Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006)

ACM SIGSOFT/FSE-14
14th ACM Symposium on the
Foundations of Software Engineering
Portland, Oregon, USA
November 10-11, 2006

Technical Report #06-29, Department of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, IA 50011-1041, USA

SAVCBS 2006 PROCEEDINGS

Specification and Verification of Component-Based Systems

http://www.cs.iastate.edu/SAVCBS/

November 10-11, 2006
Portland, Oregon, USA

Workshop at ACM SIGSOFT/FSE-14
14th ACM Symposium on
Foundations of Software Engineering

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006), November 10-11, 2006, Portland, Oregon, USA.

SAVCBS 2006
TABLE OF CONTENTS

SAVCBS 2006
ORGANIZING COMMITTEE

Mike Barnett (Microsoft Research, USA)
Mike Barnett is a Research Software Design Engineer in the Foundations of Software Engineering group at Microsoft Research. His research interests include software specification and verification, especially the interplay of static and dynamic verification. He received his Ph.D. in computer science from the University of Texas at Austin in 1992.

Dimitra Giannakopoulou (RIACS/NASA Ames Research Center, USA)
Dimitra Giannakopoulou is a RIACS research scientist at the NASA Ames Research Center. Her research focuses on scalable specification and verification techniques for NASA systems. In particular, she is interested in incremental and compositional model checking based on software components and architectures. She received her Ph.D. in 1999 from the Imperial College, University of London.

Gary T. Leavens (Dept. of Computer Science, Iowa State University, USA)
Gary T. Leavens is a professor of Computer Science at Iowa State University. His research interests include programming and specification language design and semantics, program verification, and formal methods, with an emphasis on the object-oriented and aspect-oriented paradigms. He received his Ph.D. from MIT in 1989.

Natasha Sharygina (CMU and SEI, USA; Lugano, Switzerland)
Natasha Sharygina is a senior researcher at the Carnegie Mellon Software Engineering Institute and an adjunct assistant professor in the School of Computer Science at Carnegie Mellon University, and an assistant professor at the University of Lugano. Her research interests are in program verification, formal methods in system design and analysis, systems engineering, semantics of programming languages and logics, and automated tools for reasoning about computer systems. She received her Ph.D. from The University of Texas at Austin in 2002.

SAVCBS 2006
PROGRAM COMMITTEE

Jonathan Aldrich (School of Computer Science, Carnegie Mellon Univ., USA)
Jonathan Aldrich chaired the program committee for SAVCBS 2006. He is an assistant professor in the School of Computer Science at Carnegie Mellon University. His research interests are in lightweight software verification using programming language and program analysis techniques. He received his Ph.D. in Computer Science from the University of Washington in 2003.

Program Committee:
Jonathan Aldrich (Carnegie Mellon University), Program Committee Chair
Michael Barnett (Microsoft Research)
Patrice Chalin (Concordia University)
Robert Chatley (Kizoom, London)
David Coppit (The College of William and Mary)
Ivica Crnkovic (Mälardalen University)
Stephen Edwards (Virginia Tech)
Timothy J. Halloran (Air Force Institute of Technology)
Marieke Huisman (INRIA Sophia Antipolis)
Joseph Kiniry (University College Dublin)
Matthew Parkinson (Middlesex University)
Corina Pasareanu (QSS/NASA Ames Research Center)
Andreas Rausch (University of Kaiserslautern)
Robby (Kansas State)
Heinz Schmidt (Monash University)
Wolfram Schulte (Microsoft Research)
Natasha Sharygina (Lugano and Carnegie Mellon)
Tao Xie (North Carolina State)

Sponsors:

Microsoft® Research

SAVCBS 2006
WORKSHOP INTRODUCTION

This workshop is concerned with how formal (i.e., mathematical) techniques can be or should be used to establish a suitable foundation for the specification and verification of component-based systems. Component-based systems are a growing concern for the software engineering community. Specification and reasoning techniques are urgently needed to permit composition of systems from components. Component-based specification and verification is also vital for scaling advanced verification techniques such as extended static analysis and model checking to the size of real systems. The workshop will consider formalization of both functional and non-functional behavior, such as performance or reliability.

This workshop brings together researchers and practitioners in the areas of component-based software and formal methods to address the open problems in modular specification and verification of systems composed from components. We are interested in bridging the gap between principles and practice. The intent of bringing participants together at the workshop is to help form a community-oriented understanding of the relevant research problems and help steer formal methods research in a direction that will address the problems of component-based systems. For example, researchers in formal methods have only recently begun to study principles of object-oriented software specification and verification, but do not yet have a good handle on how inheritance can be exploited in specification and verification. Other issues are also important in the practice of component-based systems, such as concurrency, mechanization and scalability, performance (time and space), reusability, and understandability. The aim is to brainstorm about these and related topics to understand both the problems involved and how formal techniques may be useful in solving them.

The goals of the workshop are to produce:

1. An outline of collaborative research topics,
2. A list of areas for further exploration,
3. An initial taxonomy of the different dimensions along which research in the area can be categorized. For instance, static/dynamic verification, modular/whole program analysis, partial/complete specification, soundness/completeness of the analysis, are all continuums along which particular techniques can be placed, and
4. A web site that will be maintained after the workshop to act as a central clearinghouse for research in this area.

We enthusiastically thank the authors of submitted papers; their quality contributions and participation are what make a workshop like SAVCBS successful. We thank the program committee for their careful reading and reviewing of the submissions. Our PC members have expertise in a wide variety of sub-disciplines related to specification and verification of component-based systems; they include established research leaders and promising recent Ph.D.s; they come from both industry and academia, and hail from all over the world.

We received 13 submissions, of which 3 were withdrawn, leaving 10 to be reviewed. All papers were reviewed by 3 PC members, with PC member papers were reviewed by 4 PC members and held to a higher-confidence standard. Ultimately 8 papers were accepted, after PC discussions via email. As in previous years, we accepted additional submissions as poster presentations, reflecting the role of SAVCBS to promote discussion and incubation of new ideas for which a full paper may be premature.

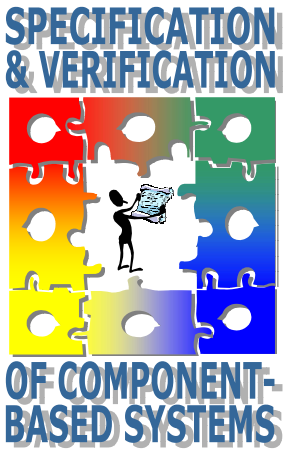
This year our program also includes solutions to a specification and verification challenge problem posed to workshop attendees. The problem focused on the specification of iterators in collection libraries such as those in Java or C#. In these systems multiple iterators can be created over a collection, and can access that collection simultaneously as long as it is modified. However, if the collection is modified, all iterators are invalidated (except—for Java—the iterator through which the change was made, if any). While familiar to many programmers, this problem poses real challenges for specification and verification systems such as state aliased between the iterators and the collection. Four-page challenge problem solutions were each read and reviewed by two members of the program committee, to ensure quality and help the authors improve their presentation; we accepted all 5 submissions.

This year we also were pleased to have an invited presentation by Josh Berdine of Microsoft Research titled “Variance Analyses from Invariance Analyses.”

Jonathan Aldrich (Program Committee Chair)

Mike Barnett (Organizing Committee)
Dimitra Giannakopoulou (Organizing Committee)
Gary T. Leavens (Organizing Committee)
Natasha Sharygina (Organizing Committee)

SAVCBS 2006 PAPERS



Performance Analysis Based upon Complete Profiles

Joan Krone
Denison University
Granville, Ohio 43023
+1 740 587 6484
krone@denison.edu

William F. Ogden
The Ohio State University
Columbus, Ohio 43210
+1 614 292 6004
ogden@cse.ohio-state.edu

Murali Sitaraman
Clemson University
Clemson, SC 29634
+1 864 656 3444
murali@cs.clemson.edu

ABSTRACT
A system for engineering and verifying component-based software must include mechanisms for specifying abstractly not only the complete functionality of components but their exact performance as well. This paper introduces *profiles* as a first-class construct for complete, independent specification of performance in higher-level languages. Using profiles, a developer can select from an assortment of implementations for a particular functionality the one that best suits his needs with respect to speed and memory usage. Equally importantly, he can define the expected performance of larger scale components using compositions of the profiles of their constituent (possibly as yet unimplemented) components. To support scalability, the profile construct facilitates abstraction in performance specifications as well as performance composition and analysis.
.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Metrics – *performance measures*

General Terms
Software Engineering, specification, verification.

Keywords
Components, performance, reuse, specification.

1. INTRODUCTION
In order to have an effective system for engineering component-based software, it is essential to have a specificational framework that supports description of those aspects of a component that are relevant to its deployment and that implicitly supports suppression of other irrelevant aspects. The functional aspect is typically the most important, and so developing a framework for its specification has been the focus of much research. However, a framework is not adequate until it includes a mechanism for completely describing component performance.

SAVCBS 2006. November 10-11, Portland, Oregon, USA.

Factoring out performance specifications seems to be a common practice in the engineering of components. An auto manufacturer, for example, sets functional limits on the dimensions of tires that can be used but leaves to tire suppliers such performance specifications as traction, tread life, etc. As in the case of auto tires, a good conceptualization of functional behavior will admit a broad assortment of realizations with varying performance characteristics.

Currently performance specifications for software components are usually treated in a rather off-hand manner. Often they're given as gross Big-O estimates, typically in terms of imprecisely-specified parameters ill suited to object oriented programming (a problem we addressed in [11]). Alternatively, they're presented as exact durations for particular "representative" examples run on particular hardware, which data isn't ordinarily of much use for predicting behavior in future applications of components. In [11], we introduced language mechanisms for including exact performance specifications within each realization for a given component. We used an enhancement for a stack component to emphasize the important role of abstraction by showing that our approach permitted performance specification to be established without knowledge of how the stack component was implemented.

Subsequently we have found that there are important advantages to separating performance specifications not only from the component concepts but also from the realizations for the concept. Since the functionality of a component can be employed independent of the performance characteristics of its various implementations, those various performance specifications obviously don't belong in a component's conceptualization, where all its functional characteristics are formally specified. The principal advantage of separating performance specifications from particular realizations is that it supports additional reuse of specifications. As we've discovered, the performance of alternative implementations for a component often differs only in ways that can easily be parameterized in an appropriately abstract specification. Such a separation of specifications also makes it easier to document the performance of hardware components that are often constituents of larger (embedded) systems.

