Class Refinement for Sequential Java

Ana Cavalcanti[1] and David A. Naumann[2]

[1] Centro de Informática
Universidade Federal de Pernambuco (UFPE), Box 7851 50740-50 Recife PE Brazil
alcc@cin.ufpe.br www.cin.ufpe.br/~alcc
[2] Department of Computer Science
Stevens Institute of Technology, Hoboken NJ 07030 USA
naumann@cs.stevens-tech.edu www.cs.stevens-tech.edu/~naumann

Keywords: class refinement, modular specification and verification, inheritance and dynamic binding, refinement calculi, semantics

1 Introduction

This extended abstract describes progress in an ongoing project on refinement calculus for sequential Java. Predicate transformer semantics is being used to validate correctness-preserving transformations for use in program development, verification, design refactoring, and compilation. We focus here on the semantics and its application in showing soundness of forward simulation for class refinement, the foundation of behavioral subclassing.

This section is an overview of project objectives and recent progress. Section 2 addresses the language and its semantics. Section 3 discusses class refinement, Section 4 presents our ideas for future work.

Our work is being done in the context of a collaboration involving others at UFPE (P. Borba and A. Sampaio) and Birmingham (U. Reddy), and our research assistants.[1] Our long-term goal is development of tools and methods for specification, construction, modular verification, restructuring, and compilation of Java programs. Current work uses an idealized language ROOL based on the sequential part of Java.

Refinement calculus is the unifying framework for the work. In refinement calculi, the specification statement $x : [pre, post]$ is treated as an "imaginary command". For commands c and c', the algorithmic refinement $c \sqsubseteq c'$ means that c' satisfies any specification that c does. Ordinary correctness is expressed using specification statements: we have that $x : [\phi, \psi] \sqsubseteq c$ holds just if c meets the specification "modifies x, requires ϕ, ensures ψ". Refinement laws formalize development by stepwise refinement from specifications [Mor94].

One of our objectives is to extend this method to encompass object-oriented programs, and in particular design patterns and refactoring transformations [Fow99], including those that involve several classes at once. In a case study applying our results, we restructure an object-oriented application to follow a

[1] The work is funded by National Science Foundation under Grant No. 9813854, and by CNPq under grants 520763/98-0 and 680032/99-1.

layered architecture [VB99]. Borba and his students are developing a tool to automate the application of design transformations.

Specification statements, including the special cases known as assertions and assumptions, provide flexible annotation of program fragments. This is useful not only for verification but also for static checking [DLNS98] and program transformation. Sampaio, Cavalcanti, and their students are developing a compiler based on the normal-form approach [HHS93,Sam97], which exploits specification statements in transformation of code fragments. In the past year, a normal form has been devised for a virtual machine based on JVM. Compilation is based on normal-form laws, and some of these have been proved using our semantics.

A major objective is to derive design and compilation laws from basic laws proved sound in predicate transformer semantics [BS00]. Weakest precondition semantics is of direct use in verification tools and it is well suited to proving refinement laws. Eventually we plan to prove soundness of this semantics with respect to an operational semantics, hopefully one already developed by another research group.

To this end, and in order to check proofs of laws and of results discussed in the sequel, we are using PVS to encode the typing system and semantics of our language. The encoding is purely definitional. We are using a deep embedding of program expressions including predicate in specification statements. In accord with the new semantics described in Section 2, commands act on state sets in PVS so this part is a shallow embedding.

Many design laws involve data refinement, for which we use an intrinsic definition [HHS86,dRE98], and behavioral subclassing, which is similar to a data refinement of coexisting classes. The primary means for establishing data refinement and behavioral subclassing is (forward) simulation. The existing literature falls short of the simulation results we need. Our new results on soundness and preservation of simulation are the main topic in the sequel.

