Bart Jacobs, Gary T. Leavens, Peter Müller, Arnd Poetzsch-Heffter (Editors)

Formal Techniques for Java Programs

Proceedings, Lisbon, Portugal (June 14, 1999)

Mathematik und Informatik

Informatik-Berichte
251 – 05/1999

Formal Techniques for Java Programs

Proceedings, Lisbon, Portugal
June 14, 1999

Bart Jacobs, Gary T. Leavens, Peter Müller, and
Arnd Poetzsch-Heffter (editors)

Contents

Preface

This is the proceedings of the first workshop on *Formal Techniques for Java Programs*, June 14, 1999, held in Lisbon, Portugal. The workshop is affiliated with the *13th European Conference on Object-Oriented Programming*, ECOOP 99. Papers in the proceedings are included here based on the reviews of the workshop organizers. This proceedings will also be available from

`www.informatik.fernuni-hagen.de/import/pi5/publications.html`

The objective of the workshop is to bring together people developing formal techniques and tool support for Java. Formal techniques can help to analyze programs, to precisely describe program behavior, and to verify program properties. Applying such techniques to object-oriented programming is especially interesting because:

1. The OO-paradigm forms the basis for the software component industry with their need for certification techniques.
2. It is widely used for distributed and network programming.
3. The potential for reuse in OO-programming carries over to reusing specifications and proofs.

Java is an excellent target to bridge the gap between formal techniques and practical program development. It plays an important role in these areas and is on the way to becoming a de facto standard because of its reasonably clear and simple semantics.

Bart Jacobs
Gary T. Leavens
Peter Müller
Arnd Poetzsch-Heffter

A Formal Specification of the Java Bytecode Semantics using the B method

Ludovic Casset[1]

Phone: +33 (0)4.42.36.54.52
Ludovic.Casset@gemplus.com

Jean Louis Lanet[2]

Phone: +33.(0)4.42.36.64.22
Jean-Louis.Lanet@gemplus.com

Introduction

The new platforms (*i.e.*, Java Card, MultOS and Smart Card for Windows) allow dynamic storage and the execution of downloaded executable content, which is based on a virtual machine for portability across multiple smart card microcontrollers and for security reasons. Due to the reduced amount of ressources, a specific Java has been specified for the Java card industry, known as the Java Card 2.1 standard. The Java card specification describes the smart card specific features of the virtual machine (*i.e.*, Applet Firewall, Shareable Interfaces, Installer...).

All those mechanisms prevent hostile applets to break the security of the smart card. However the smart card security is based on the assumptions that the JCRE (Java Card Runtime Environment) is correctly implemented. The correctness of the Applet Firewall which is an important part of the JCRE is crucial. It is the means to avoid an applet to reference illegally another applet objects. In fact not only the Applet Firewall but also the complete JCRE and the virtual machine must be correctly implemented. In order to prove such a correctness we have to use formal methods to insure that the implantation is a valid interpretation of the specification.

In the specification, it is not explicitly explain how and when the different controls are done (*i.e.*, type checking, control flow...). A defensive virtual machine where all the checks are performed at runtime has too poor performances. Thus, the smart card industry proposes an architectural design where the checks are performed off-card. The developpers have to extract the static and the dynamic semantics. The static constraints are performed with an off-the-shelf verifier and the on-card interpreter implements the dynamic semantics. If we want to formally implement the interpreter we have to expect that the verifier has been correctly implemented. We propose hereafter a model based on the refinement technique that avoid this potential incoherence.

After a brief presentation of related work, we present the bytecode subset used in our model. Then, we define the state of the defensive virtual machine using the B method [Abr-

[1] ESIL, Ecole Supérieure d'Ingénieurs de Luminy, département informatique, Luminy case 925 - 13288 Marseille cedex 09.

[2] Gemplus Research Lab, Av du Pic de Bertagne, 13881 Gémenos cedex.

96]. An example of instruction refinements is provided. Then, we conclude with the extension of our work.

Related Work

There has been much work on a formal treatment of Java and specifically at the Java language level by [Nip-98], [Dro-97] and [Sym-97]. They define a formal semantics for a subset of Java in order to prove the soundness of their type system. A closer work to our approach has been done by [Qia-98]. The author consider a subset of the bytecode and its work aims to prove the runtime type correctness from their static typing. Using its specification he proposes a proof of a verifier that can be deducted from its virtual machine specification.

The Kimera project [Sir-98] proposes a verifier implementation that has been carefully designed and tested but not based on formal methods. An interesting work has been partially done by [Coh-96] in order to formally implement a defensive virtual machine. It is possible to prove that this model is equivalent to an angressive interpreter plus a sound bytecode verifier.

A new approach

Our approach is based on the Defensive Java Virtual Machine (DJVM) split in order to obtain in the one hand the bytecode verifier and in the other hand the interpreter. At the abstract level, we define the DJVM. By successive refinements, we extract the runtime checks in order to de-synchronize verification and execution process. Then, we obtain invariants representing the formal specification of the static checks. We implement those specifications with an on-the-shelf type inference algorithm.

The Freund and Mitchell subset

Freund and Mitchell introduce in [Fre-98] a bytecode subset. Instructions in this subset are choosen to represent, at the control flow and data levels, most of the bytecode instructions. We use a small variant of this subset. The difference comes from the specialization of instructions **Istore** and **Iload** which load or store local variables of type integer. In these instructions, one can find instructions allowing integer manipulations and also instructions allowing object creations, initializations and uses. Informal specification of these instructions is given below (Fig.1).

By describing operational and static semantics, Freund and Mitchell prove that this subset is sufficient to study object initialization, flow and data-flow controls.

Inc adds one to the *integer* in top of stack.	**Push0** pushes *integer* on stack.
Pop removes the top element of the stack.	**If L** jumps to L or to next instruction according to the value of the *integer* L.
Istore x removes the *integer* from the top of stack and puts it into lacoal variable x.	**Iload x** loads value from local variable x and puts it on top of stack.
Halt terminates program execution.	**New** σ allocates a new uninitialized object of type σ on the top of stack.
Init σ initializes the object of type σ on the top of stack.	**Use** σ performs an operation on a initialized object of type σ.

Fig.1. *Informal specification of the instruction subset.*

Flow control and type correctness

Checking a program means insuring that all instructions are executed in a safe way. We first begin with executing controls on flow and types. We assume we work on a subset of Java types: *integer*, *addri* (unitialized object) and *addr* (initialized object). For such a work, we define a state and its properties. A state is defined by:

- the *pc*, the program counter which value is included in method domain,
- the *type stack*, type of the element of the stack,
- the *type frame* containing types of local variables.

The expected properties of the program are:

- confinement: a program cannot access objects or part of the program out of its workspace,
- stack access: no overflow or underflow during stack manipulation,
- initialisation: an object must be initialized once and only once. The access of an uninitialized object is not allowed.
- type correctness: it is forbidden to convert an integer into a référence; and no arithmic is allowed on pointer.

Assuming such constraints guarantee the correct state. Then, we use transfert functions related to each instruction to change to another correct state. The static semantics gives the constraint set, as the operational semantics gives the transfert functions. We define a complete lattice with the three types described previously. To implement an algorithm checking types, such as the one presented by Dwyer in [Dwy-95], we need such a lattice to organize types and to have relations between them. This algorithm is implemented in the off-card verifier.

The B model of the defensive machine

We explain the model on a particular instruction, the instruction **Inc**. An informal specification of this instruction can be: ***Inc*** *add one to the integer in the top of the stack and let the rest of the stack unchanged.* Clearly, the instruction, on flow level, increments the *pc*

to go to the next instruction. For type verification, it checks that the type on top of stack is an integer.

Our abstract model represents the DJVM: we perform checks on *pc* domain and on types and then we execute the instruction (Fig.2)

```
ins_iload = SELECT (methode (apc) = iload)
  THEN
    IF  (apc < size (methode) ∧ top_stack < max_stack
    ∧ parametre(apc) ∈ dom(types_frames)
    ∧ types_frames(parametre(apc))= INTEGERS)
    THEN
      apc := apc + 1
      || top_stack := top_stack+1
      || types_stacks:=types_stacks⩤{top_stack+1↦INTEGERS}
    END
  END;
```

Fig.2. *Instruction* ***Iload*** *in the DJVM machine.*

Then, we refine until all checks appears in the invariant. The execution is done if the variable **unchecked** set by the invariant is false.

After two refinements, it is possible to express the checks with the following invariant (Fig.3).

```
∀kd.( (kd ∈ dom(methode)) ∧ methode(kd) =iload ∧ unchecked = FALSE
      ⇒kd < size(methode) ∧ SSTtop_stack(kd) < max_stack
      ∧ SSTtop_stack(kd)=SSTtop_stack(kd+1)-1
      ∧ SSTtypes(kd+1)(SSTtop_stack(kd+1))=INTEGERS
      ∧ SSTtypes(kd)⩤{SSTtop_stack(kd)+1 ↦ INTEGERS}=SSTtypes(kd+1)
      ∧ parametre(kd) ∈ dom(SSTtypes_frames(kd)) ∧ parametre(kd)≤max_frame
      ∧ parametre(kd)≥0 ∧ SSTtypes_frames(kd)(parametre(kd))=INTEGERS
      ∧ SSTtypes_frames(kd)=SSTtypes_frames(kd+1))
```

Fig.3. *The invariant for* ***Iload*** *after two refinements.*

Then we obtain an offensive interpreter for the instruction **Iload,** *i.e.*, we just verify that previously the program passed successfully the verifier (see below).

```
ins_iload = SELECT(methode (apc) = iload ∧ unchecked = FALSE)
  THEN
    apc := apc + 1
    || top_stack := top_stack+1
    || types_stacks:=types_stacks⩤{top_stack+1↦INTEGERS}
  END;
```

Fig.4. *The operation for instruction* ***Iload*** *in the last refinement.*

With this approach, we bring to the fore that we split the original defensive machine. We introduce another abstract machine to initialize the variable unchecked by performing static

checks on the bytecode. This machine is in fact the specification of our verifier. The last refinement of the defensive machine appears to be our offensive interpreter. We have 489 Proof Obligations (PO). the project is entirely proved.

The fixed point calculus for type correctness

Computing the right type for a given *pc* is rather difficult because several paths can lead to this *pc* in the tree of possible executions. So, as performing the verification, one must checks that the type obtained is the right one and no error will occur during execution.

The method we use is to compute a fixed point. It means that, considering all paths leading to a given pc, we search the type satisfying all of them. If the program is correct, such a type exists and is usable. Otherwise, checks raise an error.

To complete such a work, we introduce a lattice (Fig.5) over types used in the bytecode. In our study, we have three different usable types: INTEGERS, Addri and Addr. To obtain a complete lattice [Dwy-95], we add a top value TOP, a bottom value $\perp$, a partial-order $\subseteq$ and a binary operator *Meet* Π. We assume that TOP and $\perp$ are non usable type.

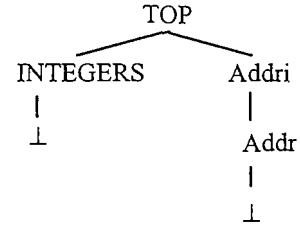


Fig.5. *The complete lattice.*

According to the partial-order over the lattice, we have the relations:

- $\perp \subseteq$ INTEGERS $\subseteq$ TOP,
- $\perp \subseteq$ Addr $\subseteq$ Addri $\subseteq$ TOP.

With such a lattice, we can solve the flow equations:

$$Types[r] = \perp$$
$$\forall\ n \neq r,\ Types[n] = \Pi\ \{\ f_i\ (Types[i])\ \mid\ i \in Preds(n)\}$$

where r is the root node of the tree, *Types* gives the type of the node n, f_i is the transfert function of the node i and *Preds* the set of all predecessors of node n. The transfert function associated to each node fits with the instruction of the given node. In our study, we have ten instructions and ten transfert functions.

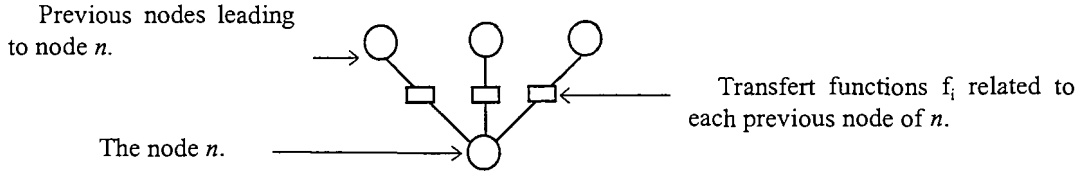


Fig.6. *The representation of the flow equation problem.*

We choose the algorithms presented by Dwyer [Dwy-95] because he proves that his algorithms converge on the greatest fixed point. The complexity of his algorithm is $O(h.N^2)$ where h is the height of the tree and N is the number of nodes.

In the B model we present the specification of the flow equation for the fixed point calculus. First, we introduce the *Meet* operator which, in fact represents the complete lattice over types, the partial-order and the binary operator (Fig.7).

$$
\begin{array}{l}
Meet \in JTYPES \times JTYPES \rightarrow JTYPES \wedge \\
\forall tt.(\, tt \in JTYPES \Rightarrow \forall tp.(tp \in JTYPES \Rightarrow Meet(tt,tp)=Meet(tp,tt))) \wedge \\
\forall tt.(\, tt \in JTYPES \Rightarrow Meet(tt,tt)=tt) \wedge \\
Meet(addr,addri)=addri \wedge Meet(addr,INTEGERS)=TOP \wedge Meet(addr,TOP)=TOP \wedge \\
Meet(addri,TOP)=TOP \wedge Meet(addri,INTEGERS)=TOP \wedge Meet(INTEGERS,TOP)=TOP
\end{array}
$$

Fig.7. *The Meet operator definition*

Then, we specify the set *Preds*. This set is made of all predecessors of a given *pc*. In our model, we add a new feature. For a given *pc*, we associate its predecessors and, for each predecessor we associate the supposed type it attributes to the different variables in the stack and in the frames through the transfert function (Fig.8).

$$
\begin{array}{l}
Preds \in 1..\mathbf{size}(methode) \rightarrow (1..\mathbf{size}(methode) \nrightarrow JTYPES) \\
Preds(ka+1)=Preds(ka+1) \lhd \{ka \mapsto INTEGERS\}
\end{array}
$$

Fig.8. *Definition and example of use of Preds.*

Finally, we translate the flow equation as follow (Figure 9). For each *pc*, we add this element to *Preds* as predecessors of *pc+1*. We associate to *pc* the type of *pc+1* using this path. Then, we compute a partial fixed point thanks to the set *Preds* by combining types through the complete lattice.

$$
\begin{array}{l}
\forall ii.(ii \in \mathbf{dom}(Preds(ka+1)) \wedge Preds(ka+1) \neq \varnothing \wedge \\
Preds(ka+1)(ii) \neq SSTtypes(ka+1)(SSTtop\_stack(ka+1)) \\
\Rightarrow SSTtypes(ka+1)=SSTtypes(ka) \lhd \{SSTtop\_stack(ka) \\
\mapsto Meet(SSTtypes(ka+1)(SSTtop\_stack(ka+1)),Preds(ka+1)(ii))\})
\end{array}
$$

Fig.9. *The flow equation in B*

At the end of the program, type of variables for every *pc* is computed. If no unsuable type remains, the program is correct for types point of view. Otherwise, the verifier raises an error.

Conclusions and Future Work

We entirely proved the defensive machine model at the flow and type control level. We are modelizing the two different parts of the defensive machine, the verifier and the interpreter. The work is already done for the flow control and we are integrating the type

control for the instruction subset and in particular the calculus of the fixed point as presented. The integration of the fixed point calculus is proved at 90% and we are still working on it to improve the model.

In the meantime, we use the results of A. Requet [Req-98] on the JavaCard 2.1 bytecode specification. With his work, we bring to the fore the static and the dynamic semantics of each real instruction. Integrating all these studies, we complete our model to present a defensive machine, a bytecode verifier and an interpreter, matching the JavaCard 2.1 standard.

References

[Abr-96] J. R. Abrial, *The B Book. Assigning Programs to Meanings*, Cambridge University Press 1996.

[Coh-96] Cohen, *Defensive Java Virtual Machine Specification*
`http ://www.cli.com/software/djvm`

[Dro-97] S. Drossoppoulou, S. Eisenbach, *Java is Type Safe - Probably.*

[Dwy-95] M. Dwyer, *Data Flow Analysis for verifying correctness properties of concurrents programs*, Phd thesis, University of Massachusetts, Sept 95.

[Fre-98] S. N. Freund, J. C. Mitchell *A type System for Object Initialization in the Java Bytecode Language* In. Proc. Conf. On Object-Oriented Programming, Systems, Languages, and Applications, pages 310-328. ACM Press 1998. `http ://theory.standford.edu/~freunds.`

[Qia-98] Z. Qian, *Least Types for Memory Locations in Java Bytecode*, Kestrel Institute, Tech. Report, 1998.

[Nip-98] T. Nipkow, D. Oheimb, *Javalight is Type-Safe - Definitely*
25th ACM symposium on Principle of Programming Languages, Jan-1998.

[Req-98] A. Requet, *Spécification Formelle en B d'un Convertisseur de Bytecode pour Applets Javacard*, Rapport de DEA, Université de Nantes, Septembre 1998.

[Sir-98] E. G. Sirer, A. J. Gregory, B. N. Bershad, *Kimera: A Java System Architecture.*
`Http ://kimera.cs.washington.edu/`, 1998

[Sym-97] D.Syme, *Proving Java Type Soundness*, Technical report, University of Cambridge Computer Laboratory, 1997.

Towards a modular denotational semantics of Java

Pietro Cenciarelli

Darmstadt Technical University, Department of Computer Science
Wilhelminenstrasse 7, D-64283, Darmstadt, Germany.
pietro@dvs1.informatik.tu-darmstadt.de

Abstract Applying modular techniques based on monads a denotational semantics of a Java-like programming language featuring concurrent objects is sketched.

1 Introduction

In denotational semantics programs are given a mathematical interpretation in domains with suitable computational structure. In [Mog91], E. Moggi proposed a categorical semantics where the concrete structure of such domains is viewed abstractly as the underlying functor of a strong monad T, where TX is the domain of programs of type X. The formal system associated with this semantics, the *computational lambda calculus*, features a type constructor T, an operator $val_A : A \to TA$ lifting values to computations, and an operator let_T to compose programs of the form $A \to TB$, parametric in A, with programs of type A, which live in the domain TA. By a suitable axiomatization, let_T corresponds to composition in the Kleisli category of T.

Because of the above abstraction, semantics can be approached modularly: complex models of computation are engineered by combining simple monad constructions, each providing the semantic structure to interpret a computational feature: one monad for exceptions one for side-effects, and so forth.

The constructions of Moggi's modular approach are called *monad constructors* [Mog90a]. These are functions $\mathcal{F}$ mapping monads to monads and satisfying certain naturality conditions. When such an $\mathcal{F}$ comes equipped with machinery to lift operations defined for the monad T to operations for $\mathcal{F}$T (see Section 2), then we speak of *semantic constructors* [Mog90b,CM93,Cen95,Cen98].

We sketch the interpretation of a Java-like programming language featuring concurrent objects. In Section 2 we define the building blocks of the proposed model: the semantic constructors for side-effects, resumptions and continuations. In Section 3 we show a few non-trivial semantic equations (notably method invocation), referring to progressively more elaborate models. The use of monads in presenting semantics allows us to introduce computational structure only when needed, without ever rewriting semantic equations. In Section 4 we point to further developments among which a possible link with the structural operational semantics of Java proposed in [CKRW98].

2 Semantic constructors

We address categorical structure by using the computational lambda calculus as metalanguage. Types of the metalanguage stand for objects in the category $\mathcal{C}$. A term of type τ with free variables in a context $x_1 : \tau_1, x_2 : \tau_2, \ldots x_n : \tau_n$ denotes a morphism $\tau_1 \times \tau_2 \times \ldots_n \to \tau$ (see [Mog91] for more detail). We omit indices when undrestood.

Side-effects. The monad constructor $\mathcal{F}$ for side-effects maps a monad T on a cartesian closed category $\mathcal{C}$ to the monad:

$$F_T A = (\mathcal{F}T)A = (T(A \times S))^S,$$

where S is some object in $\mathcal{C}$ interpreting *states* of computation. The operator let_{F_T} of Kleisli composition for F_T is defined as follows in terms of let_T:

$$let_{F_T}\ x \Leftarrow M\ in\ N[x] \stackrel{\text{def}}{=} \lambda s.\, let_T\ (a, s') \Leftarrow (M\ s)\ in\ N[a]\, s'.$$

Resumptions. Let $\mathcal{C}$ be a category with sums and initial algebras $\alpha : F\mu_F \to \mu_F$ for suitable covariant functors $F : \mathcal{C} \to \mathcal{C}$. By a well-known result of Lambek [Lam68], such an algebra is necessarily an isomorphism and therefore it can be interpreted as "minimal" solution to the recursive domain equations $X \cong FX$. *Algebraically complete categories*, which are introduced in [Fre91], have the required properties. We write $\mu X.F[X]$ for μ_F, where $F[X]$ is an expression (with a free "type variable" X) describing a functor F.

Resumable computations are elements of the domain

$$G_T A = (\mathcal{G}T)A = \mu X.T(A + X).$$

Using the isomorphism $\alpha : T(A + G_T A) \to G_T A$, its inverse γ and let_T, one can define an operation $C_{A,B} : (A \to G_T B) \times (G_T A \to G_T B) \times G_T A \to G_T B$ of case analysis as follows:

$$C(f, g, M) \stackrel{\text{def}}{=} \alpha(let_T\ z \Leftarrow \gamma(M)\ in\ \gamma(case\ z\ of\ inl(a)\ .\, f(a)\ \ inr(u)\,.\, g(u))).$$

Intuitively, C applies let_T to run "one step" of the resumable computation M (viewed through γ as an T computation) and then analyses the result z: if it is value a, that is if the computation of M is completed, then it returns $f(a)$, otherwise it applies g to whatever of M is left to do, that is u. We may leave the isomorphisms α and γ implicit when understood.

