al. [70]	-	+	+	-	+
Lauder et al. [93]	+	+	-	+	-
Orejas et al. [116]	+	+	-	+	-
this work	+	+	+	+	-

Table 4.1: Comparing our consistency restoration approach to related work based on TGGs

Finally, heuristics to improve the quality of consistency restoration (in particular with respect to information preservation capabilities) seem to be the next gap to close. It requires again formal arguments for when and how to apply such heuristics for a fully-fledged consistency restoration approach. In this sense, it is probably not a coincidence that the +/- entries in the *formal foundation* column and the *quality heuristics* column of Table 4.1 are exactly the opposites in the current landscape.

QVT-R: Similar to consistency checking, QVT-R and its operationalization as introduced in [120] form an available standard for consistency restoration in MDE. Arguably due to the semantics issues (especially the uncertainty about whether and how non-confluent consistency restoration is supported as pointed at in [134]), however, a scarcity of acceptance and practical tool support can be observed here as well. Hence, bypassing the standard operationalization in [120] and resorting to alternative formalisms has been the main strategy for consistency restoration (as is the case with consistency checking as well).

In [103], consistency restoration with QVT-R (in fact a subset of QVT-R) is reduced to model finding via logical constraints. While model finding can yield different valid results, the approach utilizes a notion of *graph edit distance* (by counting the added and deleted elements) to restore consistency with minimal change regarding the previous version of the updated model. Capturing graph structures, their meta-model conformance as well as inter-model consistency as logical constraints, however, the main practical challenge is that it easily comes to an explosion of the formulated constraints and consequently of the effort in model finding.

Though not directly addressing QVT-R but introducing an own *QVT-R-like* syntax, a similar approach to consistency restoration via model finding, namely the *Janus Transformation Language* (JTL), is introduced in [32]. In JTL, the individual relations between a pair of source and target models are operationalized to *answer set programming*, a declarative paradigm to tackle search problems. A unique characteristic of the approach is that all possible target models that correspond to a changed source model are captured in the output, while we take decisions already at rule application time among different outcomes.

Bidirectional Programming: A further notion of BX is that of bidirectional programming which has its roots in *reversible computation models*, e.g., reversible Turing machines [14, 15, 106]. In the context of BX, functional programming languages such as *Janus* [153], *BiGUL* [87], and *UnQL+* [69] shape a representative group for reversible computation. In this setting, a consistency tool developer has to design consistency via functions in one direction (e.g., in the forward direction) which, indeed, might be intuitive from a software engineering point of view. The inverse functions are accordingly derived. Having ultimately a mechanism that is as powerful as a reversible Turing machine, these languages offer fine grained control over consistency restoration, while approaches such as TGGs and QVT-R can be qualified as *declarative* (at least in a relative sense). Reversible computation typically requires determinism in both directions which, however, does not hold in consistency restoration scenarios where different outcomes are possible. Moreover, addressing potential information loss in both directions also goes beyond the scope

of these techniques. That is, at least in one direction (in the programmed direction) the output model should always be entirely derivable from the input model.

Lenses: A *lens* [51] is a pair of functions, usually called get and put (which are meant for the forward and backward direction when transferred to the TGG terminology). In their original sense as introduced in [51], the get function takes a source as input and gives a (more abstract) view as the output, while the put function takes a source together with its "changed" view as input and gives an updated source as output. This is referred to as the *asymmetric* case as a view can entirely be derived from a source but the inverse does not hold. In other words, information loss exists only in one direction. The more general case of *symmetric* lenses has been introduced in [72] where information loss can exist in both directions. While the preliminary work on lenses focuses solely on models as the input and output of the get and put functions, *delta lenses* [37] yield an alternative understanding where not only models but also deltas become first class citizens when evaluating these functions.

Lenses, in fact, shape a mathematical framework for describing BX rather than a concrete BX language as an alternative to TGGs. *Well-behavedness laws* formulated over the get and put functions define the "sanity" of consistency restoration, and the question is then whether a concrete BX approach conforms to these laws. The most basic laws are referred to as the PutGet and the GetPut laws. Intuitively, the PutGet law demands that the result of get after put should be the same view that served as input for put (the GetPut law is analogously defined).

While the formal guarantees of consistency restoration as introduced in this thesis have a grammatical focus (where we discuss correctness and completeness over the language of a TGG), well-behavedness laws of lenses offer an alternative viewpoint in this sense. Relating TGGs to lenses, i.e., checking whether a TGG implementation forms a well-behaved lens, however, makes only sense after eliminating at least one discrepancy. In particular, consistency restoration with TGGs is inherently not a function but different outcomes are possible due to alternative rule applications (and we refer to an algorithm as correct as long as it produces one of the possible results within the language of a TGG). For a functional consideration of TGGs as is the case with lenses, therefore, a component must be assumed that deterministically decides which rules to apply when restoring consistency.

Overall, it is advantageous to have the heterogeneous viewpoints of formal properties, shaped by the grammatical characteristics of a TGG on the one hand and by the well-behavedness laws of lenses on the other hand. Experience, however, shows that the latter usually requires a problem-specific investigation for a concrete TGG operating in a concrete TGG implementation. For the general fulfillment of the well-behavedness laws, the aforementioned group of bidiectional programming approaches seems to be closer to the idea of lenses (especially their restrictions are well-fitting in this sense). Nevertheless, some restrictive cases of consistency restoration with TGGs are introduced in [10] and [68] as instantiations of lenses.

Unidirectional Transformations: Model transformations shape a wide research area in MDE. While BX can be considered as a special case, the majority of the approaches addresses the unidirectional case. We exemplarily refer to [26, 47, 79, 136] for surveys of model transformation approaches and consider in the following

a subset with support for incremental change propagation (which are more related to our consistency restoration purposes).

PROGRES [127] is one of the earliest representatives in the sense of an incremental execution mode. Its incremental capabilities capture *derived* attributes and edges which are computed from other information, e.g., from other attributes and/or edges. As introduced in [82], if model changes invalidate derived structures, the algorithm lazily re-evaluates them (delayed until the first read access). Indeed, this can be considered as consistency restoration among attribute values within graphs. Furthermore, an own incremental pattern matching concept is employed to observe *invariants* (graph constraints that must be met in a graph all the time). Possible repair actions are then incrementally activated as a reaction to model changes when invariants are violated.

The *VIATRA* framework [18] uses an incremental pattern matcher as the underlying technique for its incremental mode and has been one of the most inspirational works for our consistency restoration approach. Similar to our case, a model transformation is realized as reactions to the reports of an incremental pattern matcher. In fact, it has been suggested multiple times (e.g., in [121, 143]) that TGGs can profit from this technique. This clearly defined the focus of our contribution.

The *Atlas Transformation Language* (ATL) [78] is a further prominent example for a unidirectional model transformation approach where its incremental mode is introduced in [77]. The incremental algorithm reacts to atomic model changes directly (by using a notifier mechanism attached to the model elements) and re-evaluates the affected rule applications.

Event-driven graph grammars [61], finally, form another incremental approach resembling an operationalized TGG (i.e., operating in one direction). The focus of the approach lies in user interaction components in the context of visual languages. The utilized graph grammar rules exploit correspondences and describe how event objects in frontend can incrementally be transformed to backend data. The underlying transformation engine *Atom*^{3} [36] is started whenever an event is received, i.e., incrementality is ensured on the client side but not in the engine itself.

4.8 Summary and Future Work

In this section, we have

- exemplarily demonstrated the main subtasks of consistency restoration which amount to revoking rule applications from former runs and performing new ones,
- identified search space problems in applying available rule applications in consistency restoration and made use of application conditions to block undesired rule applications,
- stated what is to be observed to detect which rule applications are to be revoked or performed and made this observation task amenable to incremental pattern matching techniques,

- proposed a straightforward procedure for consistency restoration which solely reacts to the reports of its underlying incremental pattern matcher,
- discussed the termination, correctness, and completeness of this procedure,
- given an overview of related BX approaches to consistency restoration as well as unidirectional transformations with an incremental mode.

Algorithm 2 constitutes the main contribution of this section where its added value lies in its simplicity and viability from a practical point of view. We reduce consistency restoration to a reactive component that relies on "reports" concerning forward rule applications. This way, consistency restoration is made amenable to state-of-the-art incremental pattern matching techniques and, most importantly, does not introduce additional dependency analyses or heuristics to handle deltas. The reactive concept reflects a novel understanding of what is to be done for consistency restoration. This understanding did not only require years of experience with TGGs but also had to wait for powerful pattern matching techniques.

We mainly focus on runtime tasks of consistency restoration but at least two aspects of static construction and analysis techniques are left open (which rather concern the specification time). First, requiring marking-complete forward rules, we have discussed application conditions to ensure this. In our examples, we rely on existing construction techniques [64, 85] to generate (negative) application conditions. These construction techniques suffice to avoid wrong choices of forward rules with regard to incident edges of a vertex. More generalized approaches, however, are needed to handle wrong choices that arise from more complex situations (and not only from single edges). Second, and coupled with this point, how to check whether a set of forward rules is marking-complete is yet to be generalized. While confluence [65] and local-completeness [9] are two static analysis techniques towards this goal, both can be too restrictive (rejecting a set of forward rules although they are marking-complete). State space exploration over the forward rules, nevertheless, remains a further option to check marking-completeness. Overall, we provide our statements orthogonally to how forward rules are made marking-complete or checked to be marking-complete (and expect to remain compatible to future static construction and analysis techniques).

One of the open runtime issues (not only in our approach but in general) is to increase information preservation capabilities of consistency restoration. In our context, Algorithm 2, being a reactive strategy for consistency restoration, might be "overreacting" to a source delta. As illustrated in our last example in this section, elements can be deleted (by revoking forward rule applications) just in order to be created again. This is especially the case when multiple forward rules basically do the same with slight differences. Practical (but as yet to be formalized) measures to preserve information as much as possible include *reusing* deleted elements before creating new ones [60] and *repairing* a forward rule application instead of revoking it [70]. After investigating how these measures can be applied while retaining correctness arguments in the first place, our reactive conception again can be exploited for such extensions, e.g., by introducing new types of repair reactions to invalidated forward rule applications (besides revoking them).

5

TOOL SUPPORT, EXPERIMENTAL EVALUATION, AND PRACTICAL APPLICATION

This section discusses our practical contributions. We first present a meta-tool, namely eMoflon (`http://www.emoflon.org`), that supports consistency specification with TGGs and executing consistency checking as well as restoration as introduced in the previous two sections. Subsequently, we quantitatively evaluate consistency checking and restoration in their currently implemented form with particular regard to performance-related research questions. Finally, we report on a consistency project with Siemens AG as the industrial partner where we used eMoflon to check and restore consistency between computer-aided design (CAD) and mechatronic simulation models. The insights gained from this project serve to qualitatively discuss the applicability of TGGs from a tooling point of view and to draw the lessons learned from an industrial context.

5.1 The Meta-Tool eMoflon

The predecessor of eMoflon is MOFLON [3], developed prior to this work and mainly until 2011. MOFLON was an extension to the FUJABA tool suite [113] which supported Java code generation from *programmed graph transformation* specifications. In this setting, graph grammar rules are embedded into classical control flow structures including if-else branches and for-each loops. MOLFON used this infrastructure to generate Java code that searches matches for and applies the operational rules of a TGG (in particular forward and backward rules). Its capabilities with TGGs were limited to forward and backward transformations from scratch (i.e., it was rather a graph translator and not a consistency restorer in the incremental sense). While MOFLON was compatible to a Java-based implementation of the MOF standard to represent (meta-)models, the chance was being missed to comply with a more prominent and modern MDE tool landscape, namely the Eclipse Modeling Framework (EMF) [39]. In [6], therefore, the developers at that time report on re-engineering MOFLON to its EMF-based successor eMoflon.

Similar to MOFLON, eMoflon in its initial version compiles operational TGG rules to programmed graph transformation specifications which then are compiled to Java code. Most importantly, consistency checking and restoration are the two new features implemented based on this infrastructure. For incrementality of consistency restoration, however, a hand-crafted bookkeeping mechanism is used which, roughly spoken, "mocks" an incremental pattern matcher by detecting available and invalidated forward (and backward) rule applications. As this increased the complexity of the tool and the coupled maintenance efforts for future develop-

ments, yet another re-engineering process had to be undertaken to profit from the simplifications via incremental pattern matching as argued in this thesis. Therefore, a new version of eMoflon is implemented that uses a Rete-based incremental pattern matcher (which is developed in the same research group) instead of an own bookkeeping mechanism entangled in programmed graph transformation.

From now on, eMoflon relying on programmed graph transformation is referred to as eMoflon::TiE, while the new eMoflon relying on incremental pattern matching is referred to as eMoflon::ibex.[1] Currently, both tools are available for users, whereas eMoflon::ibex is at the centre of ongoing development efforts. The present author is a main contributor to both eMoflon::TiE and eMoflon::ibex which altogether represent the implementation work for this thesis. Figure 5.1 depicts the components and artifacts involved in using eMoflon::TiE (top) and eMoflon::ibex (bottom). Bold arrows indicate transformation of an artifact to another, while simple arrows indicate input, output, or, use-relationships among components, artifacts, and users.

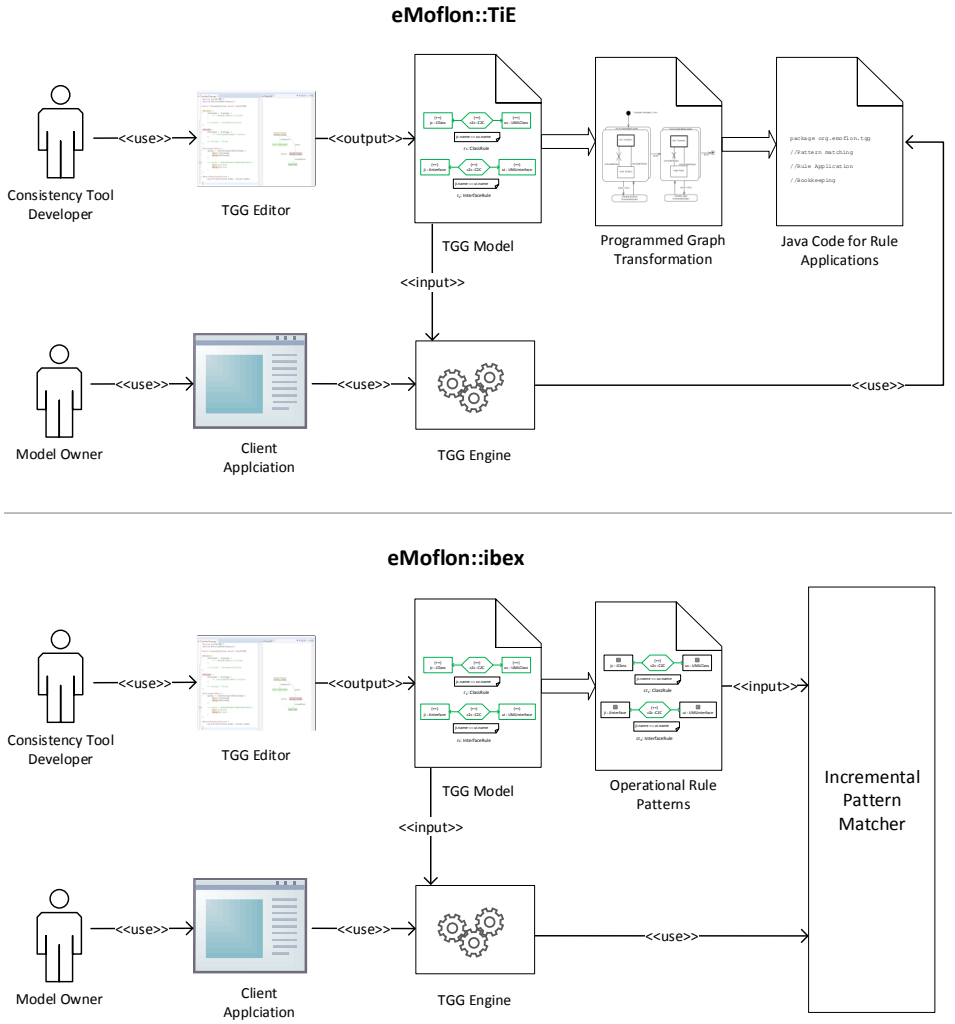


Figure 5.1: Involved components and artifacts when working with eMoflon::TiE (top) and eMoflon::ibex (bottom)

[1] TiE is an acronym for "Tool Integration Environment" and has been used in [84] for the first time, while ibex solely refers to goat species which also inspired the eMoflon logo.

Both tools provide a common *TGG editor*, a textual and Eclipse-based editor to specify a TGG (consisting of correspondence types and a set of rules). Of course, the question arises why to use textual syntax after exemplifying TGGs visually throughout the entire thesis. The arguments (and counter-arguments) are not different than those from the common debates in software engineering. We choose a textual editor in particular to reduce maintenance effort (of a component that does not lie at the heart of our research objectives) and to profit from mature editor frameworks of Eclipse with advanced features including syntax highlighting, syntax validation, and code completion.