2 Syntax and semantics

A program in our language is a sequence *cds* of Java-like class declarations followed by a main program c whose free variables may include objects of classes in *cds*. Attributes can be private, protected, or public, like in Java, and they can be mutually recursive. Methods are regarded as public. Mutual recursion between methods is not allowed, to simplify the semantics of method calls and the proof of laws. Methods are defined as parameterized commands [Bac87,CSW99] using call by value, result, and value-result (with copy semantics).

In [CN99,CN00a] we defined a weakest precondition semantics for ROOL. In that work, we regarded a predicate transformer as a function on formulae. We extended traditional weakest precondition semantics and gave an account of method calls that is both abstract and operationally intuitive. This semantics is appropriate for the proof of refinement laws, a work that is well under way [BS00].

For the proof of the soundness of simulation for data refinement, however, we find the syntactic approach to predicates to be a problem. In this context, it

is natural to consider the coupling invariant (or rather, a simulation relation) to be a formula relating the private attributes of the abstract and concrete classes. The proof of soundness of simulation requires a comparison of programs that differ only by the fact that the concrete class is substituted for the abstract one. We cannot, however, say that the semantics of the fixed client classes is equal in both programs. Since their semantics depends on the semantics of methods defined in the simulated classes, the proper relation between them is that of a simulation as well. To define this simulation, we need what we call a generalized coupling invariant to relate states of client classes.

We find it difficult to give a definition for this generalized invariant syntactically, but on the other hand, its definition as a relation on states is very intuitive and straightforward [CN00b]. Also, a data refinement proof technique should involve the definition of the coupling invariant by the developer, but not the definition of the generalized coupling invariant. So there is not really a justification to have it as a formula. For this reason, we have given a new semantics to our language where predicates are regarded as sets of states, and predicate transformers as functions on these sets.

The definitions in this new semantics are very similar to those of our previous work. We use type-theoretic techniques to organize the semantic definitions. If a command c can occur in the methods of a class N, we use a typing judgement $\Gamma, \Sigma, N \rhd c$. The typing environment Γ records the classes in context, including N, and the signature Σ includes the variables in scope for c: attributes of N, parameters, and local variables. The typing rules reflect Java's restrictions on scope and subsumption. The semantics is defined by induction on typing derivations.

As expected, the challenge was the definition of the semantics of method calls. As before, we have an environment that records the semantics of methods and that is defined by a fixpoint construction. The semantics recorded is that of the behaviour of the method when called from inside the class where it is available. We use this semantics directly to define the meaning of calls $\mathbf{self}.m(e)$. For a call of the form $x.m(e)$, it must be adapted.

At the point where the call $x.m(e)$ occurs, the state space includes x as well as attributes of the calling object, parameters of the calling method, and locals of the calling method. In a state where the dynamic type of x is N', the environment η gives a meaning $\eta N' m$ for the called method, but that meaning acts on the state space consisting of attributes of N' and parameters of m. So we have to adjust the postcondition at the point of call so that $\eta N' m$ is applicable. Roughly, this adjustment extracts the attributes of x to get a state of the right kind and ensures that state variables other than x are unchanged. The definition for a pre-state σ and a postcondition ψ is as follows.

$$\sigma \in [\![\Gamma, \Sigma, N \rhd x.m(e) : \mathbf{com}]\!]\eta\ \psi \Leftrightarrow \{x\} \cup \mathit{rvrargs} \lhd \sigma \in \mathit{pt}\ (\mathit{adapt}\ \sigma\ \psi)$$

The environment η provides the transformer pt determined as $pt = \eta\ N'\ m\ \mathit{arglist}$, where N' is the class of x defined by σ, and *arglist* is the list of arguments resulting from evaluating the expressions e in σ. The predicate transformer pt is for a local signature that contains only the attributes of N' and the parameters. On

the other hand, the predicate ψ is on Γ, Σ, N, where Σ is the state space of the caller. As already said, we need to reconcile these differences before applying pt. This is the role of the function *adapt*. The method call can only affect the value of x and of the result and value-result arguments *rvrargs*. We require that the state resulting from the domain restriction ($\lhd$) of σ to x and *rvrargs* satisfies the precondition. The function *adapt* considers the conjunction of ψ with the predicate that requires that the value of all variables, except x and those in *rvrargs*, are the same as in σ. Moreover, it transforms the resulting predicate into another one on the attributes of N' and on the result and value-result parameters, by extracting the attributes of N' (or one of its subclasses) from $\sigma\ x$ and the value of the parameters from the arguments. This new semantics combines elements from [CN00a] and [Nau00].