Kleisli composition can be expressed in terms of C as follows:

$$let_{G_T}\ x \Leftarrow M\ in\ N \stackrel{\text{def}}{=} C(\lambda x.\, N,\ \lambda w.\, let_T\ x \Leftarrow w\ in\ N,\ M).$$

Let $or : G_T A \times G_T A \to G_T A$ be an operator of nondeterministic choice between two resumable computations (see below). The interleaved execution of two programs can be expressed in this computational setting by means of the following operation $|| : G_T A \times G_T B \to G_T(A \times B)$ of parallel composition:

$$M||N \stackrel{\text{def}}{=} C(\lambda a : A.\, let\ b \Leftarrow N\ in\ \eta(a,b),\ \lambda w : RA.\, w||N,\ M)\ or \\ C(\lambda b : B.\, let\ a \Leftarrow M\ in\ \eta(a,b),\ \lambda u : RB.\, M||u,\ N).$$

Intuitively, $M||N$ is the computation which either executes one step of M and puts what is left of it in parallel with N, or it executes one step of N and puts what is left of it in parallel with M.

Continuations. The monad constructor $\mathcal{H}$ for continuations maps a monad T to

$$(\mathcal{H}T)A = T(Res)^{T(Res)^A},$$

where Res is an object interpreting the type of final results. A computation of type $(\mathcal{H}T)A$ takes a *continuation* in $T(Res)^A$ as input and returns a T-computation of a final result. In this setting, Kleisli composition is as follows:

$$let\ x \Leftarrow M\ in\ N \stackrel{\text{def}}{=} \lambda k.\, M(\lambda x.\, N[x]\, k).$$

Redefining operations. The semantic constructors defined above allow the reinterpretation of cerain operations in the new computational setting (see [Cen98] for a discussion). In particular, the constructors $\mathcal{F}$ and $\mathcal{G}$ lift operations $op : TA \times TA \to TA$ respectively to $(\mathcal{F}op) : F_T A \times F_T A \to F_T A$ and $\mathcal{G}op : G_T A \times G_T A \to G_T A$ as follows:

$$(\mathcal{F}op)\, M\, N = \lambda s.\, op\ (M\, s)\, (N\, s) \\ (\mathcal{G}op)\, M\, N = \alpha(op\ \gamma(M)\, \gamma(N)).$$

Nondeterministic choice between resumable computations, as required above, can be obtained by applying $\mathcal{G}$ to a "union" operator $\cup_A : PA \times PA \to PA$, where P is some power object construction.

A computation $K : TA$ lifts to a computation $\mathcal{H}K : (\mathcal{H}T)A$ with continuations as follows:

$$(\mathcal{H}K)\, k = let_T\ a \Leftarrow K\ in\ k(a).$$

For example, let TA be $(\mathcal{F}\, Id)A = (A \times S)^S$, let 1 be the trivial domain $\{*\}$, and let $upd_v : N \to T1$ be the operator which assigns a number to a variable v, that is: $upd_v\, n\ s = (*, s[v \mapsto n])$. Spelling out the reinterpretation of upd_v in a continuation passing semantics, that is in the computational setting $\mathcal{H}(\mathcal{F}\, Id)$, we have:

$$(\mathcal{H}\ upd_v)\, n\, k\, s = k(*)\, s[v \mapsto n].$$

Noticing that the continuation of a statement expects a value (the only) in 1, the assignment of n to v is the computation which, given a continuation k and a state s, runs the rest of the program, that is $k(*)$, in the new state $s[v \mapsto n]$.

3 Interpretation

We shall first focus on the notion of state. At this stage, we assume that a class is interpreted as an object A equipped with pairs of operations $upd_{A,i} : A \times X \to A$ for updating, and $ct_{A,i} : A \to X$ for reading, one such pair for each instance variable i of type X. This interpretation may be described as a *coalgebra* of a suitable functor. For example, a class declaring a single instance variable may be interpreted as a coalgebra of the functor $FY = Y^X \times X$, where the coalgebra map yields the morphisms upd and ct. Of course this example is oversimplified since there is more to encapsulate in an object's state space than just its instance variables. In Java, for example, there are structures for locking and unlocking objects. Moreover, there are copies of instance variables stored in the working memory of each thread, and which are as much part of the global state as are the master copies.

Let Obj be a set of object identifiers, and let τ be a map assigning to each identifier o the object of $\mathcal{C}$ interpreting the class of o. We make use of the "dependent types" $\tau(o)$ to define the object $\mathcal{M}$ of stores:

$$\mathcal{M} \stackrel{\text{def}}{=} \Sigma O \subseteq Obj.\, \Pi o \in O.\, \tau(o).$$

We similarly obtain the object $\mathcal{R} \stackrel{\text{def}}{=} \Sigma I \subseteq Ide.\, I \to Val$ of environments, where Ide is the type of ("local") variable identifiers and Val is the type of storable values, and define states to be:

$$S \stackrel{\text{def}}{=} \mathcal{R} \times \mathcal{M}.$$

This roughly corresponds to the separation in Java of stack and heap.

Interpretation is described by translating the programming language into the computational lambda calculus augmented with suitable operations, such as upd, associated with a specific notion of computation. As advocated in [Mog91], the translation, written $[\![\_]\!]$, maps terms of type σ to terms of type $T[\![\sigma]\!]$, where T is a strong monad:

$$[\![\_]\!] : Term(\sigma) \to T[\![\sigma]\!].$$

For sketching the interpretation of expressions, assignments and method calls, it is enough to work with the monad $TA = (\mathcal{F}\, Id)A = (A \times S)^S$:

$$\begin{aligned} [\![M + N]\!] &= let\ x \Leftarrow [\![M]\!]\ in\ (let\ y \Leftarrow [\![N]\!]\ in\ val(x + y)) \\ [\![o.i = M]\!] &= let\ x \Leftarrow [\![M]\!]\ in\ UPD_{o.i}(x) \end{aligned}$$

where $UPD_{o.i} : X \to R1$ is the computation defined by

$$UPD_{o.i}(x) \stackrel{\text{def}}{=} \lambda\, \rho\, \mu.\, \{(inl(*), (\rho, \mu[o \mapsto upd_{\tau(o),i}(\mu(o), x)]))\}.$$

Let $\tau(o) = A$, let m be a method of a class interpreted by A, let $x : \sigma_1$ be its parameter, and let $M[x] : \sigma_2$ be its body. We write $[\![o.m]\!]$ for the morphism $[\![\sigma_1]\!] \to R[\![\sigma_2]\!]$, defined as: $[\![o.m]\!](a) = \lambda\rho, \mu.\, [\![M]\!](\rho[this \mapsto \mu(o)]\,[x \mapsto a], \mu)$.

$$[\![o.m(N)]\!] = let\ a \Leftarrow [\![N]\!]\ in\ [\![o.m]\!](a).$$

Models of a simple language with assignments to a global memory and an *explicit* operator of parallel composition are obtained in a category $\mathcal{C}$ with the structure described in Section 2 by composing the constructions $\mathcal{F}$ and $\mathcal{G}$, that is by using the monad

$$RA = (\mathcal{G}(\mathcal{F}P))A = \mu X.\, P((A + X) \times S)^S.$$

Because of the abstraction introduced in the interpretation by the use of *let* and *val*, we don't have to rewrite the previous semantic equations. Moreover, we can lift the operator $\cup_A$ to $or_A = \mathcal{G}(\mathcal{F}\cup_A) : RA \times RA \to RA$ and interpret parallel composition as shown above.

However, this is not what one needs to model a language like Java, where parallel threads of computation are implicitely activated by the invocation of a void method *run*. In fact, one cannot have

$$[\![o.run\,(); M]\!] = [\![o.run\,()]\!] \;||\; [\![M]\!] \qquad (\ddot{\frown})$$

because this semantics would not be compositional with respect to the operator ";" of sequential composition. On the other hand, if ";" is to be taken seriously, then $[\![o.run\,(); M]\!] = let\ x \Leftarrow [\![run\,()]\!]\ in\ [\![M]\!]$, which is no parallel computation at all.

A solution is to introduce continuations, which allow us to start a new thread, while the current thread continues computation with whatever follows the invocation of *run*. Applying the constructor $\mathcal{H}$ of Section 2 to R, we obtain:

$$QA = (\mathcal{H}R)A = [\mu X.\, P((L1 + X) \times S)^S]^{[\mu X.\, P((L1+X)\times S)^S]^A},$$

where $L1$ is the domain of finite lists of $*$s, which is taken here as finite results. This choice is consistent with the operational semantics of [CKRW98] where a *successfully* terminating computation of n threads produces a sequence of n asterisks. If exceptions are considered, the domain of results must be accordingly enlarged. Now we can define the semantics of *run*:

$$[\![o.run]\!]\, a\, k = k(*) \;||\; \lambda\rho, \mu.\, [\![M]\!]\, (\rho[this \mapsto \mu(o)], \mu)\, \epsilon.$$

The semantics of method invocation is just as above, but we assign a special morphism $1 \to Q1$ to $[\![o.run]\!]$: the current continuation k is given the expected return value $*$, while the body M, which is passed the empty continuation ϵ, is run in parallel.

4 Conclusions

The sketch of semantics proposed above tries to lay the structure for interpreting a small but meaningful subset of Java. To claim success for this attempt, many basic language features, which are part of the traditional repertoire of denotational semantics, are still to be considered. Blocks and exceptions are striking examples. What we believe is also to be achieved is a match with operational semantics. A possible development in this direction is the adoption as notion of state of the *event spaces* introduced in the SOS of [CKRW97] (also based on interleaving). By replacing the simple *upd* and *ct* of Section 3 with the event space operations *read*, *write*, *load*, *store*, *lock* and *unlock*, suitably axiomatised in [CKRW98], working memories and synchronization can be introduced in the proposed model. This approach is currently under investigation.

References

[Cen95] P. Cenciarelli. *Computational applications of calculi based on monads.* PhD thesis, Department of Computer Science, University of Edinburgh, 1995. CST-127-96. Also available as ECS-LFCS-96-346.

[Cen98] P. Cenciarelli. An Algebraic View of Program Composition. In Armando Haeberer, editor, *Proceedings of 7th International Conference on Algebraic Methodology and Software Technology (AMAST'98)*, 1998. LNCS 1548, Springer.

[CKRW97] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. From Sequential to Multi-Threaded Java: an Event-Based Operational Semantics. In Michael Johnson, editor, *Algebraic Methodology and Software Technology (proc. of AMAST'97)*, pages 75–90, 1997. LNCS 1349, Springer.

[CKRW98] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An Event-Based Structural Operational Semantics of Multi-Threaded Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, 1523 LNCS. Springer, 1998.

[CM93] P. Cenciarelli and E. Moggi. A syntactic approach to modularity in denotational semantics. In *Proceedings of 5th Biennial Meeting on Category Theory and Computer Science.* CTCS-5, 1993. CWI Tech. Report.

[Fre91] P. Freyd. Algebraically complete categories. In A. Carboni, M.C. Pedicchio, and G. Rosolini, editors, *Category Theory - Proc. of the Int'l Conf. held in Como, Italy, July 1990*, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104. Springer-Verlag, Berlin, 1991.

[Lam68] J. Lambek. A fixed point theorem for complete categories. *Math. Zeitschr.*, 103:151–161, 1968.

[Mog90a] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, University of Edinburgh, Comp. Sci. Dept., 1990.

[Mog90b] E. Moggi. Modular approach to denotational semantics. Unpublished manuscript, November 1990.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

A Formal Approach to the Specification of Java Components

S. Cimato and P. Ciancarini

Dipartimento di Scienze dell'Informazione,
Università di Bologna
Mura Anteo Zamboni 7,
I – 40127 Bologna, Italy

Abstract. The goal of our work is to provide a formal notation for the specification and the verification of software components and a methodology for the design of software architectures for Java-based applications. We introduce an algebraic style of specification for modules written in Java, and explore the use of such a notation for the definition of components to be successively employed in the building of concrete applications. We also provide an architectural language which exploits the components descriptions based on the previous approach and provides a formal framework where it is possible to state and analyze system global properties.

1 Background

We are working in the field of distributed software architecture investigating formal methodologies for the specification and the design of distributed applications and techniques for their analysis. Our research on Java has been conducted towards the development of a methodology for the formal specification and verification of Java components and architectures. At this aim, we have been developing a formal framework for the design and the analysis of Java component based applications which is based on an interface specification language for the component descriptions and an architectural language to combine such components. Furthermore we provide a formal semantics for Java programs, considering all the innovative features of the Java framework such that the resulting model allows the dynamic analysis of Java applications; for this purpose we rely on a "chemical" framework [BB92] which provides facilities for modeling concurrency and distribution [Cim99a].

The motivation and the starting point of our research was the recognition that while Java is becoming one of the most used programming languages, software designers have few specific tools and techniques at their own disposal to formally reason on the resulting system. Generic specification languages, such as Z or UML , can be effectively exploited as well as architectural description languages, such as Wright [All97] or Darwin [MDEK95], but the generality of the model does not catch all the features of the underlying programming language (Java) and the abstractions needed to make formal descriptions practical and effective. On

the other hand, practical approaches for component-based software development based on Java such as JavaBeans [Sun96], or on other languages (VisualBasic or Delphi), have a restricted application domain and offer frameworks for the composition of software modules but suffer from the lack of a well-understood formal foundation.

An open question to be answered is how practical development can be influenced by the application of formal development methodologies and how much the additional costs in terms of time and money produced by the formalization efforts are repayed by the resulting product. The benefits expected from the application of formal methods are the removal of ambiguities in the design and the capability offered to the designer for the analysis and the verification of properties of the design. The capability to reason about Java programs and to state the correctness of the implementations with respect to their specification is of increasing importance especially for applications where safety is a critical requirement, such as e-commerce or secure document interchange. We need lightweight notations and methodologies to reduce as much as possible the impact of the integration of the formalization in the software development process. Among the main advantages coming from the integration of a formal approach in the software development process, we expect to:

- enhance the reuse of both specifications and code relying on the modularity present in the Java programming language;
- provide efficient and user-friendly tools to reason on and verify program properties;

Reuse as a goal of component based software development is also hampered by the poor use of formal notations. Informal or under-specified software components force designers to look into implementation details to consider their behavior and the side effects they entail on the whole system. We expect the workshop to focus on the state of the art of the notations, techniques and tools available for Java programming and to discuss how Java programmers can benefit from a component oriented approach supported by a formal framework allowing analysis of component and architectural properties.

2 A formal framework for Java components and architectures

The description of a system as composition of reusable and reconfigurable components has a number of benefits, coming from the possibility for the specifier to subdivide complex problems in smaller and easier solvable subproblems and to reason separately on organizational and functional aspects of a system. The architectural level of design deals with the high level organization of computational elements and their interactions. Our notion of software architecture is substantially based on the Shaw and Garlan's model [SG96]. which is more suitable for a practical integration of architectural design within the software

development process. We recognize *components* and *connectors* as basic elements of an architectural description. *Configurations* combining the instances of those computational elements and defining their interaction, provide a complete description of a system architecture.

Complex applications may be thought as collection of components interacting via connectors, which collaborate to achieve a result. Once selected the architectural model we must also define a *methodology* guiding the design of components and a *formal framework* making developers able to reason about the properties both of the components and of the whole application. For this purpose we introduce (in the next section) a formal specification language which allows us to specify each component by supplying both an algebraic description of its internal state and a specification of the interface it provides to the external environment. Behavioral specifications alone are insufficient to determine the kind of a system being specified, since different kinds of systems can exhibit the same observable behavior [Lam89]. The description of the functiona;l aspects of each operation must be integrated with specific interface information describing how the operation may be invoked, specifying for example for each function the number and the types of its arguments [Win87].

3 Ljala: a specification language for Java components

The Larch Java interface language (Ljala for short), has been developed building on the Larch approach to specification [GH93]. The peculiarity of Larch with respect to other specification languages is the "two tiers" (or layers) approach it uses. The kernel tier, language-independent, is based on the Larch Shared Language, an algebraic specification language which provides a mathematical vocabulary defining the properties of useful abstractions like sets, stack or other. The other tier, language dependent, is based on a behavior interface language in which predicates on pre and post conditions describe the effect of the execution of the operations on the state of a program module [GH93,Win87]. The advantage of such an approach is that separating the specifications of abstractions from the specifications of the state transformations, reuse and clarity are improved. The algebraic components can be easily included in different applications, since they do not depend on the particular state or model of computation or programming language. Having an interface specification language close to the target programming language makes the component designer able to reason in terms of language dependent issues and eases the task of the implementor.

Since in Java classes are the basic unit of programming, in Ljala each specification module specifies the interface and the behaviour of a class. The syntax of a class declaration is much the same of a Java class declaration, consisting of a class header and a body. The header denotes the modifiers and the name of the class. The body may contain an **uses** clause which defines the traits that are used in the class providing the vocabulary to specify its behavior. An optional **invariant** clause specifies a property that must be true for all objects of the class, restricting the space of the abstract values for that class. Methods are

specified by providing a header which gives the interface in Java syntax and a body which provides the behavior in terms of state changes. Each method body is composed of a sequence of **requires**, **modifies**, **ensures** clauses which introduce the preconditions for the execution of the operation, the list of modifiable objects and the postconditions which must hold after the operation, respectively. We model Java concurrent features by providing each object model with a *waiting set*, where threads waiting for synchronization conditions to hold are added and with a *lock* to support mutual exclusion. An additional **when** clause provides the mechanism for checking synchronization conditions which must hold when an operation is waiting to be scheduled.

3.1 Specification of components and connectors

According to [Nie95], a software component is an "abstraction with plugs", i.e., a component encapsulates both data and independent behaviour with a well-defined way to interact with the external environment and the other components. In our notation, the abstraction contained in the component definition is expressed by the algebraic part of the specification provided by the included traits. Designers can enrich the component description, setting out the basic functionalities and the desired properties. The interface specification expressed by the Ljala module provides the description of the component behaviour in terms of the allowed operation.

To give an explicit specification of interactions between components, we consider connectors as first class entities in the design of the system. Even if connectors are not strictly necessary from a logical point of view (they could be regarded as particular types of components), from a methodological point of view, connectors match the abstractions which designers use to describe system architectures. They correspond roughly to the lines connecting the computational elements in the informal diagrams which usually provide the description of a system; formalizing their role in the overall design and providing them with a well defined semantics, support the understanding and the analysis of the behavior of the system [Agh98,All97]. Connectors bridge the gap between the low level control mechanism offered traditionally by the target programming language, in our case Java, and the high level coordination mechanism needed to capture interaction patterns between autonomous objects.

Traits for a generic component and a generic connector respectively are given in figure 1. Basically, components and connectors are both active elements, each owning a set of *ports* and *roles*, respectively. A *port* is an interface between each component and its environment; a *role* is an interaction point among participating components. Starting from the traits for a generic component (connector), more specific descriptions of components can be derived by extension and/or parameterization of generic theories, exploiting the usual mechanism for the inclusion or instantiation of theories. The idea is to provide a hierarchy of theories whose leaves are the components (connectors) descriptions to be effectively used for the specification of the system architecture.

*component:***trait**
includes *Set(iport, Set[iport]), Set(oport, Set[oport])*
component **tuple of** *inports: Set[iport], outports: Set[oport]*

*connector:***trait**
includes *Set(irole, Set[irole]),Set(orole, Set[orole])*
connector **tuple of** *inroles: Set[irole], outroles: Set[orole]*

Fig. 1. Traits for generic components and connectors

3.2 Configurations

To define the interrelationships between components and to give a description of the overall system obtained assembling its components, we need a further level of specification. We propose a Ljala Architectural Language, which can be used to specify the structure of a software system in terms of the configurations of interacting Ljala components. A *configuration* module lists the instance of design elements which form our system, the Ljala modules which are used for the specification and the attachments between ports of the components and roles of the connectors. The structure of a configuration module is showed in figure 2.

system-name:*configuration*
component:
list of component instances
connector:
list of connector instances
attached:
list of connections among ports and roles
behaves:
activation rules
properties:
topological constraints

Fig. 2. Structure of a configuration module

The configuration module has the task to define the topology of the system being built. The *component* and *connector* parts of the module serve to name the instance of the components and the connectors, respectively, used for the description of the system. For each type a trait must have been provided. The *attached* part lists the connections among components and connectors. Each

component may be reused in the same system by defining multiple instantiations and opportunely connecting their ports through the connectors constituting a system. A number of *behave* clauses can be stated for each component in order to specify its behaviour. Each clause is composed by a precondition on the state of the component which acts as a trigger for the activation of the rule. If the preconditions hold, the operations described in the remaining part of the rule are enabled and can be executed by the object scheduler; each operation is then performed according the specification provided in the behaviour module of the component.

4 Verification

The use of Ljala notation in the specification phase simplifies considerably the verification process since its syntax is strictly related to the target programming language. Namely, Ljala interfaces specifications can be directly translated in Java, making the bridging of the gap between specification and concrete implementation an easier task for software developers.

Ljala specifications provide a formal model for components and applications obtained as suitable composition of interacting components. We are interested in verifying properties and invariants which are contained in the specifications and which can be tested with respect to the theories resulting from the combination of the traits describing the included components. At this aim, inspection and proof of properties are possible by using the Larch Prover tool [GG91], an interactive theorem prover designed for LSL modules. Detection of inconsistencies ensures that the given specifications or the given combination of elements does not fulfill the requested properties of violates the asserted invariants.