We introduce the *profile* then as a first class specificational construct for recording performance characteristics. Profiles have the virtue of allowing the designer of a component implementation to summarize its expected performance in a concise form that masks implementation details. At the same time, a prospective client for the functionality of the component

can use the profiles of its various implementations to select the one that best suits his performance objectives.

Since a profile is to act as a *performance contract* between client code and implementation code, it should become an artifact of the software development process with an importance similar to that of a functionality specification contract for a component. This makes it an entity that is independent of top-down or bottom-up development methodologies. Typically, development of a a good profile demands simultaneous attention to the desires of the clients and to the possibilities open to the implementers, regardless of whether implementation or client code exists at the time. In the same way that abstract specifications of functional behavior provide essential guideposts in development of modular component-based systems, profiles provide analytic yardsticks for checking the adequacy of system performance.

Since a performance profile is of necessity expressed in the context of a functional specification, it is not surprising that performance specification and verification potentially involves every complexity that can arise in functionality specification and verification. Moreover, since overall performance depends upon every detail of an implementation, its specification poses several new challenges. One of them is to aggregate these details into simplified, abstract specifications so that that clients can keep their focus on the larger picture as they the higher level code. Another is to formulate expressions for the performance of generic components, since performance specifications for their parameters are not fixed when their profiles are written. Our examples illustrate how to cope with all this complexity.

2 A PROFILE EXAMPLE

In order to ensure the generality of our profile mechanism proposed in this paper, we have tested it by creating performance specifications for a variety of software components, including a layered component-based system which addresses the issue of scalability. Since our objective here is to introduce the basic ideas in developing complete profiles that only make sense in the context of a thoroughly understood component, we will forego complexity of more sophisticated components and instead use the familiar generic stack component, as we did in [11]. For it, the functional specifications are given in Figure 1 in Resolve.

```
Concept Stack_Template( type Entry;
                        evaluates Max_Depth: Integer );
        uses String_Theory;
    requires Max_Depth > 0;
  Type_Family Stack ⊆ Str(Entry);
    exemplar S;
    constraints |S| ≤ Max_Depth;
    initialization ensures S = Λ;
  Operation Push( alters E: Entry; updates S: Stack );
    requires |S| < Max_Depth;
    ensures S = ⟨#E⟩∘#S;
  Operation Pop( replaces R: Entry; updates S: Stack );
    requires |S| > 0 ;
    ensures #S = ⟨R⟩∘S;
  Operation Depth_of( restores S: Stack ): Integer;
    ensures Depth_of = ( |S| );
      ⋮
end Stack_Template;
```

Figure 1: Specification for a Stack_Template

Figure 1 shows a formal, conceptual client view of a generic bounded Stack component, parameterized by the type of entries to be contained in stacks and the maximum depth to which a stack can grow. (The **evaluates** mode is used to indicate that an expression may be passed as the maximum Stack depth.)

The Stack Template concept uses mathematical String Theory, a development of which is given in [20], to formalize stacks. The notation **Type_Family** is used where the stack formalization is introduced in order to highlight the generic nature of the concept by reminding that it involves a whole family of Stack types, which differ depending upon the particular Entry type and *Max_Depth* parameters supplied at the time of instantiation.

The concept provides specifications of typical Stack operations, each specified by a **requires** clause (precondition), which is an obligation for all callers, and an **ensures** clause (postcondition), which is a result guarantee from any correct implementation.

For example, the *Pop* operation updates the value of the stack parameter *S* by removing its top entry and using it to replace the value of the parameter *R*. This result is guaranteed by the **ensures** clause $\#S = \langle R \rangle \circ S$ once we know that $\#S$ refers to the previous value of *S*, that $\langle R \rangle$ is the single entry string containing *R*, and that $\circ$ is the concatenate operation for strings. The *Clear* operation gives stack *S* the initial stack value Λ (empty), and it gets this specification not based upon an **ensures** clause but instead based upon the **clears** parameter mode.

The important point here is that, by conceiving of stacks as strings, it is possible to give a complete and coherent explanation of all of the operations on stacks. Absolutely no reference to details of any particular implementation such as arrays, pointers, or linked lists is needed. This hiding of client irrelevant information by reconceptualization of objects is an equally critical feature for any satisfactory performance specification mechanism.

2.1 A Performance Profile for the Stack

In [11], we addressed the basic problems of adding performance specifications to **realization** code and of developing a reasoning system to verify that such specifications are accurate. That work was sound as far as it went and served as the basis for subsequent work on specification of performance properties in JML, and analysis of dynamic heap space usage in [1]. However, the earlier work doesn't fully address the larger software engineering scalability concerns of separating out concise and comprehensible summaries of the performance of component implementations and of structuring them in such a fashion that they support the derivation of analogous specifications for large components produced as compositions of smaller ones. Here we write performance specifications called ***profiles*** that represent a class of implementations, thereby removing these specifications from individual realizations, and remaining at a level of abstraction allowing for multiple realizations.

Since some alternative implementations of generic concepts such as stacks provide substantively different performance trade-offs, they will of necessity have different ***profiles***. The performance ***profile*** in Figure 2, named *SSC*, is suitable for a class of *S*tack implementations that are *S*pace-*C*onscious, i.e., ones that consider space to be more important than time. The profile is written without making any assumptions about the generic type *Entry* or *Max_Depth*, and therefore, the expressions have to be compositional and presented in terms of these parameters.

One of the key elements in the specification of a ***profile*** that's free of unnecessary implementation details, is the notion of a **defines** specification clause. Whereas a typical (mathematical) definition provides an immediate definiens for its definiendum, the **defines** clause allows a profile to name a definiendum for use within the profile, but to defer to each implementation the provision of a particular definiens. An implementation can then provide a specific definiens for each **defines** deferred definiendum based upon its exact code. So the **defines** construct provides a second mechanism whereby profiles can achieve appropriate independence. Whereas *Entry* and *Max_Depth* are traditional parameters whose values come down from clients, the deferred constants SSC_I, SSC_D, etc. seen here can be viewed as parameters whose values come up from implementations.

A performance ***profile*** is intended to document the behavior of a class of implementations in terms understandable to clients of the concept and generally simpler than an exhaustive description of each implementation. A ***profile*** provides the following information. For each operation, there is a **duration** clause – a non-negative real number valued expression – that places a bound on the time taken by the operation in terms of the parameters supplied to the operation.

For each operation, there is a manipulation displacement clause (abbreviated as **manip_disp),** a natural number valued expression that bounds the minimum additional space that is necessary to execute the operation above and beyond what is occupied by all objects currently in scope. Since memory usage may increase and decrease during the execution of a complex procedure, this clause expresses the "high water mark" in terms of the parameters to the operations. In order to use this information to determine whether there is enough space to execute the next call with a certain collection of arguments, a caller needs to be able to determine the space occupied by all current objects. Thus, ***profiles*** for implementations that provide types (and therefore permit creation of objects) include a displacement clause – also a natural number – that describes how much space is used by a variable (e.g., a Stack variable), given its abstract value (a string of entries). We begin the discussion with this clause, following Figure 2.

Profile SSC **short_for** Space_Conscious **for** Stack_Template;

Defines SSC_I, SSC_{I1}, SSC_F, SSC_{Po}, SSC_{Pu}, SSC_C, SSC_{C1}, SSC_{Dp}, SSC_{RC}: $\mathbb{R}^{\geq 0}$;

Defines SSC_D, SSC_{MI}, SSC_{MF}, SSC_{MPo}, SSC_{MPu}, SSC_{MC}, SSC_{MDp}, SSC_{MRC}: $\mathbb{N}$;

Type_Family Stack;

Definition Cnts_Disp(α: Str(Entry)): $\mathbb{N}$ = ($\sum_{E: Entry}$ Occurs_Ct(E,α)·Entry.**Disp**(E));

Displacement SSC_D + Cnts_Disp(S) + (Max_Depth – |S|)·Entry.**I_Disp**;

Initialization;

duration SSC_I + (SSC_{I1} + Entry.**I_Dur**)·Max_Depth;

manip_disp SSC_{MI} + Entry.**IM_Disp** + (Max_Depth – 1)·Entry.**I_Disp**;

Oper Pop(**replaces** R: Entry; **updates** S: Stack);

duration SSC_{Po} + Entry.**I_Dur** + Entry.**F_Dur**(#R);

manip_disp SSC_{MPo} + Max(Entry.**IM_Disp**, Entry.**FM_Disp**(#R));

Oper Push(**alters** E: Entry; **updates** S: Stack);

ensures Entry.**Is_Init**(E);

duration SSC_{Pu};

⋮

end SSC;

Figure 2: A Performance Profile

The Displacement Clause

We note that the same ideas discussed here suffice whether or not stacks are bounded a priori, as also noted by Atkey[1]. For example, if the stack elements are allocated only when needed instead of initially in an array, then the displacement will be less and it would not include the last term seen here. However, to make our discussions concrete, we consider implementations that allocate and initialize an array of entries of size *Max_Depth* whenever a new *Stack* is created. An implementation might use a simple representation such as the one shown below:

Type Stack = **Record**
Contents: **Array** 1..Max_Depth **of** Entry;
Top: Integer
end;

Within this context, one class of implementations can be characterized as placing high priority on minimizing space usage for a *Stack* variable, by following a space-conscious convention (or representation invariant): All entries in array locations beyond those that correspond to the conceptual stack value are kept initialized. For a stack containing complex objects such as trees, for example, this convention leads to minimal space usage because unused array locations contain only empty trees instead of arbitrary trees.

Though we have divulged the representation details above in order to provide a concrete example for readers of this paper, a performance profile must be understandable to users based only upon the mathematical conceptualization of stacks as strings as given in Figure 1. Accordingly, the displacement clause in this performance profile expresses the space occupied by a stack S using only its abstract string value:

Displacement SSC_D + Cnts_Disp(S) + (Max_Depth – |S|)·Entry.**I_Disp**;

There are three terms in this expression. The first term is the constant SSC_D, and it represents the fixed space overhead in any Stack object (e.g., an Integer index into the array that is used to keep track of the current top). The actual definition for this constant is implementation-specific and will be specified within the implementation; the profile merely provides a placeholder for this constant and others by listing them in the **defines** clause. The second term captures the space occupied by the entries that have been pushed onto a stack. To express this term, we have introduced a locally defined contents displacement function *Cnts_**Disp**(S),* which totals for each entry *E* in a stack *S* its displacement *Entry.**Disp**(E)* times *Occurs_Ct(E, S*), the number of times *E* occurs in *S*.