Figure 5.2 is intended to give a brief glimpse of this textual editor and our textual syntax. A TGG rule is shown to the left, basically consisting of four textual fragments representing the source, correspondence, and target parts and attribute conditions of the rule. Similar to our visual rule diagrams, black and green lines of text represent individual context and created elements, respectively, while additional ++-markups again indicate created elements for a monochrome representation. For the visually inclined, nevertheless, a real-time and read-only (but admittedly modest) visualization of a TGG rule is provided (to the right in Figure 5.2) depicting source and target elements in different colors and correspondences as dashed lines.

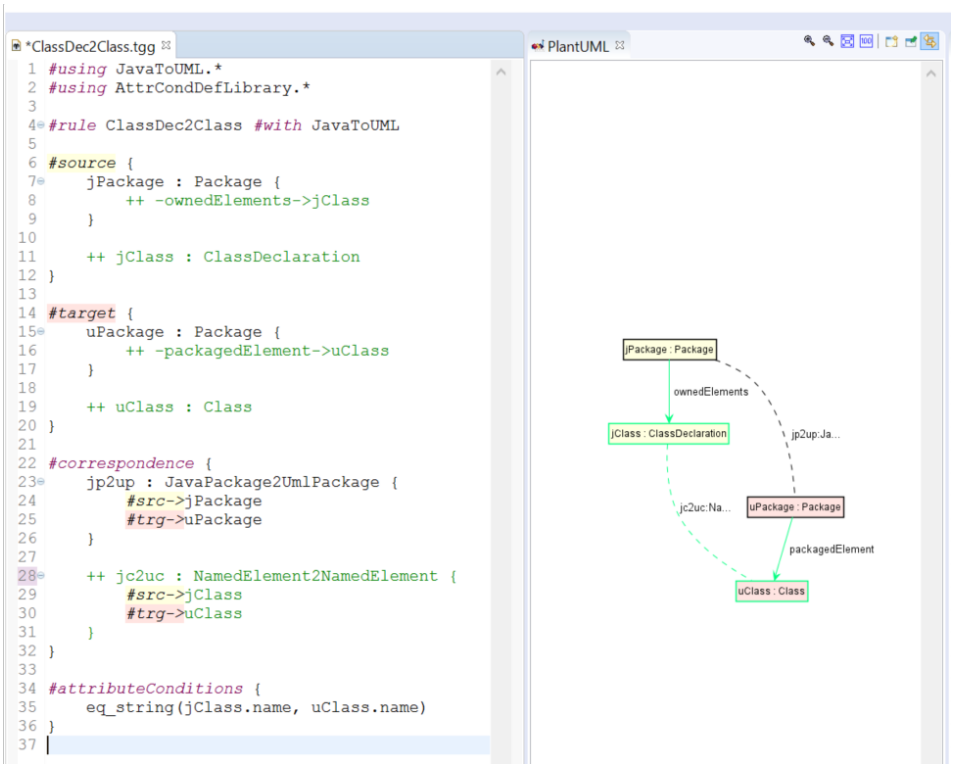


Figure 5.2: A glimpse of the textual TGG editor (left) and a respective visualization (right)

Coming back to the overall picture of components and artifacts shown in Figure 5.1, a *TGG model* is extracted in both tools by parsing the textual specification. This model indeed conforms to a *TGG meta-model* describing how TGG rules are structured and organized. That is, a consistency specification between two models

forms a model on its own (and the MDE vision suggesting to regard "everything as a model" becomes appreciable here to process the consistency specification).

The TGG model is the input for the operationalization step which makes the first main difference between eMoflon::TiE and eMoflon::ibex. In eMoflon::TiE, the TGG model is transformed to a programmed graph transformation which in turn is transformed to Java code representing the ultimate program for performing rule applications (i.e., the generated Java code is specific to the rules of a TGG). In eMoflon::ibex, on the contrary, patterns of operational rules are generated instead of a program. These patterns include the left and right-hand sides and possibly application conditions of forward, backward, and consistency rules. Overall, the entirety of these patterns represents what is to be observed and reported by an incremental pattern matcher when applying or revoking operational rules.

At runtime, both tools employ a *TGG engine* whose realization makes the second main difference between the tools. In eMoflon::TiE, the TGG engine interacts with the generated Java code to detect available rule applications. Moreover, a bookkeeping structure for the rule applications is maintained and used to detect invalidated rule applications when the involved models change. In eMoflon::ibex, the TGG engine solely reacts to its incremental pattern matcher and applies/revokes the respective operational rules. Finally, a *client application*, with which the model owners interact, uses this TGG engine as the underlying component for consistency management. Both eMoflon::TiE and eMoflon::ibex generate the plainest client application, a code fragment that serves as the entry point for executing consistency checking and restoration. Custom client applications, possibly with a graphical user interface as iconically implied in Figure 5.1, can be built upon this entry point depending on the needs of the model owners.

Being currently less mature and less optimized than eMoflon::TiE, scalability is not yet an advantage of eMoflon::ibex. Overall, simplifying consistency restoration and naturally addressing its subtasks via a reactive component remain the main improvement of eMolfon::ibex as we have carefully motivated so far. The underlying incremental pattern matcher has not yet reached an advanced state to handle the match-intensive task of consistency checking (at least not in the size of our experiments). For evaluating consistency checking in the following, therefore, eMoflon::TiE is our choice of tool which can stand the size of our experiments. For consistency restoration (which has been the main argument for eMoflon::ibex), furthermore, we use both tools and provide a comparison revealing the necessary improvements with regard to the scalability of eMoflon::ibex.

5.2 Experimental Evaluation of Consistency Checking

We first investigate the applicability of our consistency checking approach as implemented with eMoflon::TiE. In particular, we state the following two research questions related to its performance:

- **RQ-1:** How does consistency checking by combining TGGs and linear optimization scale? What are the main scalability barriers of this combination?

- **RQ-2:** How is the required runtime of individual subtasks in consistency checking (rule application and linear optimization) affected by the model sizes and the numbers of collected/chosen marking steps?

Furthermore, we compare our consistency checking approach with custom solutions of model differencing by stating the following research question below. Note that model differencing refers to the special case of consistency checking where consistency is defined as isomorphism between two models that are conform to the same meta-model (in fact, an extreme case of consistency checking due to its search space, while the regarded model differencing solutions have tailored algorithms as they do not consider the general case of consistency checking).

- **RQ-3:** How does our consistency checking approach compare to existing model differencing solutions with regard to accuracy and required runtime?

Experiment set-up: We approach **RQ-1** and **RQ-2** with an extended version of our running example, from now on referred to as JavaToUML-TGG, and a second TGG that defines consistency as isomorphism between two UML models, from now on referred to as UMLDiff-TGG. Consistency checking results with UMLDiff-TGG, moreover, are compared to the results of two model differencing tools, namely EMF compare [45] and SiDiff [137], to address **RQ-3**. That is, UMLDiff-TGG is developed for the purpose of model differencing between two UML models (and is thus used for a comparison with EMFCompare and SiDiff), while JavaToUML-TGG represents an average case of consistency checking relating to two different meta-models (and thus goes beyond the use cases of EMFCompare and SiDiff).

JavaToUML-TGG consists of 28 rules and relates packages, classifiers, attributes, methods, and parameters from Java and UML models. Method bodies in Java models are ignored as they do not have any counterpart in UML models. UMLDiff-TGG, furthermore, consists of 34 rules relating UML elements with the same type and attribute values. Moreover, the rules of UMLDiff-TGG are rather of atomic nature. Mainly inspired by the guidelines presented in [12], we distinguish between *island*, *bridge*, and *extension* rules (all depicted in Figure 5.3) in our design principle.

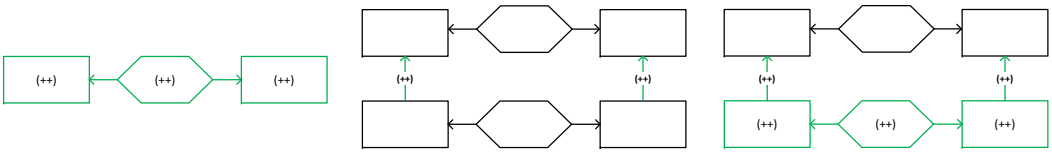


Figure 5.3: Design principles used for UMLDiff-TGG: Island rules (left), bridge rules (middle), and extension rules (right)

Island rules are used to relate pairs of vertices while bridge rules create edges between already related pairs of vertices. Using mainly these two types of rules in UMLDiff-TGG, vertices and edges in UML models are marked separately allowing for fine-grained consistency checking (and thus fine-grained model differencing in this particular case). Most importantly, two vertices can be related even if some of their edges do not have any counterparts (and these edges are detected as model differences). Requiring a minimal context (if any) in each rule, it most likely comes

to an explosion of search space making consistency checking even more challenging (and allowing us to explore the limits of our implementation). For some leaf elements, e.g., for parameters of methods, however, we make a compromise and use extension rules that create vertices together with their incident edges.

To conduct experiments with JavaToUML-TGG and UMLDiff-TGG, we extracted Java and UML models from real-world and synthetically generated software projects using the MoDisco tool [25]. In the upper part of Table 5.1, the four real-world software projects are listed together with the numbers of their contained packages, classifiers, attributes, methods, and parameters. The list of the software projects includes the core of our own TGG implementation (tgg.core), the Java discoverer of the Modisco tool (modisco.java), and two further Eclipse plugins (eclipse.graphiti and eclipse.compare). All these projects are representatives of different sizes (which is crucial to approach our performance-related research questions).

	# packages	# classifiers	# attributes	# methods	# parameters
tgg.core	114	372	197	614	813
modisco.java	75	561	263	1,423	1,556
eclipse.graphiti	74	611	683	2,438	4,079
eclipse.compare	99	1,115	1,913	4,050	6,205
synthetic-25	2	11	0	25	325
synthetic-50	2	11	0	50	1275
synthetic-75	2	11	0	75	2,850
synthetic-100	2	11	0	100	5,050

Table 5.1: Four real-world and four synthetic software projects used in our experiments

```
class MyClass
{
   void do(int p1)
   void do(int p1, int p2)
   …
   void do(int p1, int p2, …, int pn)
}
```

Besides the real software projects, we systematically generated synthetic software projects consisting of solely one class with lots of method overloadings (leading to search space problems for consistency checking as we exemplified in Section 3). These synthetic software projects (listed in the lower part of Table 5.1) contain 10 primitive Java types and one class with n overloaded methods. The overloaded methods have one to n parameters all adhering to the same naming convention as depicted in the figure above (p1, . . . , pn as parameter names). We, therefore, get a quadratically growing number of possibilities for relating methods and parameters (this applies to both JavaToUML-TGG and UMLDiff-TGG). From now on, we identify a synthetic software project with n overloaded methods as synthetic-n, e.g., synthetic-25 refers to the synthetic software project containing 25 overloaded methods with one to 25 parameters with the same naming convention.

Finally, we measured the required runtime of consistency checking on Intel i5-4200U @ 1.60GHz, Windows 8.1 (64 bit), and Java 8 with 10 GB available memory. In our plots, we show the median of five repetitions.

Experiment results and discussion: In order to get an impression of the involved search space before discussing the runtime measurements, Table 5.2 shows the respective numbers of all and chosen consistency rule applications (abbreviated as

CRA) with JavaToUML-TGG. Figure 5.4, subsequently, shows the required runtime for consistency checking in the form of a bar chart. We use stacked bars to particularly stress the distribution of the total runtime over consistency rule applications and ILP solving. In the case of real-world software projects, however, the share of ILP solving is not visible in the bars as it indeed amounts to negligible values. Therefore, the exact values of the required runtime for each individual component are explicitly given under the bars. Finally, we use a dashed line to separate the measurements with real-world and synthetic software projects.

	# all CRAs	# chosen CRAs
tgg.core	2,007	1,919
modisco.java	29,977	3,791
eclipse.graphiti	8,819	7,271
eclipse.compare	11,670	10,700
synthetic-25	6,162	362
synthetic-50	45,437	1,137
synthetic-75	149,087	2,937
synthetic-100	348,362	5,162

Table 5.2: Counted rule applications for consistency checking with JavaToUML-TGG

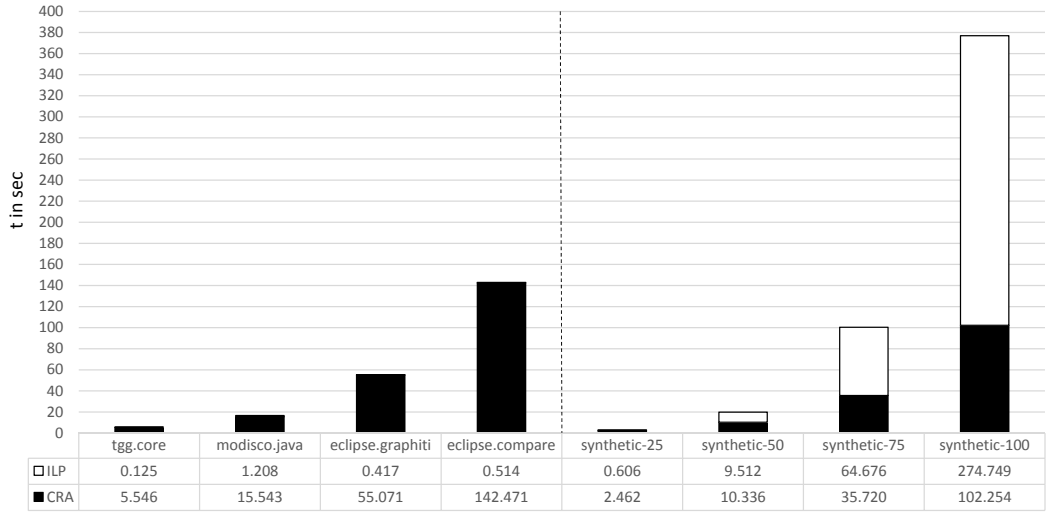


Figure 5.4: Runtime measurements for consistency checking with JavaToUML-TGG

Considering, for example, the measurements for tgg.core, consistency rule applications terminate in 5.5 seconds while ILP solving requires 0.1 of a second. As Table 5.2 shows, the search space for tgg.core involves approximately 2K consistency rule applications whereas 1.9K of them are chosen to mark the model pair entirely. The remaining consistency rule applications are wrong choices occurring due to method overloadings as we have demonstrated in our examples. The largest search space among the real-world software projects is observed in the case of modisco.java containing ca. 30K consistency rule applications whereas ca. 3.8K of them are chosen, i.e., ca. 87% of the collected consistency rule applications are wrong choices. A closer look at modisco.java revealed that the project makes excessive use of method overloading (due to the well-known visitor pattern leading to a lot of visit methods). In this case, collecting all consistency rule applications takes

15.5 seconds while ILP solving takes only 1.2 seconds. Applying consistency rules, however, reaches its limits in larger projects, e.g., taking more than 2 minutes in the case of eclipse.compare while ILP solving takes only 0.5 of a second.

Overall, we observe that, in the case of real-world software projects, ILP solving is quite performant while applying consistency rules constitutes the main bottleneck. In the case of synthetic projects, however, the situation is reversed. For synthetic-100, e.g., consistency rule applications require 102.2 seconds while ILP solving requires 274.7 seconds. The optimization step chooses ca. 5K consistency rule applications out of ca. 348K in this case, i.e., the share of misleading consistency rule applications in the search space amounts to ca. 98% representing an extreme case.

Runtime measurements for model differencing with UMLDiff-TGG are conducted twice. In the first round, we copied the model files and checked with eMoflon, EMF Compare, and SiDiff whether the models are identical with their copies. In the second round, we shuffled the orderings of some collections representing the references in the copies and performed the same check with the same tools. We only shuffled unordered references such as contained elements of a UML package and contained methods of a UML class, and respected ordered references such as the parameters of a UML method. Hence, our shuffled models are consistent to their originals (just like copied models).

The shuffled case indeed represents a realistic situation where two models are independently created and, therefore, do not have exactly the same order of elements in reference collections. In line with our expectation, however, changing the orderings in reference collections makes a significant difference for EMF Compare and SiDiff. Before discussing the runtime measurements, therefore, Table 5.3 first shows our test results showing which tools can detect the equivalence or the minimal difference between two UML models (we had added solely one vertex to one of the models when expecting the minimal difference). Note that the same TGG specification is used in eMoflon for both copied and shuffled models as neither TGGs in general nor our implementation relies on the orderings of reference collections.

	eMoflon	**EMF Compare**	**SiDiff**
copied models	pass	pass	pass
shuffled models	pass	fail	no default support

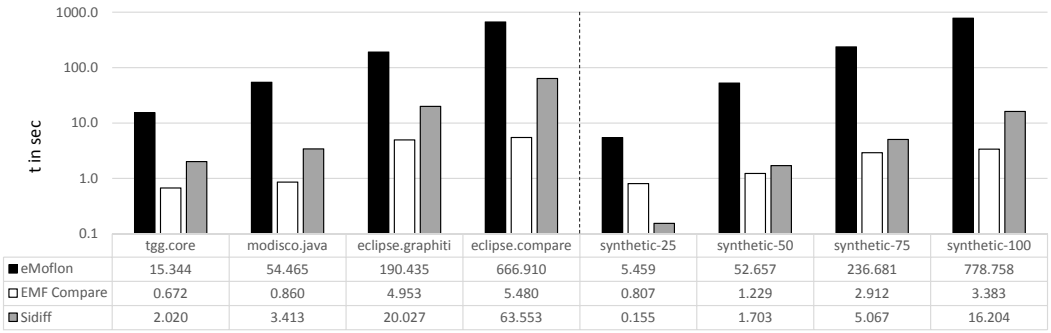
Table 5.3: Test results with UMLDiff-TGG for copied and shuffled models

While eMoflon (with UMLDiff-TGG) provides correct results for both copied and shuffled models, EMF Compare and SiDiff can provide correct results only in the copied cases. EMF Compare passes tests with shuffled models only if the models are very small consisting of a few elements which shows that the shuffled case is actually intended to be supported. It, however, fails with all of our models extracted from the aforementioned real-world and synthetic software projects. Instead of detecting equivalence or minimal differences, EMF Compare concludes that the majority of the model elements is deleted and new ones are added in the shuffled case.