On the other hand we would like to be able to prove the correctness of the implementations with respect to the Ljala specifications. For this purpose we need a formal model of the programming language which is provided us by the operational semantics for Java described in [Cim99a]. The state of the Java Virtual Machine executing the program is modeled by a tuple $\mathbb{C}, \mathcal{M}, \Gamma$ which describes the set of valid class declarations loaded into runtime, the memory where values can be retrieved and the current environment where bindings among variables and values are stored, respectively. Execution of Java statements is modeled by a set of transition rules which describe the modification of the state caused by the performing of the operation. A Java class, which must be structured as the given specification, is a consistent implementation of the specification if for every operation, each method invocation occurred in a state $S = \mathbb{C}, \mathcal{M}, \Gamma$ which satisfies the assertions in the formula ψ_r causes a transition of the JVM in a state S' which satisfies the assertion in $\psi_m \wedge \psi_e$:

$$for\ every\ S\ s.\ t.\ S \models \psi_r,\ \ then\ \ S' \models \psi_m \wedge \psi_e\ holds$$

where ψ_r,ψ_e, and ψ_m are pre, post and modifies conditions respectively expressed in the Ljala specification. Intuitively a state S satisfies a formula ψ if, after have substituted the current values in the state with the abstract values

they represent, the formula is still valid in the equational theory of the traits included in the module. To relate the concrete values manipulated by the Java program with the abstract values which are in the specification we must provide an *abstraction function* [LG86,Lea91], which maps each value of the implementation type to the abstract value of the corresponding sort. To state that the formula ψ is satisfied in a state S, the formula ψ_S, obtained after the replacing of the variables contained in ψ with the abstract values corresponding to the current values in state S, must be verified in the equational theory of the specification. Let us denote with $\mathcal{T}$ the theory of all the traits used by the interface specification module, then:

$$S \models \psi \iff \mathcal{T} \models \psi_S$$

meaning that ψ_S must be a logical consequence of the assertions in $\mathcal{T}$, i.e. ψ_S is true for every model of the axioms in $\mathcal{T}$.

5 Conclusion

The decomposition of complex software systems in a collection of easy combinable computational elements with well defined responsibilities has a number of benefits in a reuse-oriented development process. In our approach, each component specification integrates both the description of its interface and its functional behavior supported by an algebraic model in which a more abstract description can be given. Those descriptions may be used as the basis for the analysis and as a guide for the designers which have to produce the implementations of the components and to integrate the different parts of the system to work together. Differently from other approaches based on very abstract notations such as Z [AAG95], CSP [AG97,All97] or the π-calculus [MDEK95], Ljala notation achieves an acceptable compromise between formality and practice, providing at the same time both a formal framework where it is possible to reason on the system properties, and an easy way to refine a specification into an implementation. In effect, the overload on the development process due to the formalization effort has to be balanced by the benefits gained in terms of clearness of the design and analysis capabilities offered to the designers. This is particularly true for reusable software components, since the effort to write formal specifications is largely repayed from having complete models which ease their reuse in the building of new applications. On the other hand, Ljala can be used as an "annotation" language [GMP90,LBR98] for Java classes providing a powerful technique to add formal documentation to existing software. Documenting class libraries, frameworks and Application Programmer Interfaces (the JavaBeans framework can be efficiently modeled in our notation [Sun96,Cim99b]), Ljala formalism and, more in general behavioral interface languages, provides a way to construct practical and effective formal specifications.

References

[AAG95] G. Abowd, R. Allen, and D. Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, October 1995.

[AG97] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, June 1997.

[Agh98] G. Agha. Compositional Development from Reusable Components Requires Connectors for Managing both Protocols and Resources. In *Workshop on Compositional Software Architectures*, January 1998.

[All97] R. J. Allen. *A Formal Approach to Software Architecture.* PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, May 1997.

[BB92] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[Cim99a] S. Cimato. *A Methodology for the Specification and Verification of Java Components and Architectures.* PhD thesis, Dept. of Computer Science, University of Bologna, Italy, February 1999.

[Cim99b] S. Cimato. Specifying component-based java applications. In *Proceedings of Third International Conference on Formal Methods for Open Object-Based Distributed Systems FMOODS'99*, Florence, Italy, March 1999.

[GG91] S. Garland and J. Guttag. A guide to LP, the Larch Prover. Technical Report RR 82, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, December 1991.

[GH93] J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, Berlin, 1993.

[GMP90] D. Guaspari, C. Marceau, and W. Polak. Formal Verification of Ada Programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, September 1990.

[Lam89] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[LBR98] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A Behavioral Interface Specification Language for Java. Technical Report TR-98-06a, Iowa State University, 1998.

[Lea91] G. T. Leavens. Modular Specification and Verification of Object Oriented Programs. *IEEE Software*, 8(4):72–80, July 1991.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, Cambridge, MA, 1986.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin.

[Nie95] O. Nierstrasz. Component-Oriented Software Technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object Oriented Software Composition*, pages 3–28. Prentice-Hall, December 1995.

[SG96] M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[Sun96] Sun Microsystems Inc. Java Beans 1.0, October 1996.

[Win87] J. Wing. Writing Larch Interface Language Specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

Formal Refinement and Proof of a Small Java Program

Tony Clark

Department of Computing, University of Bradford, UK, BD7 1DP
a.n.clark@scm.brad.ac.uk

Abstract. The main components of a formal technique for specifying, refining and proving properties of object-oriented programs are presented. The technique is based on a λ-notation whose semantics is given using standard categorical constructs. An example of the formal development of a small Java program is presented.

1 Introduction

The aim of this work is to provide a rigorous framework for step-wise object-oriented software development which supports specification, refinement, proof and implementation. The framework takes the form of a categorical semantics of object-oriented system behaviour and a design language based on λ-notation.

This paper gives an overview of the main components of the framework using a simple system requirements and producing a Java program. It is not possible to give a full analysis of the approach in a paper of this length, the reader is directed to work by the author in the area of OO systems: [Cla96] [Cla94] [Cla97], [Cla98], [Cla99a], [Cla99b] and [Cla99c] and related work: [Ken99], [Ken97], [Eva98], [Eva99], [Bic97], [Lan98] and [Rui95] in formal methods for object-oriented development. The reader is directed to [Bar90], [Ehr91], [Gog75], [Gog89], [Gog90], [Pie96] and [Ryd88] for related work using category theory in systems development.

2 Development Framework

An *object state* is $<\alpha, \tau, \rho>$ where α is the object's type, τ is the object's identity and ρ is a partial function mapping attribute names to values. A *message* is $<\tau_s, \tau_t, \nu>$ where τ_s is the identity of the source object, τ_t is the identity of the target object and ν is a data value. Object-oriented system computation occurs in terms of state transitions resulting from message passing: $\ldots \longrightarrow \Sigma_1 \stackrel{(I,O)}{\longrightarrow} \Sigma_2 \longrightarrow \ldots$ in which a set of object states Σ_1 receives a set of input messages I producing a transition to states Σ_2 and output messages O. Since the behaviour of a system design may be non-deterministic it can be represented as a graph whose nodes are labelled with sets of object states and whose edges are labelled

with pairs of sets of messages. This leads to a category **Obj** whose objects are graphs and whose arrows are graph homomorphisms.

System construction is described by standard categorical constructions in **Obj**. Given two behaviours O_1 and O_2 in **Obj** the product $O_1 \times O_2$ exhibits both O_1 and O_2 behaviour subject to structural consistency constraints. The co-product $O_1 + O_2$ exhibits either O_1 or O_2 behaviour. Equalizers, pull-backs and push-outs can be used to express constraints such that two or more behaviours are consistent. Computational category theory provides an algorithm for computing the behaviour of a system of inter-related components using limits.

Any behaviour O can be viewed as a category in which the objects are behaviour states and arrows are sequences of message pairs. Category-hood follows from: every object Σ has an identity arrow $[]$; and for every pair of arrows $f : \Sigma_1 \rightarrow \Sigma_2$ and $g : \Sigma_2 \rightarrow \Sigma_3$ there is an arrow $g \circ f : \Sigma_1 \rightarrow \Sigma_3$ which is constructed as $f +\!\!+ g$; and the associativity of $\circ$ follows from the associativity of $+\!\!+$. A refinement R is expressed as adjoint functors $R : O_1 \rightarrow O_2$ and $U : O_2 \rightarrow O_1$:

The diagram 1 states that performing a computation in the source object is the same as translating the source state, performing the computation in the target object and then translating the target state. Given any Σ_1 the refinement is *sound* if for every f there exists a g and is *complete* if for every g there is an f [Sab97].

$$\begin{array}{ccc} \Sigma_1 & \xrightarrow{f} & U(\Sigma_2) \\ \downarrow & & \uparrow \\ R(\Sigma_1) & \xrightarrow[g]{} & \Sigma_2 \end{array} \qquad (1)$$

Object-oriented designs are expressed using a λ-notation [Lan64] whose semantics is given by **Obj**. A behaviour is denoted by a functions M and is supplied with type, identity, attribute and message information: $M(\alpha)(\tau)(v)(I) = \bigcup_{i=1,n} \{(P_i, O_i)\}$ where P_i are *replacement behaviours* and O_i are corresponding output messages. This approach is essentially the same as that of Actor Theory [Agh86] [Agh91]. The basic model of message handling is *asynchronous*, however syntactic sugar can be used to express *synchronous* message passing. The following example shows how a behaviour function (left) which synchronously sends a message e_1 is translated to a behaviour function (right) which uses a replacement *wait*:

```
letrec agent(α)(τ)(σ)(m) =
    case m of
        p1 →
            let p2 ← e1
            in e2
    end
```

```
letrec agent(α)(τ)(σ)(m) =
    case m of
        p1 → (agent(α)(τ)(σ) + wait, e1)
            whererec wait(m) =
                case m of
                    p2 → e2
                    else (wait, ∅)
                end
    end
```

3 Development of a Java Program

The requirements for a library system are defined. An initial object-oriented design is constructed. A single refinement step is performed and verified. A simple system property is established. The design is analysed prior to translating it to an implementation in Java (appendix A).

Software to control a library is required. The library has readers who may borrow copies of books. At any given time each reader has a number of books on loan. New readers may join the library at any time. The library has a number of copies of books. Each book has a unique title. A copy is either on the shelf in the library or is being borrowed by a reader. Libraries operate a shares readership policy whereby joining one library permits readers to borrow books at all participating libraries.

A library system consists of a single object with a state (R, B) consisting of readers R and books B. Each reader is a pair (n, C) where n is a name and C is a set of borrowed copies. Each book is a pair (n, i) where n is a name and i is the number of shelved copies. Initially we treat R and B as lookup tables. Let T be a table with keys $dom(T)$, lookup is $T \bullet k$, extension is $T[k \mapsto v]$. Adding table values is defined as follows (removing is similarly defined):

$$T[k \oplus v] \equiv \begin{cases} T[k \mapsto T \bullet k \cup \{v\}] & \text{when } isSet(T \bullet k) \\ T[k \mapsto T \bullet k + v] & \text{when } isInt(T \bullet k) \end{cases}$$

Initial system behaviour can be decomposed into the success and failure modes. The design operator + allows us to define these modes separately and then combine them. Success mode is defined as follows:

letrec $libOk(\alpha)(\tau)(R, B)(m) =$
  case m **of**
    $addReader(n) \rightarrow (libOk(\alpha)(\tau)(R[n \mapsto \emptyset], B), \emptyset)$ **when** $n \notin dom(R)$
    $addBook(n) \rightarrow (libOk(\alpha)(\tau)(R, B[n \mapsto 0]), \emptyset)$ **when** $n \notin dom(B)$
    $addCopy(n) \rightarrow (libOk(\alpha)(\tau)(R, B[n \oplus 1]), \emptyset)$ **when** $n \in dom(B)$
    $borrow(n_1, n_2) \rightarrow (libOk(\alpha)(\tau)(R[n_1 \oplus n_2], B[n_2 \ominus 1]), \emptyset)$
      when $n_1 \in dom(R)$ & $n_2 \in dom(B)$
    $return(n_1, n_2) \rightarrow libOk(\alpha)(\tau)(R[n_1 \ominus n_2], B[n_2 \oplus 1]), \emptyset)$
      when $n_1 \in dom(R)$ & $n_2 \in dom(B)$
    else $(libOk(\alpha)(\tau)(R, B), \emptyset)$
  end

Given a state (R, B) in the source behaviour, a refinement acts as identity on R and transforms $B = \{n_1 \mapsto i_1, \ldots, n_k \mapsto i_k\}$ into a set of object identifiers $\{\tau_1, \ldots, \tau_k\}$ and introduces new objects $\tau_1 \mapsto (n_1, i_1), \ldots, \tau_k \mapsto (n_k, i_k)$ to the system state. A book behaviour is as follows:

letrec $book(\alpha)(\tau)(n, i)(m) =$
  case m **of**
    $<\tau', \tau, getName> \rightarrow (book(\alpha)(\tau)(n, i), \{<\tau, \tau', n>\})$
    $borrow \rightarrow (book(\alpha)(\tau)(n, i-1), \emptyset)$ **when** $i > 0$
    $addCopy \rightarrow (book(\alpha)(\tau)(n, i+1), \emptyset)$
    else $(book(\alpha)(\tau)(n, i), \emptyset)$
  end

The successful library behaviour is modified to take account of book objects. The initial design uses set membership to test for the existence of a book. This must now be implemented as a private method of the library:

$$
\begin{array}{l}
<\tau', \tau, \mathit{findBook}(\emptyset, n)> \rightarrow (\mathit{libOk}(\alpha)(\tau)(R, B), \{<\tau, \tau', \mathit{noBook}>\}) \\
<\tau', \tau, \mathit{findBook}(\{o\} \cup S, n_1)> \rightarrow \\
\quad \textbf{let } n_2 \leftarrow <\tau, o, \mathit{getName}> \\
\quad \textbf{in if } n_1 = n_2 \\
\quad\quad \textbf{then } (\mathit{libOk}(\alpha)(\tau)(R, B), \{<\tau, \tau', \mathit{book}(o)>\}) \\
\quad\quad \textbf{else } (\mathit{libOk}(\alpha)(\tau)(R, B), \{<\tau', \tau, \mathit{findBook}(S, n_1)>\})
\end{array}
$$

When a library receives an *addBook* message with a name n which does not already exist then a new book object is created. We assume that τ'' is a new object identifier and that β is the type tag for books:

$$
\begin{array}{l}
\mathit{addBook}(n) \rightarrow \\
\quad \textbf{let } \mathit{noBook} \leftarrow <\tau, \tau, \mathit{findBook}(n)> \\
\quad \textbf{in } (\mathit{libOk}(\alpha)(\tau)(R, B \cup \{\tau''\}) \times \mathit{book}(\beta)(\tau'')(n, \emptyset), \emptyset)
\end{array}
$$

To verify the refinement step the following source state is used: $\{\tau \mapsto (R, B)\}$ where R is a set of readers and B is the set $\{n_1 \mapsto i_1, \ldots, n_k \mapsto i_k\}$. The corresponding target state is $\{\tau \mapsto (R, T)\} \cup O$ where T is the set of object identifiers $\{\tau_1, \ldots, \tau_k\}$ and O is the state $\{\tau_1 \mapsto (n_1, i_1), \ldots, \tau_k \mapsto (n_k, i_k)\}$. The refinement of *addBook* is sound and complete when the following diagram commutes (see diagram 1):

$$
\begin{array}{ccc}
\{\tau \mapsto (R, B)\} & \xrightarrow{\quad \mathit{addBook}(n) \quad} & \{\tau \mapsto (R, B[n \mapsto 0])\} \\
\Big\downarrow & & \Big\uparrow \\
\{\tau \mapsto (R, T)\} \cup O & \xrightarrow[\quad c \circ \mathit{addBook}(n) \quad]{} & \{\tau \mapsto (R, T \cup \tau'')\} \cup \\
 & & O[\tau'' \mapsto (n, 0)]
\end{array}
\qquad (2)
$$

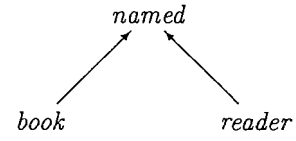
A proof of 2 is by induction on the size of the set B and the length of the computation c. Further refinement identifies a class of behaviours for *reader* and adds a private method *findReader* to the library.

The design language is given a formal semantics in terms of standard constructions in **Obj**. A design language proof theory provides a framework for establishing program properties. The proof theory views a behaviour function as a mapping from input messages and states to output messages and states. Proofs typically are by induction on the length of a messages stream. Since refinement is formally defined, it is possible to show that properties are preserved by refinement transformations.

Consider the following theorem. For any library (R, B), if b is a book borrowed by a reader then $b \in \mathit{dom}(B)$. The proof is by induction in the length of the input message stream. The theorem holds for library $(\emptyset, \emptyset)$ and the empty

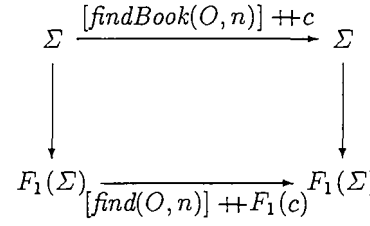
stream. Assume by induction that the theorem holds for library (R, B) and messages ms. Now show by case analysis on m that the theorem holds for all messages $ms +\!\!+ [m]$. We conclude that the theorem holds.

Consider the behaviours *book* and *reader*. Both provide a state component n which is used to index into collections of behavioural instances using the message *getName*. This indicates that there is a common behaviour *named* and projection morphisms. In an implementation *named* will occur as a super-class of both *book* and *reader*.

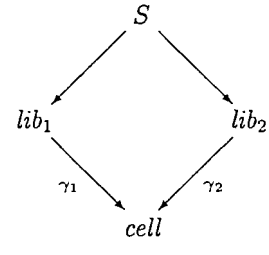


Consider a behaviour functor F_1 which acts on system states by projecting all book objects to equivalent named objects by forgetting the copy count. F_1 acts as identity on all arrows except that $findBook(O, n)$ is replaced by $find(O, n)$, $book(b)$ is replaced by $found(b)$ and $noBook$ is replaced by $notFound$.

In order for F_1 to be valid, it must be sound and complete with respect to indexing into collections of books. Therefore, for any system state Σ, the diagram on the right must commute. Similarly, a behaviour functor F_2 is defined to project states and calculations involving indexing readers. This leads us to replace the behaviours for *findBook* and *findReader* with a single behaviour *find*.



The shared readership policy is expressed as a pull-back S on a diagram showing two (or more) libraries which project onto a behaviour *cell* containing their readers. The pull-back ensures that both libraries have the same readers. There are a number of implementation choices for the shared readership policy whose behaviour is defined by S. If the programming language supports shared data between class instances (such as `static` in Java) then the R component of a library class may be shared.



References

[Agh86] Agha, G.: Actors: *A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[Agh91] Agha, G.: The Structure and Semantics of Actor Languages. In proceedings of REX School/Workshop on Foundations of Object-Oriented Languages, LNCS 489, Springer-Verlag, 1991.

[Bar90] Barr, M. & Wells, C.: *Category Theory for Computing Science*. Prentice Hall International Series in Computer Science, 1990.

[Bic97] Bicarregui, J., Lano, K. & Maibaum, T.: Towards a Compositional Interpretation of Object Diagrams. Technical Report, Department of Computing, Imperial College, 1997.

[Cla94] Clark, A. N.: A Layered Object-Oriented Programming Language. GEC Journal of Research, 11(3), The General Electric Company p.l.c., pp 173 – 180, 1994.

[Cla96] Clark, A. N.: *Semantic Primitives for Object-Oriented Programming Languages.* PhD Thesis, QMW, University of London, 1996.

[Cla97] Clark, A. N. & Evans, A. S.: Semantic Foundations of the Unified Modelling Language. In the proceedings of the First Workshop on Rigorous Object-Oriented Methods: ROOM 1, Imperial College, June, 1997.

[Cla98] Clark, A. N.: Type Checking OCL Expressions. Technical Report, 1998.

[Cla99a] Clark, A. N.: A Semantics for Object-Oriented Systems. Presented at the Third Northern Formal Methods Workshop. September 1998. To appear in BCS FACS Electronic Workshops in Computing, 1999.

[Cla99b] Clark, A. N.: A Semantics for Object-Oriented Design Notations. Technical report, submitted to the BCS FACS Journal, 1999.

[Cla99c] Clark, A. N.: A Semantic Framework for Object-Oriented Development. Technical report, submitted to the L'Objet journal special issue on formal object-oriented development, 1999.

[Ehr91] Ehrich, H-D., Goguen, J. A. & Sernadas, A.: A Categorical Model of Objects as Observed Processes. In the proceedings of REX School/Workshop on Foundations of Object-Oriented Languages, LNCS 489, Springer-Verlag, 1991.

[Eva98] Evans, A. S.: Reasoning with UML Class Diagrams. In WIFT '98, IEEE Press, 1998.

[Eva99] Evans, A. S. & Lano, K. C.: Rigorous Development in UML. To appear in the proceedings of the ETAPS '99, FASE Workshop, 1999.

[Gog75] Goguen, J.: Objects. Int. Journal of General Systems, 1(4):237–243, 1975.

[Gog89] Goguen, J.: A Categorical Manifesto. Technical Report PRG-72, Programming Research Group, Oxford University, March 1989.

[Gog90] Goguen, J. A.: Sheaf Semantics for Concurrent Interacting Objects. Mathematical Structures in Computer Science, 1990.

[Ken99] Kent, S. & Gil J.: Visualising Action Contracts in Object-Oriented Modelling. To appear in the IEE Software Journal, 1999.