The last term in the displacement expression is the product (*Max_Depth* − |*S*|)·*Entry.**I_Disp***), and it accounts for the space taken by unused array entries (all of which are assumed by this profile to have initial values). Here, *Entry.**I_Disp*** denotes the space used by an entry with an initial value. Using the given expression, it is easy to see that for an empty stack with abstract string value Λ, the displacement *Stack.Disp(*Λ*)* becomes SSC_D + *Max_Depth*·*Entry.**I_Disp***.

Specification of Initialization

In the class of implementations under discussion here, when a Stack variable is initialized, *Max_Depth* number of entries are created and initialized. Therefore, initialization duration includes the factor *Entry.**I_Dur***·*Max_Depth*, which is the product of the duration for initializing a variable of type Entry, i.e., *Entry.**I_Dur*** and *Max_Depth*, the number of entries to be initialized. The expression includes additional constant overhead per entry, denoted by SSC_{I1}, as well an overall constant overhead denoted by SSC_I. The actual definitions for these implementation-specific constants will be given in the implementations. (If the Stack elements are allocated only when needed instead of using an array, then initialization will take a constant time, and the cost of object creation will be moved to the *Push* operation.)

The initialization **manip_disp** clause expresses the minimum storage space necessary to create a new stack variable. Recall that *Entry.**I_Disp*** denotes the space taken by an entry with an initial value. To create a Stack representation with *Max_Depth* initial entries, the necessary displacement is roughly *Entry.**I_Disp***·*Max_Depth*. The expression given in the profile differs slightly because the procedure to create an initial entry might need more space than what is strictly necessary for storing an initial entry. This would be the case if Entry is a non-trivial type, and creating an initial value for it requires creation and use of other local variables. Therefore, suppose that *Entry.**IM_Disp*** denotes the manipulation space necessary for initial entry creation. Then the highest watermark in space usage during *Stack* initialization occurs when *Max_Depth* − 1 new entries have been created and the *Entry* initialization operation is being invoked to initialize the last entry. Therefore, this is the minimum space necessary to initialize a new *Stack*. The expression includes an implementation-specific constant as well.

Specification of Pop

To explain the expressions for *Pop*, we consider the following code that might have been written for a space-conscious implementation.

```
Procedure Pop( replaces R: Entry; updates S: Stack );
    Var Fresh_Val: Entry;
    R :=: S.Contents(S.Top);
    S.Contents(S.Top) :=: Fresh_Val;
    S.Top := S.Top – 1;
end Pop;
```

In this implementation, we have used the swap operator ":=:", instead of assignment, to move *Entry* values and to access array contents. The reasoning and efficiency advantages of swapping over reference assignment and representation assignment of arbitrary entries, respectively, are discussed in detail elsewhere [8]: Swapping enables reasoning without introducing aliasing; its implementation is efficient because compliers can represent large objects internally using references and merely exchanging the references in constant time. (If entries are copied, then the same principles of specifying performance expressions would still be adequate, except that the performance expressions need to account for copying.)

The second swap statement in the code is necessary to satisfy the space-conscious convention. By declaring a local Entry variable (which is automatically initialized) in the *Pop* procedure and swapping it into the array, we make sure that the arbitrary entry *R* that might have been supplied as the incoming parameter to *Pop* does not go into the array and violate the convention. At the end of the code, the local variable that then contains the incoming value of parameter *R* is released or finalized. The performance specification of Pop is expressed in user-oriented terms in the profile:

```
Operation Pop( replaces R: Entry; updates S: Stack );
    duration SSC_Po + Entry.I_Dur + Entry.F_Dur(#R);
    manip_disp SSC_MPo +
            Max( Entry.IM_Disp, Entry.FM_Disp(#R) );
```

The **duration** expression includes the time to initialize a new Entry variable. Finalization depends on the Entry that is finalized, and thus, the time to finalize is given in terms of the incoming value of parameter *R*. The definition for the deferred constant SSC_{Po} in the duration expression for *Pop* code is given internally in each implementation. For the present example, it might be defined as:

Definition SSC_{Po}: $\mathbb{R}^{\geq 0}$ = $\mathbf{Dur_{Call}}(2)$ + 2·Array.**Dur**$_{:=:}$ +
6·Record.**Dur**$_{.}$ + Int.**Dur**$_{:=}$ + Int.**Dur**$_{-}$;

This constant includes the *time to call a procedure with 2 parameters, denoted by* $\mathbf{Dur_{Call}}(2)$, array and record accesses, and Integer operations. This definition is relegated to the implementation because it provides too much information to include in a profile for clients and it is expressed in terms of implementation details that should not be visible to them. Placing the definition in the **profile**, in addition to hard wiring it, would seriously compromise information hiding and hinder modularity in reasoning.

How much space is necessary to call *Pop* beyond what is already taken up by its parameters? It is the maximum of the displacement necessary to initialize a new variable, i.e., *Entry.**IM_Disp*** (Entry initialization manipulated displacement) or finalize the incoming parametric entry, i.e., *Entry.**FM_Disp**(#R)*.

One other aspect of interest in the performance **profile** is the additional **ensures** clause for the Push operation. In particular, using the predicate *Entry.**Is_Init**(E)* that is true only if *E* has an initial *Entry* value[1], the ensures clause tells a user that *E* will be initialized after a call to *Push(E, S)*. While this information, which appears only in the performance profile, cannot be used by a client program in establishing functional correctness, it can be used for reaching **displacement/duration** conclusions, as illustrated in Section 3. Unlike *Pop*, the *Push* and *Depth_of* procedures have constant performance expressions.

Performance **profiles** are useful for component clients, enabling them to select prudently from among a variety of

[1] We use a predicate here instead of asserting *E* = *Entry.**Init*** or equivalent, because initializations may be specified to give an object one of many initial values.

implementations for a particular concept that provide interesting performance trade-offs. They are also important for independent development and modular analysis of component-based systems in the same way that abstract specifications of functional behavior are useful. For example, performance of other components that reuse the *Stack* **concept** can be derived from the performance profile of the chosen *Stack* implementation. To illustrate how profiles for a component built on other components can be presented parametrically, we analyze code for a component built on *Stack* objects and operations. The example specification for a *Flip* operation to invert a stack is given below. It is an **enhancement** or conceptual extension to the *Stack_Template* described previously. In the **ensures** clause, *Rev* denotes the mathematical string reversal operator.

Enhancement Flipping_Capability **for** Stack_Template;

Operation Flip(**updates** S: Stack);

ensures $S = \#S^{Rev}$;

end Flipping_Capability;

2.2 Profile Specification of Flip

A given implementation of *Flip* may exhibit different performance behaviors, depending on the profile of the *Stack* implementation that is used in conjunction with *Flip*. It becomes possible to express this performance dependence of one component upon another quite elegantly, if profiles are available as first class constructs in a language. To illustrate how this is done, we show profile *SSCF* for *Flip* based on the *SSC* profile of *Stack_Template*.

Profile SSCF **short_for** Space_Conscious_Stack_Flip **for**

Flipping_Capability **for** Stack_Template **with_profile** SSC;

Defines $SSCF_{F1}$, $SSCF_{F2}$: $\mathbb{R}^{\geq 0}$;

Defines $SSCF_{FMC1}$, $SSCF_{FMC2}$: $\mathbb{N}$;

Operation Flip(**updates** S: Stack);

duration $SSCF_{F1}$ + Entry.**I_Dur** + Stack.**I_Dur** + Entry.**F_IV_Dur** + Stack.**F_IV_Dur** + ($SSCF_{F2}$ + Entry.**I_Dur** + Entry.**F_IV_Dur**)·|#S|;

manip_disp ($SSCF_{FMC1}$ + Entry.**I_Disp** + Stack.**I_Disp**) + Max($SSCF_{FMC2}$, Entry.**IM_Disp**, Entry.**F_IVM_Disp**);

end SSCF;

The abstract performance specifications in the profile above are given in terms meaningful to clients of the *Flipping_Capability*. In particular, the profile of *Flip* can be understood, without knowing any implementation details of either the *Stack_Template* or the *Flipping_Capability* enhancement.

To motivate the specifics of the particular performance expressions in the profile, we consider a concrete implementation of *Flip* in this subsection. The implementation contains concrete definitions for constants used in the SSCF profile, such as $SSCF_{F1}$ and $SSCF_{F2}$. The loop is annotated with the **maintaining** (loop invariant) and **decreasing** (progress metric) clauses necessary for an automated system to prove that the code satisfies its functional specification for flipping the Stack. In addition, the loop specification includes **elapsed time** and **manipulated displacement** expressions [11] needed to prove the correctness of the code with respect to its performance **profile**.

Due to space constraints, we present and analyze just the timing-related assertions. Since the code for *Flip* relies only on the specification of operations in the *Stack_Template* and not on any particular implementation, modular reasoning about the functional correctness of the code can be done regardless of the *Stack* implementation chosen.

Realization Obvious_F_C_Realiz **for** Flipping_Capability

with_profile SSCF **of** Stack_Template **with_profile** SSC;

Definition $SSCF_{F1}$: $\mathbb{R}^{\geq 0}$ = ($\mathbf{Dur}_{\mathbf{Call}}(1)$ + (SSC_{Dp} + Int.$\mathbf{Dur}_{\neq}$) + $\mathbf{Dur}_{:=:}$);

Definition $SSCF_{F2}$: $\mathbb{R}^{\geq 0}$ = (SSC_{Dp} + Int.$\mathbf{Dur}_{\neq}$ + SSC_{Po} + SSC_{Pu});

Definition $SSCF_{FMC1}$: $\mathbb{N}$ = ⋯

Definition $SSCF_{FMC2}$: $\mathbb{N}$ = ⋯

Procedure Flip(**updates** S: Stack);

Var Next_Entry: Entry;

Var S_Flipped: Stack;

While (Depth_of(S) ≠ 0)

affecting S, S_Flipped, Next_Entry;

maintaining $\#S = S\_Flipped^{Rev} \circ S$ **and**

Entry.**Is_Init**(Next_Entry);

decreasing |S|;

elapsed_time ($SSCF_{F2}$ + Entry.**I_Dur** + Entry.**F_IV_Dur**)·|S_Flipped|;

manip_disp ⋯

do

Pop(Next_Entry, S);

Push(Next_Entry, S_Flipped);

end;

S :=: S_Flipped;

end Flip;

end Obvious_F_C_Realiz;

2.3 Durational Analysis of Flip

The **duration** expression for *Flip*, in addition to a constant term $SSCF_{F1}$, has three parts: duration for local variable initialization, for local variable finalization, and for loop execution. First we assume that a *Stack* component with profile *SSC* is used. The duration expression to initialize the two local variables – an entry and a stack – is straightforward, and it is the sum of *Entry.**I_Dur*** and *Stack.**I_Dur***. Unlike initialization, the time for finalization of the two local variables depends on the values of the local variables at the time of finalization. Therefore, we need to understand what their values would be at the end of the code. Here, the Stack *S_Flipped* that is finalized is empty, because *S* is empty just before the swap statement. Therefore, the duration expression also includes the term *Stack.**F_IV_Dur*** – the time to finalize a stack with initial value. The local variable *Next_Entry* also has an initial value just before finalization. To

see why, notice that the loop maintains the invariant *Entry.**Is_Init**(Next_Entry)*, based on the extended **ensures** clause for the *Push* operation in the profile *SSC*, which in our version of Stack, guarantees that after *Push* the parametric Entry is initialized. Therefore, the duration of finalizing the Entry at the end of the code is *Entry.**F_IV_Dur*** – the time to finalize an entry with an initial value.