3 Class Refinement

Algorithmic refinement of programs and commands is defined in the usual way as the pointwise order on predicate transformers. In [CN00a], we define two relations of class refinement. Here, we are focusing on the relation $cds \rhd cda \preccurlyeq_{=} cdc$ that captures the situation in which the abstract class *cda* is data refined by the concrete class *cdc* in the context of the sequence of class declarations *cds*. They both introduce the same class *Ns* with the same superclass.

Definition 1 (Class Refinement). *For a sequence of class declarations cds, and class declarations cda, and cdc, that introduce a class called Ns, for instance, we define* $cds \rhd cda \preccurlyeq_{=} cdc$ *if and only if*

- *the sequences of class declarations cds cda and cds cdc are both well-formed;*
- *for all commands c that use only methods in cds and cda and whose global components have types that are Ns-free, if c is a well-typed main program for cds cda, then*
  - *c is well-typed for cds cdc; and*
  - $(cds\ cda \bullet c) \sqsubseteq (cds\ cdc \bullet c)$.

A sequence of class declarations is well-formed if all methods, or rather, the commands in their bodies, are well-typed and there is no mutual recursion. The global components are the free variables, and, inductively, components of attributes of the object-valued free-variables. Intuitively, a type is N-free if any variable declared to have such a type cannot have attributes of the class N.

If c has global components that are not N-free, then the program refinement $(cdscda \bullet c) \sqsubseteq (cdscdc \bullet c)$ is not even well-defined because the programs act in different state spaces. For this reason, no global variables of object types are allowed in the result of [Nau01b], which is the closest result in the literature to what we need. There, structural subtyping is used, so there is no way to define a notion like N-free.

Forward simulation (including abstraction functions) is the standard proof technique for class refinement. We define class simulation in the context of private

attributes *avs* and *cvs* of *cda* and *cdc*, respectively, and of a coupling invariant *ci* defined as a relation from states of *cda* to states of *cdc*. The classes *cda* and *cdc* are assumed to provide exactly the same methods.

Coupling invariants have to satisfy certain healthiness conditions. For instance, only states for the same class can be related. Also, the initial states of the classes are related. More stringent conditions are motivated by the proof of soundness of simulation and are discussed later on.

Simulation for predicate transformers is defined in the usual way [GM91], but in terms of a generalized coupling invariant. First, if the class declarations *cda* and *cdc*, or rather, the states of these classes, are related by the coupling invariant *ci*, we define a relation $ogci\ T$, coupling values of a type T. If the type T is primitive, then $ogci\ T$ is the identity: the values of such a type are the same in both contexts. If T is either *Ns* or one of its subclasses, then $ogci\ T$ is the coupling invariant itself. Finally, if T is a class N that does not inherit from *Ns*, then it has the same attributes in both contexts. In this case, we relate an object o of N in the context of *cda* to an object o' in the context of *cdc*, if the values of the corresponding attributes of o and o' are related themselves.

The definition of the generalized coupling invariant for states is shown below.