[Ken97] Kent, S.: Constraint Diagrams: Visualising Invariants in Object-Oriented Models. In the proceedings of OOPSLA 97, ACM Press, 1997.

[Lan64] Landin P.: The Next 700 Programming Languages. Communication of the ACM, 9(3), 1966, pp 157 – 166.

[Lan98] Lano, K. & Bicarregui, J.: UML Refinement and Abstraction Transformations. In the proceedings of the Second Workshop on Rigorous Object-Oriented Methods: ROOM 2, Bradford, May, 1998.

[Pie96] Piessens F. & Steegmans E.: Categorical Semantics for Object-Oriented Data Specifications. In *Formal Methods and Object Technology*, (eds.) Goldsack, S. J. & Kent, S. J., Springer-Verlag, 1996, pp 302 – 316.

[Rui95] Ruiz-Delgado, A., Pitt, D. & Smythe, C.: A Review of Object-Oriented Approaches in Formal Specification. The Computer Journal, 38(10), 1995.

[Ryd88] Rydeheard, D. E. & Burstall, R. M.: *Computational Category Theory.* Prentice Hall International Series in Computer Science, 1988.

[Sab97] Sabry, A. & Wadler, P.: A Reflection on Call-by-Value. ACM Transactions on Programming Languages and Systems, 19(5), pp 111 – 136,1997.

A Library Implementation in Java

Each independent behaviour is defined as a Java class. The state components of the behaviour are defined as fields and the message handlers are defined as methods. Any common behaviour is defined using inheritance. The main features are: the class `Named` defines the common behaviour for readers and books; attribute `readers` in `Library` is declared `static` so that libraries implement the shared readership policy; class `Library` defines a method `find` that is used to index both readers and books.

```
class Named {
  private String name;
  public Named(String name) { this.name = name; }
  public String getName() { return name; }
}

class Book extends Named {
  private int copies = 0;
  public Book(String name) { super(name); }
  public void borrow()
  {
    if(copies > 0)
      copies = copies - 1;
    else throw new Error("no copies left");
  }
  public void addCopy() { copies = copies + 1; }
}

class Reader extends Named {
  private Vector copies = new Vector();
  public Reader(String name,Vector copies)
  {
    super(name);
    this.copies = copies;
  }
  public void borrow(String name) { copies.addElement(name); }
  public void ret(String name) { copies.removeElement(name); }
}

class Library {
  private static Vector readers = new Vector();
  private Vector books = new Vector();
  public void addReader(String name) { readers.addElement(new Reader(name,new Vector())); }
  public void addBook(String name) { books.addElement(new Book(name)); }
  public void addCopy(String bookName)
  {
    Book book = (Book)find(bookName,books);
    if(book != null)
      book.addCopy();
    else throw new Error("cannot find book");
  }
  private Named find(String name,Vector table)
  {
    Named named = null;
    for(int i = 0; (named == null) && (i < table.size()); i++) {
      Named n = (Named)table.elementAt(i);
      if(n.getName().equals(name))
        named = n;
    }
    return named;
  }
  public void borrow(String readerName,String bookName)
  {
    Reader reader = (Reader)find(readerName,readers);
    Book book = (Book)find(bookName,books);
    if((reader != null) & (book != null)) {
      reader.borrow(bookName);
      book.borrow();
    } else throw new Error("illegal name in borrow");
  }
  public void ret(String readerName,String bookName)
  {
    Reader reader = (Reader)find(readerName,readers);
    Book book = (Book)find(bookName,books);
    if((reader != null) & (book != null)) {
      reader.ret(bookName);
      book.addCopy();
    } else throw new Error("illegal name in ret");
  }
}
```

Software Development with Object-Z, CSP and Java: A Pragmatic Link from Formal Specifications to Programs

Clemens Fischer

University of Oldenburg
Fachbereich Informatik
PO-Box 2503, 26111 Oldenburg, Germany
fischer@informatik.uni-oldenburg.de

Abstract. Object-Z and CSP are high level specification languages which offer powerful formal support for the design of distributed, communicating systems. Java is an ideal implementation language for such systems. But developing provably correct Java implementations from these specifications is notoriously difficult. To bridge this gap we suggest to use Jass, which extends Java with assertions, as an intermediate language. These assertions can be generated automatically from Object-Z and CSP specifications. This does not guarantee a provably correct implementation, but allows an easy way of testing and linking error messages directly to the formal specification.

1 Introduction

Java is well suited for designing distributed systems which must meet high correctness requirements. But applicable methods for building distributed high quality systems are not widely accepted nor available.

A pragmatic approach for better software quality is Meyer's 'design-by-contract' [Mey97]. The idea is to write predicates that specify properties of systems into the code and to check these predicates during run time.

Predicates at the beginning of a method (preconditions) describe the properties that a user of a method must obey. In return, the developer of the code guarantees some property (postcondition) at method termination.

Missing design-by-contract support in Java is discussed in SUN's list as bug number 4071460. At the time of writing this paper, it's voted second by Java developers in the 'Request For Enhancements'-list. The Jass (Java with assertions) compiler [Jas99,Bar99] overcomes this problem. Jass programs are normal Java programs augmented with assertions placed in specific comments. The compiler translates these assertions to Java statements that check the predicate during run time.

Design-by-contract, however, is limited to specifying functional, sequential aspects of systems. It is not possible to capture dynamic aspects of communicating distributed systems and it cannot be used to provide high-level interface specifications that not only hide functional implementation details but also conceal architectural design decisions or underlying network technology.

Therefore we suggest a new link from the high level formal specification language CSP-OZ [Fis97] to Jass.

CSP-OZ is a combination of Object-Z [DRS95] and CSP [Ros97]. Object-Z is strong at specifying the state space and methods of a class in a predicative way. This matches nicely the predicates used in a design-by-contract approach. But Object-Z has a powerful type system, schema calculus and a toolkit framework that go far beyond the predicate language used in Jass or Eiffel. It is, however, weak at describing dynamic aspects of distributed communicating systems. This loophole can be filled with the process algebra CSP, which comes with many operators and a mature theory of communicating, distributed systems. CSP, on the other hand, has no high level concepts for the specification of data.

The combination CSP-OZ [Fis97] takes the best of these two worlds. It is a wide range specification language for complex distributed systems like telecommunication, satellite, or rail-road systems. CSP-OZ has powerful methods for building provably correct systems: like transformation rules and data refinement [FH97] or model checking support [FW99]. In principle, it is also possible to transform CSP-OZ specifications into code. However, all these tasks require expertise in using formal methods and tools and require often significant interaction. This problem holds for many formal methods and limits the chance for industrial success stories.

The key idea we present here, is to generate Jass assertions from CSP-OZ specifications. This task can be automated given a CSP-OZ specification with implementable data types. An overview of this method can be found in Fig. 1. The three ellipses CSP-

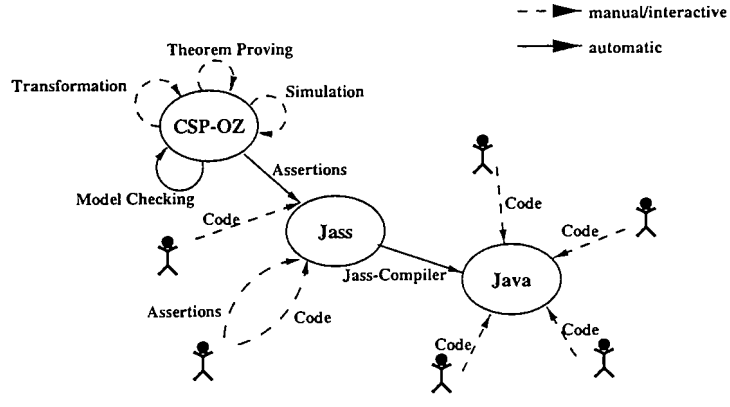


Fig. 1. Overview over the Jass-Method

OZ, Jass and Java represent the three design levels. An initial CSP-OZ specification can be transformed using verified rules. These steps can be, but don't have to be, proven correct using model checking or theorem proving. If the data types match Java data types, method headers and assertions can be generated automatically. The implementation of a method body has to be done by hand. The architectural information of a specification can also be translated automatically if it meets predefined design patterns. Otherwise the architecture has to be translated manually, too.

The step from CSP-OZ to Jass does not guarantee the correctness in a mathematical sense. But it combines the advantages of a fully formal specification of a distributed system with the simplicity of a design-by-contract approach. Errors can be found earlier during testing and any assertion violation can be linked directly to the part of the formal specification it stems from, making error interpretation much easier.

Furthermore, it can be introduced step by step in an evolutionary fashion into the software development process. As indicated in Fig. 1, not all parts of the system have to be specified using CSP-OZ. Some part, safety critical for example, can be designed using CSP-OZ, but other part of the code can be developed traditionally using design-by-contract (Jass) or pure Java.

The rest of this paper sketches the development of an Internet based game to discuss this idea in more detail.

2 CSP-OZ

As case study, we use a distributed version of the game Tic-Tac-Toe where two players can play via the Internet. The following class models the basic data aspects of the system. We start with a high level specification of the user interface without any details of the underlying distributed implementation we are aiming at.

CSP-OZ specifications begin with introducing the basic types and constants. We need symbols, positions and the board.

$$Symbol ::= cross \mid circle$$
$$Pos == \{1,2,3\} \times \{1,2,3\}$$
$$Board == Pos \pfun Symbol$$

TTT_{Data}

State

$b : Board$
$lp : Pos$

$lp \in \operatorname{dom} b \lor b = \varnothing$

Init

$b = \varnothing$

win

$\Delta(lp)$
$w! : Symbol$

$(\exists j : 1..3 \bullet \forall i : 1..3 \bullet b(i,j) = w!) \lor$
$(\exists j : 1..3 \bullet \forall i : 1..3 \bullet b(j,i) = w!) \lor$
$(p(1,1) = w! \land p(2,2) = w! \land p(3,3) = w!) \lor$
$(p(3,1) = w! \land p(2,2) = w! \land p(1,3) = w!)$

The class TTT_{Data} has three schemas: `State` specifies the state space of the object. A board (b) and the last position used (lp) are stored. The invariant $lp \in \operatorname{dom} b$ (dom is the domain of the function b) guarantees that a symbol is on the position lp. The initial schema `Init` specifies the initial states. The operation schema *win* outputs the winner of the game. It might change the value lp ($\Delta(lp)$) and outputs the value $w!$. The behaviour of *win* is specified by the predicate below the line: The symbol $w!$ wins if it occupies a vertical, a horizontal or one of the diagonal rows. As lp is not restricted by this predicate, any value that fulfils the invariant is possible for lp after executing *win*.

The class TTT_{Data} has actually more operation schemas – like *move* and *upd* – which are omitted for lack of space here.

TTT_{Game}

– Interface Declaration omitted –

$\texttt{main} = moveA \rightarrow (updB \rightarrow B \,\Box\, winA \rightarrow WIN)$

$B = moveB \rightarrow (updA \rightarrow \texttt{main} \,\Box\, winB \rightarrow WIN)$

$WIN = updA \rightarrow \texttt{skip} \;|||\; updB \rightarrow \texttt{skip}$

TTT_{Data}

$\texttt{enable\_}winA \mathrel{\widehat{=}} \text{pre } win[cross/w!]$

$\texttt{effect\_}winA \mathrel{\widehat{=}} win \ldots$

The complete behaviour of the system is specified in the class TTT_{Game}. It has the methods *move*, *win* and *upd*, one for each player A and B. The CSP processes *main*, B and *WIN* specify the possible traces of the system: A move of A is followed by an update of B's board or ($\Box$) A wins the game. When someone wins, the screen is updated in any order ($|||$) before termination (`skip`).

TTT_{Game} inherits TTT_{Data}; the actual behaviour of the operations is specified with the enable and effect schemas which correspond somehow with pre- and postconditions.[1] E.g. the method *winA* should only happen, if cross is the winner.

The next step is to develop $TTT_{DisGame}$, the distributed version of the game. It's overall structure can be found in Fig. 2. The two objects $TTT(A)$ and $TTT(B)$ communicate over a socket to provide the same service for the players as the class TTT_{Game}. Note that the Fig. 2 has a precise semantics based on the CSP operators for parallel composition and hiding.

CSP-OZ offers powerful tools to prove formally, that $TTT_{DisGame}$ is indeed a refinement of TTT_{Game}. However, CSP-OZ can beneficially just be used as a specification language to document the different design steps.

The structure of $TTT_{DisGame}$ can already be implemented using Java. But some data refinement steps on the class TTT, which are not shown here, have to be done to yield implementable Java data types.

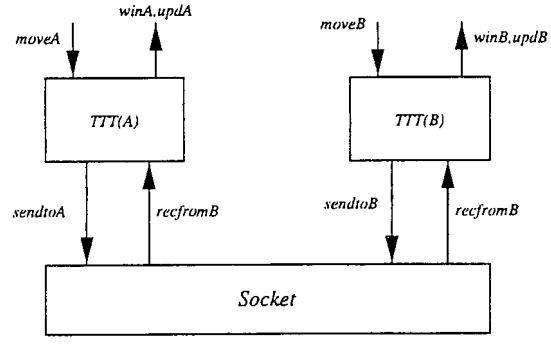


Fig. 2. The System $TTT_{DisGame}$

3 Jass

We now sketch the Jass implementation of the class TTT. Light fonts stem from CSP-OZ assertion generation; the parts written with normal black font (two lines in the code below) have to be provided manually. Beside the class invariant and ensure and effect

[1] Check [Fis97,Fis98] for the relations between enable/effect predicates and pre-/postconditions.

predicates which are well known from design-by-contract, Jass has a new trace assertion (which is generated from the CSP part of a class) to check dynamic aspects of systems during run time.[2] Note that the history of the game (h) and the number of moves (n) are stored in addition to the CSP-OZ specification above. To generate the assertions, the developer must provide the mapping from CSP-OZ names to Java expressions. E. g. *sendtoA* from $TTT_{DisGame}$ is mapped to `S.sendto` in the Jass code below.

```
public class TTT {
  private byte b[]= new byte[3][3];
  private byte h[] = new byte[9];
  private byte lx, ly, n, sym;
  /** invariant n <= 9 **/
  /** trace main = moveA -> S.sendto ->
        S.recfrom -> (A.upd -> main | A.win -> SKIP) **/
  public TTT(byte s){
    sym = s;
    /** ensure (forall x:{1..3} # forall y:{1..3}#
                                b[x][y]==0); n==0; **/
  }
  public void move(byte x, byte y, byte s){
   /** require b[x][y]==0 **/
   b[x][y]=s; h[n]=x+3*y+9*s;  n = n+1
   /** ensure nochange **/
  }
}
```

Jass offers further features, we have not presented here, that are used for the translation from CSP-OZ or to improve the quality of the hand written code.

- Two keywords can be used in postconditions: The construct `changeonly(x,y)` specifies that only variables `x` and `y` are changed by the method. It corresponds to the $\Delta(x, y)$ expression used for Object-Z methods. The keyword `nochange` is equivalent to `changeonly()`.
  The object `old` is a copy of the state before the method invocation. Thus `old.x` refers to the old value of the variable `x`.
- The quantifiers `forall` and `exists` can be used for quantification over finite sets or any finitely enumerable object.
- Interference checks help to avoid any unwanted writes on global variables in the hand written code.
- Loop-invariants and variants and simple assertions placed anywhere in the code improve the quality of the handwritten code.
- A Javadoc service extends class-invariants, pre- and postconditions and trace assertion with html-tags and moves them into the right position such that they are used by Javadoc for the documentation generation.

[2] This feature is the only one described here, that's not yet implemented in the Jass compiler [Bar99].

4 Limitations

The idea proposed here cannot be used for all Java developments. Systems with dynamic communication structures do not fit within the static channels from CSP-OZ. If only a limited number of new communication links is created some tricks can be used to model these systems, but CSP-OZ is not well suited for this purpose. E. g. to design a site where many Tic-Tac-Toe players can meet and play is hard in CSP-OZ. Similar problems occur if an unbounded number of new objects can be created. But it is always possible to use CSP-OZ for designing a high level interface specification of the core game and refine it towards a distributed implementation and to wrap the result with Java code that organises the dynamic socket creation, for example. This part would not be covered by the formal specification.

Concerning the communication mechanism, CSP-OZ generated assertions only make sense with fully synchronised threads. Uncontrolled concurrent writing on global objects is hard to marry with the synchronous communication mechanism of CSP-OZ. However, unsynchronised threads are not used much in safety critical systems anyway.

Errors in the transformation of the architectural information from CSP-OZ cannot be recognised by the translation procedure at the moment. But such structural errors should occur early during basic testing.

5 Future work

The ideas presented here would fit nicely into a UML framework. Or to be more precise, a UML extension with an Object-Z like predicate language and a precise formal semantics could replace CSP-OZ as the starting formalism to generate Jass assertions from. This could help to establish a closer link between UML specifications and programms. Furthermore this might be a solution to overcome the limited ability of CSP-OZ to deal with dynamic communication structures.

The difficulties to put Formal Methods into practice are well known. A helpful strategy to bridge that gap is the education of students. If every computer science graduate knows something about applicable Formal Methods chances are better that they use them while developing industrial software. I believe that the work I presented here is not only usable but also nice to teach. Z and CSP are well known formal methods that are supported by very good books. They cover important aspects of formal methods. The combination of both can be easily motivated during a course. Mixing this with Java attracts students very much. Worked out material would help to spread such a course.

The intermediate language Jass had to be implemented as an extra tool. Design by contract could be supported much better if it is part of the official language specification. Therefor we try to persuade SUN to integrate Jass-like concepts into the official Java language.

References

[Bar99] D. Bartetzko. Parallelität und Vererbung beim 'Programmieren mit Vertrag'. Master's thesis, University of Oldenburg, May 1999. in German.

[BHL+96] J. Bowen, C. A. R. Hoare, H. Langmaack, E.-R. Olderog, and A. P. Ravn. A ProCoS II Project Final Report: ESPRIT Basic Research project 7071. *Bulletin of the EATCS*, 59:76–99, 1996.

[DRS95] R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[FH97] C. Fischer and S. Hallerstede. Data-Refinement in CSP-OZ. Technical Report TRCF-97-3, University of Oldenburg, June 1997.

[Fis97] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.

[Fis98] C. Fischer. How to combine Z with a process algebra. In J. Bowen, A. Fett, and M. Hinchey, editors, *ZUM'98 The Z Formal Specification Notation*, volume 1493 of *LNCS*, pages 5–23. Springer, 1998.

[FW99] C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In *Proceedings of Integrated Formal Methods (IFM)*, 1999. to appear.

[Jas99] Jass: Java with assertions, May 1999. `http://semantik.informatik.uni-oldenburg.de/~jass`.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. ISE, 2. edition, 1997.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

A Personal Background

Clemens Fischer holds a master in computer science and one in mathematics from the University of Oldenburg. Since 1995 he works in the group of E.-R. Olderog. He participated in the projects CoCoN (Provably Correct Communication Networks) with the Philips Research Laboratories Aachen, Germany, and UniForM (Universelle Entwicklungsumgebung für Formale Methoden) with the University of Bremen and the Elpro GmbH. During this time he developed the language CSP-OZ. The work presented here will be part of his PhD thesis.

The group of E.-R. Olderog has a strong background in formal methods for concurrent and real time systems with expertise in transformational design, graphical specification of real time requirements, specification and code generation of PLC programs and related real time applications, model checking and application of formal methods to telecommunication systems.

The work presented here is influenced by the language MIX developed in the ProCoS project (Provably Correct Systems) [BHL+96]. MIX provides a detailed set of transformation rules to develop OCCAM implementations from CSP-OZ like specifications in a pure transformational way.

A case study in class library verification: Java's vector class

(Abstract)

MARIEKE HUISMAN, BART JACOBS, JOACHIM VAN DEN BERG

Computing Science Institute, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
{marieke,bart,joachim}@cs.kun.nl

Abstract. This paper presents a verification of an invariant property for the `Vector` class from Java's standard library. The property says (essentially) that the actual size of a vector is less than or equal to its capacity. This property is maintained by all methods of the `Vector` class, and it holds for all objects created by the constructors of the `Vector` class. The verification relies on two tools: the proof tool PVS is used for reasoning, and the LOOP tool is used for an automatic translation of Java into PVS. This project shows the feasibility of tool-assisted verification for non-trivial Java classes.

1 Introduction

One of the reasons for the popularity of object-oriented programming is the possibility it offers for reuse of code. Usually, the distribution of an object-oriented programming language comes together with a collection of ready-to-use classes, in a class library. Typically, these classes contain general purpose code, which can be used in many applications. Before using such classes, a programmer usually wants to know how they behave and when their methods throw exceptions. One way to do this, is to study the actual code, but since this is time-consuming and requires understanding all particular ins and outs of the implementation, this is often not the most efficient way. Another approach is to study the documentation provided. As long as the documentation is clear and concise, this works well, but otherwise one still is forced to look at the actual code.

An alternative approach is to verify suitable properties of standard classes, and add these to the documentation. Examples of properties that can be verified are termination conditions (in which cases will a method terminate normally, in which cases will it throw an exception), pre-post-condition relations and class invariants. Once sufficiently many properties have been verified, one only has to understand these properties, and there is no need anymore to study the actual code, in order to be able to use the class safely.