The loop executes |#S| times. The time for each iteration includes a constant term arising from calls to *Depth_of*, *Push*, and the loop branching activity. In addition, we note from the *SSC* profile that every call to *Pop(R, S)* takes time SSC_{Po} + *Entry.**I_Dur*** + *Entry.**F_Dur**(#R)*. In the code given above, the *Next_Entry* that is supplied to Pop is the entry resulting from the previous to call to *Push*. Since the ensures clause for *Push* in SSC profile guarantees that *Push* initializes its *Entry* parameter, we are guaranteed that *Pop* is only supplied initial entries in every call. Therefore, *Pop* needs to finalize only initial entries and the time for each call to *Pop* simplifies to SSC_{Po} + *Entry.**I_Dur*** + *Entry.**F_IV_Dur***. Given these considerations and the matching definitions of constants $SSCF_{F1}$ and $SSCF_{F2}$, the elapsed time estimate for the loop is documented in the implementation as:

$(SSCF_{F2}$ + Entry.**I_Dur** + Entry.**F_IV_Dur**$)\cdot|$S_Flipped$|$

2.3 Validity of the Elapsed Time Estimate

This elapsed time estimate is used in proving the performance correctness of *Flip*. A part of the proof that verifies that the given elapsed time estimate is valid is given in the table below.

State	Path Condition	Assume	Confirm
While (Depth_of(S) ≠ 0) **affecting** S, S_Flipped, Next_Entry; **maintaining** #S = S_Flipped$^{Rev}\circ$S **and** Entry.**Is_Init**(Next_Entry); **decreasing** |S|; **elapsed_time** ($SSCF_{F2}$ + Entry.**I_Dur** + Entry.**F_IV_Dur**)$\cdot|$S_Flipped$|$; **do**			
2	$\|S_2\| \neq 0$	Entry.**Is_Init**(Next_Entry$_2$) ∧ **ET**$_2$ = ($SSCF_{F2}$ + Entry.**I_Dur** + Entry.**F_IV_Dur**)$\cdot|$S_Flipped$_2|$ ⋯	…
Pop (Next_Entry, S);			
3	$\|S_2\| \neq 0$	$S_2 = S_3\circ$<Next_Entry$_3$> ∧ S_Flipped$_3$ = S_Flipped$_2$ ∧ **ET**$_3$ = **ET**$_2$ + (SSC_{Po} + Entry.**I_Dur** + Entry.**F_Dur**(Next_Entry2)) ⋯	…
Push (Next_Entry, S_Flipped);			
4	$\|S_2\| \neq 0$	Entry.**Is_Init**(Next_Entry$_4$) ∧ $S_4 = S_3$ ∧ S_Flipped$_4$ = S_Flipped$_3\circ$<Next_Entry$_3$> ∧ **ET**$_4$ = **ET**$_3$ + SSC_{Pu} ⋯	…
Confirm ET4 = ($SSCF_{F2}$+ Entry.**I_Dur** + Entry.**F_IV_Dur**)$\cdot|$S_Flipped$_4|$ ∧ ⋯			
end;			

The table shows only a part of an inductive proof: verification conditions corresponding to the inductive potion of the proof to confirm the invariance of the elapsed time estimate. In the table, which is based on [26]], we **assume** at the beginning of the loop (numbered state 2 in the figure) the elapsed time estimate holds. We then **confirm** at the end of the loop (state 4) that the estimate when evaluated there is correct. The assumptions in states 3 and 4 come from the functional and performance specifications of operations *Push* and *Pop*. Variable names are subscripted with the state number to distinguish their values in different states. The *verification variable* **ET** stands for the elapsed time. Given the assumptions, a verifier can conclude that **ET**$_4$ satisfies its equation if **ET**$_2$ satisfies its equation. We have omitted the base case for the inductive proof, assertions outside the loop, and functionality-related assertions, not necessary for the above proof.

3. SCALING UP

Two important scalability questions arise in generalizing the utility of the profile construct:

1. Can profiles for layered components be expressed abstractly?
2. How complicated will profiles get when components are used to put together a layered system?

To address these questions we designed and specified a *spanning forest* component that we built using a *prioritizer* and a *coalescable equivalence relation* component, among others, and specified all components fully for both functionality and performance.

We answer the first question affirmatively noting that it was possible to write a fully descriptive profile for the top layer of the system without filling in the details for the components upon which it was layered.

The second question is one of concern, since the stack example may give the impression that the number of lines of specifications in a profile may approach the number of lines of executable code. However, we note that the stack component has an unusually small number of lines of code, and that the complexity of the profile is dominated by its parameterization. Moreover, although it may seem counter-intuitive, it turns out that when layering up, the profile for a higher-level component is usually no longer than that for a lower level one, while the aggregate number of lines of executable code grows considerably. For example, in the case of the spanning forest, the ratio of lines of performance specification to executable code is closer to one to three, rather than one to one, indicating that the depth of layering in a system is not an indicator of the need for longer profiles.

Our research has also shown that the profile construct is essential for documenting concisely the various performance specifications of a layered component, such as the spanning forest component, that result when alternatives are considered for the performance of a constituent component such as the prioritizer.

4. RELATED WORK AND DISCUSSION

The importance of performance considerations for software engineering (e.g., [4], [14], [17]), in general, and for software components, in particular, has been widely acknowledged. Designers of languages and developers of component libraries have emphasized the need for alternative implementations in order to provide performance trade-offs [3], [16], [18]. The importance

of generic programming and of alternative implementations is being increasingly recognized, as is evident from the evolving designs of C#, C++, and Java.

In order for component users to choose from multiple implementations and analyze performance of component-based systems in a modular fashion, a formal system for performance specification is necessary. Balsamo, et al., in surveying various efforts in performance analysis [2], note that "Although several of these approaches have been successfully applied, we are still far from seeing performance prediction integrated into ordinary software development" and conclude that one of the unresolved problems is the lack of software notations that allow for easily expressing performance. The profile construct proposed here for extending specification (and programming) languages to support specifying performance is a contribution to integrating performance considerations into software development.

A general performance specification system should be flexible, allowing specifiers to express performance in terms of abstractions that are appropriate for the problem at hand. This emphasis on abstraction and generic components in specifying both time and space usage of components also makes the ideas discussed in this paper quite different from the work in the real-time community (e.g., [7], [23]) where timing deadlines and concurrency are the focus.

Expression of tight timing constraints is an active area of research [6], [15]. Elsewhere, we have detailed how the expressiveness issues that arise in tight specification of performance at the source code level can be addressed using intermediate abstraction models [28].

Hehner has built on the work of Shaw [22], to formalize time and space analysis of a recursive procedure at the source code level [9]. Our earlier work and the work of Schmidt and Zimmermann [21] have considered space complexity issues for components. Working within the context of functional programs Unnikrishnan, et al. and Hofmann and Jost have addressed issues in bounding the space usage of functional programs under various assumptions using program-level source code analysis [10], [27]. Ultimately, compositional performance analysis needs to be combined with advances in verification of functional behavior in the presence of data abstractions (e.g, [5], [19], [25], [26]) because assertions from functional correctness are necessary for establishing performance correctness.

We have introduced profiles as a first class language construct for modular specification and analysis, providing a vocabulary for stating time and space constraints. The construct supports both generics and compositionality. Based on the construct, as Atkey [1] has shown recently, mechanisms for other behavioral specification language and implementation language combinations can be developed, provided the particulars of the language features are carefully accommodated in specifications.

5. ACKNOWLEDGMENTS

Several members of our research groups have contributed important ideas to this work. Our special thanks are due to Gary Leavens and Bruce Weide for their comments. We gratefully acknowledge financial support from the U.S. National Science Foundation under grant CCR-0113181 and a grant from the U.S. National Aeronautics and Space Administration through the SC Space Grant Consortium.

6. REFERENCES

[1] Atkey, J., "Specifying and Verifying Heap Space Allocation with JML and ESC/Java2", *Proceedings of the ECOOP Workshop Formal Techniques for Java-like Programs*, Nantes, France, July 2006; available at: http://www.disi.unige.it/person/AnconaD/FTfJP06/

[2] Balsamo, S., Di Marco, A., and Inverardi, P., "Model-Based Performance Prediction in Software Development: A Survey", *IEEE Transactions on Software Engineering,* 30(5), May 2004, 67-82.

[3] Booch, G. *Software Components With Ada.* Benjamin/Cummings, Menlo Park, CA, 1987.

[4] Cheng, A. M. K., Clemens, P., and Woodside, M., eds. Special section: Workshop on Software and Performance. *IEEE Trans. on Software Engineering 26*, 11/12, November/December, 2000.

[5] Ernst, G. W., Hookway, R. J., and Ogden, W. F., "Modular Verification of Data Abstractions with Shared Realizations", *IEEE Transactions on Software Engineering 20,* 4, April 1994, 288-307.

[6] Gomez, G. and Liu, Y. A., "Automatic time-bound analysis for a higher-order language," *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '02*), Portland, Oregon, USA, January 14-15, 2002, ACM SIGPLAN Notices 37(3), March 2002.

