# Simulation and class refinement for Java; extended abstract for WFTJP

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

[1] Centro de Informática
Universidade Federal de Pernambuco (UFPE), Box 7851 50740-50 Recife PE Brazil
Phone: +55 81 271 8430 Fax: +55 81 271 8438
alcc@di.ufpe.br www.di.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**: behavioral subclassing, modular specification and verification, inheritance and dynamic binding, refinement calculi, semantics

## 1 Introduction

This document is an extended abstract of ongoing work on reasoning about correctness and behavioral subclassing for programs using some of the challenging features of Java. We include dynamic binding and inheritance, visibility control, and mutually recursive classes and recursive methods (but no pointers or concurrency). In [?] we give a predicate-transformer semantics for a language with these features as well as specification constructs from refinement calculi [Mor94,?]. Here we discuss intrinsic notions of behavioral subclassing and refinement, and we discuss soundness of proof by forward simulation.

This section describes the context and objectives for our project, lightly touching on related work for contrast. Section 2 presents the language and semantics; Section 3 discusses of refinement; Section 4 sketches the background of the authors and our expectations for the workshop.

Our work is part of an ongoing collaboration involving others at UFPE (P. Borba and A. Sampaio) and Birmingham (U. Reddy) and our research assistants. A long-term goal is to use refinement laws as the basis both for a development method and for tools supporting program analysis, verification, and compilation. In our current project the focus is on semantics, behavioral subclassing and refinement, and refinement laws that are sufficient to reduce programs to a normal form. Laws, however, are beyond the scope of this abstract.

The specification statement $x : [pre, post]$ is treated as a command (albeit one that is not executable). The relation $\sqsubseteq$ of (algorithmic) refinement subsumes the usual notion of satisfaction of specifications; in particular, $x : [\phi, \psi] \sqsubseteq c$ holds just if $c$ meets the specification "modifies $x$, requires $\phi$, ensures $\psi$". Rules of Hoare logic are reformulated in terms of refinement laws.

Intermixing specifications with programs is very natural for use in abstract classes. For stepwise development of object-oriented programs, one needs a notion of refinement to account for correctness-preserving replacement of one definition for a class by another. We write $cds \bullet c$ for a program $c$ using classes declared in a collection $cds$ of class declarations. For a class declaration $cd$ to be behaviorally refined by another class $cd'$ should mean that

$$cds, cd \bullet c \sqsubseteq cds, cd' \bullet c \tag{1}$$

for suitable main programs $c$.

A complementary notion of behavioral refinement is that of behavioral subclassing: here, two classes $cd$ and $cd'$ coexist, with $cd'$ declaring a class by extension from $cd$. Syntactic subclassing allows objects from $cd'$ to be used in place of objects from $cd$, so the behavior of $cd'$ objects should refine the behavior of $cd$ objects. If this is not the case, it can be extremely difficult to reason about program behavior, and many programs will exhibit undesirable behavior.

Adequate formalization of refinement depends on language features used in programs and specifications, e.g, unbridled use of type test "**is**" (`instanceof`) and casts can thwart behavioral subclassing because their semantics is based on naming. For example, in a context where $cd'$ declares a subclass $C'$ of the class $C$ declared by $cd$, the main program **if** $x$ **is** $C'$ **then abort else skip fi** is ill behaved even if $cd$ and $cd'$ define identical behavior.

By making a careful study of behavioral refinement in a language with many of the challenging features of Java, we believe we will contribute to the advance of formal techniques for analysis and verification of Java programs. Many of the issues apply to both notions of behavioral refinement described above. Our results shed light on issues relevant beyond the relatively narrow and long-term goals of program development based on refinement. In particular, our use of a relatively abstract semantic model can illuminate issues that are obscured in detailed operational models, just as other abstractions like Hoare logic, coalgebras, and monads are useful conceptual tools.

On the other hand, abstract semantic models must be justified in terms of a suitable connection with concrete operational semantics, and ideally that connection is relatively transparent. Weakest precondition predicate transformers are attractive because, for imperative programs, the connection with operational semantics is well known (and crucial in completeness proofs connecting Hoare logics with operational semantics). We have chosen where possible to use definitions which extend standard definitions of weakest preconditions for sequential programs. One of the most exciting moments in our work came when we found a semantics for dynamic dispatch that is, on the one hand, a transparent reflection of the operational mechanism, and, on the other hand, compatible with the usual definitions of weakest preconditions for ordinary commands [CN99a,CN99b,**?**].