This paper describes a case study verification of one particular library class, namely `Vector`, which is in the standard distribution of the programming language Java [AG97,GJS96]. The `Vector` class basically consists of an array of objects, which is internally replaced by an array of different size, according to needs[1]. The choice for the `Vector` class is in fact rather arbitrary: it serves our purposes well because it involves a non-trivial amount of code (including the code from its surrounding classes from the library), and gives rise to an interesting invariant. However, other classes than `Vector` could have been verified. The investment of formal verification for library classes can be justified, because these classes are used extremely often. The result of such a verification may be detection of errors, and also improvement of documentation. This particular verification effort did not bring forward errors in the `Vector` class—which would have been unlikely, given how often it is used. However, it pointed out several places where the documentation could be improved.

This verification project makes use of two tools: the PVS [ORR+96,ORSvH95] proof tool, developed at SRI, and the LOOP [JvdBH+98,HHJT98] translation tool. The latter is a compiler

[1] Arrays in Java have a fixed size; vectors are thus useful if it is not known in advance how many storage positions are needed.

which translates Java classes into logical theories in the higher-order logic of PVS. Development of this tool is part of the so-called LOOP project, for Logic of Object-Oriented Programming, in which all three the authors are involved. Initially, this project aimed at reasoning about class specifications (see [HHJT98]), but a new branch of this project concentrated on reasoning about Java [JvdBH+98].

The LOOP tool translates Java classes into appropriate definitions in the language of PVS, by computing a semantical value $[\![s]\!]$, for each (legal) Java expression s. It also generates auxiliary definitions and results. Of particular importance for this paper are invariance definitions, which are generated for each class. Actual verification benefits from auxiliary results, which can be used for automatic rewriting. The series of logical theories that is generated when the compiler is applied to a series of Java classes, can be loaded into PVS. After type checking, the user can state the properties (s)he would like to prove about these Java classes, and subsequently (try to) prove them, using the full power of PVS.

The underlying Java semantics that is used in the automatic translation is based on so-called coalgebras [JR97,Rei95,Jac96]. These are special functions, which are useful for describing state-based dynamical systems. In the theory of coalgebras there are standard notions of invariance and bisimulation. Java classes are translated into coalgebras, acting on a single (global) memory (type), consisting of an infinite series of cells for storing objects. The language constructs of Java, like `if-else`, `while`, `try-catch-finally`, are represented in PVS, in what we call a semantic prelude. It is standardly loaded in the theories of translated Java classes. More information about the underlying semantics of Java can be obtained from [HJ99].

Current work in the LOOP project is on optimising the translation from Java to PVS, on designing efficient proof rules, and on extending the compiler to generate theories which are understood by the proof tool Isabelle [Pau94].

The contribution of the work presented in this paper is two fold. First of all, it shows the feasibility of tool-assisted verification of (standard library) classes in Java. The verification results could be used to improve the class documentation and make it exact, in contrast to informal explanations in ordinary language. Secondly, it is an illustration of the use (and capabilities) of the LOOP translation tool [JvdBH+98]. Although the translation does not cover all of Java yet—threads are not incorporated at the moment—it already allows reasoning about real-life Java programs. This is the first time, such a large verification has been done within this project. An important point, worth making explicit, is that this verification is not about programs written is some clean, mathematically civilised, abstract programming language, but about actual Java programs with all their messy details. We consider it a challenge to be able to handle such details.

There are relatively few references on formal verification for object-oriented languages. Specific logics for reasoning about object-oriented programs are proposed in [Boe99,AL97,Lei98]. When it comes to Java, one can distinguish between (1) reasoning about Java as a language, and (2) reasoning about programs written in Java. In the first category there is work on, for example, safety of the type system [NvO98,Sym97], or bytecode verification [Pus99]. But the present paper falls in the second category. There is related work in [PHM99], but in its current state of development, this does not cover abrupt termination (caused, for instance by exceptions). Being able to reason also about abrupt termination (see also [HJ99]) is crucial for the verification in this paper.

This paper is organised as follows. Section 2 describes the interface of Java's `Vector` class and its surrounding classes in the Java library, and how these classes are translated. Then, Section 3 discusses the invariant. Due to space restrictions we cannot really go into the details of how it is established in PVS. Finally, Section 4 gives some conclusions and possibilities for future work.

2 Java's `Vector` class and its translation to PVS theories

Java's `Vector` class[2] is part of the `java.util` package. It can be found in the sources of the JDK distribution. The class as a whole is too big to describe here in detail. It contains three fields,

[2] We use version number 1.38, written by Lee Boynton and Jonathan Payne, under Sun Microsystems copyright.

```
public class Vector implements Cloneable, java.io.Serializable {
   // fields
   protected Object elementData[];
   protected int elementCount;
   protected int capacityIncrement;

   // constructors
   public Vector(int initialCapacity, int capacityIncrement);
   public Vector(int initialCapacity);
   public Vector();

   // methods
   public final synchronized void copyInto(Object anArray[]);
   public final synchronized void trimToSize();
   public final synchronized void ensureCapacity(int minCapacity);
   private void ensureCapacityHelper(int minCapacity);
   public final synchronized void setSize(int newSize);
   public final int capacity();
   public final int size();
   public final boolean isEmpty();
   public final synchronized Enumeration elements();
   public final boolean contains(Object elem);
   public final int indexOf(Object elem);
   public final synchronized int indexOf(Object elem, int index);
   public final int lastIndexOf(Object elem);
   public final synchronized int lastIndexOf(Object elem, int index);
   public final synchronized Object elementAt(int index);
   public final synchronized Object firstElement();
   public final synchronized Object lastElement();
   public final synchronized void setElementAt(Object obj, int index);
   public final synchronized void removeElementAt(int index);
   public final synchronized void insertElementAt(Object obj, int index);
   public final synchronized void addElement(Object obj);
   public final synchronized boolean removeElement(Object obj);
   public final synchronized void removeAllElements();
   public synchronized Object clone();
   public final synchronized String toString();
}
```

Fig. 1. The interface of Java's `Vector` class

three constructors, and twenty-five methods. Most of the method bodies consist of between five and ten lines of code. The fields in class `Vector` are: an array `elementData` of type `Object` in which the elements of the vector are stored, an integer `elementCount` which holds the number of elements in the vector, and an integer `capacityIncrement` which indicates the amount by which the vector will be incremented when its size (`elementCount`) becomes greater than its capacity (length of `elementData`). If `capacityIncrement` is greater than zero, every time the vector needs to grow the capacity of the vector will be incremented by this amount, otherwise the capacity will be doubled. These fields are all protected, so that they can only be accessed in (a subclass of) `Vector`.

Space restrictions prevent us from describing the constructors and methods of the `Vector` class in detail. Therefore, we refer to standard documentation [AG97] for more information, and we will only list the interface of the `Vector` class, see Figure 1. The names and types give an idea of what these methods are supposed to do.

The following Java classes are used in the `Vector` class, in one way or another: `Cloneable`, `ArrayIndexOutOfBoundsException`, `InternalError`, `CloneNotSupportedException`, `Integer`, `Object`, `StringBuffer`, `String`, `System` (all from the `java.lang` package) `Enumeration` and `NoSuchElementException` (both from the `java.util` package), and `Serializable` (from the `java.io` package). These additional classes are relevant for the verification, since they also have to be translated into PVS. They are intertwined via mutual recursion.

To keep the size of generated theories maintainable, in the surrounding classes only the methods that are actually needed, are translated by the LOOP tool. In this way, 10K of Java code, excluding documentation, remains to be translated. The LOOP tool turns it into about 750K of PVS code[3].

Java's `Object` and `System` classes have several native methods. A native method lets a programmer use some already existing (non-Java) code, by invoking it from within Java. In the `Vector` class two native methods are used, namely `clone` from `Object`, and `arraycopy` from `System`. We insert our own PVS code as translation of the method bodies of these native methods.

Mutually recursive classes do not present a problem for our Java semantics—although PVS does not have mutually recursive types. The reason is that objects are handled as references, and not as values. Thus, an occurrence of a class or interface type in Java is translated into a special type of references in PVS. The latter does not contain objects, but references to objects, given as natural numbers pointing to memory cells in which objects are stored[4].

The current version of our LOOP tool handles practically all[5] of "sequential" Java, *i.e.* Java without threads. But the possible use of vectors in a concurrent scenario is not relevant for the translation and verification of the `Vector` class. The `synchronized` keyword in the method declarations is simply ignored.

3 The invariant for class `Vector`

After translation of the `Vector` class (and all surrounding classes), the generated theories are loaded into PVS and the verification effort starts. As suggested by the documentation in the `Vector` class, a class invariant should be: the number of elements in the array of a vector object never exceeds its capacity. Let us call this property `VectorIntegrity?`. Our goal is to show that `VectorIntegrity?` is indeed an invariant.

The precise formulation of what it means to be an invariant for a particular Java class depends on the interface, *i.e.* on the types of the methods in the class. Briefly, an invariant is a predicate

[3] This may seem a formidable size multiplication, but it does not present problems in verification; it only means that typechecking takes a long time. Reductions in size may still be possible by making more efficient use of parametrisation in PVS code generation.

[4] More precisely: the type of references, as defined in PVS and used for the translation of Java reference types, consists of either the null reference, or a proper reference containing a location in memory, given as `objpos?`, a run-time type for the object stored in this location, given as `clname?`, and possibly a length (for array references), given as `len?`.

[5] It does not cover static initialisers, for example.

on a state space, which, once it holds in a state x, will continue to hold in successor states x' of x, obtained by method invocations. In Java, a method may either hang (*i.e.* not terminate at all), terminate normally, or terminate abruptly. In the first case, no successor state is produced, but in the second and third cases of normal and abrupt termination one does have a successor state. Abrupt termination in Java is caused by either an exception, a return, a break or a continue. Standard Java compilers (from the JDK) enforce that a method in a Java class can only terminate abruptly because of an exception; break, continue and return abnormalities are caught inside method bodies, resulting in a normal return state.

Thus, `VectorIntegrity?` is an invariant of class `Vector`, if for each method `m(A1 a1,...,An an)` of `Vector` one has: if `VectorIntegrity?` holds for a state x, then for all appropriately typed actual parameters `a1`, ..., `an`,

1. if running `m` with these parameters in state x terminates normally, resulting in a successor state x', then `VectorIntegrity?` holds in x';
2. if running `m` with these parameters in state x terminates abruptly (because of an exception) with successor state x', then `VectorIntegrity?` holds in x'.

The second requirement is fairly strong. Often a class invariant expresses certain integrity constraints on the instance variables of the class. When a method throws an exception, the second requirement demands that this should be done before any data is corrupted, so that the invariant still holds in the resulting (abnormal) state. This ensures that if an exception is eventually caught, the resulting (normal) state still satisfies the invariant.

We also show that `VectorIntegrity?` holds after invoking a constructor of the `Vector` class.

In fact, the `VectorIntegrity?` predicate does not simply consist of a formalisation of the statement: "the integer field `elementCount` is less than or equal to the length of the object array field `elementData`". It should also incorporate trivial, but essential properties like: `elementCount` is non-negative, and `elementData` is a non-null reference. Without such additional properties, it cannot be shown that `VectorIntegrity?` is maintained by all methods. We shall describe all ingredients in words, not in PVS language. Some of the points are closely related to the representation of references in our semantics of Java.

The predicate `VectorIntegrity?` consists of the following eight points, the last of which is most interesting.

1. The number of elements of a vector, stored in the integer field `elementCount`, is always positive;
2. The array field `elementData` in which the data elements are stored is a proper, non-null reference;
3. This reference contains a length field;
4. The elements of the array `elementData` are stored in allocated memory;
5. The elements are stored at positions that are different from the position of the array itself.
6. The array `elementData` is an array of `Object`'s;
7. For each element in the array, if it is a non-null reference, then its (run-time) class is a subclass of `Object`.
8. The number of elements of a vector, stored in `elementCount`, is less than the length of the array `elementData`.

Notice that this predicate `VectorIntegrity?` does not say anything about the value of the field `capacityIncrement`. One would expect it to be positive, but this is not needed, since the only time `capacityIncrement` is actually used (in the body of the method `ensureCapacityHelper`), it is first tested whether its value is greater than zero. The documentation for this field states that "if the capacity increment is 0, the capacity of the vector is doubled each time it needs to grow", but a more precise statement would be "if the capacity increment is 0 *or less*, ...".

Let us consider an example of how we show that `VectorIntegrity?` is preserved by all the methods in class `Vector`. The following fragment from the `Vector` class describes the method `copyInto` together with its documentation.

Method/constructor invocation	Terminates normally if
`Vector(initialCapacity, capacityIncrement)`	`initialCapacity` ≥ 0
`Vector(initialCapacity)`	`initialCapacity` ≥ 0
`Vector()`	always
`setSize(newSize)`	`newSize` ≥ 0
`contains(elem)`	`elementCount` > 0 implies `elem` is non-null
`indexOf(elem)`	`elementCount` > 0 implies `elem` is non-null
`indexOf(elem, index)`	`elem` is non-null and `index` ≥ 0, or `index` $\geq$ `elementCount`
`lastIndexOf(elem)`	`elementCount` > 0 implies `elem` is non-null
`lastIndexOf(elem,index)`	`elem` is non-null and `index` $<$ `elementData.length`, or `index` < 0
`elementAt(index)`	$0 \leq$ `index` $<$ `elementCount`
`firstElement()`	`elementCount` > 0
`lastElement()`	`elementCount` > 0
`setElementAt(obj,index)`	$0 \leq$ `index` $<$ `elementCount`
`removeElementAt(obj,index)`	$0 \leq$ `index` $<$ `elementCount`
`insertElementAt(obj,index)`	$0 \leq$ `index` $\leq$ `elementCount`
`removeElement(obj)`	`elementCount` > 0 implies `obj` is non-null

Fig. 2. An overview of the termination conditions for some methods in class `Vector`.

```
/**
 * Copies the components of this vector into the specified array.
 * The array must be big enough to hold all the objects in this vector.
 *
 * @param   anArray   the array into which the components get copied.
 * @since   JDK1.0
 */
public final synchronized void copyInto(Object anArray[]) {
    int i = elementCount;
    while (i-- > 0) {
        anArray[i] = elementData[i];
    }
}
```

This method will throw an exception in each of the following cases.

- The field `elementCount` is greater than zero, and the argument array `anArray` is a null reference;
- `elementCount` is greater than zero, `anArray` is a non-null reference, and its length is less than `elementCount`;
- `elementCount` is greater than zero, `anArray` is a non-null reference, its length is at least `elementCount`, and there is an index `i` below `elementCount` such that the (run-time) class of `elementData[i]` is not assignment compatible with the (run-time) class of `anArray`.

The first of these three cases produces a `NullPointerException`, the second case produces an `ArrayIndexOutOfBoundsException`, the third one an `ArrayStoreException`[6]. This last case is subtle, and is not documented at all; it can easily be overlooked. But in all three cases, no data is corrupted, and the predicate `VectorIntegrity?` still holds in the resulting (abnormal) state.

[6] See the explanation in [GJS96], Subsection 15.25.1, second paragraph on page 371. This exception occurs for example during execution of the following (compilable) code fragment.

```
Vector v = new Vector();
v.addElement(new Object());
v.copyInto(new Integer[1]);
```

Many `Vector` methods always terminate normally. These are: `trimToSize`, `ensureCapacity`, `ensureCapacityHelper`, `capacity`, `size`, `isEmpty`, `elements`, `addElement`, `removeAllElements`, `clone`, `toString`. The conditions for normal termination of the remaining methods (excluding `copyInto`, discussed above) and of all three constructors are summarised in Figure 2.

In all these cases of normal termination, the predicate `VectorIntegrity?` holds in the result state, if it is assumed to hold in the original state (in which the method was invoked). The same holds for the cases of abrupt termination (caused by an exception). Thus we conclude that `VectorIntegrity?` is indeed an invariant of Java's `Vector` class. In this verification we greatly benefited from the use of high-level Hoare logic proof rules, tailored for Java [HJ99].

The verification of the class invariant shows that the `Vector` class documentation often is inprecise or even incomplete. We think that it could greatly benefit from extending it with some more formal results, on the basis of verification. Naturally, these formal statements can never replace the informal documentation, but it can help to clearify and disambiguate it. These formal statements could be added at several places, giving rise to an 'annotated Java' language (similar to JML [LBC99]).

4 Conclusions

We have presented an overview of a verification in PVS of an invariant property of the `Vector` class from Java's standard library, as a case study in class library verification. The case study is part of a wider project for reasoning about Java programs [JvdBH+98]. It shows the feasibility of verification of classes in Java with the use of modern and powerful tools. Formal verification results can be used to improve class documentation, since it is exact, in contrast to informal explanations in ordinary language. Adding formal verification results to class documentation could result in an 'annotated Java'. A formalisation of such a language is a topic of further research. Other topics of further research are the further development of high-level proof rules, which ease the verification process, and of techniques for dealing with late binding in such a way that proofs can be re-used.

References

[AG97] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 2^{nd} edition, 1997.

[AL97] M. Abadi and K.R.M. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 682–696. Springer-Verlag, 1997.

[Boe99] F.S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Foundations of Software Science and Computation Structures*, number 1578 in LNCS, pages 135–149. Springer, Berlin, 1999.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[HHJT98] U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In *Proceedings of European Symposium on Programming (ESOP)*, volume 1381 of *LNCS*, pages 105–121. Springer-Verlag, March 1998.

[HJ99] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. Manuscript, 1999.

[Jac96] B. Jacobs. Objects and classes, co-algebraically. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Acad. Publ., 1996.

[JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[JvdBH+98] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 329–340. ACM Press, 1998.

[LBC99] G.T. Leavens, A.L. Baker, and C.Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06c, Iowa State University, Department of Computer Science, January 1999.

[Lei98] K.R.M. Leino. Data groups: specifying the modification of extended state. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 144–153. ACM Press, 1998.

[NvO98] T. Nipkow and D. von Oheimb. $Java_{light}$ is type-safe—definitely. In *Principles of Programming Languages (POPL)*, pages 161–170, 1998.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification (CAV '96)*, volume 1102 of *LNCS*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Pau94] L.C. Paulson. *Isabelle - a generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag, 1994. With contributions by Tobias Nipkow.

[PHM99] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S.D. Swierstra, editor, *Programming Languages and Systems*, LNCS, pages 162–176. Springer, Berlin, 1999.

[Pus99] C. Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W.R. Claeveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, number 1579 in LNCS, pages 89–103. Springer, Berlin, 1999.

[Rei95] H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.

[Sym97] D. Syme. Proving Java type soundness. Technical Report 427, University of Cambridge Computer Laboratory, 1997.

Threads and Main Memory Semantics

Vishnu Kotrajaras and Susan Eisenbach
Department of Computing, Imperial College

March 31, 1999

Abstract

In this paper we take the descriptions given in the Java Language Specification [1] about the relationship between threads and the main memory into a more formal notation. From the notation, we illustrate conditions that allow data inconsistency. We show data consistency holds in a single-threaded environment. We then investigate the differences between volatile declarations and synchronized statements. Volatile variables do not maintain data consistency while synchronization provides it. We also propose a synchronization structure which allows deterministic debugging and reasoning.

1 Introduction

The Java Language Specification [1] defines how a thread's working memory should interact with the main memory. It is crucial that programmers understand those rules in order both to predict correctly the behaviour of concurrent programs and to implement correct concurrent programs. Moreover, it is crucial for implementors to follow those definitions in order to implement a correctly behaved runtime system.

In this study, we examine descriptions given in chapter 17 of [1] and prove properties regarding low level working-main memory behaviour. Our main objective is to formalize those descriptions and to use the formalization to investigate data consistency among threads and the main memory. We formally introduce conditions which guarantees safe and efficient execution of concurrent programs.

Related work: Drossopoulou and Eisenbach [2, 3, 4] formulated an operational semantics of Java. However, their work did not include concurrent Java. Börger and Schulte [5] used the Abstract State Machine to describe the behaviour of Java threads in a pseudocode-like notation. Their thread-memory description is a good implementation that corresponds to the thread-memory rules. However, they did not formalize the thread-memory rules or study the rules' properties. Coscia and Reggio [6, 7, 8] developed their own version of the operational semantics for concurrent Java. Their work, like [5], focused mainly on execution that synchronizes on every variable access, therefore ignoring the working-main memory rules. [9, 10] transformed the thread-memory rules into logic sentences and went on to use the transformed rules to present the operational semantics for multi-threaded Java. However, the possible behaviour of Java programs was not studied.

Our notations and formal rules that we used for proofs are listed in appendix A.

2 Single-Threaded Environment

Definition 2.1 *Data consistency is when a thread uses the latest assigned value.*

Lemma 2.1 *For a single-threaded Java environment, data consistency is always guaranteed.*

The proof is illustrated in appendix B.

3 Multi-Threaded Environment

In a multi-threaded environment, we need to take the main memory into account.

Definition 3.1 *Data consistency in a multi-threaded environment is when:*

- *The latest assigned value will always be used.*
- *The latest assigned value will always cause the latest write to the main memory.*

Hence there are two possible cases of data inconsistency.

Case 3.1 *The use action uses an out of date value from the working memory.*

Within this case there are three execution sequences which allow data inconsistency.

Subcase 3.1 *A thread does not load a value before performing a use action.*

Subcase 3.2 *After the latest assign action by a thread to a variable, another thread loads an old value of that variable before the first thread writes to the main memory.*

Subcase 3.3 *After a thread reads a variable from the main memory but before it carries out the corresponding load action, another thread assigns to that variable. Since the load action gets a value from the read action, the assign action will be lost.*

Case 3.2 *The most recent* assign *action is not the last action that writes to the main memory. Some previous* assign *action is the last that updates the main memory.*

In order to achieve the correct value for using and updating, an execution sequence must obey the two axioms below:

Axiom 3.1 *If thread T_0 assigns to a variable V and thread T_1 later uses V, there is only one way T_1 is guaranteed to read the assigned value:*

1. *T_0 writes the updated value into the main memory.*

2. *After the write action T_0 finishes, T_1 then attempts to read V from the main memory before actually using it.*

Axiom 3.2 *If thread T_0 assigns to a variable V and thread T_1 assigns to V later on, data consistency is guaranteed only if the write action of T_1 happens after the write action of T_0.*

The formal notation of execution sequences in this section is illustrated in appendix C.