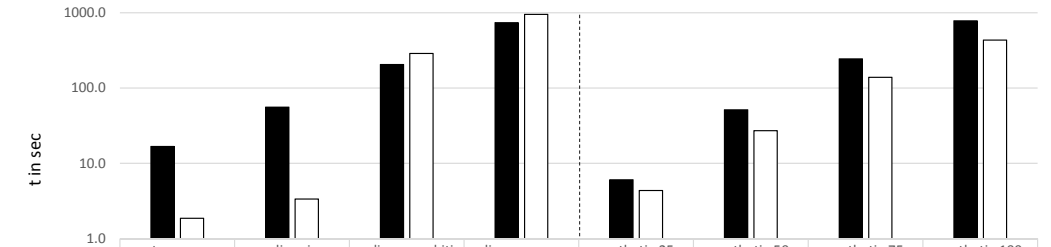
More crucially, the shuffled case (at least for UML models) goes beyond the default support of model differencing with SiDiff. While SiDiff has a special *model matcher* designated for copied models, none of the default model matchers is applicable to model differencing with shuffled UML models. The closest one to our

purposes, the so-called named-element matcher, requires unique name attributes of model elements which is not always the case with UML models (some vertices even do not have a name when representing, e.g., inheritance, dependency, or multiplicity). The named-element matcher of SiDiff consequently ignores a substantial part of our models, i.e., it does not attempt to match model elements with ambiguous names or to calculate their differences in contrast to EMF Compare. In order to avoid misleading conclusions, therefore, we take SiDiff out of comparison for the shuffled case and measure its required runtime only for the copied case.[2]

The required runtime for detecting the equivalence of two models is given in Figure 5.5. The upper diagram shows the measurement results with copied models (where both EMF Compare and SiDiff are involved in the comparison) and the lower one shows the measurement results with shuffled models (where only EMF Compare is involved). Note that we use a logarithmic time axis due to the large differences in measurement results especially in the case of copied models.



	tgg.core	modisco.java	eclipse.graphiti	eclipse.compare	synthetic-25	synthetic-50	synthetic-75	synthetic-100
eMoflon	15.344	54.465	190.435	666.910	5.459	52.657	236.681	778.758
EMF Compare	0.672	0.860	4.953	5.480	0.807	1.229	2.912	3.383
Sidiff	2.020	3.413	20.027	63.553	0.155	1.703	5.067	16.204



	tgg.core	modisco.java	eclipse.graphiti	eclipse.compare	synthetic-25	synthetic-50	synthetic-75	synthetic-100
eMoflon	16.766	55.704	205.019	736.960	6.046	51.313	244.211	781.147
EMF Compare	1.862	3.352	288.044	951.979	4.351	27.122	138.698	433.328

Figure 5.5: Runtime measurements for model differencing with copied models (at the top) and shuffled models (at the bottom)

For copied models, EMF Compare is not only reliable but also performant requiring under 6 seconds for all models. In contrast, EMF Compare does not only fail in the case of shuffled models but also faces scalability issues requiring, e.g., more than 15 minutes for the software project eclipse.compare. Runtime measurements with eMoflon do not have such a significant differentiation between copied and shuffled models. The required runtime amounts to 5-15 seconds for small software

[2] A further option would be to develop a custom matcher for (shuffled) UML models and SiDiff would only act as a framework calling this custom matching algorithm. This, however, is not only a cumbersome task but also does not fit into our experiment purposes as we want to locate consistency checking with TGGs into the landscape of "existing" approaches.

projects (tgg.core and synthetic-25) and more than 10 minutes for large software projects (eclipse.compare and synthetic-100). SiDiff with copied models, finally, requires under 5 seconds for small and mid-sized software projects and ca. 1 minute for the largest model (eclipse.compare).

Before finally answering the research questions, we furthermore take a closer look at the runtime measurements with eMoflon for UMLDiff-TGG. First, Table 5.4 shows the counted numbers of all and chosen consistency rule applications in our experiments. Comparing these values to those in Table 5.2, it can be observed that consistency checking with UMLDiff-TGG involves much larger search spaces than with JavaToUML-TGG. In the case of eclipse.compare, for example, 27K of 350K consistency rule applications are chosen via ILP solving, i.e., 92% of the consistency rule applications are misleading. Even more extremely, 5K of 696K consistency rule applications are chosen in the case of synthetic-100 amounting to 99% wrong choices of consistency rule applications. These larger search spaces of UMLDiff-TGG (as compared to JavaToUML-TGG) are due to the atomic nature of its rules as depicted in Figure 5.3, i.e., more vertices and edges from both sides seem to be relatable as the rules require a minimal context.

	# all CRAs	# chosen CRAs
tgg.core	7,724	4,612
modisco.java	68,712	8,005
eclipse.graphiti	56,982	15,497
eclipse.compare	350,167	27,159
synthetic-25	12,335	410
synthetic-50	90,885	1,410
synthetic-75	298,185	3,035
synthetic-100	696,735	5,285

Table 5.4: Counted rule applications for consistency checking with UMLDiff-TGG

In Figure 5.6, runtime measurements with UMLDiff-TGG (previously presented in Figure 5.5 by the eMoflon bars) are depicted in more detail using a stacked bar chart. We again distinguish between consistency rule applications and ILP solving as two components whose required runtime amounts to the overall runtime. Both components last longer than it was the case with JavaToUML-TGG. Applying consistency rules reaches its limits in the case of eclipse.compare taking ca. 10 minutes. ILP solving, furthermore, reaches its limits in the case of synthetic-100 taking, again, ca. 10 minutes.

In line with these measurement results and observations, we draw the following conclusions to answer our research questions:

- ***RQ-1:*** *How does consistency checking by combining TGGs and linear optimization scale? What are the main scalability barriers of this combination?*

  Consistency checking scales well in the case of realistic models and TGGs with moderate (i.e., non-explosive) search space, terminating in the order of seconds with small and mid-sized models and in the order of 1-2 minutes with large models (when considering the measurement results with JavaToUML-TGG and real-world models). In the challenging cases with large models

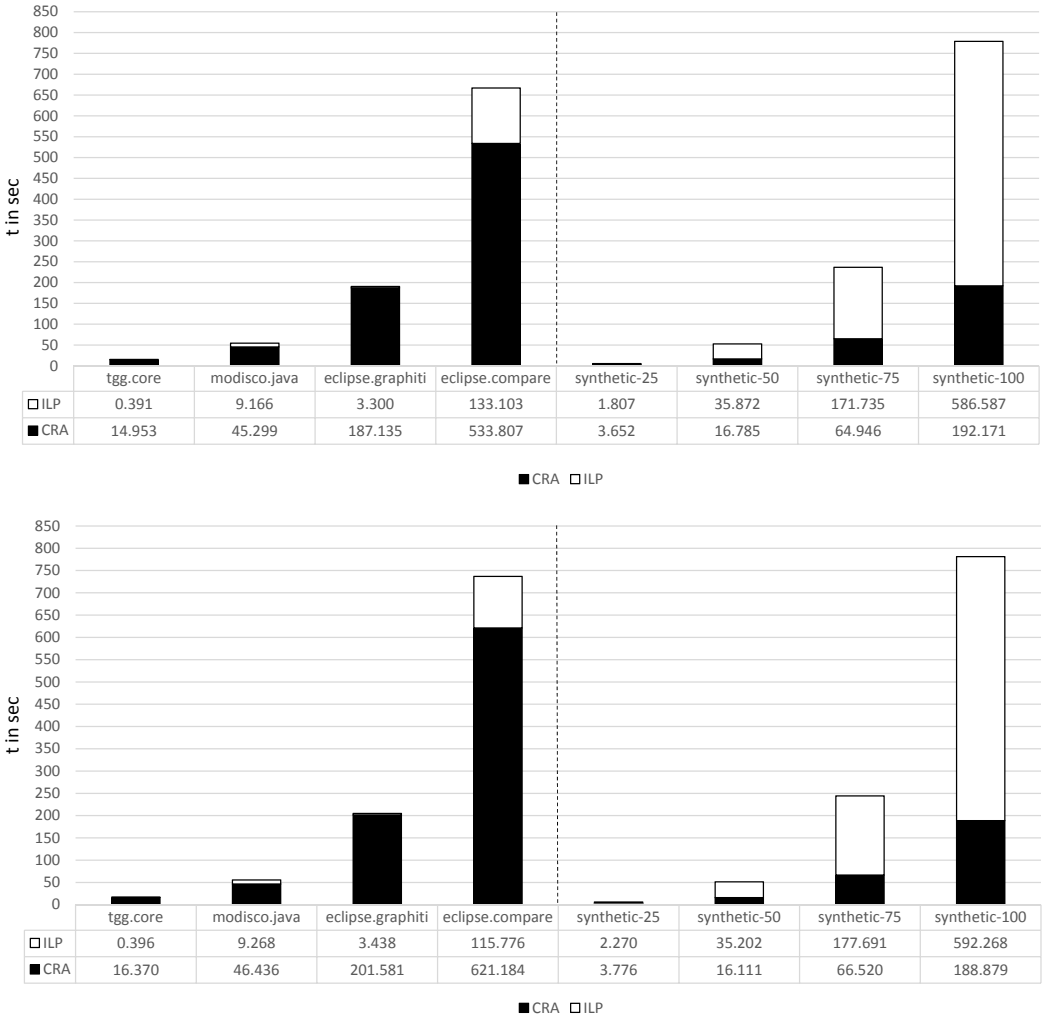


Figure 5.6: Detailed runtime measurements for eMoflon with UMLDiff-TGG (copied models at the top and shuffled models at the bottom)

and explosive search spaces including up to ca. 700K consistency rule applications, however, scalability issues arise and consistency checking requires 5-15 minutes. The main scalability barriers are (i) finding all possible consistency rule applications when the models are large and (ii) ILP solving when the search space is explosive involving more than 90% wrong choices.

- ***RQ-2:*** *How is the required runtime of individual subtasks in consistency checking (rule application and linear optimization) affected by the model sizes and the numbers of collected/chosen marking steps?*

  The required runtime for consistency rule applications highly depends on the model sizes rather than the number of applied consistency rules. Especially the measurements with JavaToUML-TGG make this abundantly clear: Performing 30K consistency rule applications for modisco.java takes only 15 seconds, while performing 11K consistency rule applications for eclipse.compare takes 142 seconds. Pattern matching, i.e., exploring the possible consistency rule applications between large models, is apparently the expensive task here, and

performing (lots of) consistency rule applications is not the main challenge. Runtime of ILP solving, on the contrary, highly depends on the number of collected consistency rule applications which, at the same time, defines the number of integer variables in the formulated optimization problem. Measurement results with UMLDiff-TGG furthermore show that the required runtime of ILP solving abruptly increases (e.g., under 3.5 seconds for up to 57K consistency rule applications but ca. 10 minutes for up to 700K consistency rule applications).

- ***RQ-3:*** *How does our consistency checking approach compare to existing model differencing solutions with regard to accuracy and required runtime?*

  Having experimented with two prominent model differencing tools, we observe a gap in sufficiently addressing accurate model differencing. Both EMF Compare and SiDiff (in their default support) highly rely on the assumption that one of the models is the copy of the other and thus preserves the orderings of (non-deleted) elements. Consistency checking with TGGs does not employ such assumptions and purely relies on the notion of graphs and linear optimization to provide accurate results. The price for this is that the required runtime exceeds ten minutes for large models, while EMF Compare as well as SiDiff require a few seconds for the cases they can pass. Nevertheless, eMoflon is not the only tool challenged by model differencing from a performance point of view. In the case of shuffled models where its internal heuristics and assumptions are not applicable, EMF Compare does not only fail but also becomes slower than eMoflon with increasing model sizes.

We finally discuss the threats to validity of our evaluation in the following two groups: *internal validity* concerning possibly unaccounted for factors that could distort the results, *external validity* concerning the generalizability of the results.

With regard to external validity, generalizability of our results requires further non-trivial case studies. We, nevertheless, argue that our experiments are the ultimately challenging cases of consistency checking with large models and explosive search spaces. Hence, the required runtime for an "average" application can expectedly stay within the spectrum of our measurements.

Internal validity is also a justified concern. Even if all runtime measurements are performed on the same hardware resources, available memory still plays an important role for the comparisons. Consistency checking exhaustively exploits the available memory in the case of larger models and its required runtime is thus influenced by the Java garbage collector that must regularly free some space. More or less available memory can, therefore, improve or worsen runtime results (this especially applies to consistency checking with large models).

5.3 Experimental Evaluation of Consistency Restoration

We experimentally evaluate in the following the scalability of consistency restoration with both eMoflon::TiE and eMoflon::ibex. Our experiments are of comparative nature and intend to reveal the current drawbacks of eMoflon::ibex being less

mature than eMoflon::TiE which is optimized throughout a period of years. For comparisons that span multiple tools (i.e., tools other than the different versions of eMoflon), we refer to [97] (specific to TGGs) and to [13] (different BX approaches). While such comparisons have only been possible with academic "toy" examples (in order to find a common ground for all involved tools), our experiments are conducted with industry-sized examples and thus focused on eMoflon.

In our experiments, we primarily distinguish between initial transformation of a model from scratch and delta propagation between two already existing models. Note that the first is a special case of consistency restoration (where the empty triple graph can be regarded as the formerly consistent state), while the latter represents the incremental case that has been the focus in our examples so far. In line with our evaluation goals, we state the next three research questions:

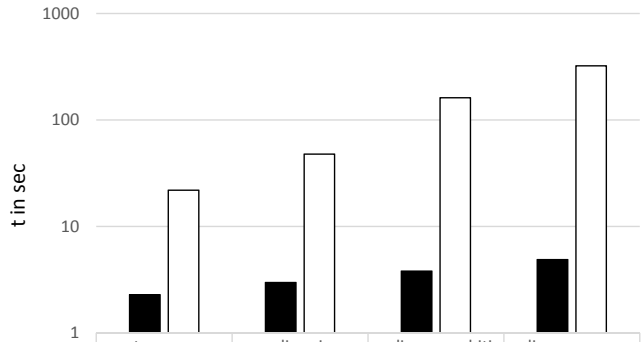
- **RQ-4:** How does the initial transformation of a model from scratch scale with increasing model size when performed with eMoflon::TiE and eMoflon::ibex?
- **RQ-5:** How does delta propagation from one model to an existing related model scale with increasing model size and delta size when performed with eMoflon::TiE and eMoflon::ibex?
- **RQ-6:** What are the current drawbacks and improvement potentials of using an incremental pattern matcher in eMoflon::ibex?

Finding answers to **RQ-6** is particularly crucial as eMoflon::ibex further on remains an actively developed tool within an ongoing research project on TGGs, while this work presents its very first version.

Experiment set-up: We use JavaToUML-TGG (which has been used for evaluating consistency checking in the previous subsection) to address our research questions. For **RQ-4**, we transform the Java models extracted from the real-world software projects listed in Table 5.1 to UML models. For **RQ-5**, furthermore, we take existing model triples as starting point and propagate the addition and then the random deletion of 1, 5, 10, 50, and 100 Java elements. While deltas with 1 element relate to a single method parameter, larger additive deltas (with 5, 10, 50, and 100 elements) comprise a mixture of packages, classes, methods, parameters, and attributes. For deletions, we remain on the level of leaf elements (parameters and attributes) to have a control over the number of deleted elements (i.e., to avoid further implicit deletions). To answer **RQ-6**, moreover, we performed additional runs of these experiments with a Java profiling tool detecting the hotspots in eMoflon::ibex. Finally, we used in these experiments the same platform as done for evaluating consistency checking (Intel i5-4200U @ 1.60GHz, Windows 8.1 64 bit, and Java 8 with 10 GB available memory).

Experiment results and discussion: In Figure 5.7, the runtime measurements for the initial transformation are depicted which show that eMoflon::ibex is especially challenged in this case (note the logarithmic time axis).