[7] Hayes, I.J. and Utting, M., "A Sequential Real-Time Refinement Calculus," Acta Informatica 37, 2001, 385-448.

[8] Harms, D.E., and Weide, B.W., "Copying and Swapping: Influences on the Design of Reusable Software Components," *IEEE Transactions on Software Engineering,* Vol. 17, No. 5, May 1991, 424-435.

[9] Hehner, E. C. R., "Formalization of Time and Space," *Formal Aspects of Computing*, Springer-Verlag, 1999, 6-18.

[10] Hofmann, M. and Jost, S., "Static Prediction of Heap Space Usage for First-Order Functional Programs," *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2003, 185-197.

[11] Krone, J., Ogden, W. F., and Sitaraman, M., "Modular Verification of Performance Correctness", *SAVCBS Workshop Proceedings*, 2001, 60-67.

[12] Krone, J., Ogden, W.F., "Abstract OO Big O," *SAVCBS Workshop Proceedings*, 2003, 80-84.

[13] Leavens, G. T., Baker, A. L., and Ruby, C., "JML: A Notation for Detailed Design," *Behavioral Specifications of Businesses and Systems*, H. Kilov and B. Rumpe and I. Simmonds, eds., Kluwer Academic Publishers, Boston, 1999.

[14] Leavens, G.T., Abrial, J., Batory, D., Butler, M., Coglio, A., Fisler, K., Hehner, E., Jones, C., Miller, D., Peyton-Jones, S., Sitaraman, M., Smith, D.R., and Stump, A.: Roadmap for Enhanced Languages and Methods to Aid Verification. Department of Computer Science, Iowa State University, TR #06-21. July 2006.

[15] Lim, S-S, Bae, Y. H., Jang, G. T., Rhee, B-D, Min, S. L., Park, C. Y., Shin, H., Park, K., Moon, S-M, and Kim, C. S., "An accurate worst case timing analysis for RISC processors," *IEEE Transactions on Software Engineering,* Vol. 21, No. 7, July 1995, 593 - 604.

[16] Meyer, B., *Object-Oriented Software Construction,* Prentice Hall PTR, Upper Saddle River, New Jersey, 1997.

[17] Meyer, B., "The Grand Challenge of Trusted Components," *Procs. 25th Int. Conference on Software Engineering*, Portland, OR, May 2003, 660-667.

[18] Musser, D.R., Derge, G.J., and Saini, A. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, 2001.

[19] Muller, P. and Poetzsch-Heffter, A., "Modular Specification and Verification Techniques for Object-Oriented Software Components," in *Foundations of Component-Based Systems*, eds. G. T. Leavens and M. Sitaraman, Cambridge University Press, 2000.

[20] Ogden, W.F., *The Proper Conceptualization of Data Structures,* Dept. Computer and Information Science, Ohio State University, 2000.

[21] Schmidt, H. and Zimmermann, W., "A Complexity Calculus for Object-Oriented Programs," *Journal of Object-Oriented Systems*, 1994, 117-147.

[22] Shaw, M., *A Formal System for Specifying and Verifying Program Performance*, Carnegie-Mellon University Technical Report CMU-CS-79-129, June 1979.

[23] Shaw, A. C., Reasoning About Time in Higher-Level Language Software, *IEEE Transactions on Software Engineering 15*, 1989, 875-889**.**

[24] Smith, C. U., *Performance Engineering of Software Systems*, Addison-Wesley, 1990.

[25] Sitaraman, M., Ogden, W.F., and Weide, B.W., "On the Practical Need for Abstraction Relations to Verify Abstract Data Type Representations," *IEEE Trans. Software Eng 23*, 3, Mar. 1997, 157-170.

[26] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W., Long, T. J., Bucci, P., Heym, W., Pike, S., and Hollingsworth, J. E., "Reasoning About Software-Component Behavior," *Procs. Sixth Int. Conf. on Software Reuse*, IEEE Computer Society, 2000.

[27] Unnikrishnan, L., Stoller, S. D., and Liu, Y. A., "Automatic Accurate Live Memory Analysis for Garbage-Collected Languages," *Procs. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2001.

[28] Weide, B. W., Ogden, W. F., and Sitaraman, M., "Expressiveness Issues in Compositional Performance Reasoning," *Procs. Sixth ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, Portland, OR, May 2001, 85 - 90

Performance Modeling of a JavaEE Component Application using Layered Queuing Networks: Revised Approach and a Case Study

Alexander Ufimtsev
Performance Engineering Laboratory
School of Computer Science and Informatics,
University College Dublin, Belfield, D4, Ireland
alexu@ucd.ie

Liam Murphy
Performance Engineering Laboratory
School of Computer Science and Informatics,
University College Dublin, Belfield, D4, Ireland
Liam.Murphy@ucd.ie

ABSTRACT

Nowadays component technologies are an integral part of any enterprise production environment. Performance and scalability are among the key properties of such systems. Using Layered Queuing Networks (LQN), one can predict the performance of a component based system from its design. This work revises the approach of using LQN templates, and offers a case study by using the revised approach to model a realistic component application.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*; D.2.1 [**Software Engineering**]: Requirements/ Specifications—*Methodologies*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; D.2.9 [**Software Engineering**]: Management—*Software quality assurance*

Keywords

performance modeling, JavaEE, component systems, ECPerf, Layered Queuing Network.

1. INTRODUCTION AND MOTIVATION

Many large software development projects fail to deliver the product on time, within budget, and with satisfactory QoS. Useful software engineering practices such as model checking, verification, and continuous testing help satisfy the functional requirements of the projects. However, some of the non-functional requirements can only be checked when integrated with other components and during system testing, which is typically done during the final stages of development. *Performance* is one of the non-functional requirements that is commonly difficult to check outside a test environment.

Software products nowadays include various components developed by third parties and running on a stack of multiple software layers. Such complexity makes it difficult to provide software performance guarantees - moving functionality to middleware and third party components reduces the overall understanding of the system. Nowadays 'buggy' and poorly implemented code contributes less problems to enterprise software development than a problematic architecture or "short-sighted design", as well as poor capacity planning [9]. Unfortunately, design problems are not easily resolvable at the final development stages. Therefore, a system designer should be able to assess the performance properties of the system (both software and hardware) early. He or she needs to 'plan ahead', leaving room for possible future improvements and requirement changes.

Use of Layered Queuing Networks for modeling of component-based software can help system designers to assess the outcome of performance decisions, starting from very early stages of development. This work builds upon 'layered queuing network templates' [8, 11] by analyzing key strengths and weaknesses of the approach and also by conducting a case study on a realistic application.

2. LQN OVERVIEW

Layered Queuing Networks (LQN) is an extension of Queuing Networks that allows the software to be represented in layers, thus separating resources from a model and dividing a model into multiple submodels. LQN is abstract enough not to suffer from drawbacks of other popular formalisms, notably Stochastic Petri Nets (SPN).

Basic LQN notation consists of three basic elements: *circles* denote resource type (CPU, disk, network), *rectangles* - software blocks. The right rectangle usually denotes an object/bean, while rectangles to the left from it denote an action happening with that object, such as method or function call. Also, control rectangle usually indicates the type of resource it utilizes. *Arrows* depict control flow within the network.

Key advantages of LQN:

- Layered architecture of LQN 'naturally' maps to componentized, multi-tiered, and multi-layered enterprise level software stacks;
- LQN is easily extend able to include newly discovered

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006), November 10-11, 2006, Portland, Oregon, USA.

bottleneck resource or device into existing model;

- LQN avoids state explosion as some other formalisms (SPN-like) via higher level of abstraction;
- LQN is a formalism and provides robust analytical and simulation-based tools.

LQN's disadvantages:

- Does not allow any type of dynamism in the running system, provides only steady state solutions;
- Does not have a notion or possibility to model timeouts, locks, and workload variation within a run. To introduce some sort of dynamism to LQN one must have to solve LQN for different states first and then bind them together with a form of Markov chain, with probability of the system changing its state from one to another.
- Requires a lot of data to be accurately collected, interpreted, and put into a model to produce accurate results.

3. PROBLEM STATEMENT AND APPROACH

The purpose of this work was to understand how suitable were LQN templates for modeling JavaEE applications and what improvements to the approach are needed. Java Enterprise Edition (JavaEE) is a superset of Java Standard Edition (JavaSE), designed for multi-tier solutions [6]. It provides developers with the underlying infrastructure required by the enterprise systems. J2EE's core is a family of component models: on the client side, JavaBeans and applets; on the web server tier, J2servlets and Java Server Pages (JSPs); on the application server tier Enterprise JavaBeans (EJB).

Java EE's ECperf application was selected to be modeled for a number of reasons. First, ECperf is an industry-standard JavaEE benchmark meant to measure the scalability and performance of JEE servers and containers. It stresses the ability of EJB containers to handle the complexities of memory management, connection pooling, passivation/activation, and caching. ECperf is highly portable and runs on majority of application servers, which makes it perfect for future evaluation of the approach on different software and hardware configurations. Second, ECPerf creators stress its ability to represent real-life business applications. ECPerf is designed as a typical web business application that permits customers direct specification of product configuration, ordering, and status checking. It also automates manufacturing, inventory, supplier chain management, and customer billing. Third, as a benchmark application it provides necessary workload drivers and all the performance data aggregators in useful and professional manner.

4. APPLICATION DESIGN

Originally developed by Sun Microsystems, ECPerf is now being developed and maintained by SPEC Corporation[1]. It is currently available from SPEC under the name of *SPECjAppServer2004*. ECPerf is designed to be a typical enterprise application. It has four implemented domains in its code: Manufacturing, Supplier & Service Provider, Customer, and Corporate. Each domain has separate database and applications. They provide the foundation for the ECperf workload. Customers contact the business through any number of methods, including directly through the web. All of the worldwide offices and plants make frequent access to data held in the other offices or plants, and must, at times, compare/collate/verify their data against that held worldwide. The company also interacts with completely separate supplier companies. Each supplier has its own independent set of computing resources. The overall setup can be seen in Figure 1.