## 2 Syntax and semantics

We present the syntax by means of a small contrived example.

$$\begin{array}{l}\textbf{class } C \quad \textbf{pri } x\!:\!\textbf{int};\ nxt\!:\ C \\ \quad \textbf{meth } Dec \mathrel{\widehat{=}} \textbf{avar } y \bullet x : [x = y, x < y]\textbf{end } y \\ \quad \textbf{meth } Inc \mathrel{\widehat{=}} (\textbf{res } r \bullet x := x + 1;\ nxt := (\textbf{new } C;\ nxt : \textbf{self}.nxt);\ r := x) \\ \bullet\ c.Inc()\end{array}$$

The main program works in a state space with variable $c$ of type $C$, in a context where $C$ has private attributes $x$ and $nxt$. Public method $Dec$ is specified to decrease $x$, leaving $nxt$ unchanged; the specification uses "angelic varible" $y$ as a logical constant. As in most other work, our expressions have no side effects, although they can fail, as in null dereferencing. Values are returned via result parameters as in method $Inc$. Pointers are a complication we choose to set aside. In method $Inc$, the expression **new** $C$ constructs a fresh object, and in the asignment to $nxt$ there is an update expression which creates a copy of **new** $C$ with its $nxt$ field set to a copy of $nxt$ from the enclosing object. For brevity we omit constructors and related issues throughout this abstract.

As in other current work (e.g, [**?**]) our formalism is not fully compositional: to reason about method calls we need the complete collection of existing classes. But we do allow the main program to have variables of declared object types (not just primitive data) which allows compositional reasoning about main programs in a fixed context. Public attributes can be used to specify such programs.

Semantically, commands denote predicate transformers, i.e, functions from and to formulas. We have also given a set-theoretic semantics for expressions, which we use to prove type-correctness results; eventually this semantics will be used as part of the connection between predicate transformer and state transformer semantics. In the semantics of expressions, object states are tuples, finitely nested in the case of recursive classes like $C$ above.

To organize the semantic definitions we adopt type-theoretic techniques. A complete program $cds \bullet c$ is typed in the context of its global variables, as in $x : T \rhd cds \bullet c$. For command $c$ that can occur in method bodies in class $N$, we use a typing judgement $\Gamma;\ x\!:T, N \rhd c : \mathbf{com}$ where $\Gamma$ contains information about declared classes and subclassing, attributes, and method signatures; here $x$ are attributes, parameters, and local variables. A similar judgement is used for expressions. We drop "**com**" in the sequel for brevity. The typing rules embody Java's rules for visibility and for subsumption in assignments and parameter passing.

The semantics of a typing judgement is given by induction on the typing derivation. The semantics of a command depends on an environment that gives the semantics of methods. Because commands can have recursive calls, the environment is obtained by a fixpoint construction. Given the environment $\eta$, the semantics $[\![\Gamma;\ x\!:T, N \rhd c]\!]\eta$ is a function from and to formulas, with the property that if $\psi$ is typable for $\Gamma;\ x\!:T, N$ then so is $[\![\Gamma;\ x\!:T, N \rhd c]\!]\eta\psi$.

To simplify the interpretation of a method body in the context of its calls, self is treated as an implicit value-result parameter. A call $x.m(e)$, passing parameters $e$ to method $m$ of object $x$, has the following semantics.

$$[\![\Gamma, N \rhd x.m(e)]\!]\eta\ \psi = (\vee_{N' \leq_\Gamma N''} \bullet\ x\ \textbf{isExactly}\ N' \wedge f_{N'}\psi)$$

where $N''$ is the type of $x$, and for each $N' \leq_\Gamma N''$ function $f_{N'}$ is the semantics for a call of method $m$ as it is declared or inherited in class $N'$, taking into account parameter passing including $x$ passed as self.[1]

Note that this semantics is based on the actual semantics of method bodies: an exact type test is used so, operationally, the method body applied is exactly that determined by the dynamic type of the object. To avoid unsoundness [MS98] and simplify logic [**?**] it is natural to reason in terms of a base class specification, and some work goes so far as to require a specification which is then taken to be the semantics of a method call [Lei98]. We want to study and justify reasoning methods rather than presupposing them in semantic definitions.

Algorithmic refinement of commands is defined in terms of the pointwise order on predicate transformers: omitting typing context we have $c \sqsubseteq c'$ iff $[\![c]\!]\psi \Rightarrow [\![c']\!]\psi$ for all suitably typed predicates $\psi$. Typing of predicates poses an interesting issue with respect to visibility. If we consider only predicates that respect the visibility rules, then we cannot distinguish between commands with inadequate visible state. For the example program, no postcondition for $c.Inc$ could distinguish the given method body from one in which *Inc* sets *nxt* to null. As a result, our semantics would fail to be operationally sound. Thus in the semantics we use an extended typing system in which visibility constraints are dropped.