While eMoflon::TiE requires less than 5 seconds for all of our input models, the required runtime of eMoflon::ibex is multiplied by a factor up to 15 for small and



	tgg.core	modisco.java	eclipse.graphiti	eclipse.compare
■ eMoflon::TiE	2.287	2.975	3.808	4.877
□ eMoflon::ibex	21.866	47.752	161.805	322.438

Figure 5.7: Runtime measurements for the initial transformation

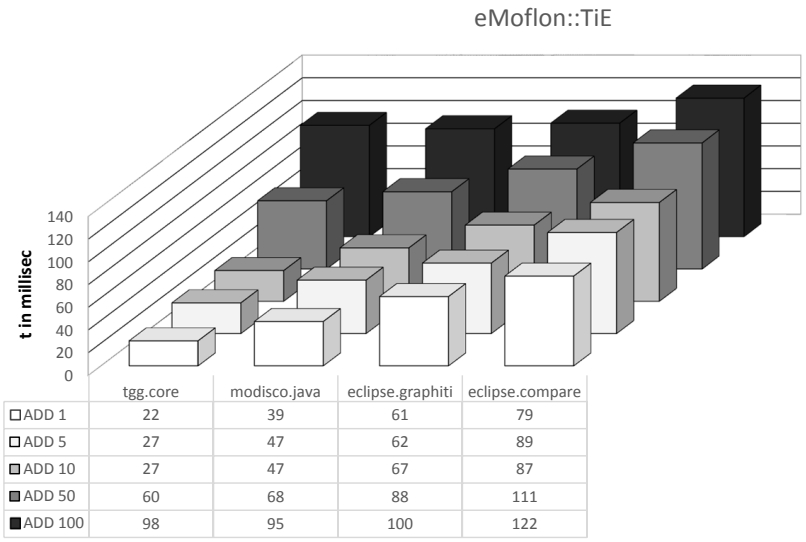
mid-sized models. For larger models, more critically, the initial transformation with eMoflon::ibex runs on exhaustively filled memory most of the time and requires more than 5 minutes. Profiling shows that runtime spent for filling the data tuples for (partial and complete) matches within the Rete network mainly leads to this difference. In fact, we can arguably expect that an incremental pattern matcher introduces an overhead for the initial transformation (for the sake of an incremental match maintenance in later runs). Reducing this overhead, however, seems to be critical for improving eMoflon::ibex.

In Figure 5.8 and 5.9, the runtime measurements for propagating additions and deletions, respectively, are depicted. We show the required runtime of both tools separately and use 3D bar charts as delta size now becomes a further dimension. For eMoflon::ibex, we use a logarithmic time axis due to the differences between the least and the greatest measurement values.

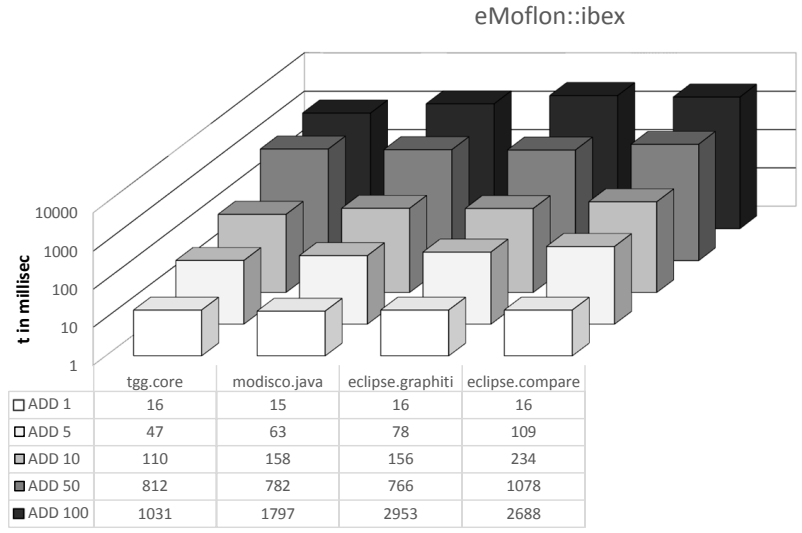
The required runtime of consistency restoration (when adding or deleting up to 100 elements) is in the order of milliseconds in most cases and at least does not exceed 4 seconds in the worst case (the worst case is deleting 100 elements from eclipse.compare and propagating this with eMoflon::ibex). Hence, both implementations back up the performance-related motivation of consistency restoration as compared to a transformation from scratch.

Again, eMoflon::TiE is our faster implementation and is especially performant when propagating deletions (requiring less than 100 milliseconds in almost all cases). This is mainly because eMoflon::TiE does not resort to pattern matching at all in the case of deletions but accesses its bookkeeping structures to calculate matches concerned by the deletions. While former versions of bookkeeping suffered from the scalability (in particular memory consumption) of Java collections, an alternative collection API, namely Trove [138], substantially contributed to this optimized state.

Currently, the only case where eMoflon::ibex performs better than eMoflon::TiE is the propagation of adding one element. In this case, eMoflon::ibex almost constantly requires 16 milliseconds for all models. Such a behavior, in fact, reflects an "ideal" consistency restorer whose propagation time solely depends on the delta size (and not on the model size). This, however, cannot be observed in other measurement

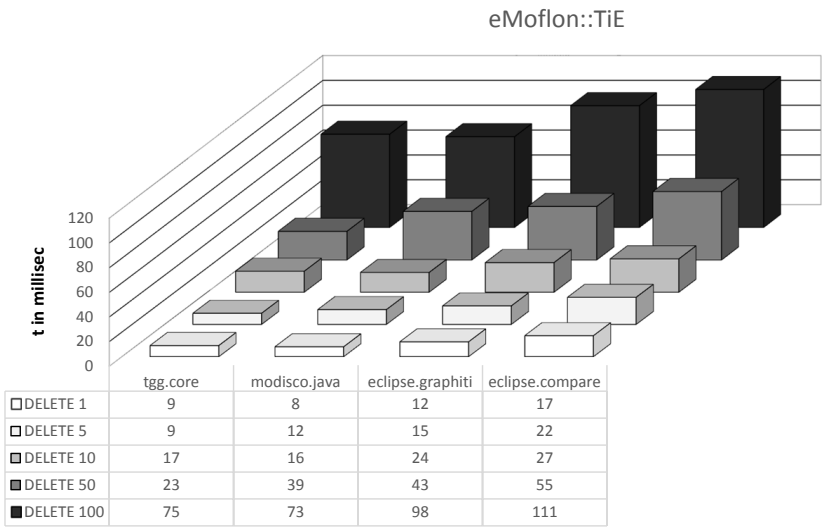


	tgg.core	modisco.java	eclipse.graphiti	eclipse.compare
ADD 1	22	39	61	79
ADD 5	27	47	62	89
ADD 10	27	47	67	87
ADD 50	60	68	88	111
ADD 100	98	95	100	122

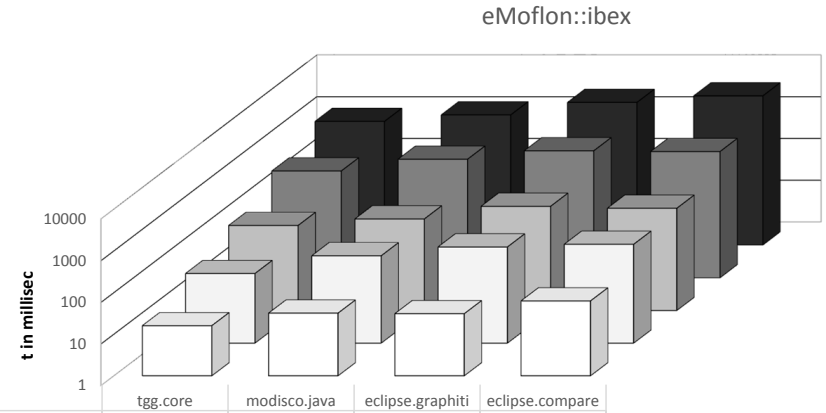


	tgg.core	modisco.java	eclipse.graphiti	eclipse.compare
ADD 1	16	15	16	16
ADD 5	47	63	78	109
ADD 10	110	158	156	234
ADD 50	812	782	766	1078
ADD 100	1031	1797	2953	2688

Figure 5.8: Runtime measurements with eMoflon::TiE (top) and eMoflon::ibex (down) for propagating additions of elements



	tgg.core	modisco.java	eclipse.graphiti	eclipse.compare
DELETE 1	9	8	12	17
DELETE 5	9	12	15	22
DELETE 10	17	16	24	27
DELETE 50	23	39	43	55
DELETE 100	75	73	98	111



	tgg.core	modisco.java	eclipse.graphiti	eclipse.compare
DELETE 1	16	32	31	63
DELETE 5	47	125	203	235
DELETE 10	109	156	313	281
DELETE 50	360	687	1094	1047
DELETE 100	906	1297	2578	3703

Figure 5.9: Runtime measurements with eMoflon::TiE (top) and eMoflon::ibex (down) for propagating deletions of elements

results (also not for eMoflon::TiE). Overall, eMoflon::ibex has a stronger tendency to be dependent on the delta size instead of the model size (having similar lengths of the bars on the same row from left to right in Figure 5.8 and 5.9). The absolute values of the required runtime, however, currently speak for a better scalability of eMoflon::TiE in general. The main drawback in using the incremental pattern matcher is again the scalability of updating data tuples of matches (which has to be done more frequently when the delta size grows).

In conclusion, we answer **RQ-4**, **RQ-5**, and **RQ-6** as follows:

- ***RQ-4:*** *How does the initial transformation of a model from scratch scale with increasing model size when performed with eMoflon::TiE and eMoflon::ibex?*

  Initial transformations scale well with increasing model size in eMoflon::TiE. Although our input models represent diverse sizes, the required runtime grows only with a factor of ca. 2 from the smallest model to the largest one. The same scalability, however, is currently not provided in eMoflon::ibex facing, in the first place, excessive memory usage in the case of larger models. The growth factor is consequently ca. 15 from the smallest to the largest model.

- ***RQ-5:*** *How does delta propagation from one model to an existing related model scale with increasing model size and delta size when performed with eMoflon::TiE and eMoflon::ibex?*

  Both tools propagate addition and deletion deltas with a size up to 100 elements within milliseconds in most cases. When propagating addition or deletion of 100 elements, the required runtime of eMoflon::ibex amounts to seconds but never exceeds 4 seconds overall. Both tools have a growth in the required runtime when fixing the model size and increasing the delta size. When fixing the delta size and increasing the model size, again a growth can be observed (but rather a moderate one as compared to increasing the delta size). The only exception is the propagation of adding one element where eMoflon::ibex manages to provide constant time over different models.

- ***RQ-6:*** *What are the current drawbacks and improvement potentials of using an incremental pattern matcher in eMoflon::ibex?*

  Memory consumption of incremental pattern matching is currently the most obvious drawback with regard to scalability and, in fact, becomes the decisive factor for the required runtime when the Java garbage collector must regularly intervene. This, however, is rather the symptom of a more fundamental problem, namely the effort put in maintaining partial and complete matches in the Rete network. New ways of reducing this effort must be found and define the focus of future developments. In particular, construction of data tuples for matches must be optimized (e.g., the number of maintained data tuples must be decreased). In this sense, optimization ideas such as tree-like representations of matches [74] are worthwhile to consider. Further matching algorithms such as TREAT [108] or LEAPS [109] may also provide additional performance improvements but an appropriate tool support tailored for the

MDE landscape is currently missing. Nonetheless, it is most likely that optimizations specific to TGGs can provide the most effective speedup (whereas the currently used incremental pattern matcher is general-purpose). Similar optimizations based on the knowledge of what specialities a triple graph exhibits as compared to any other model structure and what is expected to change during consistency restoration, in fact, brought eMoflon::TiE to its current performant state. Though not a fundamental solution but at least pragmatically mitigating the symptoms, furthermore, using more scalable alternatives to Java collections in pattern matching has also improvement potential (at least experience from the bookkeeping mechanism of eMoflon::TiE points at this potential).

External validity of our results, again, requires further case studies with different types of models and TGGs. Nevertheless, our experiments represent a large-scale consistency scenario and thus a suitable starting point to evaluate the scalability of eMoflon::TiE and eMoflon::ibex. Internal validity, furthermore, requires a special care especially in the case of **RQ-6** demanding a fine-grained investigation of eMoflon::ibex (besides its overall scalability). We, therefore, carefully utilized a Java profiling tool to conclude what are the main bottlenecks in consistency restoration with eMoflon::ibex in regard to required runtime and memory consumption.

5.4 The GraTraM Project

Funded by the Federal Ministry of Education and Research (Germany), the formal results of this thesis have been implemented and applied to an industrial case study provided by Siemens AG. The micro-project[3] is called GraTraM, an acronym for "A **Gra**ph grammar-based approach to **Tra**ceability of related **M**odels".

The focus of the GraTraM project is to explore how the consistency management techniques as discussed in this thesis can be applied to support interdisciplinary engineering landscapes. The case study involves, on the one hand, CAD models developed and maintained by mechanical engineers. A CAD model is a technical drawing and conveys basically physical information on the dimensions and materials of a system, e.g., an excavator or a water supply line as exemplified in the project. On the other hand, the same system is represented by multi-domain simulation models developed and maintained by mechanical, electrical, and automation engineers. A simulation model does not focus on the physical appearance of the system but rather on interaction and behavior of components with regard to, among others, mechanical dynamics, energy flows, electrical signal processing, and controllers.

The most important aspect here, distinguishing this project from previous ones with TGGs such as [5, 21, 67, 126], is that both models initially exist, i.e., consistency management becomes relevant after concurrently developing the models. There are at least two reasons for this: First, the amount of information shared between a pair of CAD and simulation models is rather small as compared to the entire size

[3] Industrial projects within the scope of the parent organization Software Campus (`http:\www.softwarecampus.de`) are commonly referred to as micro-projects involving grants up to 100,000 Euros and directly led by a PhD student (the present author in the case of GraTraM).

of the models. While the models must agree on geometrical structures, dimensions, and material properties of individual components, it always requires human intelligence and domain-specific knowledge to lift the models onto an advanced state. Hence, deriving one model from another entirely for an initial start as assumed so far in BX is not feasible. Second, it is a common practice that engineers start with already existing solutions on the CAD and/or simulation side and adjust their models to match customer specific needs (instead of modeling an excavator or a water supply line each time from scratch).

The workflow investigated in the project is depicted in Figure 5.10 using the Business Process Model and Notation [27]. We vertically distinguish between two environments, namely those of the CAD engineer and the simulation engineer both having the model owner role. The individual runs of consistency checking or restoration in the workflow are labeled with ❶, ❷, and ❸ in Figure 5.10.

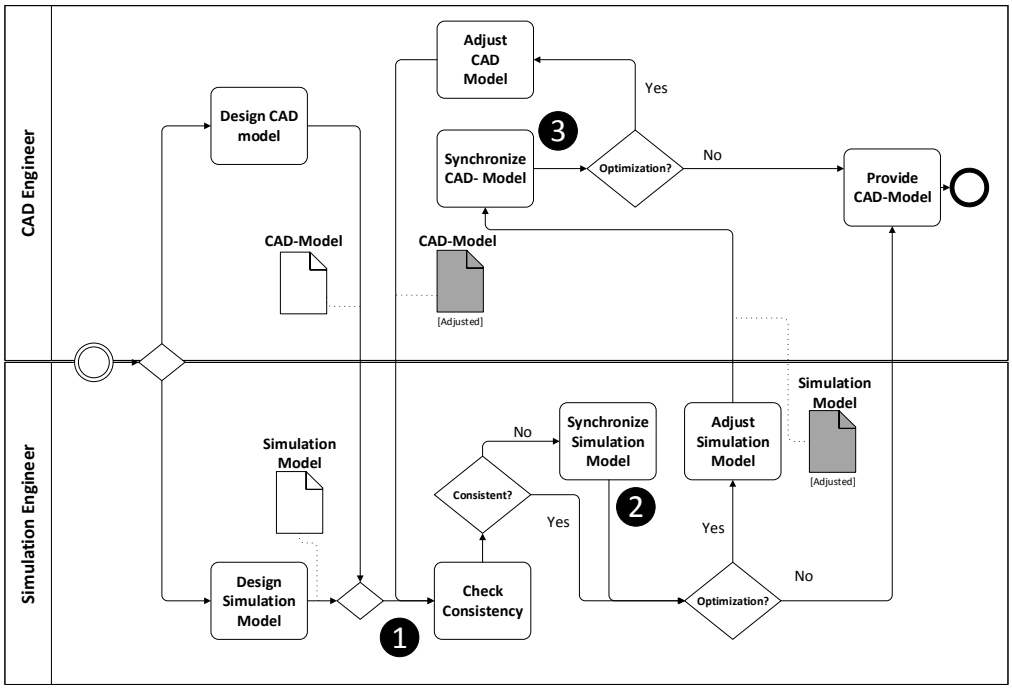


Figure 5.10: The workflow investigated in the GraTaM project

Starting with a consistency checking run (❶), the first goal is to make the simulation model consistent to the CAD model, i.e., design decisions of the CAD engineer has a higher priority at this stage. If the models are inconsistent, which is very likely if some legacy solutions are intended to be used, therefore, a consistency restoration run (❷) is performed to synchronize the simulation model.