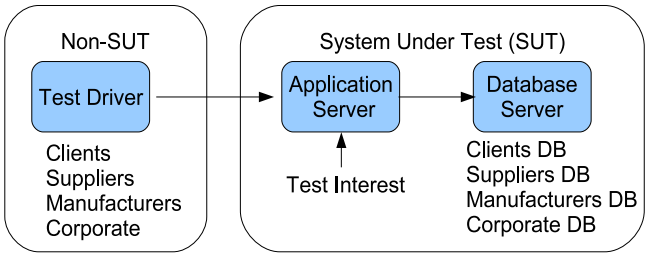


Figure 1: High-Level Overview of Testing System

Customer Domain

This domain emulates business interactions with the clients. Clients can be of two types: individuals and distributors. Both interact with by placing orders. The difference between the two is in the quantity of items ordered. Approximately 57% of work on the system is related to orders from distributors (i.e contain large number of items), 43% is from individual customers. Customer domain implementation of ECperf contains seven beans: OrderSes, OrderEnt, OrderLineEnt, ItemEnt, OrderCustomerSes, OrderCustomerEnt, and CartSes.

Manufacturing Domain

The manufacturing domain emulates business product lines, which process the orders received by customer domain. There are two types of production lines: Planned lines and Largeorder lines. The planned lines run on schedule and produce a pre-defined number of widgets. On the other hand, the largeorder lines run only when a large order is received from a customer such as a distributor. This domain is implemented with ten beans: WorkOrderSes, LargeOrderSes, ReceiveSes, PartEnt, AssemblyEnt, WorkOrderEnt, LargeOrderEnt, ComponentEnt, InventoryEnt, and BOMEnt.

Supplier Domain

The Supplier Domain decides which supplier to choose based on the parts that need to be ordered, the time in which they are required and the price quoted by suppliers. It is implemented in the system with seven beans: BuyerSes, ReceiverSes, SupplierEnt, SupplierComponentEnt, POEnt, POLineEnt, and SComponentEnt.

[1]http://www.spec.org

Table 1: Top cumulative time-consuming methods

Domain	Name	Time (%)
mfg	WorkOrderEnt.process	19
mfg	WorkOrderCmpEJB.process	19
mfg	WorkOrderSesEJB.scheduleWO	16.1
supplier	BuyerSes.purchase	5.3
supplier	BuyerSesEJB.purchase	5.3
mfg	ComponentEnt.takeInventory	5.3
orders	OrderSesEJB.newOrder	5.3
orders	OrderEntHome.create	5.3
orders	OrderCmpEJB.ejbCreate	5.3
mfg	WorkOrderSesEJB.scheduleWO	4.8

Table 2: Top average method time

Domain	Name	Time (%)
mfg	WorkOrderEntHome.create	0.4
orders	OrderCustomerEntHome.create	0.2
supplier	ReceiverSesHome.create	0.1
mfg	WorkOrderEntHome.create	0.1
orders	OrderCustomerEntHome.findByPK	0.1

Corporate Domain

This domain manages the global list of customers, parts and suppliers and is implemented in three beans: CustomerEnt, DiscountEnt, RuleEnt. The domain is used for obtaining customer credit status, various discounts and billing.

5. PERFORMANCE AND CODE ANALYSIS

While performing analysis it is important to concentrate on so-called "critical paths" within the system. Critical paths in ECPerf were obtained by analysing its source code and execution traces while running the system with minimal possible workload. The latter was necessary to avoid abnormal behavior due to monitoring overhead and clarity of the derived call graph. It should be noted that despite the use of smallest possible workload the amount of users in the system was equal to eight. Therefore, concurrent resource usage behavior was preserved. The source code was analysed using Juliet [2], while execution graph and times were obtained with JProbe [3] tool. The measurements had shown that that only two domains make a significant performance impact: cumulative execution time of beans in manufacturing and customer domains consume around 80% of overall ECPerf execution time (See Table 1). The other two domains did not seem to have any significant influence on the overall system performance. Also, Table 2 shows top five non-cumulative averaged method times. It can be noted that even the worst performing individual methods use very little resources on their own. Since a lot of complex services was pushed to middleware, what becomes important is the "orchestration" of services provided by other methods and containers.

5.1 Workload Generation

[2]http://infotectonica.com/
[3]http://jprobe.quest.com

ECperf workload is generated by so-called 'Driver' script, that runs 'agents'. Standard ECperf configuration features three agents, one per domain: *ordersAgent* (customers), *mfgAgent* (manufactures), and *loAgent* (large orders). Strictly speaking, *loAgent* is not a new domain, but rather belongs to both customers and manufactures. Agents, in their turn, start up and control the client instances that issue requests to the servers (See Figure 2). The number of planned lines and order lines depends on scale parameter.

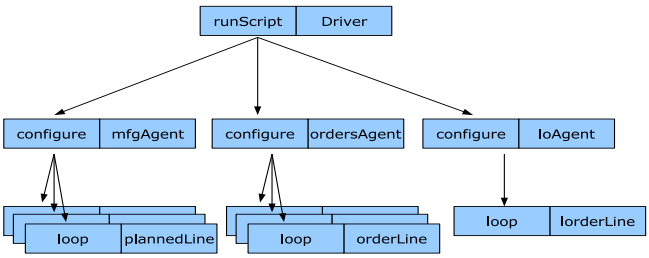


Figure 2: ECperf Workload Overview

All the loops wait for a certain period of time. It is hard-coded for one second for large order agent, configured to 100 ms for mfgAgent and for orderEntry it is set to a negative exponential distribution

$$Tc = -ln(x)/I_r \quad (1)$$

where ln - natural log (base e), x - random number with at least 31 bits of precision, I_r - mean Injection Rate.

Figure 3 shows an overview of manufacturing activity in LQN notation. System sleep is necessary to introduce emulation of activity stations that gradually change workorder state from 'scheduled' to 'updated', and finally to 'completed'. *WorkOrderSes* session bean also creates, searches, and updates a few entity beans.

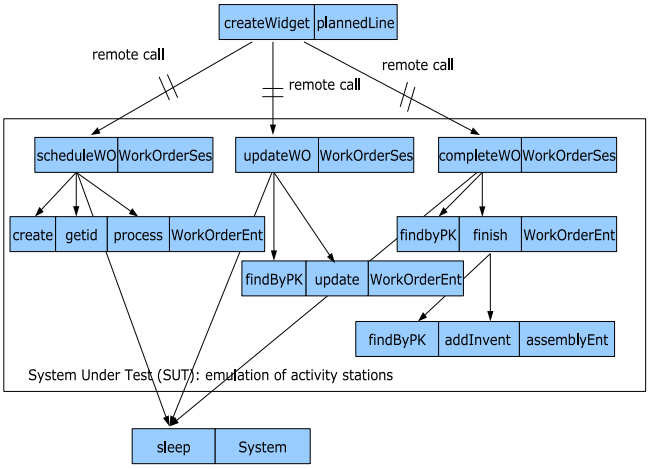


Figure 3: High-level overview of Manufacturer System Activity

Large orders agent just pulls information from the database about large orders once per second and updates its statistics

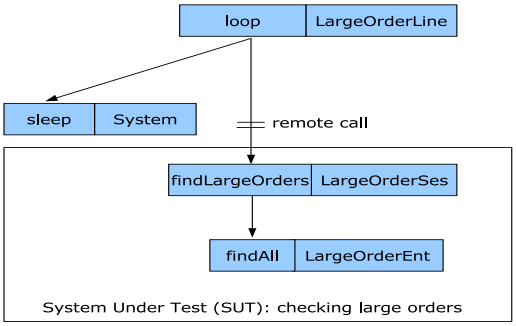


Figure 4: Large Orders Agent activity

Order entry represents a significant amount of system activity. The stream of order entry transactions is split in the following way:

- newOrder - 50%
- getOrderStatus - 20%
- changeOrder - 20%
- getCustStatus - 10%

There is also 10% chance that order is going to be large, 50% chance that people buy goods right away and another %50 - that they will use a shopping cart. In 90 percent of cases, people *delete* the content of their shopping cart. The Metric for the Customer Domain is Transactions/min. The metric for the Manufacturing Domain isWorkorders/min, whether produced on the Planned lines or on the LargeOrder lines. The numbers described are based on real world experience of Sun Microsystems building typical enterprise applications. This particular transaction mix and probabilities of state changes is built into ECPerf and should not be altered.

newOrder call path is presented in Figure 5. Please note that other paths, such as *changeOrder*, *orderStatus*, *customerStatus*, and *createNewShoppingCart* were not expanded due to cluttering of space. Numbers on call arrows show probabilities.

5.2 Addressing Ambiguity

A few features or implementation details of ECperf presented a modeling challenge. First, since ECperf was designed after typical web-based enterprise applications, it intentionally did not keep the database size stable. In fact, it kept growing throughout each test and had to be reinitialized before a new one. Second, even the initial database size, *e.g.* the initial number of customers and products depends on the expected (configurable) workload. This is another reason why databases have to be swept clean and repopulated with data upon startup of another test. Third, ECperf authors made some provisions for unstable environment. If a transaction fails for whatever reason (timeouts, database lock, *etc.*), ECperf handles this exception and retries it from 5 to 20 times before failing.

All of the above uncertainties had to be abstracted in order to keep the scope of the use case feasible. Database size for the model was chosen as an average of the real database size before and after test run. We presume this is correct since database never becomes a bottleneck device throughout the test. We also had to average in all the retries and transaction rollback that happened.

The rest of the testsuit was modeled using the following refined principles:

- *Communication* is broken into two types: local and remote. Remote one is modeled with a network resource/processor. Local calls that don't exercise Remote Method Invocation (RMI) are modeled as simple LQN calls without any resource consumption.
- *Container services* is an aggregate term for any additional activities performed by container. It is not modeled separately, but spread across execution times.
- *Reflection* - included in container services times.
- *Connection pooling* - two essential queues are modeled: container threads and database threads.
- *Transaction manager* - again, included in overall container services for model simplicity. No specific model for the actually transaction rollback is specified.
- *Security* - security checks are modeled with submodels of beans.
- *Garbage Collection (GC)* is not addressed in the current version of model, since the complexity and closed source code of JVM makes it hard to derive correct models. GC time is generally spread across container services.
- *Naming* - modeled through container services
- *Database* - modeled with an average response time. It is possible, however, to model it with any specified distribution of response times.

6. MODELING APPROACH

The proof of concept use of LQN for modeling of EJB-based applications has been demonstrated by Xu *et al.* [11]. Produced LQN-EJB templates can be instantiated according to specific function requirement in each scenario for system usage, and then be assembled into a complete LQN model for the whole scenario. General information on functional aspects of EJB technology as well as specific models for each bean type were presented in [11].