Definition 2 (Generalized Coupling Invariant). *For a class N and local variables in scope vs, we define gci N vs to relate states σ for N and vs in the context of cda with states σ' for the same class and local variables, but in the context of cdc.*

$$
\begin{array}{ll}
(\sigma, \sigma') \in gci\ N\ vs \Leftrightarrow (\alpha(vs) \lhd \sigma, \alpha(vs) \lhd \sigma') \in ci \wedge & \\
\quad \forall x : \alpha(vs) \bullet (\sigma\ x, \sigma'\ x) \in ogci\ T & \text{if } N \text{ is a subclass of} Ns \\
(\sigma, \sigma') \in gci\ N\ vs \Leftrightarrow dom\ \sigma = dom\ \sigma' \wedge \sigma\ myclass = \sigma'\ myclass \wedge & \\
\quad \forall x : dom\ \sigma \setminus \{myclass\} \bullet (\sigma\ x, \sigma'\ x) \in ogci\ T & \text{otherwise}
\end{array}
$$

where T is the type of x in the context of N.

If N is a subclass of *Ns* we cannot simply define $gci\ N\ vs$ to be *ci* because of the extra local variables *vs*. If we disregard them, by considering the states $\alpha(vs) \lhd \sigma$ and $\alpha(vs) \lhd \sigma'$, then we can require the resulting states to be related by *ci*. The set $\alpha(vs)$ contains the local variables, as opposed to *vs* which is their declaration. We use the operator $\lhd$ (domain subtraction) to remove those variables from the states. The values assigned to the variables of *vs* have to be related by *ogci*. For the case in which N is not a subclass of *Ns*, we require the states to give values to the same variables ($dom\ \sigma = dom\ \sigma'$), to be for the same class ($\sigma\ myclass = \sigma'\ myclass$), and finally give related values to corresponding attributes. Besides declared attributes, a state σ has a special attribute *myclass* that designates its class. The states for a class include all the states for its subclasses.

To define simulation for the classes *cda* and *cdc* we consider the method

environments η and η' determined by *cds cda* and *cds cdc*.

Definition 3 (Class Simulation). *We define*

$$cds, avs, cvs, ci \rhd cda \preccurlyeq cdc$$

if and only if for each method m of cda and cdc, we have that

$$cds, cda, cdc, avs, cvs, ci, Ns \rhd (\eta\ Ns\ m) \preccurlyeq (\eta'\ Ns\ m)$$

We require that the meaning recorded in η for each method of *cda* and *cdc* is simulated by the meaning recorded in η'.

The meaning of a method recorded in the environment is a curried function from argument values to predicate transformers. Simulation for these functions is defined in terms of simulation of predicate transformers. We require that if the corresponding arguments are related by simulation, the resulting predicate transformers are as well. Simulation of arguments amounts to simulation of values, for value arguments, and the identity, for variables passed by result or value-result.

Our main theorem is stated below.

Theorem 1 (Soundness of Simulation). *If* $cds, avs, cvs, ci \rhd cda \preccurlyeq cdc$, *then* $cds \rhd cda \preccurlyeq_{=} cdc$.

The proof of this theorem relies mainly on two facts. The first is preservation: the semantics of the commands of the client classes of *cda* and *cdc* are related by simulation. This implies simulation for any main program. The second is an identity extension lemma: the generalized coupling invariant is the identity when the global components in context are *Ns*-free. Therefore, simulation of a main program implies algorithmic refinement, as required by Definition 1.

The identity extension result is simple and rather straightforward. The proof of preservation, on the other hand, brought to light a few surprises. The syntactic approach to the semantics requires the inclusion of equality on objects as a primitive function. We need that to define, for instance, the semantics of assignment. Such an expression, however, does not preserve data-refinement as it relies on equality of private attributes. Luckily it is not needed in the present semantics and it was eliminated from the language.

For variable blocks, result and value-result parameterization, and specification statements, the coupling invariant has to be surjective. The representation of an object value has to include values of private attributes, even though they are hidden. The semantics of a variable block, for instance, considers all initial values that a local object variable can have, including the different values for its private attributes. If a variable block declares a variable whose type is that being refined, then to relate the concrete block to the abstract block, we have to relate every possible concrete value of the variable to a corresponding abstract value. This requires the coupling relation to be surjective. This requirement is unnecessary, and incomplete, for simple imperative programs [HHS86,dRE98].