Having a consistent state (no matter whether directly after ❶ or first after ❷), the simulation engineer performs simulations and checks the properties of the system (e.g., answering questions such as "How much stress can the excavator bucket handle?" or "What amount of flow can be reached with the water supply line?"). If the simulation engineer "approves" the system regarding the simulation results, no further consistency restoration is needed until the models are again changed for

some reasons. If not, however, the simulation engineer adjusts parameters (possibly including geometric properties) until the simulation results are satisfactory. This again gives rise to the need for a further consistency restoration run (❸) where this time the decisions of the simulation engineer have a higher priority, i.e., the CAD model must be synchronized. The CAD engineer either releases the synchronized CAD model or has some new modifications in the meantime which again require a new iteration starting with consistency checking (❶).

In the first iteration of the project, the consistency between a pair of CAD and simulation models representing an excavator is investigated. The CAD model is specified with the CAD tool NX [114], while the Amesim platform [4] represents the simulation side. The shared information between the two models is given by the actuators of the excavator, in particular a chain of three hydraulic components that move back and forth. In Figure 5.11, excerpts from the CAD model (left) and the simulation model (right) representing the excavator are depicted. The hydraulic cylinders that are at the centre of the consistency tasks are marked via black rectangles in both models.

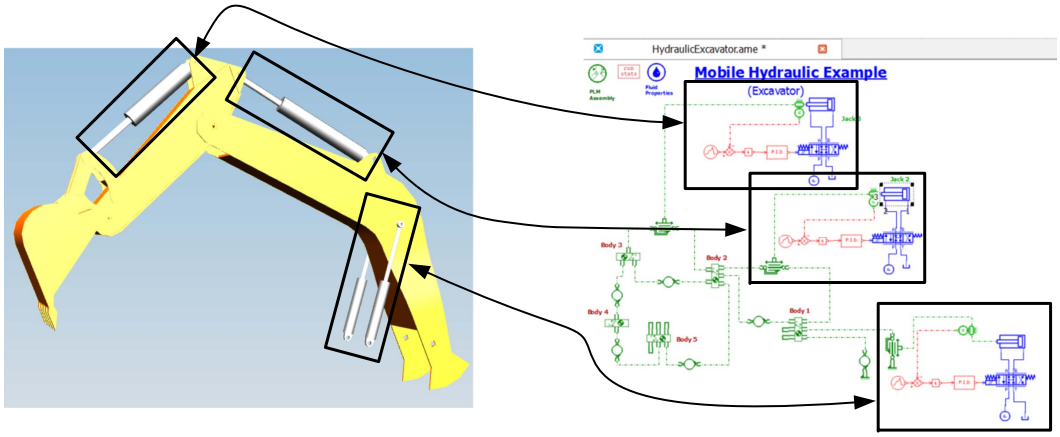


Figure 5.11: The CAD (left) and simulation (right) model of the excavator

In the CAD model, the hydraulic cylinders are technically drawn as a combination of a cylinder body and a piston rod. In the simulation model, however, a group of several components from the Amesim library is used representing, among others, a mechanical piston, a hydraulic piston, a displacement sensor (observing the piston position), and a PID controller (for piston movements). Especially the electrical components (such as sensors and controllers) are not represented in the CAD model but are integral parts of the consistency notion. Hence, examining their existence via consistency checking as well as retaining them incrementally are crucial for the simulation purposes. Furthermore, two parallel hydraulic cylinders in the CAD model (that are meant to move always in the same direction with the same displacement) can be reduced to one group of elements in the simulation model. This is the case for the lowermost rectangles in Figure 5.11. Finally, besides this structural understanding of consistency, geometric parameter values such as length, radius, and initial displacements of the cylinder bodies as well as the pistons constitute the remaining objects of consistency tasks. Overall, the expressiveness of TGGs (as implemented in eMoflon) suffices to address these requirements.

Figure 5.12 gives an overview of the involved components and artifacts in the excavator case study. Considering the eMoflon part, the TGG specification consists of 15 rules capturing the aforementioned consistency notion of cylinder bodies, piston rods, and their geometric parameters. The derived consistency checking and restoration tool operates on EMF models. Hence, the actual CAD and simulation models must be converted to an EMF representation and also vice versa to reflect consistency restoration actions on the EMF representation again in the actual models. Components addressing this task are called *tool adapter* and play generally a crucial role to exploit an MDE tool for the ultimate engineering documents.

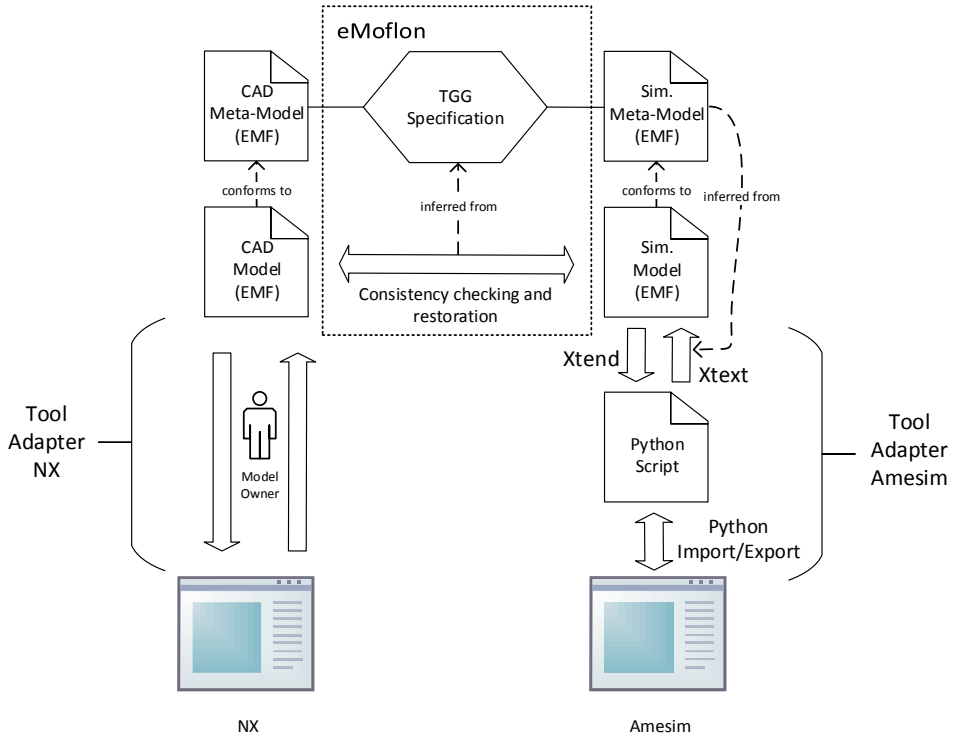


Figure 5.12: Overview of the tooling for the excavator case study

To implement a bidirectionally operating tool adapter, it is important to have access to the data represented in the respective tool. This can either be achieved by directly parsing and generating the user files of the tool, referred to as *offline tool adapter* (as the tool itself does not have to run in the background), or by a plug-in interacting with the programming interface of the tool, referred to as *online tool adapter*. For the Amesim side, an offline tool adapter enables a bidirectional conversion between Amesim models and their EMF representation. The Python export/import facility of Amesim is used as a subcomponent for this where the excavator is represented as a sequence of Python commands (which are executed by Amesim to create the excavator model step by step). The Python commands are parsed via a grammar specified with Xtext [151] which induces, at the same time, an EMF-based meta-model for the simulation side (the relationship between the Xtext component and the simulation meta-model is depicted explicitly in Figure 5.12). Accordingly, the parser produces a model conforming to this meta-model. For the

reverse direction, i.e., for producing Python commands from the EMF representation, Xtend [150] is used as a template-based code generation technology.

For the NX side, however, the same level of automation is not feasible for a tool adapter. Different CAD file formats have been considered for an offline tool adapter but all of them turned out to be unsuitable for parsing. CAD files are either of binary nature or represent a technical drawing at the lowest level consisting of atomic drawing instructions. Therefore, for example, recognizing a cylinder and its geometric properties is not possible with a justifiable effort. A similar level of difficulty also applies to an online tool adapter in the case of NX which would go beyond the scope and research goals of the project. Therefore, the tool adapter for NX is not automated but the relevant portions of the CAD model are manually reproduced in the EMF representation.

Considering overall the individual runs ❶, ❷, and ❸ with our consistency checking and restoration tool as depicted in Figure 5.10, consistency checking (❶) is coupled with the manual effort of creating the EMF representation of the CAD model. Restoring consistency by synchronizing the simulation model (❷) is then fully automated. Operating incrementally as introduced in the previous section, this step only changes the inconsistent parts of the simulation model and keeps all other irrelevant information (which is necessary though for executing a simulation). From a demonstration point of view, therefore, the combination of ❶ and ❷ enjoys the most positive reception from the industrial partner in this first iteration. Restoring consistency on the CAD side (❸), finally, is again coupled with the manual effort of feeding the changes on the EMF representation back into NX.

In the second iteration of the project, a further case study based on a water supply line has been investigated (instead of going deeper into the excavator case study). This time, Parasolid [117] represents the CAD side, while Amesim further on remains the simulation side. Mitigating the tool adapter challenges experienced in the first iteration, the industrial partner has a custom solution for Parasolid that extracts an XML representation of a CAD model. Hence, the CAD model is not provided as a technical drawing but as an XML file in the second iteration. The industrial partner indeed utilizes such XML files to derive sketches of simulation models, while the GraTraM project additionally introduces consistency checking and incremental consistency restoration aspects into the tooling.

In Figure 5.13, model excerpts of the water supply line and their relations are depicted. The CAD model has an XML-based representation describing a water supply line consisting of individual pipe components as well as their connectors such as bends and T-junctions. In the simulation model, special components from the Amesim library again represent the system for simulation purposes. Besides the existence of pipes, bends, and T-junctions, their relative locations within the network and geometric properties constitute the consistency notion that must be checked and, if necessary, restored.

Figure 5.14 depicts the components and artifacts involved in addressing consistency for the water supply line models, differing from those of the excavator case study mainly on the CAD side. With the help of the provided XML export, the initial consistency checking and synchronizing the simulation model are fully automated (❶ and ❷ in Figure 5.10). The conversion between the XML and EMF

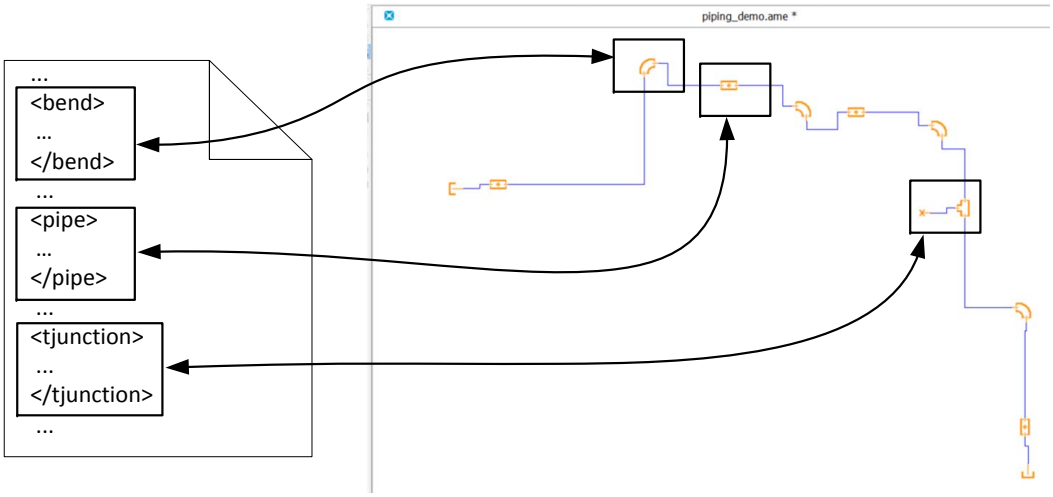


Figure 5.13: The XML-based CAD (left) and simulation (right) model of the water supply line

representations of the CAD model is implemented with Java using standard XML parsing and generation libraries. Feeding the changes on the XML model back into Parasolid after consistency restoration on the CAD side (❸) remains the only aspect that requires manual effort.

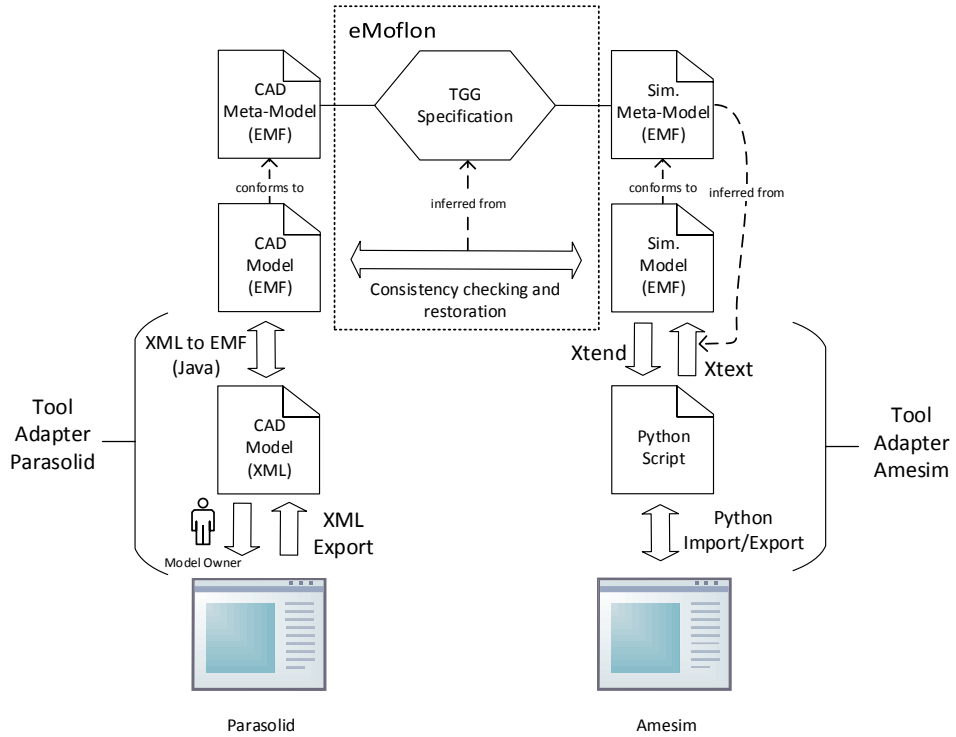


Figure 5.14: Overview of the tooling for the water supply line case study

Overall, we draw the following conclusions from the GraTraM project. Note that the first two points have been decisive for our research and development activities

and had a considerable impact on the contents of this thesis. The third point, furthermore, rather relates to the tooling aspect in an industrial context.

- Even though models in an interdisciplinary engineering landscape are said to be "related", the amount of shared information might refer to a small (nevertheless crucial) subpart of the models. Therefore, all involved domains exhibit their own unique justification for relying on human intelligence, and the ambitious objective of deriving one model from the other automatically does not necessarily address consistency challenges adequately. Instead, consistency management approaches should allow concurrent developments by human intelligence and should accept the advanced state after concurrent developments as the starting point. Our consistency checking approach is the foundational step towards this conception.

- While custom solutions exist for information exchange between two different models, the incrementality aspect is the most difficult one to address without a systematic practice. Coupled with the previous point, however, models generally exhibit private information which are lost without incrementality. Considering the feedback from the industrial partner, the added value of our prototypes as compared to existing solutions especially lies in their incrementality. This is not due to efficiency reasons (at least not in the investigated case studies) but rather due to information preservation. Hence, a fully-fledged BX approach to consistency restoration (be it with TGGs or other formalisms) must attach importance to preserving as much information as possible when restoring consistency.

- Engineering tools must consequently make use of MDE in order to profit from its vision for consistency management. That is, suitable abstractions (models) that conform to a well-defined specification (meta-model) must be provided so that any MDE tool can operate on these. In the project, especially the CAD side was challenging in this regard representing relevant engineering data mostly in low-level and atomic drawing instructions. This makes the development of appropriate tool adapters very difficult if not impossible. While a custom XML export serves as a helpful subcomponent in the second iteration of the project, general solutions (in particular file formats) are necessary to make CAD data more accessible. This aspect, of course, goes beyond the scope of the project but indeed shapes a relevant research challenge on its own (for some preliminary efforts, we exemplarily refer to [16, 19, 54]). For the fully automated applicability of consistency management, addressing these issues is crucial from a tooling point of view.

5.5 Summary and Future Work

In this section, we have

- presented our meta-tool eMoflon, currently having the two different versions eMoflon::TiE and eMoflon::ibex,

- experimentally evaluated our consistency checking approach by (i) investigating the distribution of overall runtime over rule applications and linear optimization and (ii) comparing consistency checking to model differencing solutions via dedicated examples,
- experimentally evaluated our consistency restoration approach via a comparison between eMoflon::TiE and eMoflon::ibex revealing the future directions in terms of scalability for eMoflon::ibex,
- reported on an industrial project where we applied our consistency checking and restoration approach to maintain consistency between CAD and mechatronic simulations models.