A system is modeled by presenting the beans as tasks with estimated parameters, then instantiating the template to wrap each class of beans in a container, and finally adding the execution environment including the database. Calls between beans, and calls to the database, are part of the final assembly. The model may be calibrated from running data, or by combining

- knowledge of the operations of each bean;
- pre-calibrated workload parameters for container services, communication, and database operations.

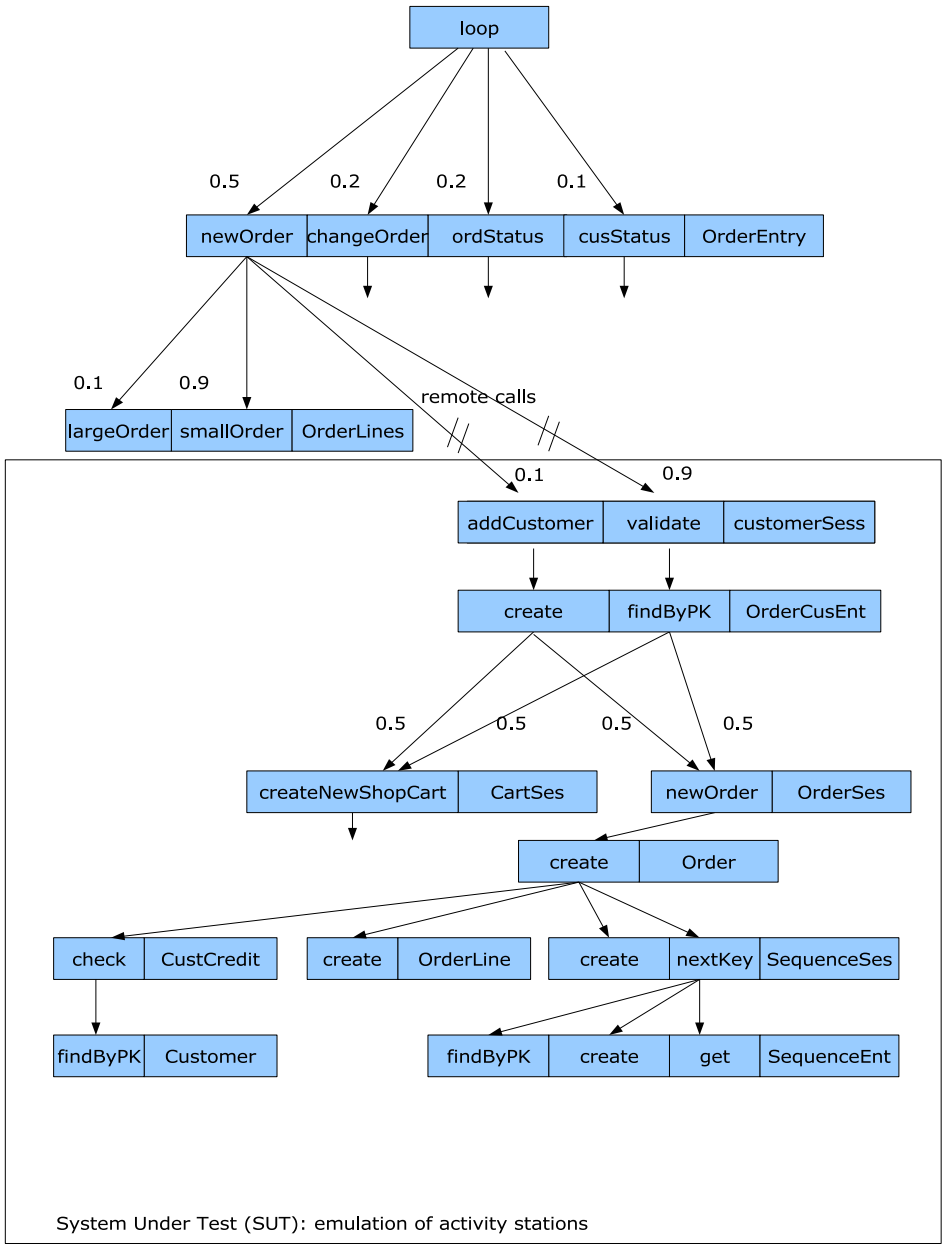


Figure 5: Part of the Orders Domain

7. MODEL CALIBRATION AND TESTING

7.1 Hardware platform

The testing environment includes four x86 machines:

- *application server* Pentium III-866 Mhz with 512 Mb RAM;
- *database* Pentium III-800Mhz with 512 Mb RAM; and client
- *client* Pentium IV-2.2 Ghz, 1024 Mb RAM.

The client machine is more or as powerful as servers to ensure it does not become a bottleneck when generating the test load.

7.2 The software environment

The following software was used for testing purposes:

- *operating system*: Debian GNU/Linux 3.1 'sarge', kernel v 2.6.8-3;
- *database server*: MySQL v. 5.0.7beta-1;
- *application server*: JBoss v. 4.01sp1;
- *JVM*: Java2SDK 1.4.2_09.

Measurements on the container and program execution were obtained by running JProbe 5.2.1 Freeware profiler for Linux. The following options were used for JVM startup:

- The initial Java heap size was $480MB$;
- parameter *-XX:+PrintCompilation* was set to monitor the runtime behavior of the JVM.

7.3 Benchmarking results

Performance benchmarking of ECPerf produced the following results (See Figure 6). The X-axis shows the value of SCALE parameters while Y-axis shows the number of business operations per minute (BBops/min).

Scale value is not equal to number of users, but rather $user\_num = f(scale)$. In our case, $user\_num = 5 * scale$. The minimal number of users in the system is 5, and the maximum measured is 250. The second line of in benchmarking figure shows standard deviation of the results obtained. It can be noted that results become quite unstable once SCALE goes over 10, while overall performance does not seem to increase. Also, once load reaches 50 users (SCALE=10) the system starts producing errors (time outs, etc) due to overload. Therefore, system achieves its peak performance with SCALE=7, or 35 users. Figure 7 shows response times for Manufacturing and Orders for the respective workload.

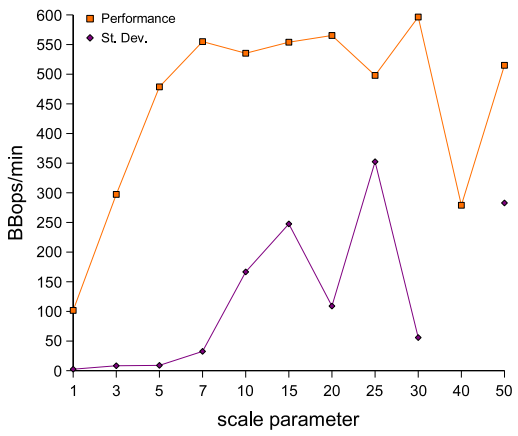


Figure 6: Results of ECPerf benchmarking

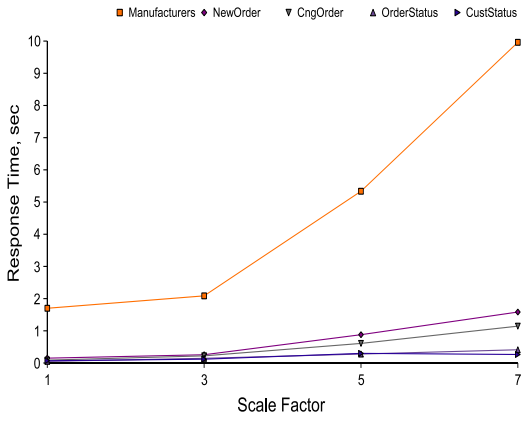


Figure 7: Response Time of ECPerf

7.4 Resource Utilization

Various resource were used during benchmarking, including CPUs of the test machines, network, HDDs, etc. Except for SCALE=1, the CPU of the application server was the bottleneck. CPU usage of client and db machines reached 17 percent at max with average utilization of 5-7 per cent. Network utilization was around 1%. Disk usage for both database and application server was also negligible.

7.5 Model Calibration

The model constructed in Section 7 was calibrated from the profiling data under a minimal workload. During the measurement phase JProbe profiling tool introduced significant overhead, so the execution demand values extracted from profiling data are adjusted to remove the contribution of overhead. This was done by using a *Profiling Ratio Factor* (PFC) based on the assumption that the profiling overhead is proportionally distributed across the operations within some section of the scenario. The factor was obtained for each section by measuring the service time with and without profiling and taking the ratio. For the configuration of Jboss, PFC varied from 1 (for low-level operations) to 7.49 for business method related operations. Ultimately, when profiling was on and every method call in JBoss and ECperf was logged, the response times slowed down 7.49 times. By proportionally dividing each response time obtained with the monitoring on PFC, we can get averaged execution times for individual methods. It should be noted that this approach becomes highly inaccurate for monitoring a lot of methods simultaneously. The best results are shown when $PFC \rightarrow 1$, which happens when monitoring is turned on for a very small number of methods 1...10.

The problems also included the fact that the cycle times in ECPerf were very dynamic and depended on the response time.

7.6 Result Analysis

Calibrated LQN simulation gave quite close results to the tested real-life configuration (See Figure 8). The upper line is a modeled result, while the lower - a real system test, which is identical to Figure 6 on the scale from 1 to 7. The modeled results were aggregated using predefined data in Subsection 5.1.

For small and average workloads the LQN results were a bit more pessimistic than real ones, but they are quite close. We consider that a good result taking into account that measurements for the model where conducted on the system only once with minimum workload possible, *e.g.* $SCALE = 1$. However, at the higher loads LQN result becomes more optimistic. The worrying trend is that LQN model continues to predict higher performance than the real system. At $SCALE = 7$ the real system reaches its optimal workload and its throughput stabilises. The LQN model shows almost linear increase in predicted performance. We were unable to identify the reasons for such behavior, though we noticed that the variance for the overall response time in the model greatly increased when the workload reached its peak. This could be due to lack of locking in LQN, or a missed bottleneck in the model. The most likely 'offenders' are transaction and security features of application server, and database record locking. We suspect that when the load reaches its peak, some transactions might timeout and be subsequently retried. Since transactions are resource-expensive, it might deviate the system's throughput away from the 'ideal' situation as demonstrated by the model.