These correct execution sequences occur only by chance. However, the creators of Java provide two mechanisms that they claim to enable programmers to ensure data consistency:

- Synchronization
- Volatile declarations

4 Synchronization and Data Consistency

To prove that data consistency is always maintained in a multi-threaded environment, we need to show that the sequences that guarantee data consistency always occur:

- axiom 3.1 always holds when one thread assigns to a variable and another thread attempts to use that variable later.
- axiom 3.2 always holds when two or more threads attempt to assign to a variable.

Lemma 4.1 *Data consistency on a shared variable V is guaranteed when synchronization is performed on actions that access V.*

The proof is illustrated in appendix D.

5 Volatile Declaration and Data Consistency

We examine whether volatile declaration guarantees that data consistency is always maintained.

By attempting to prove that execution sequences that lead to data inconsistency can not possibly happen, we discovered, however, that volatile declaration does not guarantee data consistency.

Lemma 5.1 *Data consistency is guaranteed when a variable is declared volatile.*

This lemma is shown to be false. The actual proof is in appendix E. The fact that volatile *does not* provide data consistency as synchronization comes as a surprise to us. A volatile declaration of a variable is used to force a thread to reconcile the working copy of that variable with the master copy every time it accesses that variable. Therefore we naturally expect data consistency to be preserved, as seems to be the case in section 8.3.1.4 of [1]. However, our proof shows that the example in section 8.3.1.4 of [1] does not necessarily behave correctly.

6 Rules for Data Consistency

We need to show that synchronization ensures data consistency. In order to guarantee full data consistency, we add two new rules to the thread-memory rules.

Let T_0 and T_1 be two different threads, V be a shared variable which both threads have access to, L be a lock. We express possible concurrent execution sequences by using a || symbol:

The new rules will be as follows:

Axiom 6.1 *When two concurrent threads T_0 and T_1 access the same variable V, if a T_0 assigns to V and T_1 uses V, then data consistency is guaranteed only if the* assign *action of T_0 and the* use *action of T_1 are synchronized on the same lock.*

$$\text{Assign}(\text{T}_0, V) \parallel Use(T_1, V) \Longrightarrow$$

$$\begin{array}{cccccc} & Lock(T_0,O) & & Unlock(T_0,O) & Lock(T_1,O) & Unlock(T_1,O) \\ (& \overset{\downarrow}{\rightarrow} Assign(T_0,V) & \overset{\downarrow}{\rightarrow} &) \parallel (& \overset{\downarrow}{\rightarrow} Use(T_1,V) & \overset{\downarrow}{\rightarrow} \quad) \end{array} \quad (1)$$

Axiom 6.2 *When two concurrent threads T_0 and T_1 assigns to the same variable V, then data consistency is guaranteed only if the* assign *actions of T_0 and T_1 are synchronized on the same lock.*

$$\text{Assign}(\text{T}_0, V) \parallel Assign(T_1, V) \Longrightarrow$$

$$\begin{array}{cccccc} & Lock(T_0,O) & & Unlock(T_0,O) & Lock(T_1,O) & Unlock(T_1,O) \\ (& \overset{\downarrow}{\rightarrow} Assign(T_0,V) & \overset{\downarrow}{\rightarrow} &) \parallel (& \overset{\downarrow}{\rightarrow} Assign(T_1,V) & \overset{\downarrow}{\rightarrow} \quad) \end{array} \quad (2)$$

6.1 Deterministic Efficient Synchronization

Consider threads T0 and T1 with accesses on variable a:

Row	T0	T1
1	Use a	Assign a
2	Assign a	Use a
3	Use a	Assign a
4	Use a	Use a
5	Use a	Use a

In order to preserve data consistency, lock and unlock pairs have to be inserted:

Row	T0	T1
	Lock a	*Lock a*
1	Use a	Assign a
	Unlock a	*Unlock a*
	Lock a	*Lock a*
2	Assign a	Use a
	Unlock a	*Unlock a*
	Lock a	*Lock a*
3	Use a	Assign a
	Unlock a	*Unlock a*
	Lock a	*Lock a*
4	Use a	Use a
	Unlock a	*Unlock a*
	Lock a	*Lock a*
5	Use a	Use a
	Unlock a	*Unlock a*

Although data consistency is guaranteed, there are some disadvantages:

- Synchronization overhead surely costs performance. In the above case, the program can be slowed down dramatically because synchronization is performed on every access.
- A concurrent program that behaves this way by allowing a variable to be updated nondeterministically before use is very hard to debug and reason about.

When programming for a thread, in order to produce a program which is deterministic and simple to reason about, a *use* action should only read the value assigned earlier by the same thread, unless that thread is specifically waiting for a certain condition, which should be stated clearly in the program structure.

From the above example, the *use* actions of T_0 in row 3,4, and 5 can use different values nondeterministically due to T_1 updating the value, but a programmer will want all of them to use the same value assigned in row 2, unless T_0 calls a wait method.

In order to

- Reduce synchronization overhead.
- Create a program that can be debugged and reasoned about without too complicated mechanisms.
- Maintain data consistency of shared variables.

The synchronization rules in section 6 will need to be updated by:

Axiom 6.3 *A Deterministic and Efficient Synchronization (DES) by two or more threads on a shared variable V is carried out by:*

- *Having each thread synchronized on the same lock over an* assign *action (say, action "A") and the following* use *actions of V.*
- *If there are no prior* assign *action to V, synchronization should start before the first* use *action.*

- *Each* use *action under synchronization must at least take place before the* assign *action that comes after "A".*

To put it simply, the above definition puts *lock* and *unlock* pair around an *assign* and any following *use* actions. The locking range must at least cover the *use* action just before the next *assign* action. Note that the locking range can cover more than one *assign* action.

The above execution will become:

Row	T0	T1
	Lock a	*Lock a*
1	Use a	Assign a
	Unlock a	
	Lock a	
2	Assign a	Use a
		Unlock a
		Lock a
3	Use a	Assign a
4	Use a	Use a
5	Use a	Use a
	Unlock a	*Unlock a*

The formal DES rules are given in appendix F.

7 Conclusion and Future work

In this paper we formalized the semantics of threads and the main memory. We formally pointed out data inconsistency situations. Using the formalized semantics rules, we were able to prove that a single-threaded environment preserves data consistency, the use of volatile declaration does not preserve data consistency, unlike the use of synchronization locks. We also present formal rules that guarantee data consistency of a concurrent program and make the program easier for debugging and reasoning.

We hope what we found out about volatile declarations will be useful for Java programmers and creators. We intend to use the derived data consistency rules for our study in Java Semantics. The current project involves developing a system that can check data consistency of Java programs.

8 About the Authors

Susan Eisenbach is Director of Studies in the Department of Computing at Imperial College. She was principal investigator of the Multimedia Network Application (BECALM) project funded by the UK Engineering and Physical Science Research Council (EPSRC), where she worked on language design for multimedia applications in large-scale distributed systems. She is co-investigator of

the EPSRC Systems Engineering project SLURP investigating Java semantics. Susan Eisenbach was program chair of the OOPSLA'98 Workshop on Formal Underpinnings of Java Semantics.

Vishnu Kotrajaras obtained his Master in Engineering at Imperial College, London, in 1997. He is now a PhD student in the SLURP project in the Department of Computing at Imperial College.

References

[1] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, August 1996.

[2] Sophia Drossopoulou and Susan Eisenbach. Java is type safe -probably. In *11 th European Conference on Object-Oriented Programming*, February 1997. http://outoften.doc.ic.ac.uk/projects/slurp/papers.html.

[3] Sophia Drossopoulou and Susan Eisenbach. Is the java type system sound? In Fourth International Workshop on Foundations of Object-Oriented Languages, October 1997. http://outoften.doc.ic.ac.uk/projects/slurp/papers.html.

[4] Sophia Drossopoulou and Susan Eisenbach. Towards an operational semantics and proof of type soundness for java, April 1998. http://outoften.doc.ic.ac.uk/projects/slurp/papers.html.

[5] Egon Börger and Wolfram Schulte. A programmer friendly modular definition of the semantics of java. Universita di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy, boerger@di.unipi.it, Universität Ulm, Fakutät für Informatik, D-89069 Ulm, Germany, wolfram@informatik.uni-ulm.de.

[6] Eva Coscia and Gianna Reggio. A proposal for an abstract semantics of shared objects in multi-threaded java. Departimento di Informatica e Scienze dell' Informazione, Universita di Genova, Via Dodecaneso, 35 - Genova 16146 - Italy, http://www.disi.unige.it.

[7] Eva Coscia and Gianna Reggio. An operational semantics for java. Departimento di Informatica e Scienze dell' Informazione, Universita di Genova, Via Dodecaneso, 35 - Genova 16146 - Italy, http://www.disi.unige.it.

[8] Eva Coscia and Gianna Reggio. A proposal for a semantics of a subset of multi-threaded good java programs. Departimento di Informatica e Scienze dell' Informazione, Universita di Genova, Via Dodecaneso, 35 - Genova 16146 - Italy, http://www.disi.unige.it.

[9] Peitro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. From sequential to multi-threaded java: an event-based operational semantics. In *6 th Conf. Algebraic Methodology and Software Technology , AMAST*, 1997.

[10] Peitro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. An event-based structural operational semantics of multi-threaded java. Formal Syntax and Semantics of Java, Springer, 1998.

Checking Java programs via guarded commands

K. Rustan M. Leino, James B. Saxe, and Raymie Stata
Compaq Systems Research Center
130 Lytton Ave., Palo Alto, CA 94301, U.S.A.
{rustan,saxe,stata}@pa.dec.com
21 May 1999

Abstract

This paper defines a simple guarded-command–like language and its semantics. The language is used as an intermediate language in generating verification conditions for Java. The paper discusses why it is a good idea to generate verification conditions via an intermediate language, rather than directly.

0 Introduction

It is well-known that the later a software error is detected, the more expensive it is to correct. The Extended Static Checker for Java (ESC/Java) is a tool for finding, by static analysis, common programming errors normally not detected until run-time, if ever [3]. ESC/Java takes as input a Java program, possibly including user annotations, and produces as output a list of warnings of potential errors. It does so by deriving a *verification condition* for each routine (method or constructor), passing these verification conditions to an automatic theorem-prover, and post-processing the prover's output to produce warnings from failed proofs.

Deriving verification conditions for a practical language, rather than for a toy language, can be complex. Furthermore, in designing a tool for automatic checking, one faces trade-offs involving the frequency of spurious warnings, the frequency of missed errors, the efficiency of the tool, and the effort required to annotate programs. To explore and exploit these trade-offs flexibly, it must be easy to change the verification conditions generated by the tool.

To manage the complexity and achieve flexibility, we chose to derive verification conditions by first translating the source language into a simple intermediate *guarded-command* language, and then using the semantics of this guarded-command language to produce verification conditions. In this paper, we describe our intermediate language, give its semantics, and discuss how it is used in our tool.

1 Translation stages

Our translation from Java to verification conditions is broken into three stages. First, we translate from Java to a sugared form of our guarded-command language that includes high-level features such as iteration and method invocation. Second, we desugar the sugared guarded commands into primitive guarded commands. Finally, we compute verification conditions from these primitive guarded commands.

In the translation from Java into sugared guarded commands, we eliminate many of the complexities found in Java, such as `switch` statements and expressions with side effects. This part of the translation is bulky and tedious. We have designed the sugared guarded-command language to make the translation easy to understand and to implement. At the same time, this part of the translation is relatively stable. We find it nice to separate this bulky but stable part of the translation process from other parts that change during experimentation.

The desugaring into primitive guarded commands is where we need a lot of flexibility. This is the principal stage of the translation where we make the kinds of trade-offs mentioned in the introduction. In section 3, we give examples of how different desugarings, possibly chosen under user control, result in different kinds of checking.

The semantics of the primitive guarded-command language is quite simple. Indeed, a naive set of equations for deriving verification conditions from primitive guarded commands fills less than half a page. However, by experimenting with different, but semantically equivalent, equations, we have achieved significant performance gains. Because this stage of the translation begins with such a simple language, we have been able to perform these experiments easily.

2 Primitive guarded-command language

Our primitive guarded-command language is a form of Dijkstra's guarded commands [2], with several important distinguishing features: exceptions [0, 6], partial commands [8, 7], and going wrong (see, *e.g.*, section 6.2 of [4]). (By including partial commands, we no longer need the "guards" that originally gave the language its name.) The syntax of commands in our guarded-command language is as follows:

cmd ::=
  variable := *expr* | **skip** | **raise** | **assert** *expr* | **assume** *expr*
| **var** *variable*$^+$ **in** *cmd* **end** | *cmd* ; *cmd* | *cmd* ! *cmd* | *cmd* □ *cmd*

where an *expr* is an expression in untyped first-order predicate calculus extended with *labels*. A labeled expression (**label** L : e) is semantically equivalent to the expression

e, but supplies the label L to the theorem-prover in order to facilitate the production of user-sensible warning messages (see section 6 of [1], which describes an extended static checker for Modula-3).

We model Java instance fields as maps from objects to values. Thus, we translate the Java expression `o.f` into $select(f, o)$, where the *select* function extracts from map f the component indexed by o.

Unlike Dijkstra's guarded commands, our commands can terminate not only *normally*, but also *exceptionally* and *erroneously*. We use exceptional termination to model Java's exceptions and also Java's control-transfer statements `break`, `continue`, and `return`. We use erroneous termination ("going wrong") to model violations of the programming discipline that ESC/Java checks.

The semantics of our primitive guarded commands is given by their *weakest liberal preconditions*. For any command C and predicates (on the post-state of C) N, X, and W, the predicate $wlp.C.(N, X, W)$ holds in exactly those initial states from which each execution of C either terminates normally in a state satisfying N, terminates exceptionally in a state satisfying X, or terminates erroneously in a state satisfying W (or doesn't terminate at all, but all of our primitive commands do terminate). We define *wlp* by the following equations:

$$
\begin{aligned}
wlp.(v := e).(N, X, W) &\equiv N[v \leftarrow e] \\
wlp.\mathbf{skip}.(N, X, W) &\equiv N \\
wlp.\mathbf{raise}.(N, X, W) &\equiv X \\
wlp.(\mathbf{assert}\ e).(N, X, W) &\equiv (e \wedge N) \vee (\neg e \wedge W) \\
wlp.(\mathbf{assume}\ e).(N, X, W) &\equiv e \Rightarrow N \\
wlp.(\mathbf{var}\ v_1 \ldots v_n\ \mathbf{in}\ C\ \mathbf{end}).(N, X, W) &\equiv (\forall v_1 \ldots v_n :: wlp.C.(N, X, W)) \\
wlp.(C0\ ;\ C1).(N, X, W) &\equiv wlp.C0.(wlp.C1.(N, X, W), X, W) \\
wlp.(C0\ !\ C1).(N, X, W) &\equiv wlp.C0.(N, wlp.C1.(N, X, W), W) \\
wlp.(C0\ \square\ C1).(N, X, W) &\equiv wlp.C0.(N, X, W) \wedge wlp.C1.(N, X, W)
\end{aligned}
$$

where in the equation for the **var** command, $v_1 \ldots v_n$ are distinct variables not occurring free in N, X, or W.

The verification condition for a routine r has the form

$$BP \Rightarrow wlp.C.(true, true, false)$$

where C is the translation of r and BP is the *background predicate*. The background predicate is a set of axioms, derived in part from declarations in the user's program, that encode various properties guaranteed by Java, such as properties of the type system (see [5] for the background predicate of a simple object-oriented language).

3 Sugared guarded-command language

At the outset of our project, we considered it fairly obvious that trying to expand Java directly into verification conditions would result in a software engineering disaster. Introducing a desugaring stage was a less obvious design decision, but one that has turned out to be valuable in managing complexity and maximizing flexibility. In this section, we give examples of constructs in our sugared language.

Checks. To achieve a flexible treatment of conditions such a null dereferences, we use a command called **check**. For example, a Java statement `v = o.f;` on line 27 translates into the sugared commands

check *Null*, 27, o != *null* ;
v = *select*(f, o)

We have several choices in the desugaring of the **check** command. If we want treat null dereferences as errors, then we desugar the **check** command into

assert (**label** *Null*@27 : o != *null*)

ESC/Java lets users suppress null dereference warnings, either selectively or globally. If null dereference warnings are suppressed on line 27, then the **check** command desugars into

assume o != *null*

Introducing this assumption (instead of, say, desugaring the **check** command into **skip**) prevents ESC/Java from, for example, generating a warning on line 28 if that line contains the dereference `o.g`. (But there are other cases where we do desugar a **check** into **skip**.)

ESC/Java enforces a programming discipline in which null dereferences are considered to be errors. If we wanted to support a programming style in which the programmer might intentionally dereference null and then handle the resulting Java exception, we would desugar the **check** command into something like

(**assume** o == *null* ; ... ; **raise**) [] **assume** o != *null*

where the "..." elides the commands that make the subsequent **raise** model the raising of a new *NullPointerException*.

Loops. The translation of Java `while`, `do`, and `for` loops produces commands that contain a sugared command of the form

loop { **invariant** J } C **end**

In contrast to exiting the loop when C can no longer be executed [8], control exits this loop when C raises an exception. The usual way of defining *wlp* for loops involves a strongest fixed point. We approximate this fixed point by considering only executions that iterate at most once. That is, we desugar the **loop** command into

check *LoopInvInit*, *loc*, J ;
C ;
check *LoopInvMaintained*, *loc*, J ;
assume *false*

where *loc* is the source code location of the Java loop statement. While this approximation is coarse, we have found that it still allows the checker to find many program errors, even when J is the trivial invariant *true* (see section 9 of [1]). By translating Java loops into commands that contain **loop** commands, we retain the flexibility to try different desugarings. For example, we could unroll a **loop** two or more times. Or, we could produce a conservative desugaring of the form

check *LoopInvInit*, *loc*, J ; **assume** *false*
▯
... ; **assume** J ; C ; **check** *LoopInvMaintained*, *loc*, J ; **assume** *false*

where the "..." assigns arbitrary values to the assignment targets of the loop. Lastly, note that our translation retains the flexibility of strengthening any programmer-declared invariant with any kind of inferred invariants, for which the literature offers numerous techniques.

Calls. Our sugared language also contains a **call** command, whose desugaring depends on the specification of the routine being called. Roughly speaking, **call** $r(e0, e1)$, where routine r is allowed to modify x, desugars into a command of the form

var $p0\ p1$ **in**
  $p0 := e0$; $p1 := e1$; **check** ...*preconditions*... ;
  var x_0 **in** $x_0 := x$; **modify** x ; **assume** ...*postconditions*... **end** ; ...
end

where **modify** x is a sugared command that desugars into

var x' **in** $x := x'$ **end**

The actual desugaring of **call** is more complicated. For example, result values and exceptions must be treated, and postconditions include both user-declared conditions and conditions guaranteed by Java.

The **modify** command uses the nondeterminism inherent in the primitive **var** command. In the desugaring of **call** (and also elsewhere in our translation), we use **assume** commands to restrict that nondeterminism. (Our translation uses the nondeterminism only of the **var** command, never of the [] command. Whenever our translation generates a [] command, the enabling conditions of the subcommands are mutually exclusive.)

4 Conclusions

Generating verification conditions for a real-world language like Java is a significant engineering challenge. Such languages provide many programmer conveniences that make the derivation bulky and tedious. Also, finding the right derivation is as much an art as a science, an art involving much trial-and-error. Thus, it is important to appropriately separate concerns both to manage complexity and to maximize flexibility. In building the ESC/Java verification condition generator, we have applied this principle in decomposing the verification condition generation into a three-stage process that seems to have served us well.

History and acknowledgements. ESC/Java was built by Cormac Flanagan, Mark Lillibridge, Greg Nelson, and the authors. Greg Nelson first suggested verification condition generation via guarded commands, almost a decade ago. Subsequently, this became the basis for the ESC/Modula-3 verification condition generator, written initially by Damien Doligez and then mainly by Dave Detlefs.

References

[0] Flaviu Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, 10:163–174, 1984.

[1] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, December 1998. Available from `www.research.digital.com/SRC/publications/src-rr.html`.

[2] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[3] Extended Static Checking home page, Compaq Systems Research Center. On the Web at `www.research.digital.com/SRC/esc/Esc.html`.

[4] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, Pasadena, CA 91125, January 1995. Technical Report Caltech-CS-TR-95-03.

[5] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997. Proceedings available from `www.cs.williams.edu/~kim/FOOL/FOOL4.html`.

[6] M. S. Manasse and C. G. Nelson. Correct compilation of control structures. Technical report, AT&T Bell Laboratories, September 1984.

[7] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[8] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.

A Logic of Recursive Objects

Bernhard Reus

Ludwig-Maximilians-Universität, Oettingenstr. 67, 80538 München, Germany,
reus@informatik.uni-muenchen.de
Phone: +49 89 2178 2178
Fax: +49 89 2178 2175

Abstract. A denotational semantics of an untyped functional object calculus according to Abadi & Cardelli is endowed with a higher-order logic and some proof principles in order to obtain a logic of (recursive) objects in the spirit of the Logic of Computable Functions (LCF). Contrary to the work of Abadi & Leino the logic also allows for recursive objects and specifications. In this paper we focus on recursive interface specifications with invariants. This is extended to an imperative object calculus that is appropriate for a sublanguage of sequential Java. It provides us with a denotational basis for the different axiomatic calculi that have been suggested by several authors.