## 3 Refinement

In this abstract we confine attention to a relation $cds \rhd cd \preccurlyeq_= cd'$ intended to capture the situation that *cd* and *cd'* declare the same class name (and superclass), with *cd'* a behavioral refinement of *cd* in the context of other classes *cds* with which there may be mutually recursive references. The intrinsic requirement for behavioral refinement is the one mentioned at start of paper: (1) should hold for all main programs $c$. Roughly speaking, we take that as the definition of $\preccurlyeq_=$.

The standard technique for proving such a relation is simulation. Even for data refinement of ordinary modules (without dynamic binding and inheritance) it is known that two forms of simulation, forward and backward, are needed for this to be a complete proof technique even when nondeterminacy is bounded [HHS86]. So it is a little surprising that forward simulation has been proposed as a definition for class refinement (e.g. [LW94,BMvW97,MS98]). In any case, simulation needs to be shown sound in the sense that it implies the relation we call $\preccurlyeq_=$. We are aware of few such results (except in the context of concurrent

[1] We use syntactic rewriting for semantics of parameter passing, and syntactic fixpoint techniques as well [LM98,CSW99].

objects using method call traces rather than pre-post specifications). One such result appears in [BMvW97] for a restricted language and main programs of a fixed shape.

Our main conjecture says that if $cds, ci \rhd cd \preccurlyeq cd'$ then $cds \rhd cd \preccurlyeq_{=} cd'$. Here $cds, ci \rhd cd \preccurlyeq cd'$ expresses that there is a forward simulation using coupling relation $ci$, as we explain below. We are confident in the result but call it a conjecture because some details have not been checked at the time of writing. Our work on this result has been fruitful in uncovering subtleties. For example, in (1) the algorithmic refinement relation $\sqsubseteq$ needs to be defined in terms of observable predicates, i.e, those that respect the visibility rules; otherwise the comparison would be between commands in different state spaces. The corresponding result for behavioral subclassing is even more interesting due to type tests and casts, as mentioned earlier.

Predicate transformers seems to be at a good level of abstraction for such proofs. Details about typing etc. are precise enough to ensure proper treatment of many necessary technical details, but still the proofs are comprehensible and tractable by hand with diligence.

For a command $c$ in the context of a method body for the refined class, the forward simulation condition for coupling invariant $ci$ takes a standard form:

$$(\exists\, vs \bullet ci \wedge [\![\Gamma, N_S \rhd c]\!]\eta\ \psi) \Rightarrow [\![\Gamma', N_S \rhd c']\!]\eta'\ (\exists\, vs \bullet ci \wedge \psi) \qquad (2)$$

for all suitable postconditions $\psi$. Here $vs$ are the abstract variables used in $cd$; the coupling invariant relates them to concrete variables $vs'$ used in $cd'$. Also, $\Gamma$ and $\eta$ (respectively, $\Gamma'$ and $\eta'$) are environments for $cds\ cd$, $N_S$, and $c$ (respectively, $cds\ cd'$, $N_S$, and $c'$), and $N_S$ is the class declared by both $cd$ and $cd'$.

Outside these classes, commands should not need to be modified, as only private attributes of the classes are changed. Nevertheless, even though these commands cannot access or affect the attributes being changed, their weakest precondition semantics is defined in terms of postconditions that can refer to these attributes. Therefore, we define a forward simulation relation for all commands. A function $gci$ is needed to generalize the coupling invariant $ci$ from the state space of the refined class to state spaces in other classes. We omit some arguments to $gci$ in the forward simulation condition for commands:

$$gci\ ci\ ([\![\Gamma, N \rhd c]\!]\eta\ \psi) \Rightarrow [\![\Gamma', N \rhd c]\!]\eta'\ (gci\ ci\ \psi) \qquad (3)$$

We show that this holds for all $c$ provided (2) holds for method bodies of the refined class. For predicates that do not refer to the private attributes of the refined class, $gci$ reduces to the identity function; this "identity extension lemma" is used to derive the main soundness result from (3).

In set theoretic models, it is easy to define the extension of a coupling relation to arbitrary data types and state spaces [BMvW97,**?**], but $gci$ must act in purely syntactic terms on formulas, which is one of the main challenges in obtaining our result. The benefit is that it is the syntactic form that is needed in proof rules anyway, if such rules are to be expressed at the level of specifications.

## 4 The authors

For reasons of funding, we expect that only Naumann can attend the workshop.

Background: Extension of wp based ref calc to proper treatment of procedures [Ana], integration with rich specification language [Ana], data structured as extensible records [D, Ana in Z?], procedures stored, which can model dyn dispatch [D]. Both of us have experience in OO prog dev; D writing textbook on DS using OO features and formal meth; A writing book on ?