Tasks for future work regarding consistency checking and restoration have been already discussed in the respective sections and also define tasks for future work from an implementation point of view. First and foremost, however, performance optimizations with regard to the incremental pattern matching in eMoflon::ibex define the first crucial step, whereas this thesis presents a first version of the tool and reveals its current scalability limits.

Furthermore, model differencing via consistency checking with TGGs indeed seems to be a promising approach (when we consider the gaps in sufficiently addressing model differencing). Hence, a custom implementation of TGGs tailored for the model differencing case is worthwhile to consider. Accordingly, more comprehensive comparisons with existing solutions to model differencing are needed to determine whether TGGs can be established as a model differencing approach.

6

CONCLUSION

This thesis addressed consistency management tasks in Model-Driven Engineering (MDE), in particular consistency checking and restoration between related models based on Triple Graph Grammars (TGGs). A TGG is a grammar-based consistency description (stating how consistent pairs of models are constructed) from which consistency checkers and restorers are automatically derived.

In the introduction, we distinguished between three different groups of stakeholders including model owners, consistency tool developers, and meta-tool developers (whereas the term "meta-tool" refers to a tool used for developing consistency tools). Furthermore, we identified their functional and non-functional requirements (in Table 1.1) which defined the contributions of this thesis. These requirements are repeated and used to structure the following discussion for concluding this thesis.

Regarding the functional requirements, the following results have been achieved:

- *Check consistency:* In Section 3, we established consistency checking as a novel use case of TGGs, while practical solutions thereof are scarce in the general landscape of MDE. In our setting, given two related models, consistency checking does not only refer to finding a yes or no answer for consistency but also detects the maximal consistent portions of the models. Based on our results, model owners can concurrently work and then check whether or to what extent their models are consistent to each other.

- *Restore consistency:* In Section 4, we introduced a mechanism for consistency restoration via delta propagation from one model to the other. A delta can be inferred either from consistency checking (i.e., model parts beyond the detected consistent portions) or from modifications made by model owners (or from a combination of both). Given a delta, our approach brings two models again to a consistent state and also maintains a consistency history that can serve as input for further runs of consistency restoration when the models are again changed.

- *Develop meta-tool:* The level of formalization in both Section 3 and 4 focuses on how to realize consistency checking and restoration, and, therefore, is addressed to meta-tool developers in the first place. For consistency checking, we identify search space problems that form an obstacle to an implementation and clear this obstacle by exploiting linear optimization techniques. For consistency restoration, moreover, our contribution is to come up with a simplified and straightforward procedure that outsources necessary tasks to an incremental pattern matcher. Available tool support for our used techniques also enables the development of a meta-tool for TGGs.

- *Develop consistency tool:* We presented our own meta-tool, namely eMoflon, in Section 5 which is available as an open source project at `http://www.emoflon.org`. The tool allows for specifying TGGs and derives consistency checking and restoration tools based on our formal results.

Furthermore, the non-functional requirements in Table 1.1 are addressed by our contributions as follows:

- *Scalability:* Dealing with challenging problems (in fact even an NP-complete problem in the case of consistency checking), we have a pragmatic understanding of scalability and refer to it as the capability of handling real-world consistency scenarios in reasonable runtime. Our experiments in Section 5 showed that consistency checking can terminate already in the order of seconds for realistic examples but can also require several minutes in the most challenging corner cases. Delta propagation in consistency restoration, furthermore, terminates in the order of milliseconds in most cases. Having currently two available versions of our tool as discussed in Section 5, we also identified necessary optimization steps for the re-engineered version (such that the matureness of the older version can be reached again).

- *Formal properties:* For both consistency checking and restoration, we resorted to the well-known construction techniques of category theory and in the case of consistency checking, furthermore, to integer linear programming. For our consistency checking and restoration procedures in Section 3 and 4, respectively, we provided formal proofs for termination, correctness (i.e., the returned result is conform to the consistency notion yielded by the TGG), and completeness (i.e., a result is always returned).

- *Usability:* Our meta-tool is based on the Eclipse Modeling Framework (EMF) and profits from the mature infrastructure of Eclipse. For specifying TGGs, we provide a textual editor with code completion, syntax highlighting, and validation. For performing consistency checking and restoration, moreover, our meta-tool generates the respective tools automatically (in line with our operationalization results) and provides entry points for the execution.

- *Validation:* While academic "toy" examples are generally used to demonstrate consistency tools, we went one step further in our experiments and focused on a real-world scenario related to consistency between Java and UML models. Furthermore, we used our consistency checking approach for model differencing purposes (where the two models represent the different versions of the same artifact). Most importantly, we further validated our approach with an industrial project from the domain of mechanical engineering and addressed consistency between computer-aided design (CAD) and mechatronic simulation models.

At the end of Section 3, 4, and 5, we discussed the important lines of research for future work which can be summarized as follows:

- For consistency checking, lifting our results to the incremental case (where markings and correspondences from former runs are reused) seems to be the next logical step. While consistency checking is formulated as choosing (via linear optimization) between alternative rule applications, the incremental case can require maintaining unchosen ones (as they might become choosable in retrospect after changing the involved models). Furthermore, incorporating user-specific constraints and objectives into our linear optimization problem can broaden the possible use cases of our formalisms (tackling different mapping problems that have their own requirements besides consistency).

- For consistency restoration, static analysis techniques must be explored to guarantee successful consistency restoration. We state sufficient properties for this but how to check them in a general manner was not within the scope of this thesis. Existing static analysis techniques [9, 65] provide a help but can be too restrictive, while state space exploration (e.g., as practically supported in [55]) seems to be a further promising choice. Moreover, increasing information preservation capabilities of consistency restoration is a necessary improvement. In particular, repairing invalidated rule applications must be explored before revoking them in order to reduce the actions taken by consistency restoration. Of course, remaining compatible to the correctness and completeness results of consistency restoration is of the utmost importance for these extensions. Finally, our understanding of consistency restoration is concerned with propagating a delta in one model to the other. Given that both models have a delta due to concurrent modifications, conflict detection and (semi-automatic) resolution between these deltas must be established (before consistency restoration) for a more generalized support for consistency management.

- From a tooling point of view, performance improvements (and possibly optimizations specific to TGGs) for incremental pattern matching define the most crucial task. To this end, especially the memory consumption in maintaining (partial and complete) matches must be reduced as our experimental evaluation results revealed.

Finally, expressiveness of TGGs is identified in Table 1.1 as a further non-functional requirement that goes beyond the scope of this thesis. Increasing the expressiveness of TGGs in order to support a larger class of consistency scenarios requires new language features, possibly adopted from the general field of graph grammars such as multi-amalgamation [57], nested quantification [123], or nested application conditions [63]. We, nevertheless, believe to have come up with a solid formal foundation for TGGs that can be compatibly extended for these features.

BIBLIOGRAPHY

[1] Neta Aizenbud-Reshef, Brian T. Nolan, Julia Rubin, and Yael Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3):515–526, 2006. doi: 10.1147/sj.453.0515. URL https://doi.org/10.1147/sj.453.0515. (Cited on page 78.)

[2] Marcus Alanen and Ivan Porres. Difference and Union of Models. In *«UML» 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings*, pages 2–17, 2003. doi: 10.1007/978-3-540-45221-8_2. URL https://doi.org/10.1007/978-3-540-45221-8_2. (Cited on page 79.)

[3] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*, pages 361–375, 2006. doi: 10.1007/11787044_27. URL https://doi.org/10.1007/11787044_27. (Cited on page 125.)

[4] Amesim, Siemens AG. http://www.plm.automation.siemens.com/en/products/lms/imagine-lab/amesim/index.shtml, 2017. [Online; accessed 20-November-2017]. (Cited on page 144.)

[5] Anthony Anjorin. *Synchronization of Models on Different Abstraction Levels using Triple Graph Grammars*. PhD thesis, Darmstadt University of Technology, Germany, 2014. URL http://tuprints.ulb.tu-darmstadt.de/4399/. (Cited on page 4, 84, 100, 119, 120, and 142.)

[6] Anthony Anjorin, Marius Lauder, Sven Patzina, and Andy Schürr. eMoflon: Leveraging EMF and Professional CASE Tools. In *Tagungsband der INFORMATIK - Lecture Notes in Informatics*, volume 192. July 2011. (Cited on page 125.)

[7] Anthony Anjorin, Andy Schürr, and Gabriele Taentzer. Construction of Integrity Preserving Triple Graph Grammars. In *Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, September 24-29, 2012. Proceedings*, pages 356–370, 2012. doi: 10.1007/978-3-642-33654-6_24. URL https://doi.org/10.1007/978-3-642-33654-6_24. (Cited on page 38.)

[8] Anthony Anjorin, Gergely Varró, and Andy Schürr. Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques. *ECEASST*, 49, 2012. URL http://journal.ub.tu-berlin.de/eceasst/article/view/707. (Cited on page 22.)

[9] Anthony Anjorin, Erhan Leblebici, Andy Schürr, and Gabriele Taentzer. A Static Analysis of Non-confluent Triple Graph Grammars for Efficient Model Transformation. In *Graph Transformation - 7th International Conference, ICGT*

2014, Held as Part of STAF 2014, York, UK, July 22-24, 2014. Proceedings, pages 130–145, 2014. doi: 10.1007/978-3-319-09108-2_9. URL https://doi.org/10.1007/978-3-319-09108-2_9. (Cited on page 105, 124, and 153.)

[10] Anthony Anjorin, Sebastian Rose, Frederik Deckwerth, and Andy Schürr. Efficient Model Synchronization with View Triple Graph Grammars. In *Modelling Foundations and Applications - 10th European Conference, ECMFA 2014, Held as Part of STAF 2014, York, UK, July 21-25, 2014. Proceedings*, pages 1–17, 2014. doi: 10.1007/978-3-319-09195-2_1. URL https://doi.org/10.1007/978-3-319-09195-2_1. (Cited on page 122.)

[11] Anthony Anjorin, Karsten Saller, Malte Lochau, and Andy Schürr. Modularizing Triple Graph Grammars Using Rule Refinement. In *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 340–354, 2014. doi: 10.1007/978-3-642-54804-8_24. URL https://doi.org/10.1007/978-3-642-54804-8_24. (Cited on page 35.)

[12] Anthony Anjorin, Erhan Leblebici, Roland Kluge, Andy Schürr, and Perdita Stevens. A Systematic Approach and Guidelines to Developing a Triple Graph Grammar. In *Proceedings of the 4th International Workshop on Bidirectional Transformations co-located with Software Technologies: Applications and Foundations, STAF 2015, L'Aquila, Italy, July 24, 2015.*, pages 81–95, 2015. URL http://ceur-ws.org/Vol-1396/p81-anjorin.pdf. (Cited on page 129.)

[13] Anthony Anjorin, Zinovy Diskin, Freédéric Jouault, Hsiang-Shang Ko, Erhan Leblebici, and Bernhard Westfechtel. Benchmarx Reloaded: A Practical Benchmark Framework for Bidirectional Transformations. In *Proceedings of the 6th International Workshop on Bidirectional Transformations, Bx 2016, co-located with The European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 29, 2017.*, 2017. (Cited on page 3 and 137.)

[14] Holger Bock Axelsen and Robert Glück. What Do Reversible Programs Compute? In *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 42–56, 2011. doi: 10.1007/978-3-642-19805-2_4. URL https://doi.org/10.1007/978-3-642-19805-2_4. (Cited on page 121.)

[15] Holger Bock Axelsen and Robert Glück. A Simple and Efficient Universal Reversible Turing Machine. In *Language and Automata Theory and Applications - 5th International Conference, LATA 2011, Tarragona, Spain, May 26-31, 2011. Proceedings*, pages 117–128, 2011. doi: 10.1007/978-3-642-21254-3_8. URL https://doi.org/10.1007/978-3-642-21254-3_8. (Cited on page 121.)

[16] Jing Bai, Shuming Gao, Weihua Tang, Yusheng Liu, and Song Guo. Semantic-based partial retrieval of CAD models for design reuse. In *Proceedings of the*

2009 ACM Symposium on Solid and Physical Modeling, San Francisco, California, USA, October 5-8, 2009, pages 271–276, 2009. doi: 10.1145/1629255.1629289. URL http://doi.acm.org/10.1145/1629255.1629289. (Cited on page 148.)

[17] Ameni ben Fadhel, Marouane Kessentini, Philip Langer, and Manuel Wimmer. Search-based detection of high-level model changes. In *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, pages 212–221, 2012. doi: 10.1109/ICSM.2012.6405274. URL https://doi.org/10.1109/ICSM.2012.6405274. (Cited on page 54.)

[18] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. Viatra 3: A Reactive Model Transformation Platform. In *Theory and Practice of Model Transformations - 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings*, pages 101–110, 2015. doi: 10.1007/978-3-319-21155-8_8. URL https://doi.org/10.1007/978-3-319-21155-8_8. (Cited on page 123.)

[19] Nestor Velasco Bermeo, Miguel González-Mendoza, and Alexander García Castro. Semantic Representation of CAD Models Based on the IGES Standard. In *Advances in Artificial Intelligence and Its Applications - 12th Mexican International Conference on Artificial Intelligence, MICAI 2013, Mexico City, Mexico, November 24-30, 2013, Proceedings, Part I*, pages 157–168, 2013. doi: 10.1007/978-3-642-45114-0_13. URL https://doi.org/10.1007/978-3-642-45114-0_13. (Cited on page 148.)

[20] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*, pages 273–280, 2001. URL https://doi.org/10.1109/ASE.2001.989813. (Cited on page 1.)

[21] Dominique Blouin, Alain Plantec, Pierre Dissaux, Frank Singhoff, and Jean-Philippe Diguet. Synchronization of Models of Rich Languages with Triple Graph Grammars: An Experience Report. In *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*, pages 106–121, 2014. URL https://doi.org/10.1007/978-3-319-08789-4_8. (Cited on page 4 and 142.)

[22] Julian C. Bradfield and Perdita Stevens. Recursive Checkonly QVT-R Transformations with General when and where Clauses via the Modal Mu Calculus. In *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 194–208, 2012. doi: 10.1007/978-3-642-28872-2_14. URL https://doi.org/10.1007/978-3-642-28872-2_14. (Cited on page 76.)

[23] Alan W. Brown. Model driven architecture: Principles and practice. *Software and System Modeling*, 3(4):314–327, 2004. URL https://doi.org/10.1007/s10270-004-0061-2. (Cited on page 1.)

[24] Gerald G. Brown and Robert F. Dell. Formulating Integer Linear Programs: A Rogues' Gallery. *INFORMS Trans. Education*, 7(2):153–159, 2007. doi: 10.1287/ited.7.2.153. URL https://doi.org/10.1287/ited.7.2.153. (Cited on page 55.)

[25] Hugo Brunelière, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. MoDisco: A model driven reverse engineering framework. *Information & Software Technology*, 56(8):1012–1032, 2014. doi: 10.1016/j.infsof.2014.04.007. URL https://doi.org/10.1016/j.infsof.2014.04.007. (Cited on page 15 and 130.)

[26] Thomas Buchmann, Alexander Dotor, Sabrina Uhrig, and Bernhard Westfechtel. Model-Driven Software Development with Graph Transformations: A Comparative Case Study. In *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers*, pages 345–360, 2007. doi: 10.1007/978-3-540-89020-1_24. URL https://doi.org/10.1007/978-3-540-89020-1_24. (Cited on page 122.)

[27] Business Process Model and Notation, Object Management Group. http://www.omg.org/spec/BPMN/, 2017. [Online; accessed 17-November-2017]. (Cited on page 143.)

[28] Glenn Callow and Roy Kalawsky. A Satisficing Bi-Directional Model Transformation Engine using Mixed Integer Linear Programming. *Journal of Object Technology*, 12(1):1: 1–43, 2013. doi: 10.5381/jot.2013.12.1.a1. URL https://doi.org/10.5381/jot.2013.12.1.a1. (Cited on page 54.)

[29] Chaos Report, The Standish Group. https://www.projectsmart.co.uk/white-papers/chaos-report.pdf, 2014. [Online; accessed 08-June-2017]. (Cited on page 1.)

[30] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956. (Cited on page 25.)

[31] Noam Chomsky. *Syntactic Structures*. Mouton and Co., The Hague, 1957. (Cited on page 25.)

[32] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. JTL: A Bidirectional and Change Propagating Transformation Language. In *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, pages 183–202, 2010. doi: 10.1007/978-3-642-19440-5_11. URL https://doi.org/10.1007/978-3-642-19440-5_11. (Cited on page 121.)

[33] Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors. *Graph-Grammars and Their Application to Computer Science and Biology, International Workshop, Bad Honnef, October 30 - November 3, 1978*, volume 73 of *Lecture Notes in Computer Science*, 1979. Springer. ISBN 3-540-09525-X. doi: 10.1007/BFb0025713. URL https://doi.org/10.1007/BFb0025713. (Cited on page 25.)

[34] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006. URL https://doi.org/10.1147/sj.453.0621. (Cited on page 3.)

[35] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *Theory and Practice of Model Transformations, Second International Conference, ICMT 2009, Zurich, Switzerland, June 29-30, 2009. Proceedings*, pages 260–283, 2009. URL https://doi.org/10.1007/978-3-642-02408-5_19. (Cited on page 3.)