A way around this problem is to consider that variables are initialized. In that case, the semantics has to consider only those initial values, and the coupling invariant only needs to be surjective for values that can be expressed in

the language. The visibility restrictions and the simulation properties ensure that differences in values of hidden attributes are not relevant. This approach, however, does not work for specification statements.

We are going to investigate a solution in which each class has an invariant and the semantics quantifies over objects satisfying this invariant only. The coupling invariant is defined as a relation on states that satisfy the invariant and the surjectivity restriction is weaker. The user has to provide class invariants and discharge the corresponding proof obligations. Nevertheless, class invariants are normal practice and have independent justification. Another alternative is to change the semantics to quantify over object values obtained by applying the methods of its class to the initial values defined by the constructor. In other words, we use the weakest invariant determined by the program, rather than requiring an explicitly declared invariant.

Angelic variable (logical constant) blocks only preserve data refinement if the coupling invariant is total. If the coupling invariant is not total, in the concrete counterpart of the block, the angelic choice is restricted. As an example, consider the block (**avar** $x : T \bullet : [x = v, \mathrm{true}]$) using a specification with empty frame. In the abstract context, the block behaves like **skip** as the angelic choice can succeed in establishing the precondition of the specification statement by choosing x to be v. If v does not have a concrete counterpart, however, the concrete block is (**avar** $x : T \bullet : [\mathrm{false}, \mathrm{true}]$), which behaves like **abort**. The approaches above can also be used to avoid the totality restriction on coupling invariants.

In summary, forward simulation is sound for all the program constructs. To extend soundness to specification statements, uninitialized variable blocks, result and value-result parameters, and angelic variables, however, we need surjectivity and totality with respect to some form of class invariant.

4 Future Work

An immediate topic for further work is the investigation of the alternatives pointed out in the previous Section to generalize our results to arbitrary coupling invariants. Besides pursuing these approaches, we are going to adapt our results for the relation $cds \rhd cd \preccurlyeq\!\!\!\neq cd'$. This is the second class refinement relation introduced in [CN00a], which captures the situation in which cd and cd' introduce classes of different names. This subsumes the relation of behavioural subclassing.

Besides the specific goals of our project, we believe that our work complements the work of others in various ways. In particular, we are using a semantic model to justify simulation techniques that are often postulated as means to achieve behavioral subclassing. As a specific example, we plan to work with Gary Leavens to interpret the core constructs of JML using our semantics. On this basis, we expect to justify JML rules for behavioral subclassing.

In the first phase of our project we decided that the scope of the language would include core features of sequential Java, including visibility controls and recursion, but excluding concurrency, exceptions, and most contentiously,

pointers. Meanwhile, Reddy and his student Hongseok Yang have worked on modular reasoning for pointer programs, extending recent work of Reynolds [Rey01,RO01,IO01,Yan00,ROY01].

This work is based on a non-standard logic, but we have recently shown how a form of spatial conjunction can be used in the setting of standard logic and predicate transformers [Nau01a]. This work focuses on reasoning about fine-grained manipulation of pointers. In particular, it localizes reasoning using partitions of the heap that can have two-way interlinking, unlike disciplines such as Universe Types [MPH00] which focus on modular reasoning at the level of classes. In the next phase of our project we plan to deal with pointers using Universe Types together with spatial conjunction.

Variations of the specification statement are used in JML [LLP+00] as "model programs" which are particularly useful in specifying calling patterns of methods, including callbacks [BW99,RL00]. Up to now, our specification constructs include only the specification statements and "angelic variables" (logical constants) of Morgan's refinement calculus [Mor94]. In the next phase, we plan to add abstract attributes and dependencies for modular specification [LN00,Mül01].

References

[Bac87] R. J. R. Back. Procedural Abstraction in the Refinement Calculus. Technical report, Department of Computer Science, Åbo - Finland, 1987. Ser. A No. 55.

[BS00] P. H. M. Borba and A. C. A. Sampaio. Basic Laws of ROOL: an object-oriented language. In *3rd Workshop on Formal Methods*, pages 33 – 44, Brazil, 2000.