Indeed, our work grew out of a more semantically oriented model for Oberon, using transformers of semantic predicates [**?**,**?**], which in turn was based on an abstract mathematical analysis of predicate transformers at higher types [**?**,**?**].

D PT math, apply to alg of progs on ind types; w Kedar rich logic for pointers and dyn types, decision with model checking

My book is on a refinement calculus for Z. I have designed of the several object-oriented Z specifications. It is called MooZ and is presented in [**?**,**?**,**?**]. I'm also interested in semantics for object-oriented languages in general [**?**]. I'm also investigating a method for transforming specification written is a combination of CSP and Z to Java programs. But I haven't published anything yet. So, I'm not sure it's interesting to mention that.

pointers... (They can be, and are in current work [Jacobs,Leino,Muller], treated using standard techniques.)

Expectations: Besides making others aware of our work... help find suitable notions of behav ref. Learn how to handle features we have set aside (e.g. pointers) in belief that our approach can be extended. Learn current thinking on key specification techniques which we expect can be used in our setting. Explore possible connections with other oper sem (our project includes state trans sem but not clear that a complete connection can be found in near term, and we'd like a more concrete one anyway). Learn about experience and available stuff in PVS for eventual use with our semantics (Shankar persuaded me semantics reasoning is practical, but not that oper sem is best.) Learn about exceptional termination (std technique as in Jacobs used in Leino's wp sem and it's the coprod in our underlying model [**?**]

## 5 STRAYS - topics thoughts not yet organized

value sem same as ptrs in absence of sharing phenomena; doing everything at level of heap loses abstraction

Want behav subclassing so reasoning can be in terms of spec, to simplify logic (ESC, Arnd, ...) and avoid unsoundness (Leonid fragile base class)

Interconnected issues: sem not well def'd w/o behav subcl but to define behav subcl need sem

The wide acceptance of the object-oriented paradigm of analysis, design, and programming has raised the interest of the formal methods community in understanding and coping with the object concept and technology. Af first, several object-oriented formal specification languages were proposed [LH94,DK92,GM87].

Nowadays, attention has turned to reasoning methods that can deal with objects, dynamic binding, and take advantage of the modularity offered by object-orientation [LW94,MS97].

In an attempt to provide a formal basis to a refinement calculus for an object-oriented language, we have recently presented a weakest precondition semantics for ROOL, an object-oriented language for refinement [CN99b,CN99a]. It is similar to Java, but also includes constructs commonly found in languages of refinement calculi. The work in [CN99b,CN99a] is based on the standard predicate transformer model of refinement, which is extented to deal with recursive classes, visibility control, dynamic binding, and recursive methods.

## References


[BMvW97] R.J.R. Back, A. Mikhajlova, and J. von Wright. Class refinement as semantics of correct object substitutability. Technical Report TUCS-TR-147, Turku Center for Computer Science, December 1997.

[CN99a] 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.

[CN99b] A. L. C. Cavalcanti and D. Naumann. A Weakest Precondition Semantics for an Object-oriented Language of Refinement - Extended Version. Technical Report CS 9903, Stevens Institute of Technology, September 1999.

[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.

[DK92] E. H. Durr and J. Katwijk. VDM++ – A Formal Specification Language of Object-oriented Design. In *Computer Systems and Software Engineering. Proceedings of CompEuro'92.*, pages 214 – 219. IEEE Computer Society Press, 1992.

[GM87] J. Goguen and J. Meseguer. Unifying Functional, Object-oriented and Relational Programming with Logical Semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-oriented Programming*, pages 417 – 477. MIT, 1987.

[HHS86] J. He, C. A. R. Hoare, and J. W. Sanders. Data Refinement Refined. In G. Goos and H. Hartmants, editors, *ESOP'86 European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 187 – 196, March 1986.

[Lei98] K. R. M. Leino. Recursive Object Types in a Logic of Object-oriented Programming. In C. Hankin, editor, *7th European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[LH94] K. Lano and H. Haughton, editors. *Object-oriented Specification Case Studies*. The Object-oriented Series. Prentice-Hall, 1994.

[LM98] K. R. M. Leino and R. Manohar. Joining Specification Statements. *Theoretical Computer Science*, 1998. To appear.

[LW94] B. H. Liskov and J. M. Wing. A Behavioural Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[Mor94] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.

[MS97] A. Mikhajlova and E. Sekerinski. Class refinement and Interface refinement in Object-oriented Programs. In *FME'97: Industrial Benefit of Formal Methods*. Springer-Verlag, 1997.

[MS98] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In E. Jul, editor, *Proceedings of ECOOP'98*, number 1445 in Lecture Notes in Computer Science. Springer-Verlag, 1998.