[36] Juan de Lara and Hans Vangheluwe. AToM^3: A Tool for Multi-formalism and Meta-modelling. In *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, pages 174–188, 2002. doi: 10.1007/3-540-45923-5_12. URL https://doi.org/10.1007/3-540-45923-5_12. (Cited on page 123.)

[37] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. From State- to Delta-Based Bidirectional Model Transformations. In *Theory and Practice of Model Transformations, Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings*, pages 61–76, 2010. doi: 10.1007/978-3-642-13688-7_5. URL https://doi.org/10.1007/978-3-642-13688-7_5. (Cited on page 122.)

[38] DOORS, IBM. http://www.ibm.com/us-en/marketplace/rational-doors, 2017. [Online; accessed 20-November-2017]. (Cited on page 77.)

[39] Eclipse Modeling Framework, Eclipse Foundation. http://www.eclipse.org/modeling/emf/, 2017. [Online; accessed 03-August-2017]. (Cited on page 31 and 125.)

[40] Hartmut Ehrig and Hans-Jörg Kreowski. Parallelism of Manipulations in Multidimensional Information Structures. In *Mathematical Foundations of Computer Science 1976, 5th Symposium, Gdansk, Poland, September 6-10, 1976, Proceedings*, pages 284–293, 1976. doi: 10.1007/3-540-07854-1_188. URL https://doi.org/10.1007/3-540-07854-1_188. (Cited on page 53.)

[41] Hartmut Ehrig, Karsten Ehrig, Annegret Habel, and Karl-Heinz Pennemann. Constraints and Application Conditions: From Graphs to High-Level Structures. In *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*, pages 287–303, 2004. doi: 10.1007/978-3-540-30203-2_21. URL https://doi.org/10.1007/978-3-540-30203-2_21. (Cited on page 37.)

[42] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006. ISBN 978-3-540-31187-4. doi: 10.1007/3-540-31188-2. URL https://doi.org/10.1007/3-540-31188-2. (Cited on page 15, 17, 18, 19, 21, 22, 24, 29, 50, and 100.)

[43] Hartmut Ehrig, Karsten Ehrig, and Frank Hermann. From Model Transformation to Model Integration based on the Algebraic Approach to Triple Graph Grammars. *ECEASST*, 10, 2008. URL http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/154. (Cited on page 39, 40, 52, 75, and 77.)

[44] Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Frank Hermann. *Graph and Model Transformation - General Framework and Applications*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2015. ISBN 978-3-662-47979-7. doi: 10.1007/978-3-662-47980-3. URL https://doi.org/10.1007/978-3-662-47980-3. (Cited on page 75 and 77.)

[45] EMFCompare, Eclipse Foundation. http://www.eclipse.org/emf/compare/, 2017. [Online; accessed 03-August-2017]. (Cited on page 79 and 129.)

[46] Claudia Ermel, Frank Hermann, Jürgen Gall, and Daniel Binanzer. Visual Modeling and Analysis of EMF Model Transformations Based on Triple Graph Grammars. *ECEASST*, 54, 2012. URL http://journal.ub.tu-berlin.de/eceasst/article/view/771. (Cited on page 4, 40, and 52.)

[47] Juergen Etzlstorfer, Angelika Kusel, Elisabeth Kapsammer, Philip Langer, Werner Retschitzegger, Johannes Schoenboeck, Wieland Schwinger, and Manuel Wimmer. A Survey on Incremental Model Transformation Approaches. In *Proceedings of the Workshop on Models and Evolution co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, Florida, USA, October 1, 2013.*, pages 4–13, 2013. URL http://ceur-ws.org/Vol-1090/1.pdf. (Cited on page 122.)

[48] Martin Fleck, Javier Troya, and Manuel Wimmer. Search-based model transformations. *Journal of Software: Evolution and Process*, 28(12):1081–1117, 2016. doi: 10.1002/smr.1804. URL https://doi.org/10.1002/smr.1804. (Cited on page 54 and 79.)

[49] Martin Fleck, Javier Troya, and Manuel Wimmer. Search-Based Model Transformations with MOMoT. In *Theory and Practice of Model Transformations - 9th International Conference, ICMT 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-5, 2016, Proceedings*, pages 79–87, 2016. doi: 10.1007/978-3-319-42064-6_6. URL https://doi.org/10.1007/978-3-319-42064-6_6. (Cited on page 54 and 79.)

[50] Charles Forgy. Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. *Artif. Intell.*, 19(1):17–37, 1982. doi: 10.1016/0004-3702(82)90020-0. URL https://doi.org/10.1016/0004-3702(82)90020-0. (Cited on page 85, 109, and 118.)

[51] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang.*

Syst., 29(3):17, 2007. doi: 10.1145/1232420.1232424. URL http://doi.acm.org/10.1145/1232420.1232424. (Cited on page 122.)

[52] Lars Fritsche, Erhan Leblebici, Anthony Anjorin, and Andy Schürr. A Look-Ahead Strategy for Rule-Based Model Transformations. In *Proceedings of the 11th Workshop on Models and Evolution co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA, September 17-22, 2017*, 2017. (Cited on page 102.)

[53] Ismênia Galvão and Arda Goknil. Survey of Traceability Approaches in Model-Driven Engineering. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), 15-19 October 2007, Annapolis, Maryland, USA*, pages 313–326, 2007. doi: 10.1109/EDOC.2007.42. URL https://doi.org/10.1109/EDOC.2007.42. (Cited on page 78.)

[54] Samer Abdul Ghafour, Parisa Ghodous, Behzad Shariat, Eliane Perna, and Farzad Khosrowshahi. Semantic interoperability of knowledge in feature-based CAD models. *Computer-Aided Design*, 56:45–57, 2014. doi: 10.1016/j.cad.2014.06.001. URL https://doi.org/10.1016/j.cad.2014.06.001. (Cited on page 148.)

[55] Amir Hossein Ghamarian, Arash Jalali, and Arend Rensink. Incremental Pattern Matching in Graph-Based State Space Exploration. *ECEASST*, 32, 2010. URL http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/520. (Cited on page 106 and 153.)

[56] Holger Giese, Stephan Hildebrandt, and Stefan Neumann. Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent. In *Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, pages 555–579, 2010. doi: 10.1007/978-3-642-17322-6_24. URL https://doi.org/10.1007/978-3-642-17322-6_24. (Cited on page 4.)

[57] Ulrike Golas, Annegret Habel, and Hartmut Ehrig. Multi-amalgamation of rules with application conditions in -adhesive categories. *Mathematical Structures in Computer Science*, 24(4), 2014. doi: 10.1017/S0960129512000345. URL https://doi.org/10.1017/S0960129512000345. (Cited on page 153.)

[58] Birgit Grammel. *Automatic Generation of Trace Links in Model-driven Software Development*. PhD thesis, Dresden University of Technology, 2014. URL http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-155839. (Cited on page 78.)

[59] Birgit Grammel, Stefan Kastenholz, and Konrad Voigt. Model Matching for Trace Link Generation in Model-Driven Software Development. In *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, pages 609–625, 2012. doi: 10.1007/978-3-642-33666-9_39. URL https://doi.org/10.1007/978-3-642-33666-9_39. (Cited on page 78.)

[60] Joel Greenyer, Sebastian Pook, and Jan Rieke. Preventing Information Loss in Incremental Model Synchronization by Reusing Elements. In *Modelling Foundations and Applications - 7th European Conference, ECMFA 2011, Birmingham, UK, June 6 - 9, 2011 Proceedings*, pages 144–159, 2011. doi: 10.1007/978-3-642-21470-7_11. URL https://doi.org/10.1007/978-3-642-21470-7_11. (Cited on page 84, 100, 120, and 124.)

[61] Esther Guerra and Juan de Lara. Event-driven grammars: relating abstract and concrete levels of visual languages. *Software and System Modeling*, 6(3): 317–347, 2007. doi: 10.1007/s10270-007-0051-2. URL https://doi.org/10.1007/s10270-007-0051-2. (Cited on page 123.)

[62] Esther Guerra and Juan de Lara. An Algebraic Semantics for QVT-Relations Check-only Transformations. *Fundam. Inform.*, 114(1):73–101, 2012. doi: 10.3233/FI-2011-618. URL https://doi.org/10.3233/FI-2011-618. (Cited on page 76 and 77.)

[63] Annegret Habel and Karl-Heinz Pennemann. Nested constraints and application conditions for high-level structures. In *Formal Methods in Software and Systems Modeling, Essays Dedicated to Hartmut Ehrig, on the Occasion of His 60th Birthday*, pages 293–308, 2005. doi: 10.1007/978-3-540-31847-7_17. URL https://doi.org/10.1007/978-3-540-31847-7_17. (Cited on page 153.)

[64] Frank Hermann, Hartmut Ehrig, Ulrike Golas, and Fernando Orejas. Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars. In *Proceedings of the First International Workshop on Model-Driven Interoperability*, MDI '10, pages 22–31, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0292-0. doi: 10.1145/1866272.1866277. URL http://doi.acm.org/10.1145/1866272.1866277. (Cited on page 101, 102, and 124.)

[65] Frank Hermann, Hartmut Ehrig, Fernando Orejas, and Ulrike Golas. Formal Analysis of Functional Behaviour for Model Transformations Based on Triple Graph Grammars. In *Graph Transformations - 5th International Conference, ICGT 2010, Enschede, The Netherlands, September 27 - - October 2, 2010. Proceedings*, pages 155–170, 2010. doi: 10.1007/978-3-642-15928-2_11. URL https://doi.org/10.1007/978-3-642-15928-2_11. (Cited on page 96, 105, 124, and 153.)

[66] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, and Yingfei Xiong. Correctness of Model Synchronization Based on Triple Graph Grammars. In *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, pages 668–682, 2011. doi: 10.1007/978-3-642-24485-8_49. URL https://doi.org/10.1007/978-3-642-24485-8_49. (Cited on page 84 and 120.)

[67] Frank Hermann, Susann Gottmann, Nico Nachtigall, Hartmut Ehrig, Benjamin Braatz, Gianluigi Morelli, Alain Pierre, Thomas Engel, and Claudia Ermel. Triple Graph Grammars in the Large for Translating Satellite Procedures.

In *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*, pages 122–137, 2014. URL https://doi.org/10.1007/978-3-319-08789-4_9. (Cited on page 4 and 142.)

[68] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, Yingfei Xiong, Susann Gottmann, and Thomas Engel. Model synchronization based on triple graph grammars: correctness, completeness and invertibility. *Software and System Modeling*, 14(1):241–269, 2015. doi: 10.1007/s10270-012-0309-1. URL https://doi.org/10.1007/s10270-012-0309-1. (Cited on page 75, 77, and 122.)

[69] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 480–483, 2011. doi: 10.1109/ASE.2011.6100104. URL https://doi.org/10.1109/ASE.2011.6100104. (Cited on page 4 and 121.)

[70] Stephan Hildebrandt. *On the performance and conformance of triple graph grammar implementations*. PhD thesis, University of Potsdam, 2014. URL http://d-nb.info/1054564477. (Cited on page 84, 100, 119, 120, and 124.)

[71] Stephan Hildebrandt, Leen Lambers, and Holger Giese. The MDELab tool framework for the development of correct model transformations with triple graph grammars. In *Proceedings of the First Workshop on the Analysis of Model Transformations, AMT@MODELS 2012, Innsbruck, Austria, October 2, 2012*, pages 33–34, 2012. URL http://doi.acm.org/10.1145/2432497.2432504. (Cited on page 4.)

[72] Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Symmetric lenses. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 371–384, 2011. doi: 10.1145/1926385.1926428. URL http://doi.acm.org/10.1145/1926385.1926428. (Cited on page 122.)

[73] Ivaylo Hristakiev and Detlef Plump. Attributed Graph Transformation via Rule Schemata: Church-Rosser Theorem. In *Software Technologies: Applications and Foundations - STAF 2016 Collocated Workshops: DataMod, GCM, HOFM, MELO, SEMS, VeryComp, Vienna, Austria, July 4-8, 2016, Revised Selected Papers*, pages 145–160, 2016. doi: 10.1007/978-3-319-50230-4_11. URL https://doi.org/10.1007/978-3-319-50230-4_11. (Cited on page 22.)

[74] Arash Jalali, Arend Rensink, and Amir Hossein Ghamarian. Incremental Pattern Matching for Regular Expressions. *ECEASST*, 47, 2012. URL http://journal.ub.tu-berlin.de/eceasst/article/view/736. (Cited on page 141.)

[75] Java Language Specification, Oracle America Inc. http://docs.oracle.com/javase/specs/jls/se7/html/index.html, 2017. [Online; accessed 26-July-2017]. (Cited on page 15.)

[76] Hsinyi Jiang, Tien N. Nguyen, Carl K. Chang, and Fei Dong. Traceability Link Evolution Management with Incremental Latent Semantic Indexing. In *31st Annual International Computer Software and Applications Conference, COMPSAC 2007, Beijing, China, July 24-27, 2007. Volume 1*, pages 309–316, 2007. doi: 10.1109/COMPSAC.2007.225. URL https://doi.org/10.1109/COMPSAC.2007.225. (Cited on page 78.)

[77] Frédéric Jouault and Massimo Tisi. Towards Incremental Execution of ATL Transformations. In *Theory and Practice of Model Transformations, Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings*, pages 123–137, 2010. doi: 10.1007/978-3-642-13688-7_9. URL https://doi.org/10.1007/978-3-642-13688-7_9. (Cited on page 123.)

[78] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: a QVT-like transformation language. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 719–720, 2006. doi: 10.1145/1176617.1176691. URL http://doi.acm.org/10.1145/1176617.1176691. (Cited on page 123.)

[79] Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Model Transformation By-Example: A Survey of the First Wave. In *Conceptual Modelling and Its Theoretical Foundations - Essays Dedicated to Bernhard Thalheim on the Occasion of His 60th Birthday*, pages 197–215, 2012. doi: 10.1007/978-3-642-28279-9_15. URL https://doi.org/10.1007/978-3-642-28279-9_15. (Cited on page 122.)

[80] Richard M. Karp. Reducibility Among Combinatorial Problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, pages 85–103, 1972. URL http://www.cs.berkeley.edu/~luca/cs172/karp.pdf. (Cited on page 74.)

[81] Marouane Kessentini, Houari A. Sahraoui, and Mounir Boukadoum. Model Transformation as an Optimization Problem. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, pages 159–173, 2008. doi: 10.1007/978-3-540-87875-9_12. URL https://doi.org/10.1007/978-3-540-87875-9_12. (Cited on page 54.)

[82] Norbert Kiesel, Andy Schürr, and Bernhard Westfechtel. GRAS: A Graph-Oriented Software Engineering Database System. In *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, pages 397–425, 1996. URL http://link.springer.com/chapter/10.1007/BFb0035688. (Cited on page 123.)

[83] Ekkart Kindler, Vladimir Rubin, and Robert Wagner. An Adaptable TGG Interpreter for In-Memory Model Transformation. In *Proc. of the 2nd International Fujaba Days 2004, Darmstadt, Germany*, volume tr-ri-04-253 of *Technical Report*, pages 35–38. University of Paderborn, 2004. (Cited on page 4.)

[84] Felix Klar, Sebastian Rose, and Andy Schürr. TiE - A Tool Integration Environment. In *Proceedings of the 5th ECMDA Traceability Workshop*, volume WP09-0 of *CTIT Workshop Proceedings*, pages 39–48, 2009. (Cited on page 126.)

[85] Felix Klar, Marius Lauder, Alexander Königs, and Andy Schürr. Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In *Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, pages 141–174, 2010. doi: 10.1007/978-3-642-17322-6_8. URL https://doi.org/10.1007/978-3-642-17322-6_8. (Cited on page 38, 101, 102, and 124.)

[86] Lilija Klassen and Robert Wagner. EMorF - A tool for model transformations. *ECEASST*, 54, 2012. URL http://journal.ub.tu-berlin.de/eceasst/article/view/768. (Cited on page 4 and 84.)

[87] Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. BiGUL: a formally verified core language for putback-based bidirectional programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 61–72, 2016. doi: 10.1145/2847538.2847544. URL http://doi.acm.org/10.1145/2847538.2847544. (Cited on page 4 and 121.)

[88] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 1–6, 2009. doi: 10.1109/CVSM.2009.5071714. (Cited on page 79.)

[89] Simon Frederick Königs, Grischa Beier, Asmus Figge, and Rainer Stark. Traceability in Systems Engineering - Review of industrial practices, state-of-the-art technologies and new research solutions. *Advanced Engineering Informatics*, 26(4):924–940, 2012. doi: 10.1016/j.aei.2012.08.002. URL https://doi.org/10.1016/j.aei.2012.08.002. (Cited on page 78.)

[90] Thomas Kühne. Matters of (Meta-)Modeling. *Software and System Modeling*, 5(4):369–385, 2006. URL https://doi.org/10.1007/s10270-006-0017-9. (Cited on page 1.)

[91] Dilshodbek Kuryazov. Delta Operation Language for Model Difference Representation. In *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland*, pages 2221–2232, 2014. URL http://subs.emis.de/LNI/Proceedings/Proceedings232/article51.html. (Cited on page 79.)