1 Introduction

Several functional and imperative object calculi have been proposed e.g. in [1]. Following the pioneering work of [10, 12], an axiomatic semantics for an imperative object calculus with global state is suggested in [2]. However, it is restricted to non-recursive objects and specifications. In [11, 17] different axiomatic calculi are presented, all based on operational semantics or in [16] even purely axiomatically. [7] suggests a wp-calculus with local state. Local state is also at the heart of the coalgebraic approach [9] where classes are specified in the "style of algebraic datatypes", but coalgebraically, using equational logic. Most of the above work deal with axiomatic semantics in terms of quite complex calculi, sometimes overloaded with huge "background predicates", and exclusively referring to *operational* semantics.

This paper is propagating a *denotational view on objects* and their properties. Just like the Logic of Computational Function [13] allows one to reason about the denotations of functional programs instead of their syntactic representation (sometimes also referred to as "Logic of Domains"), we suggest to take the same view on objects and object calculi. A similar philosophy underlies [8] where an equational logic on positive $F_{\leq}$ was used to reason about classes with self in a functional typed setting.

Once having embedded a denotational semantics of an object calculus or OO-programming language in a higher-order logic, one can reason about denotations of objects and prove their properties. This is more concise and conveys more intuition – at least for a semanticist or domain-theorist – than long and complicated definitions of Hoare-calcluli. Of course, it is highly diserable to have such calculi, but a denotational model might help developing them and to prove correctness/(relative) completeness w.r.t. to them.

2 Syntax

Let Var be the set of variables (typically $y \in \mathrm{Var}$) and Field and Meth the universe of field and method labels, respectively. Throughout the paper we assume that $I \subseteq \mathrm{Field}$

and $J \subseteq \text{Meth}$. The syntax of a simple functional object-calculus of Abadi & Cardelli [1] reads as follows:

$$o ::= y \mid [f_i = \varsigma(y)b_i \ ^{i\in I}, m_j = \varsigma(y)b_j \ ^{j\in J}] \mid o.f \mid o.m() \mid o.f := v \mid o.m \Leftarrow \varsigma(y)b$$

The f_i ($i \in I$) denote field, the m_j ($j \in J$) method labels. Whenever we do not want to distinguish we write m_k ($k \in I \cup J$). For fields $f_i = \varsigma(y)b_i$ it is assumed that b_i does not contain y otherwise it is a method. $o.m \Leftarrow \varsigma(y)b$ denotes method update. We write I_o and J_o for the fields and methods of o, respectively.

3 Denotational Semantics

We follow the self-application semantics proposed by Abadi & Cardelli [1][Chapter 8] and [3] giving, however, an appropriate domain equation. For the sake of simplicity (to avoid the numerous $\perp$s), domain equations are formulated in a category of predomains, i.e. domains (cpos) not necessarily having a least element. They can be solved in the corresponding category of lift-algebras over those predomains.

First consider the following definition of records:

$$\text{Rec}_{\mathcal{L}} Y \cong \Sigma_{L\subseteq\mathcal{L}} L \to Y$$

For a record r and label l belonging to (the first projection of) r let $r.l$ denote selection and $r[l \mapsto \ldots]$ record update (i.e. redefinition of r at l). The extension of a record r by a new label l is written $r \uplus l$. In the object calculus such an extension is not allowed, but it is needed later for describing states as records.

Assume that we have a type of basic values $\mathcal{V}$ which includes the booleans and the natural numbers.

$$\mathcal{O} \cong \mathcal{V} + (\text{Rec}_{\text{Meth}\cup\text{Field}}\mathcal{O} \rightharpoonup \mathcal{O})$$

$$[\![-]\!] : \text{Terms} \to \text{Env} \to \mathcal{O}$$

The type of environments is defined as $\text{Env} = \text{Var} \to \mathcal{O}$. In the semantics below we do not cover the error cases when a method or field is called for an element in $\mathcal{V}$.

$$[\![[f_i = \varsigma(y)b_i \ ^{i\in I}, m_j = \varsigma(y)b_j \ ^{j\in J}]]\!]_\rho = inr(I \cup J, m_k = \lambda z : \mathcal{O}.\, [\![b_k]\!]_{\rho[y\mapsto z]} \ ^{k\in I\cup J})$$

$$[\![o.m_j()]\!]_\rho = [\![o]\!]_\rho.m_j([\![o]\!]_\rho)$$

$$[\![o.f_i]\!]_\rho = [\![o]\!]_\rho.f_i([\![o]\!]_\rho)$$

$$[\![o.f_i := v]\!]_\rho = [\![o]\!]_\rho[f_i = \lambda z : \mathcal{O}.\, [\![v]\!]_{\rho[y\mapsto z]})$$

$$[\![o.m_j \Leftarrow \varsigma(y)b]\!]_\rho = [\![o]\!]_\rho[m_j = \lambda z : \mathcal{O}.\, [\![b]\!]_{\rho[y\mapsto z]})$$

If not otherwise stated we consider only the calculus *without* method update.

4 The Logic

We assume that the universe of domains is embedded in a higher-order logic with the solution of the domain equation for $\mathcal{O}$ as a datatype (can be e.g. easily established in LCF [13, 15]).

4.1 Interface Specifications

Interface specifications are e.g. treated in [2, 11, 16, 17]. Regarding the separation of method and field specifications we follow [2]. An interface specification for objects in the typed world consists of a number of field and method names, their signatures (i.e. types) and a specification of their behaviour. In our untyped setting for an object with fields f_i and methods m_j this reduces to

$$(B_k, T_k)_{k \in K \subseteq \text{Field} \cup \text{Meth}}$$

where

$$B_k \in \wp(\mathcal{O}) \to \wp(\mathcal{O}) \text{ and } T_k \subseteq \mathcal{O} \times \mathcal{O} \ .$$

The B_k are predicate transformers that given the "meaning of the interface specification" P yield the specification of the corresponding field m_k, if $k \in$ Field, and simply P, if $k \in$ Meth. The latter ensures that P is indeed an object invariant. Recursion comes into play because methods in the functional setting "return" the modified object (instead of changing it as in the imperative style) and moreover, as fields might contain objects for which the interface specification is desired to hold again[1].

The T_k are the input/output specifications of the method. Fields can be considered as "get"-methods with a fixed specification, i.e. $T_i(s, s') = (s' = s.f_i)$.

What is a possibly recursive interface specification given in terms of $(B_k, T_k)_{k \in K}$ and what does it mean for an object to fulfill it? Below we usually abbreviate $(B_k, T_k)_{k \in K}$ by $(\boldsymbol{B}, \boldsymbol{T})$ and for the index set $K \subseteq \text{Field} \cup \text{Meth}$ of $(\boldsymbol{B}, \boldsymbol{T})$ we write $K_{\text{B,T}}$.

First, we define an operator $\Phi[\boldsymbol{B}, \boldsymbol{T}] : \wp(\mathcal{O}) \to \wp(\mathcal{O})$, induced by an interface specification $(\boldsymbol{B}, \boldsymbol{T})$, as follows:

$$\Phi[\boldsymbol{B}, \boldsymbol{T}](P)(s) = \forall k \in K_{\text{B,T}}. \forall s' \in \mathcal{O}. T_k(s, s') \Rightarrow B_k(P)(s') \tag{1}$$

The semantics of the interface specification $(\boldsymbol{B}, \boldsymbol{T})$ is simply the greatest fixpoint of $\Phi[\boldsymbol{B}, \boldsymbol{T}]$, i.e.

$$\text{Spec}(\boldsymbol{B}, \boldsymbol{T}) = \nu P. \Phi[\boldsymbol{B}, \boldsymbol{T}](P)$$

The fixpoint exists as Φ is monotone.

An object o is said to fulfill the interface specification $\nu P. \Phi[\boldsymbol{B}, \boldsymbol{T}](P)$, written $o \models \nu P. \Phi[\boldsymbol{B}, \boldsymbol{T}](P)$, if, and only if, o is an element of the specification and $T_j(o.\, o.m_j())$ holds for any $j \in J_o$ (observe that $T_i(o, o.f_i)$ holds by definition for any $i \in I_o$). In other words

$$o \models \text{Spec}(\boldsymbol{B}, \boldsymbol{T}) \iff \text{Spec}(\boldsymbol{B}, \boldsymbol{T})(o) \wedge \forall j \in K_{\text{B,T}} \cap \text{Meth}. T_j(o, o.m_j())$$

Observe the separation of the method implementation part and the interface specification. As methods belong to objects (object based approach in contrast to the class based one) and the interface specification should be independent of any particular object, the T_j method specifications can be used to abstract away from method implementations. This is in accordance with the method specifications in [2, 16] and allows also for a completely axiomatic treatment without any denotational or operational model at all (cf. [16]).

[1] Note that in the functional setting there is no difference between the identity as a method and a field containing the object itself (aka a pointer to itself).

4.2 Reasoning Principles

In order to prove that an object fulfills an interface specification one needs an appropriate proof rule. It is suggested by the greatest fixpoint property of $\nu P.\Phi[\boldsymbol{B},\boldsymbol{T}](P)$:

$$\frac{o \in I \wedge I \subseteq \Phi[\boldsymbol{B},\boldsymbol{T}](I)}{o \in \nu P.\Phi[\boldsymbol{B},\boldsymbol{T}](P)}$$

This is generally called "principle of coinduction". It requires to establish an invariant I. Invariants are well-known from the verification of while-loops in the standard imperative paradigm. Unfolding the precondition we obtain:

$$I(o) \wedge \forall s \in \mathcal{O}.\, I(s) \Rightarrow \forall k \in K_{\mathrm{B,T}}.\, \forall s' \in \mathcal{O}.\, T_k(s,s') \Rightarrow B_k(I)(s')$$

which can be seen to be equivalent to

$$\begin{array}{l} I(o) \wedge \\ (\forall i \in \mathrm{Field} \cap K_{\mathrm{B,T}}.\, \forall s \in \mathcal{O}.\, I(s) \Rightarrow B_i(I)(s.f_i)) \wedge \\ (\forall j \in \mathrm{Meth} \cap K_{\mathrm{B,T}}.\, \forall s \in \mathcal{O}.\, I(s) \Rightarrow (\forall s'.\, T_j(s,s') \Rightarrow B_j(I)(s'))) \end{array}$$

4.3 Example

Consider the following object

$$o \equiv [\mathrm{p} = \varsigma(y)1, \mathrm{inc} = \varsigma(y)y.p := y.p + 1]$$

Note that the natural numbers are supposed to be contained in $\mathcal{V}$. Now consider the following interface specification

$$B_{\mathrm{p}}(P)(v) = (v > 0)$$
$$T_{\mathrm{inc}}(s,s') = (s'.p > s.p)$$

Note that $T_{\mathrm{p}}(s,s') = (s' = s.p)$ and $B_{\mathrm{inc}}(P) = P$ by default.

In order to prove $o \in \mathrm{Spec}(\boldsymbol{B},\boldsymbol{T})$ we have to single out an invariant I, here $I(s) = s.p > 0$. Obviously $I(o)$ holds so it remains to prove

$$\begin{array}{l} \forall s \in \mathcal{O}.\, I(s) \Rightarrow B_{\mathrm{p}}(I)(s.p) \wedge \\ \forall s \in \mathcal{O}.\, I(s) \Rightarrow B_{\mathrm{inc}}(I)(s) \wedge \\ \forall s \in \mathcal{O}.\, I(s) \Rightarrow (\forall s'.\, T_{\mathrm{p}}(s,s') \Rightarrow B_{\mathrm{p}}(I)(s')) \wedge \\ \forall s \in \mathcal{O}.\, I(s) \Rightarrow (\forall s'.\, T_{\mathrm{inc}}(s,s') \Rightarrow B_{\mathrm{inc}}(I)(s')) \end{array}$$

which equals

$$\begin{array}{l} \forall s \in \mathcal{O}.\, (s.p > 0) \Rightarrow s.p > 0 \wedge \\ \forall s \in \mathcal{O}.\, I(s) \Rightarrow I(s) \wedge \\ \forall s \in \mathcal{O}.\, (s.p > 0) \Rightarrow (\forall s'.\, (s' = s.p) \Rightarrow s' > 0) \wedge \\ \forall s \in \mathcal{O}.\, (s.p > 0) \Rightarrow (\forall s'.\, (s'.p > s.p) \Rightarrow (s'.p > 0)) \end{array}$$

wchich is obviously true. Also $T_{\mathrm{inc}}(o, o.\mathrm{inc}())$ holds trivially.

4.4 Partiality

Whether a method must terminate or not can be specified by requiring for a $T_j(s,s')$ that $s' \neq \bot$ or not. For cases where $T_j(s,\bot)$ yields true, condition (1) gives rise to the proof-obligation $B_j(P)(\bot)$. So, to obtain a notion of *partial* correctness one better defines B_j to contain $\bot$. In the same vein $T_j(\bot,\bot)$ should always hold for partial correctness.

4.5 Method Update

In case method update is allowed, one has to ensure that during an object's lifetime its methods m_j do, though subject to modifications, always fulfill T_j. This can be readily done by the following requirement:

$$\forall j \in K_{\mathrm{B,T}} \cap \mathrm{Meth}.\, \forall o' \in \mathcal{O}.\, (T_j(o, o') \Rightarrow \forall j \in K_{\mathrm{B,T}} \cap \mathrm{Meth}.\, T_j(o', o'.m_j()))$$

5 Imperative Object Calculus

In the imperative case fields have to be treated differently from methods as fields are evaluated by an eager strategy. Therefore method and field update will be essentially different, although the coding of fields and methods is done uniformly in the domain equation. Syntax is enriched by an operator $\mathrm{new}_{I,J}$ which creates a new object in the store with fields I and methods J. It is assumed, rather deliberately, that every field is initialized with a pointer to itself and any method is the identity. Object creation can then be derived from new, field update, and method update.

5.1 Denotational Semantics

$$R \cong \mathcal{V} + \mathrm{Loc}$$

$$St \cong \mathrm{Rec}_{\mathrm{Loc}}(\mathrm{Rec}_{\mathrm{Meth} \cup \mathrm{Field}} Cl)$$

$$Cl \cong (\mathrm{Loc} \times St) \rightharpoonup (R \times St)$$

Loc stands for locations, R for results, St for stores, and Cl for closures; $-_1$ and $-_2$ denote first and second projection of the cartesian product, respectively. Setting $\mathrm{Env} = \mathrm{Var} \to \mathrm{Loc}$ we get the following interpretation for terms of the imperative object-calculus:

$$[\![-]\!] : \mathrm{Terms} \to \mathrm{Env} \to St \to R \times St$$

$$\begin{aligned}
[\![x]\!]_\rho\sigma &= (\rho(x), \sigma) \\
[\![o.m_j()]\!]_\rho\sigma &= (([\![o]\!]_\rho\sigma)_2.([\![o]\!]_\rho\sigma)_1).m_j([\![o]\!]_\rho\sigma) \\
[\![o.f_i]\!]_\rho\sigma &= (([\![o]\!]_\rho\sigma)_2.([\![o]\!]_\rho\sigma)_1).f_i([\![o]\!]_\rho\sigma) \\
[\![o.f_i := e]\!]_\rho\sigma &= \mathrm{let}\ (v, \sigma') = ([\![e]\!]_\rho\sigma)\ \mathrm{in} \\
&\qquad (([\![o]\!]_\rho\sigma')_1, (([\![o]\!]_\rho\sigma')_2.([\![o]\!]_\rho\sigma')_1)[f_i \mapsto \lambda s : \mathrm{Loc} \times St.\, (v, s)]) \\
[\![o.m_j \Leftarrow \varsigma(y) b_j]\!]_\rho\sigma &= (([\![o]\!]_\rho\sigma)_1, (([\![o]\!]_\rho\sigma)_2.([\![o]\!]_\rho\sigma)_1)[m_j \mapsto \lambda t : \mathrm{Loc} \times St.\, [\![b_j]\!]_{\rho[y \mapsto t_1]} t_2) \\
[\![\mathrm{new}_{I,J}]\!]_\rho\sigma &= (l, (\sigma \uplus l)[l \mapsto (I \cup J, (m_k \mapsto \lambda s : \mathrm{Loc} \times St.\, (l, s))^{k \in I \cup J})])
\end{aligned}$$

5.2 Specifications

The field and method specifications in the imperative case have the following types: $B_i \in \wp(R \times St) \to \wp(R \times St)$ and $T_j \in \wp(St \times R \times St)$. The field invariants thus range over a result (the content of the field) and the underlying state, whereas the method specifications refer to input state, result, and output state. If $[\![o]\!] = (l, s)$ then what was $T_j(o, o.m_j())$ in the functional case becomes $T_j(s, ((s.l.m_j)(l, s))_1, ((s.l.m_j)(l, s))_2)$.

6 Expectations and Conclusions

The presented logic is planned to be formalised analogously to the implementation of LCF in a theorem prover [15]. In fact, a version of LCF itself might serve as a basis or, alternatively, a more constructive version of domain theory like [18]. The relations between operational, denotational and axiomatic semantics have to be established for object calculi and OO-programming languages (like Java) in the way that is meanwhile standard for functional languages or, at least, its archetypical representative PCF. It should be interesting to discuss with the authors of the different axiomatic settings, in which respect their calculi can be understood denotationally.

For Java this means also that a denotational semantics has to be proposed that fits e.g. with the ones suggested in [5, 6, 14]. By translating the classed based approach to the object based one and switching from the untyped to the typed world one should obtain a semantics and a logic for a kernel-sublanguage of Java in the style of this paper. This is ongoing work. Interesting problems occur when "inheriting invariants". It is also ongoing research how the suggested *Logic of Recursive Objects* looks like when built upon a denotational semantics using coalgebras. Thus, collaboration is desired also with the coalgebraic object semantics community.

Acknowledgement

Thanks to Thomas Streicher for fruitful discussions on this subject and his motivating interest in understanding object logics denotationally.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Springer Verlag, 1996.
2. M. Abadi and K.R.M. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *Theory and Practice of Software Development: Proceedings / TAPSOFT '97, 7th International Joint Conference CAAP/FASE*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer-Verlag, 1997.
3. Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *Principles of Programming Languages*, pages 396–409, 1996.
4. Jim Alves-Foss, editor. *Formal Syntax and Semantics of Java.* Lect. Notes Comp. Sci. Springer, Berlin, 199x. To appear.
5. Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. From Sequential to Multi-Threaded Java: An Event-Based Operational Semantics. In Michael Johnson, editor, *Proc. 6th Int. Conf. Algebraic Methodology and Software Technology*, volume 1349 of *Lect. Notes Comp. Sci.*, pages 75–90, Berlin, 1997. Springer.
6. Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. An Event-Based Structural Operational Semantics of Multi-Threaded Java. In Alves-Foss [4]. To appear.
7. F. S. de Boer. A wp-calculus for oo. In W. Thomas, editor, *Foundations of Software Science and Computations Structures*, volume 1578 of *Lecture Notes in Computer Science.* Springer-Verlag, 1999.
8. M. Hofmann and B. Pierce. Positive subtyping. *Information and Computation*, 126(1):186–197, 1996.
9. Bart Jacobs. Coalgebraic reasoning about classes in object-oriented languages. In *Special issue on the workshop Coalgebraic Methods in Computer Science (CMCS 1998)*, number 11 in Electr. Notes in Comp. Sci. Elsevier, 1998.
10. G.T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, 1991.

11. K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. Technical Report KRML 65-0, SRC, 1996.
12. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.
13. R. Milner. Implementation and application of Scott's logic of continuous functions. In *Conference on Proving Assertions About Programs*, pages 1–6. SIGPLAN 1, 1972.
14. Tobias Nipkow and David von Oheimb. Machine-checking the Java Specification: Proving Type-Saftey. In Alves-Foss [4]. To appear.
15. L.C. Paulson. *Logic and Computation*, volume 2 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1987.
16. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Technical report, Technical University of Munich, 1997. Habilitation Thesis.
17. A. Poetzsch-Heffter and P. Möller. A logic for the verification of object-oriented programs. In R. Berghammer and F. Simon, editors, *Programming Languages and Fundamentals of Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1997.
18. B. Reus. Formalizing synthetic domain theory – the basic definitions. *Journal of Automated Reasoning*, 199x. To appear in the special volume on Formal Proof.

Exception Analysis for Java

Kwangkeun Yi[1,*] and Byeong-Mo Chang[2,**]

[1] kwang@cs.kaist.ac.kr
ROPAS***
Dept. of Computer Science
Korea Advanced Institute of Science & Technology
[2] chang@cs.sookmyung.ac.kr
Dept. of Computer Science
Sookmyung Women's University

Abstract. Current JDK Java compiler relies too much on programmer's specification for checking against uncaught exceptions of the input program. It is not elaborate enough to remove programmer's unnecessary handlers (when programmer's specifications are too many) nor suggest to programmers for specialized handlings (when programmer's specifications are too general).
We propose a static analysis of Java programs that estimates their exception flows independently of the programmer's specifications. This analysis is an extension of a class analysis to Java's exception mechanism. Its cost-effectiveness balance is suggested by sparsely analyzing the program at method-level (hence reducing the number of unknowns in the flow equations).

1 Introduction

The current Java compiler relies on the programmer's specifications to check that the input program will have no uncaught exceptions at run-time. The programmers have to declare in a method definition any exception class whose exceptions may escape from its body.

The problem is that the current compiler is not elaborate enough to do "better" than as specified by the programmers. It cannot avoid programmer's unnecessary handlers nor suggest to programmers for specialized handlings. It is foreseeable for careless (or inconfident) programmers to excessively declare at every method that some exceptions can be uncaught. Then every use of the method have to be wrapped with handlers, whose installation at run-time would

* This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.
** This work is partly supported by KISTEP project "Research on Basic and Applied Technology for Information Systems Security" and STEPI project "Highspeed Computing 2G-12".
*** Research On Program Analysis System (http://ropas.kaist.ac.kr), National Creative Research Initiative Center.