[BW99] Martin Büchi and Wolfgang Weck. The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, August 1999. http://www.abo.fi/~mbuechi/publications/TR297.html.

[CN99] A. L. C. Cavalcanti and D. Naumann. A Weakest Precondition Semantics for an Object-oriented Language of Refinement. In J. M. Wing, J. C. P. Woodcock, and J. Davies, editors, *FM'99: World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1439 – 1459. Springer-Verlag, September 1999.

[CN00a] A. L. C. Cavalcanti and D. A. Naumann. A Weakest Precondition Semantics for Refinement of Object-oriented Programs. *IEEE Transactions on Software Engineering*, 26(8):713 – 728, August 2000.

[CN00b] A. L. C. Cavalcanti and D. A. Naumann. Simulation and Class Refinement for Java. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, Fernuniversität Hagen, 2000. Available from `http://www.informatik.fernuni-hagen.de/pi5/publications.html`.

[CSW99] A. L. C. Cavalcanti, A. Sampaio, and J. C. P. Woodcock. An Inconsistency in Procedures, Parameters, and Substitution in the Refinement Calculus. *Science of Computer Programming*, 33(1):87 – 96, 1999.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report Report 159, Compaq Systems Research Center, December 1998.

[dRE98] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[GM91] P. H. B. Gardiner and C. C. Morgan. Data Refinement of Predicate Transformers. *Theoretical Computer Science*, 87:143 – 162, 1991.

[HHS86] J. He, C. A. R. Hoare, and J.W. Sanders. Data refinement refined (resumé). In *European Symposium on Programming*, volume 213 of *Springer LNCS*, 1986.

[HHS93] C. A. R. Hoare, J. He, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.

[IO01] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*. ACM Press, 2001.

[LLP+00] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106. ACM, October 2000.

[LN00] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. Technical Report 160, COMPAQ Systems Research Center, November 2000.

[Mor94] Carroll Morgan. *Programming from Specifications, second edition*. Prentice Hall, 1994.

[MPH00] Peter Müller and Arnd Poetzsch-Heffter. A type system for controlling representation exposure in Java. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *ECOOP Workshop on Formal Techniques for Java Programs*. Technical Report 269, Fernuniversität Hagen, 2000. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[Mül01] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[Nau00] David A. Naumann. Predicate transformer semantics of a higher order imperative language with record subtyping. *Science of Computer Programming*, 2000. To appear.

[Nau01a] David A. Naumann. Ideal models for pointwise relational and state-free imperative programming. In *Principles and Practice of Declarative Programming*, 2001. `http: //www.cs.stevens-tech.edu/~naumann/relambda.ps`,.

[Nau01b] David A. Naumann. Soundness of data refinement for a higher order imperative language. *Theoretical Computer Science*, 2001. To appear.

[Rey01] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*. Palgrave, 2001.

[RL00] Clyde Ruby and Gary T. Leavens. Safely creating correct subclasses without seeing superclass code. In *Proceedings of OOPSLA 2000*, October 2000.

[RO01] John C. Reynolds and Peter W. O'Hearn. Reasoning about shared mutable data structure. Slides from invited talk at SPACE 2001, January 2001.

[ROY01] John C. Reynolds, Peter W. O'Hearn, and Hongseok Yang. Local reasoning about shared mutable data structure. Slides for invited talk at APPSEM 2001 workshop, 2001.

[Sam97] Augusto Sampaio. *An Algebraic Approach to Compiler Design*, volume 4 of *Algebraic Methodology and Software Technology*. World Scientific, 1997.

[VB99] E. Viana and P. Borba. Integrando Java com Bancos de Dados Relacionais. *III Simpósio Brasileiro de Linguagens de Programação*, pages 77–91, May 1999.

[Yan00] Hongseok Yang. An example of local reasoning in BI pointer logic:the schorr-waite graph marking algorithm. Draft, December 2000.