[92] Leen Lambers, Stephan Hildebrandt, Holger Giese, and Fernando Orejas. Attribute Handling for Bidirectional Model Transformations: The Triple Graph

Grammar Case. *ECEASST*, 49, 2012. URL http://journal.ub.tu-berlin.de/eceasst/article/view/706. (Cited on page 22.)

[93] Marius Lauder, Anthony Anjorin, Gergely Varró, and Andy Schürr. Efficient Model Synchronization with Precedence Triple Graph Grammars. In *Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, September 24-29, 2012. Proceedings*, pages 401–415, 2012. doi: 10.1007/978-3-642-33654-6_27. URL https://doi.org/10.1007/978-3-642-33654-6_27. (Cited on page 84, 100, 119, and 120.)

[94] Marius Paul Lauder. *Incremental model synchronization with precedence-driven triple graph grammars*. PhD thesis, Darmstadt University of Technology, Germany, 2013. URL http://tuprints.ulb.tu-darmstadt.de/3352/. (Cited on page 118.)

[95] Erhan Leblebici. Towards a Graph Grammar-Based Approach to Inter-Model Consistency Checks with Traceability Support. In *Proceedings of the 5th International Workshop on Bidirectional Transformations, Bx 2016, co-located with The European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 8, 2016.*, pages 35–39, 2016. (Cited on page 39.)

[96] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. Developing eMoflon with eMoflon. In *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*, pages 138–145, 2014. URL https://doi.org/10.1007/978-3-319-08789-4_10. (Cited on page 4.)

[97] Erhan Leblebici, Anthony Anjorin, Andy Schürr, Stephan Hildebrandt, Jan Rieke, and Joel Greenyer. A Comparison of Incremental Triple Graph Grammar Tools. *ECEASST*, 67, 2014. URL http://journal.ub.tu-berlin.de/eceasst/article/view/939. (Cited on page 137.)

[98] Erhan Leblebici, Anthony Anjorin, Andy Schürr, and Gabriele Taentzer. Multi-amalgamated Triple Graph Grammars. In *Graph Transformation - 8th International Conference, ICGT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 21-23, 2015. Proceedings*, pages 87–103, 2015. doi: 10.1007/978-3-319-21145-9_6. URL https://doi.org/10.1007/978-3-319-21145-9_6. (Cited on page 38.)

[99] Erhan Leblebici, Anthony Anjorin, Lars Fritsche, Gergely Varró, and Andy Schürr. Leveraging Incremental Pattern Matching Techniques for Model Synchronisation. In *Graph Transformation - 10th International Conference, ICGT 2017, Held as Part of STAF 2017, Marburg, Germany, July 18-19, 2017, Proceedings*, pages 179–195, 2017. doi: 10.1007/978-3-319-61470-0_11. URL https://doi.org/10.1007/978-3-319-61470-0_11. (Cited on page 83 and 96.)

[100] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. Inter-model Consistency Checking Using Triple Graph Grammars and Linear Optimization Techniques. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on*

Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, pages 191–207, 2017. (Cited on page 39.)

[101] Erhan Leblebici, Anthony Anjorin, Andy Schürr, and Gabriele Taentzer. Multi-amalgamated triple graph grammars: Formal foundation and application to visual language translation. *J. Vis. Lang. Comput.*, 42:99–121, 2017. doi: 10.1016/j.jvlc.2016.03.001. URL https://doi.org/10.1016/j.jvlc.2016.03.001. (Cited on page 38.)

[102] Andrea De Lucia, Rocco Oliveto, and Paola Sgueglia. Incremental Approach and User Feedbacks: a Silver Bullet for Traceability Recovery. In *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*, pages 299–309, 2006. doi: 10.1109/ICSM.2006.32. URL https://doi.org/10.1109/ICSM.2006.32. (Cited on page 78.)

[103] Nuno Macedo and Alcino Cunha. Implementing QVT-R Bidirectional Model Transformations Using Alloy. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 297–311, 2013. doi: 10.1007/978-3-642-37057-1_22. URL https://doi.org/10.1007/978-3-642-37057-1_22. (Cited on page 76, 77, and 121.)

[104] Patrick Mäder, Orlena Gotel, and Ilka Philippow. Enabling Automated Traceability Maintenance through the Upkeep of Traceability Relations. In *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings*, pages 174–189, 2009. doi: 10.1007/978-3-642-02674-4_13. URL https://doi.org/10.1007/978-3-642-02674-4_13. (Cited on page 78.)

[105] Usman Mansoor, Marouane Kessentini, Manuel Wimmer, and Kalyanmoy Deb. Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. *Software Quality Journal*, 25(2):473–501, 2017. doi: 10.1007/s11219-015-9284-4. URL https://doi.org/10.1007/s11219-015-9284-4. (Cited on page 79.)

[106] John McCarthy. The Inversion of Functions Defined by Turing Machines. In J. McCarthy C.E. Shannon, editor, *Automata Studies, Annals of Mathematical Studies*, number 34, pages 177–181. Princeton University Press, 1956. (Cited on page 121.)

[107] Meta Object Facility, Object Management Group. http://www.omg.org/mof/, 2017. [Online; accessed 08-June-2017]. (Cited on page 2.)

[108] Daniel P. Miranker. TREAT: A Better Match Algorithm for AI Production System Matching. In *Proceedings of the 6th National Conference on Artificial Intelligence. Seattle, WA, July 1987.*, pages 42–47, 1987. URL http://www.aaai.org/Library/AAAI/1987/aaai87-008.php. (Cited on page 141.)

[109] Daniel P. Miranker, David A. Brant, Bernie J. Lofaso, and David Gadbois. On the Performance of Lazy Matching in Production Systems. In *Proceedings of the 8th National Conference on Artificial Intelligence. Boston, Massachusetts, July 29 - August 3, 1990, 2 Volumes.*, pages 685–692, 1990. URL http://www.aaai.org/Library/AAAI/1990/aaai90-103.php. (Cited on page 141.)

[110] Model Driven Architecture, Object Management Group. http://www.omg.org/mda/, 2017. [Online; accessed 08-June-2017]. (Cited on page 2.)

[111] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969. (Cited on page 1.)

[112] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve M. Easterbrook, and Pamela Zave. Matching and Merging of Statecharts Specifications. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 54–64, 2007. doi: 10.1109/ICSE.2007.50. URL https://doi.org/10.1109/ICSE.2007.50. (Cited on page 79.)

[113] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, pages 742–745, 2000. doi: 10.1145/337180.337620. URL http://doi.acm.org/10.1145/337180.337620. (Cited on page 125.)

[114] NX, Siemens AG. http://www.plm.automation.siemens.com/en/products/nx/, 2017. [Online; accessed 20-November-2017]. (Cited on page 144.)

[115] Object Constraint Language, Object Management Group. http://www.omg.org/spec/OCL/, 2017. [Online; accessed 21-September-2017]. (Cited on page 37.)

[116] Fernando Orejas and Elvira Pino. Correctness of Incremental Model Synchronization with Triple Graph Grammars. In *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*, pages 74–90, 2014. doi: 10.1007/978-3-319-08789-4_6. URL https://doi.org/10.1007/978-3-319-08789-4_6. (Cited on page 84, 100, 119, and 120.)

[117] Parasolid, Siemens AG. http://www.plm.automation.siemens.com/de/products/open/parasolid/, 2017. [Online; accessed 20-November-2017]. (Cited on page 146.)

[118] Benjamin C. Pierce. A taste of category theory for computer scientists. Technical report, Carnegie Mellon University, Computer Science Department, 1988. (Cited on page 18.)

[119] Terrence W. Pratt. Pair Grammars, Graph Languages and String-to--Graph Translations. *J. Comput. Syst. Sci.*, 5(6):560–595, 1971. doi: 10.1016/

S0022-0000(71)80016-8. URL https://doi.org/10.1016/S0022-0000(71)80016-8. (Cited on page 4 and 26.)

[120] Query/View/Transformation, Object Management Group. http://www.omg.org/spec/QVT/, 2017. [Online; accessed 08-June-2017]. (Cited on page 4, 39, 75, and 121.)

[121] István Ráth, Gábor Bergmann, András Ökrös, and Dániel Varró. Live Model Transformations Driven by Incremental Pattern Matching. In *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008, Proceedings*, pages 107–121, 2008. doi: 10.1007/978-3-540-69927-9_8. URL https://doi.org/10.1007/978-3-540-69927-9_8. (Cited on page 123.)

[122] Raghu Reddy, Robert France, Franck Fleuery, and Benoit Baudry. Model composition - a signature based approach. In *Aspect Oriented Modeling workshop held with MODELS/UML 2005, Montego*, page 2, 2005. (Cited on page 79.)

[123] Arend Rensink. Nested quantification in graph transformation rules. In *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, pages 1–13, 2006. doi: 10.1007/11841883_1. URL https://doi.org/10.1007/11841883_1. (Cited on page 153.)

[124] Arend Rensink. The Edge of Graph Transformation - Graphs for Behavioural Specification. In *Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, pages 6–32, 2010. doi: 10.1007/978-3-642-17322-6_2. URL https://doi.org/10.1007/978-3-642-17322-6_2. (Cited on page 22.)

[125] Reqtify, Dassault Systems. https://www.3ds.com/products-services/catia/products/reqtify/, 2017. [Online; accessed 20-November-2017]. (Cited on page 77.)

[126] Sebastian Rose, Marius Lauder, Michael Schlereth, and Andy Schürr. A Multi-dimensional Approach for Concurrent Model-Driven Automation Engineering. In *Model-Driven Domain Analysis and Software Development - Architectures and Functions.*, pages 90–113. 2011. doi: 10.4018/978-1-61692-874-2.ch005. URL https://doi.org/10.4018/978-1-61692-874-2.ch005. (Cited on page 4 and 142.)

[127] Andy Schürr. Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language. In *Graph-Theoretic Concepts in Computer Science, 15th International Workshop, WG '89, Castle Rolduc, The Netherlands, June 14-16, 1989, Proceedings*, pages 151–165, 1989. doi: 10.1007/3-540-52292-1_11. URL https://doi.org/10.1007/3-540-52292-1_11. (Cited on page 123.)

[128] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings*, pages 151–163,

1994. URL https://doi.org/10.1007/3-540-59071-4_45. (Cited on page 4 and 26.)

[129] Andy Schürr and Felix Klar. 15 Years of Triple Graph Grammars. In *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*, pages 411–425, 2008. doi: 10.1007/978-3-540-87405-8_28. URL https://doi.org/10.1007/978-3-540-87405-8_28. (Cited on page 51 and 117.)

[130] Petri Selonen and Markus Kettunen. Metamodel-Based Inference of Inter--Model Correspondence. In *11th European Conference on Software Maintenance and Reengineering, Software Evolution in Complex Software Intensive Systems, CSMR 2007, 21-23 March 2007, Amsterdam, The Netherlands*, pages 71–80, 2007. doi: 10.1109/CSMR.2007.31. URL https://doi.org/10.1109/CSMR.2007.31. (Cited on page 79.)

[131] Matthew Stephan and James R. Cordy. A Survey of Model Comparison Approaches and Applications. In *MODELSWARD 2013 - Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development, Barcelona, Spain, 19 - 21 February, 2013*, pages 265–277, 2013. doi: 10.5220/0004311102650277. URL https://doi.org/10.5220/0004311102650277. (Cited on page 79.)

[132] Perdita Stevens. A Landscape of Bidirectional Model Transformations. In *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, pages 408–424, 2007. URL https://doi.org/10.1007/978-3-540-88643-3_10. (Cited on page 3.)

[133] Perdita Stevens. A Simple Game-Theoretic Approach to Checkonly QVT Relations. In *Theory and Practice of Model Transformations, Second International Conference, ICMT 2009, Zurich, Switzerland, June 29-30, 2009. Proceedings*, pages 165–180, 2009. doi: 10.1007/978-3-642-02408-5_12. URL https://doi.org/10.1007/978-3-642-02408-5_12. (Cited on page 76 and 77.)

[134] Perdita Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software and System Modeling*, 9(1):7–20, 2010. URL https://doi.org/10.1007/s10270-008-0109-9. (Cited on page 4 and 121.)

[135] Atakan Sünnetcioglu, Elisabeth Brandenburg, Uwe Rothenburg, and Rainer Stark. ModelTracer: User-friendly Traceability for the Development of Mechatronic Products. *Procedia Technology*, 26(Supplement):365–373, 2016. doi: 10.1016/j.protcy.2016.08.047. URL http://www.sciencedirect.com/science/article/pii/S2212017316303942. (Cited on page 78.)

[136] Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan De Lara, Tihamer Levendovszky, Ulrike Prange, Daniel Varro, and et al. Model Transformations by Graph Transformations: A Comparative Study. In *Model Transformations in Practice Workshop at MoDELS 2005, MONTEGO*, 2005. (Cited on page 122.)

[137] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference computation of large models. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 295–304, 2007. doi: 10.1145/1287624.1287665. URL http://doi.acm.org/10.1145/1287624.1287665. (Cited on page 79 and 129.)

[138] Trove. https://bitbucket.org/trove4j/trove, 2017. [Online; accessed 15-January-2018]. (Cited on page 138.)

[139] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.*, 98:80–99, 2015. doi: 10.1016/j.scico.2014.01.004. URL https://doi.org/10.1016/j.scico.2014.01.004. (Cited on page 85 and 109.)

[140] UML Lab, Yatta Solutions. http://www.uml-lab.com/de/uml-lab/, 2017. [Online; accessed 28-June-2017]. (Cited on page 6 and 31.)

[141] Unified Modeling Language, Object Management Group. http://www.omg.org/spec/UML/, 2017. [Online; accessed 08-June-2017]. (Cited on page 2, 15, and 16.)

[142] Gergely Varró and Frederik Deckwerth. A Rete Network Construction Algorithm for Incremental Pattern Matching. In *Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*, pages 125–140, 2013. doi: 10.1007/978-3-642-38883-5_13. URL https://doi.org/10.1007/978-3-642-38883-5_13. (Cited on page 85 and 109.)

[143] Gergely Varró, Dániel Varró, and Andy Schürr. Incremental Graph Pattern Matching: Data Structures and Initial Experiments. *ECEASST*, 4, 2006. URL http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/12. (Cited on page 123.)

[144] Szilvia Varró-Gyapay and Dániel Varró. Optimization in Graph Transformation Systems Using Petri Net Based Techniques. *ECEASST*, 2, 2006. URL http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/61. (Cited on page 80.)

[145] Antje von Knethen and Barbara Paech. A survey on tracing approaches in practice and research. Technical report, Fraunhofer IESE, 2002. (Cited on page 78.)

[146] Bernhard Westfechtel. Case-based exploration of bidirectional transformations in QVT Relations. *Software & Systems Modeling*, 2016. doi: 10.1007/s10270-016-0527-z. URL https://doi.org/10.1007/s10270-016-0527-z. (Cited on page 4.)

[147] Stefan Winkler and Jens von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and System Modeling*, 9 (4):529–565, 2010. doi: 10.1007/s10270-009-0145-0. URL https://doi.org/10.1007/s10270-009-0145-0. (Cited on page 78.)

[148] Zhenchang Xing and Eleni Stroulia. UMLDiff: An Algorithm for Object Oriented Design Differencing. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 54–65, 2005. doi: 10.1145/1101908.1101919. URL http://doi.acm.org/10.1145/1101908.1101919. (Cited on page 79.)

[149] XML Metadata Interchange, Object Management Group. http://www.omg.org/spec/XMI/, 2017. [Online; accessed 08-June-2017]. (Cited on page 2.)

[150] Xtend, Eclipse Foundation. http://www.eclipse.org/xtend/, 2017. [Online; accessed 21-November-2017]. (Cited on page 146.)

[151] Xtext, Eclipse Foundation. http://www.eclipse.org/Xtext/, 2017. [Online; accessed 21-November-2017]. (Cited on page 145.)

[152] YAKINDU, itemis. https://www.itemis.com/en/yakindu/, 2017. [Online; accessed 20-November-2017]. (Cited on page 77.)

[153] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Towards a Reversible Functional Language. In *Reversible Computation - Third International Workshop, RC 2011, Gent, Belgium, July 4-5, 2011. Revised Papers*, pages 14–29, 2011. doi: 10.1007/978-3-642-29517-1_2. URL https://doi.org/10.1007/978-3-642-29517-1_2. (Cited on page 4 and 121.)

CURRICULUM VITAE

	Personal Details
Date of Brith	31. 05. 1988
Place of Birth	Istanbul, Turkey
Nationality	Turkish
	Work Experience
07/2013–06/2018	Research and Teaching Assistant
	Technische Universität Darmstadt
09/2011–03/2012	Intern, Working Student
	msg systems AG, Frankfurt
11/2010–05/2011	Graduate Student Assistant
	Technische Universität Darmstadt
04/2009–02/2010	Undergraduate Student Assistant
	Technische Universität Darmstadt
	Education
10/2010–05/2013	M.Sc., Electrical Engineering and Information Technology, Technische Universität Darmstadt
10/2007–10/2010	B.Sc., Electrical Engineering and Information Technology, Technische Universität Darmstadt
09/2002–06/2007	German Abitur
	Istanbul High School - The German Department