P	::= C^*	program
C	::= `class` c `ext` c {`var` x^* M^*}	class definition
M	::= `method` $m(x)$ `=` e [`throws` c^*]	method definition
e	::= id	variable
	| id := e	assignment
	| `new` c	new object
	| `this`	self object
	| e ; e	sequence
	| `if` e `then` e `else` e	branch
	| `throw` e	exception raise
	| `try` e `catch` (c x e)	exception handle
	| $e.m(e)$	method call
id	::= x	method parameter
	| $id.x$	field variable
c		class name
m		method name
x		variable name

Fig. 1. Abstract Syntax of a Core of Java

be useless. Similarly, programmers can specify exceptions in too a broad sense. Programmers can declare that a method throws exceptions of the most general class `Exception` even if the actual exceptions are of much lower, specific classes. Then its handler cannot offer proper treatments specific to the exact classes of actual exceptions.

We propose a static analysis of Java programs that estimates their exception flows independently of the programmer's specifications. This analysis is an extension of a class analysis[1] to Java's exception mechanism. The class analysis estimates for each expression e_1 at method call $e_1.m(e_2)$ the classes S to which the method m belongs. The classes of uncaught exceptions from this call is then the classes of exceptions that can be raised *and* unhandled during the execution of $c.m$'s body for every class c in S.

2 Language

For presentation brevity we consider an imaginary core of Java with its exception constructs. Its abstract syntax is in Figure 1. A program is a sequence of class definitions. Class bodies consist of field variable declarations and method definitions. A method definition consists of the method name, its parameter, and its body expression. Every expression's result is an object. Assignment expression returns the object of its right hand side expression. Sequence expression returns the object of the last expression in the sequence. A method call returns

[1] You can consider our class analysis a simplified version of DeFouw et. al's analysis[3].

the object from the method body. The `try` expression

$$\texttt{try } e_0 \texttt{ catch } (c\ x\ e_1)$$

evaluates e_0 first. If the expression returns a normal object then this object is the result of the `try` expression. If an exception is raised from e_0 and its class is covered by c then the handler expression e_1 is evaluated with the exception object bound to x. If the raised exception is not covered by class c then the raised exception continues to propagate back along the evaluation chain until it meets another handler. Note that nested `try` expression can express multiple handlers for a single expression e_0. The exception object e_0 is raised by `throw` e_0.

We assume that (1) class inheritance is explicitly expanded. That is, subclass's body has all the inheritted parts from its super-classes. (2) For the self object's field varialbe x is always explicitly prefixed as `this.`x. Variable x without the prefix is only for method parameter or handler variable x in "`try` e `catch` $(c\ x\ e)$." (3) all variables are distinct.

3 Uncaught Exception Analysis

We present our exception analysis in the set-constraint framework[4]. Every expression e of the program has two set constraints: $\mathcal{X}_e \supseteq se, \mathcal{P}_e \supseteq se$. The $\mathcal{X}_e$ is for the object classes that the expression e's normal object belongs to. The $\mathcal{P}_e$ is for the exception classes that the expression e's uncaught exception belongs to. The meaning of a set constrainst $\mathcal{X} \supseteq se$ is intuitive: set $\mathcal{X}$ contains the set represented by set expression se. Multiple constraints are conjunctions. We write $\mathcal{C}$ for such conjunctive set of constraints. Collected constraints for a program guarantee the existence of its least solution (model) because every operator is monotonic (in terms of set-inclusion) and each constraint's left-hand-side is a single variable [4]. Our implementation computes the solution by the conventional iterative fixpoint method because our solution space is finite: exception classes in the program. Correctness proofs are done by the fixpoint induction over the continuous functions that are derived [1] from our constraint system.

In Section 3.1 we present a constraint system that analyzes uncaught exceptions from *every* expression of the input program. Because exception-related expressions are sparse in programs, generating constraints for every expression is wasteful. The analysis cost-accuracy balance need to be addressed by enlarging the analysis granularity. Hence in Section 3.2 we present a sparse constraint system that analyzes uncaught exceptions at a larger granularity than at every expression. Similar technique of enlarging constraint granularity has already been successfuly used in ML [5]'s exception analysis [6]. Our analysis result is the solution of this sparse constraints.

3.1 Exception Analysis at Expression-Level

Figure 2 has the rules to generate set constraints for the object classes of *every* expression. The subscript e of set variables $\mathcal{X}_e$ and $\mathcal{P}_e$ denotes the current ex-

pression to which the rule applies. The relation "$\triangleright\, e : \mathcal{C}$" is read "constraints $\mathcal{C}$ are generated from expression e."

Consider the rule for method call:

$$[\text{MethCall}]\ \frac{\triangleright\, e_1 : \mathcal{C}_1 \quad \triangleright\, e_2 : \mathcal{C}_2}{\begin{array}{r}\triangleright\, e_1.m(e_2) : \{\mathcal{X}_x \supseteq \mathcal{X}_{e_2}, \mathcal{X}_e \supseteq \mathcal{X}_{c.m} | c \in \mathcal{X}_{e_1}, \texttt{method}\ m(x)\ =\ e_m \in c\} \\ \cup \{\mathcal{P}_e \supseteq \mathcal{P}_{c.m} | c \in \mathcal{X}_{e_1}, \texttt{method}\ m(x)\ =\ e_m \in c\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

The call expression will have the objects returned from the method m. This method $m(x)\ =\ e_m$ is the one defined inside the classes $c \in \mathcal{X}_{e_1}$ of e_1's objects. Hence $\mathcal{X}_e \supseteq \mathcal{X}_{c.m}$, and similary, $\mathcal{P}_e \supseteq \mathcal{P}_{c.m}$ for uncaught exceptions. (The subscript $c.m$ indicates the index for the body expression of class c's method m.) The constraint $\mathcal{X}_x \supseteq \mathcal{X}_{e_2}$ is for parameter binding: object of e_2 is passed to the method parameter x.

Consider the rule for `throw` expression:

$$[\text{Throw}]\ \frac{\triangleright\, e_1 : \mathcal{C}_1}{\triangleright\, \texttt{throw}\, e_1 : \{\mathcal{P}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{P}_{e_1}\} \cup \mathcal{C}_1}$$

It throws exceptions e_1 or, prior to throwing, it can have uncaught exceptions from inside e_1 too.

Consider the rule for `try` expression:

$$[\text{Try}]\ \frac{\triangleright\, e_0 : \mathcal{C}_0 \quad \triangleright\, e_1 : \mathcal{C}_1}{\begin{array}{r}\triangleright\, \texttt{try}\, e_0\ \texttt{catch}\ (c_1\ x_1\ e_1) : \{\mathcal{X}_e \supseteq \mathcal{X}_{e_0} \cup \mathcal{X}_{e_1}, \mathcal{X}_{x_1} \supseteq \mathcal{P}_{e_0} \dot{\cap} \{c\}\} \\ \cup \{\mathcal{P}_e \supseteq (\mathcal{P}_{e_0} \dot{-} \{c_1\}) \cup \mathcal{P}_{e_1}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1\end{array}}$$

Normal objects are either from e_0 or from e_1 (after handling), hence $\mathcal{X}_e \supseteq \mathcal{X}_{e_0} \cup \mathcal{X}_{e_1}$. Raised exceptions from e_0 can be catched by x_1 only when their classes are covered by c_1. After this catching, exceptions can also be raised during the handling inside e_1. Hence, $\mathcal{P}_e \supseteq (\mathcal{P}_{e_0} \dot{-} \{c_1\}) \cup \mathcal{P}_{e_1}$.

The operators $\dot{-}$ and $\dot{\cap}$ are corresponding set-operations (set difference and intersection) modulo class hierarchy. We can easily catch the meaning by the following examples:

$$\{c\} \dot{-} \{c'\} = \begin{cases} \emptyset & \text{if } c = c' \text{ or } c \text{ is a subclass of } c' \\ \{c\} & \text{otherwise} \end{cases}$$

$$\{c\} \dot{\cap} \{c'\} = \begin{cases} \{c\} & \text{if } c = c' \text{ or } c \text{ is a subclass of } c' \\ \{c'\} & \text{if } c' \text{ is a subclass of } c \\ \{\} & \text{otherwise} \end{cases}$$

3.2 Exception Analysis at Method-Level

In our new, sparse constraint system, only four groups of set variables are considered: set variables for class' methods, field variables, try-expressions, and catch-variables. The number of unknowns is thus proportional only to the number of

methods, field variables, and `try` expressions, not to the total number of expressions. For each method f, set variable $\mathcal{X}_f$ is for classes (including exception classes) that are "available" at f, and $\mathcal{P}_f$ is for classes of uncaught exceptions during the call to f. Similarly $\mathcal{X}_x$ for each field variable x. Every catch-variable x of `try` expressions:

$$\texttt{try } e_g \texttt{ catch } (c\ x\ e)$$

has also a separate set variable $\mathcal{X}_x$ which will has the classes of uncaught exceptions from e_g. The try-expression e_g also has separate set variables $\mathcal{X}_g$ and $\mathcal{P}_g$, which are respectively for available and uncaught exception classes in e_g.

Figure 3 shows this new constraint system. The left-hand-side f in relation $f \rhd e : C$ indicates that the expression e is a sub-expression of method f (or try-expression f).

Consider the rule for `throw` expression:

$$[\text{Throw}]_m \ \frac{f \rhd e_1 : C_1}{f \rhd \texttt{throw } e_1 : \{\mathcal{P}_f \supseteq \mathcal{X}_f \cap \mathit{ExnClasses}\} \cup C_1}$$

The classes $\mathcal{P}_f$ of uncaught exceptions from method f are the exception classes ($\mathcal{X}_f \cap \mathit{ExnClasses}$) among the classes $\mathcal{X}_f$ available at f.

Consider the rule for `try` expression:

$$[\text{Try}]_m \ \frac{g \rhd e_g : C_g \quad f \rhd e_1 : C_1}{f \rhd \texttt{try } e_g \texttt{ catch } (c_1 x_1 e_1) : \{\mathcal{X}_{x_1} \supseteq \mathcal{P}_g \dot{\cap} \{c_1\}\} \cup C_g \cup C_1}$$

The exceptions bound to the handler variable x_1 are the uncaught exceptions $\mathcal{P}_g$ from e_g if the exception's classes are covered by c, hence $\mathcal{X}_{x_1} \supseteq \mathcal{P}_g \dot{\cap} \{c_1\}$. Note that the constraints C_1 from the try-expression e_g are derived under g.

The least model of the sparse constraints C, which are derived ($\rhd \mathit{pgm} : C$) from an input program *pgm* is our analysis result. The solutions for $\mathcal{P}_m$ has the exception classes whose exceptions might be thrown and uncaught during m's execution.

3.3 Typeful Constraints for Improved Accuracy

The method-level analysis' accuracy can be improved using types. For example, the constraint rule $[\text{MethCall}]_m$ for method call $e_1.m(e_2)$ can be sharpened using m's type: $c_1 \to c_2$:

$$\frac{f \rhd e_1 : C_1 \quad f \rhd e_2 : C_2}{f \rhd e_1.m(e_2) : \begin{array}{l}\{\mathcal{X}_f \supseteq \mathcal{X}_{c.m} \dot{\cap} \{c_2\}, \mathcal{X}_{c.m} \supseteq \mathcal{X}_f \dot{\cap} \{c_1\}, \mathcal{P}_f \supseteq \mathcal{P}_{c.m} | c \in X_f, c.m : c_1 \to c_2\} \\ \cup C_1 \cup C_2\end{array}}$$

If f's body has a method call $e_1.m(e_2)$, the classes $\mathcal{X}_f$ of available objects in f include the classes $\mathcal{X}_{c.m}$ of the objects available at the called method $c.m$ only if the classes are covered by m's return type. Similarly, method $c.m$ receives objects from current method f via the parameter passing only if their classes are covered by m's parameter type.

4 Discussion

Though we cannot claim the cost-effectiveness of our sparse analysis (Section 3.2) until its implementation [2] is tested with realistic Java programs, our experience of similarly developing a sister analysis [6, 7] for ML programs makes us positive about our design decision. Because exceptions are sparse objects in Java programs our gross estimation at the method-level (e.g., the $[\text{Throw}]_m$ and the $[\text{MethCall}]_m$ rules in Figure 3) would rarely be exposed in the analysis accuracy. First, an exception to raise is usually constructed (by `new` expression) inside the method that raises it. Second, method to call is usually explicit at the call-site. Even for dynamic-binding cases the situation that methods are exception-raising and also overridden in a class hierarchy may not be frequent.

References

1. Patrick Cousot and Radhia Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In *Lecture Notes in Computer Science*, volume 939, pages 293–308. Springer-Verlag, proceedings of the 7th international conference on computer-aided verification edition, 1995.
2. J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. *OOPSLA'96 Conference Proceedings.*
3. G. DeFouw, D. Grove, and C. Chambers, Fast interprocedural class analysis, *Proceedings of 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* pages 222-236, Januaray 1998.
4. N. Heintze, Set-based program analysis. Ph.D thesis, Carnegie Mellon University, October 1992.
5. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.
6. Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Lecture Notes in Computer Science*, volume 1302, pages 98–113. Springer-Verlag, proceedings of the 4th international static analysis symposium edition, 1997.
7. Kwangkeun Yi and Sukyoung Ryu. SML/NJ Exception Analysis version 0.98. `http://compiler.kaist.ac.kr/pub/exna/`, December 1998.

[2] We are currently implementing our analysis inside the Vortex compiler [2].

[New] $$\triangleright \texttt{new}\, c : \{\mathcal{X}_e \supseteq \{c\}\}$$

[This] $$\frac{c \text{ is the enclosing class}}{\triangleright \texttt{this} : \{\mathcal{X}_e \supseteq \{c\}\}}$$

[FieldAss] $$\frac{\triangleright e_1 : \mathcal{C}_1}{\triangleright id.x := e_1 : \{\mathcal{X}_{c.x} \supseteq \mathcal{X}_{e_1} | c \in \mathcal{X}_{id}\} \cup \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1}, \mathcal{P}_e \supseteq \mathcal{P}_{e_1}\} \cup \mathcal{C}_1}$$

[ParamAss] $$\frac{\triangleright e_1 : \mathcal{C}_1}{\triangleright x := e_1 : \{\mathcal{X}_x \supseteq \mathcal{X}_{e_1}, \mathcal{X}_e \supseteq \mathcal{X}_{e_1}, \mathcal{P}_e \supseteq \mathcal{P}_{e_1}\} \cup \mathcal{C}_1}$$

[Seq] $$\frac{\triangleright e_1 : \mathcal{C}_1 \quad \triangleright e_2 : \mathcal{C}_2}{\triangleright e_1 ; e_2 : \{\mathcal{X}_e \supseteq \mathcal{X}_{e_2}, \mathcal{P}_e \supseteq \mathcal{P}_{e_1} \cup \mathcal{P}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[Cond] $$\frac{\triangleright e_0 : \mathcal{C}_0 \quad \triangleright e_1 : \mathcal{C}_1 \quad \triangleright e_2 : \mathcal{C}_2}{\triangleright \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}, \mathcal{P}_e \supseteq \mathcal{P}_{e_0} \cup \mathcal{P}_{e_1} \cup \mathcal{P}_{e_2}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[FieldVar] $$\frac{\triangleright id : \mathcal{C}_{id}}{\triangleright id.x : \{\mathcal{X}_e \supseteq \mathcal{X}_{c.x} | c \in \mathcal{X}_{id}\} \cup \mathcal{C}_{id}}$$

[Param] $$\triangleright x : \emptyset$$

[Throw] $$\frac{\triangleright e_1 : \mathcal{C}_1}{\triangleright \texttt{throw}\, e_1 : \{\mathcal{P}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{P}_{e_1}\} \cup \mathcal{C}_1}$$

[Try] $$\frac{\triangleright e_0 : \mathcal{C}_0 \quad \triangleright e_1 : \mathcal{C}_1}{\begin{array}{l}\triangleright \texttt{try}\, e_0\, \texttt{catch}\ (c_1\ x_1\ e_1) : \{\mathcal{X}_e \supseteq \mathcal{X}_{e_0} \cup \mathcal{X}_{e_1}, \mathcal{X}_{x_1} \supseteq \mathcal{P}_{e_0} \dot{\cap} \{c\}\} \\ \qquad\qquad \cup \{\mathcal{P}_e \supseteq (\mathcal{P}_{e_0} \dot{-} \{c_1\}) \cup \mathcal{P}_{e_1}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1\end{array}}$$

[MethCall] $$\frac{\triangleright e_1 : \mathcal{C}_1 \quad \triangleright e_2 : \mathcal{C}_2}{\begin{array}{l}\triangleright e_1.m(e_2) : \{\mathcal{X}_x \supseteq \mathcal{X}_{e_2}, \mathcal{X}_e \supseteq \mathcal{X}_{c.m} | c \in \mathcal{X}_{e_1}, \texttt{method}\ m(x) = e_m \in c\} \\ \qquad\qquad \cup \{\mathcal{P}_e \supseteq \mathcal{P}_{c.m} | c \in \mathcal{X}_{e_1}, \texttt{method}\ m(x) = e_m \in c\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

[MethDef] $$\frac{\triangleright e_m : \mathcal{C}}{\triangleright \texttt{method}\ m(x) = e_m : \{\mathcal{X}_{c.m} \supseteq \mathcal{X}_{e_m},\ \mathcal{P}_{c.m} \supseteq \mathcal{P}_{e_m}\} \cup \mathcal{C}}$$

[ClassDef] $$\frac{\triangleright m_i : \mathcal{C}_i\ i = 1, \cdots, n}{\triangleright \texttt{class}\ c = \{\texttt{var}\ x_1, \cdots, x_k, m_1, \cdots, m_n\} : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

[Program] $$\frac{\triangleright C_i : \mathcal{C}_i\ i = 1, \cdots, n}{\triangleright C_1, \cdots, C_n : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

Fig. 2. Exception Analysis at Expression-Level

$[\text{New}]_m \quad f \rhd \texttt{new } c : \{\mathcal{X}_f \supseteq \{c\}\}$

$[\text{This}]_m \quad \dfrac{\text{c is the enclosing class}}{f \rhd \texttt{this} : \{\mathcal{X}_f \supseteq \{c\}\}}$

$[\text{FieldAss}]_m \quad \dfrac{f \rhd e_1 : \mathcal{C}_1}{f \rhd id.x := e_1 : \{\mathcal{X}_{c.x} \supseteq \mathcal{X}_f | c \in \mathcal{X}_f\} \cup \mathcal{C}_1}$

$[\text{ParamAss}]_m \quad \dfrac{f \rhd e_1 : \mathcal{C}_1}{f \rhd x := e_1 : \mathcal{C}_1}$

$[\text{Seq}]_m \quad \dfrac{f \rhd e_1 : \mathcal{C}_1 \quad f \rhd e_2 : \mathcal{C}_2}{f \rhd e_1 ; e_2 : \mathcal{C}_1 \cup \mathcal{C}_2}$

$[\text{Cond}]_m \quad \dfrac{f \rhd e_0 : \mathcal{C}_0 \quad f \rhd e_1 : \mathcal{C}_1 \quad f \rhd e_2 : \mathcal{C}_2}{f \rhd \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 : \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2}$

$[\text{FieldVar}]_m \quad \dfrac{f \rhd id : \mathcal{C}_1}{f \rhd id.x : \{\mathcal{X}_f \supseteq \mathcal{X}_{c.x} | c \in \mathcal{X}_f\} \cup \mathcal{C}_1}$

$[\text{Param}]_m \quad f \rhd x : \emptyset$

$[\text{Throw}]_m \quad \dfrac{f \rhd e_1 : \mathcal{C}_1}{f \rhd \texttt{throw } e_1 : \{\mathcal{P}_f \supseteq \mathcal{X}_f \cap \mathit{ExnClasses}\} \cup \mathcal{C}_1}$

$[\text{Try}]_m \quad \dfrac{g \rhd e_g : \mathcal{C}_g \quad f \rhd e_1 : \mathcal{C}_1}{f \rhd \texttt{try } e_g \texttt{ catch } (c_1 x_1 e_1) : \{\mathcal{X}_{x_1} \supseteq \mathcal{P}_g \dot{\cap} \{c_1\}\} \cup \mathcal{C}_g \cup \mathcal{C}_1}$

$[\text{MethCall}]_m \quad \dfrac{f \rhd e_1 : \mathcal{C}_1 \quad f \rhd e_2 : \mathcal{C}_2}{f \rhd e_1.m(e_2) : \{\mathcal{X}_f \supseteq \mathcal{X}_{c.m}, \mathcal{P}_f \supseteq \mathcal{P}_{c.m}, \mathcal{X}_{c.m} \supseteq \mathcal{X}_f | c \in X_f\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$

$[\text{MethDef}]_m \quad \dfrac{m \rhd e_m : \mathcal{C}_m}{\rhd \texttt{method } m(x) = e_m : \mathcal{C}_m}$

$[\text{ClassDef}]_m \quad \dfrac{\rhd M_i : \mathcal{C}_i \quad i = 1, \cdots, m}{\rhd \texttt{class } c = \{\texttt{var } x_1 \cdots x_n \; M_1 \cdots M_m\} : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_m}$

$[\text{Program}]_m \quad \dfrac{\rhd C_i : \mathcal{C}_i \quad i = 1, \cdots, n}{\rhd C_1 \cdots C_n : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$

Fig. 3. Exception Analysis at Method-Level