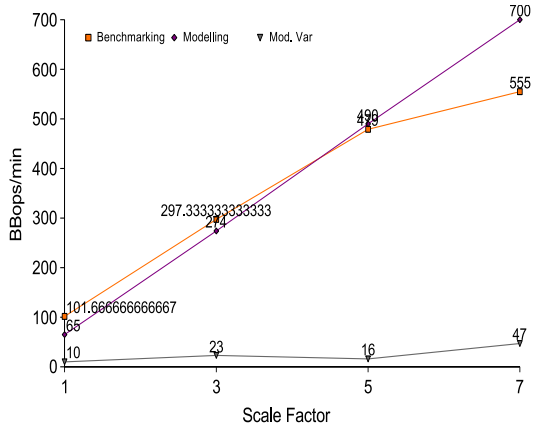


Figure 8: ECPerf Response Time

8. RELATED WORK

The following solutions have been offered so far. Lui *et al.* [7] proposed a method for predicting performance of J2EE applications at design level, which seems to rely mostly on profiling information instead of models. Denaro *et al.* [2] came up with generic framework for performance testing of distributed component architectures. The evaluation of the approach is not formal and based on creating a stub application, which then is run in the real environment.

9. CONCLUSIONS AND FUTURE WORK

We have presented an evaluation of a Layered Queuing Network (LQN) templates approach by building a model of ECPerf - a JavaEE component application. Various modeling problems have been addressed within the limitations allowed by LQN formalism. We showed that despite our efforts, ECperf performance prediction model was overoptimistic when compared to a real system. This suggests a missed resource congestion point due to inaccurate modeling assumptions or LQN limitations.

We plan to improve the existing model by adding JavaEE container services, such as transaction & security. This way we hope to eliminate the current inaccuracies within the LQN templates model. Additional services outside JVM layer can also be added to the model. For instance, Virtual Memory Manager was shown to have a significant effect on component-based application performance [10].

10. ACKNOWLEDGMENT

The support of the Informatics Commercialization initiative of Enterprise Ireland is gratefully acknowledged.

11. REFERENCES

[1] Cecchet, E., Marguerite, J., Zwaenepoel, W.: Performance and Scalability of EJB Applications Proc of 17th ACM Conference on Object-Oriented Programming, Seattle, Washington, (2002).

[2] Denaro, G., Polini, A., and Emmerich, W.: Early performance testing of distributed software applications Proc of the Fourth international Workshop on Software and Performance, Redwood Shores, California, (2004) 94-103

[3] Descripton of LQN XML Schema http://www.sce.carleton.ca/rads/lqn/lqn-documentation/schema/

[4] ECperf Kit Sun Microsystems http://java.sun.com/developer/releases/j2ee/ecperf/

[5] Gorton, I., Liu, A.: Performance Evaluation of Alternative Component Architectures for Enterprise JavaBean Applications in IEEE Internet Computing, vol.7, no. 3, pages 18-23, 2003

[6] Java Enterprise Edition Sun Microsystems http://java.sun.com/javaee/

[7] Liu, Y., Fekete, A., Gorton, I.: Predicting the Performance of middleware-based applications at the design level Proc of Fourth International Workshop on Software and Performance, Redwood Shores, California (2004) 166-170.

[8] Oufimtsev, A. and Murphy, L.: Predicting Performance of EJB-based Systems Using Layered Queueing Networks Proc. of OOPSLA conference, ACM, Oct. 2004 (poster)

[9] The State of J2EE Application Management: Analysis of 2003 Benchmark Survey Survey Analysis by Ptak, Noel & Associates http://ptaknoelassociates.com/members/J2EE_app_mgmt_survey.pdf

[10] Ufimtsev, A., Murphy, L., Kucharenka A.: Impact of Virtual Memory Managers on Performance of J2EE Applications In Proceedings of Component-Based Software Engineering (CBSE) conference, Vesteras, Sweden, June 2006

[11] Xu, J., Oufimtsev, A., Woodside, M., Murphy, L.: Performance Modeling and Prediction of Enterprise JavaBeans with Layered Queuing Network Templates Proc of SAVCBS Workshop, FSE, Lisbon (2005)

Soundness and Completeness Warnings in ESC/Java2

Joseph R. Kiniry, Alan E. Morkan and Barry Denby
School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland

ABSTRACT

Usability is a key concern in the development of verification tools. In this paper, we present an usability extension for the verification tool ESC/Java2. This enhancement is not achieved through extensions to the underlying logic or calculi of ESC/Java2, but instead we focus on its *human interface* facets. User awareness of the *soundness* and *completeness* of the tool is vitally important in the verification process, and lack of information about such is one of the most requested features from ESC/Java2 users, and a primary complaint from ESC/Java2 critics. Areas of unsoundness and incompleteness of ESC/Java2 exist at three levels: the level of the underlying logic; the level of translation of program constructs into verification conditions; and at the level of the theorem prover. The user must be made aware of these issues for each particular part of the source code analysed in order to have confidence in the verification process. Our extension to ESC/Java2 provides clear warnings to the user when unsound or incomplete reasoning may be taking place.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software VerificationÑprogramming by contract; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Design, Languages, Theory, Verification

Keywords

Extended Static Checking, Java Modeling Language, JML, Soundness, Completeness

1. INTRODUCTION

ESC/Java2 [7] is a programming tool that attempts to partially verify JML [3] annotated Java programs by static analysis of the program code and its formal annotations. Users can control the amount and kinds of checking that ESC/Java2 performs by annotating their programs with specially formatted comments called pragmas.

In order for the user to have confidence in the verification process, s/he must have confidence in the feedback from the tool. However, ESC/Java2 is neither sound nor complete. ESC/Java2 being unsound means that it emits false positives. That is, it misses errors that are actually present in the program it is analysing. As ESC/Java2 is an extended static checker rather than a program verifier, some areas of unsoundness are incorporated into the checker by design, based on intentional trade-offs of unsoundness with other properties of the checker, such as efficiency and the frequency of false alarms. ESC/Java2 being incomplete means that it emits false negatives. That is, it warns of potential errors when it is impossible for these error to occur in any execution of the program. Since ESC/Java2 attempts to check program properties that are, in general, undecidable, some degree of incompleteness is inevitable. In addition, the developers of ESC/Java2 were willing to accept some avoidable areas of incompleteness in order to improve performance and to keep the tool simple.

This paper presents an extension to ESC/Java2 that attempts to improve the usability of the tool by providing warnings in cases where the reasoning of the tool is either unsound or incomplete. These warnings should give the user greater confidence in using ESC/Java2.

Unfortunately, such user interaction and feedback is very rarely incorporated in static analysis tools, and in formal methods tools in general. Indeed, there is very little published related work in this field. Many tools are only used by a small community and are not designed for broad adoption, especially across computing disciplines (including students and programmers in industry). In addition, user feedback needs to be "honest". Although, many tools aim for soundness and a high level of completeness, it is uncommon for them to openly declare to the user the limitations of the tool.

ESC/Java2, on the other hand, is aimed at a broad number of users. It reasons about an established industrial-strength language, detecting common programming errors, while allowing users to determine the amount of checking performed by providing pragmas in a straightforward behavioural specification language (JML)[1]. In addition, the extensions described in this paper are inspired by "honesty". It is essential that the user be aware of the limitations of ESC/Java2, much the same as any verification tool that they use. Finally, efforts to make ESC/Java2 more user friendly are continuous. More details of this can be found in Section 4.

The rest of this paper is organised as follows: Section 2 describes the soundness and completeness of ESC/Java2. A detection and

[1]JML is also considered the *de facto* standard specification language for Java

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006), November 10–11, 2006, Portland, Oregon, USA.

warning system for areas where the reasoning of ESC/Java2 is potentially unsound or incomplete is presented in Section 3. Future work is considered in Section 4 and Section 5 concludes.

2. LIMITATIONS OF ESC/JAVA2

Although ESC/Java2 contains a full Java program verifier, the goal of ESC/Java2 is not to provide formally rigourous program verification. Rather, its aim is to help programmers find some kinds of errors more quickly than they might be found by other methods, such as testing or code reviews. Consequently, ESC/Java2 embodies engineering trade-offs among a number of factors including the frequency of missed errors, the frequency of false alarms, the amount of time used by the tool and the effort required to implement the tool. These trade-offs mean that ESC/Java2 is neither sound nor complete[2].

It is important to note that, when discussing program verification the words "soundness" and "completeness" are often used imprecisely. Referring to a single concept "soundness" or a single concept "completeness" hides the various layers of each concept that exist in a verification environment. Firstly, there is the soundness and completeness of the underlying logic in which the verification conditions will be generated. Secondly, there is the soundness and completeness of the translation of program constructs into verification conditions. Finally, there is the soundness and completeness of the theorem prover that disposes the verification conditions.

In this section, we discuss the various instances of unsoundness and incompleteness in ESC/Java2, paying special attention to the category to which it belongs.

2.1 Forms of Unsoundness

This section presents the areas of unsoundness in ESC/Java2 classified according to the underlying cause.

2.1.1 Semantics

There are a number of constructs in Java and JML whose semantics are not treated in a sound manner by ESC/Java2. These are:

Unsound Pragmas. The use of unsound pragmas such as `assume` and `axiom` allow the user to introduce assumptions into the verification process. ESC/Java2 trusts them, assuming them to be true. When these assumptions are invalid, the verification is unsound.

Arithmetic Overflow. ESC/Java2 reasons about integer arithmetic as though machine integers were of unlimited magnitude. This is unsound. However, it simplifies the checker and reduces the annotation burden for the user, while still allowing ESC/Java2 to catch many common errors.

Inherited pragmas. The `also_modifies` and `also_requires` pragma are unsound because they allow an overriding method to have a weaker specification than the method it overrides.

Constructor Leaking. There are a number of ways (often involving exceptional behaviour) in which a constructor can make the new object under construction available in contexts where its instance invariants are assumed to hold, but without actually having established those instance invariants.

[2]A description of some of the soundness and completeness issues in the original release of ESC/Java can be found here: http://secure.ucd.ie/products/opensource/ESCJava2/ESCTools/docs/ESCJAVA-UsersManual.html

Shared Variables. ESC/Java2 assumes that the value of a shared variable stays unchanged if a routine releases and then re-acquires the lock that protects it, ignoring the possibility that some other thread might have acquired the lock and modified the variable in the interim.

String Literals. Java's treatment of string concatenation is not accurately modeled by ESC/Java2. This is a source both of unsoundness and incompleteness.

2.1.2 Verification Methodology

Additionally, there are a number of ways in which ESC/Java2 does not translate the semantics of the constructs in a Java program into appropriately sound verification conditions.

Loops. ESC/Java2 does not consider all possible execution paths through a loop. It considers only those that execute at most one complete iteration, together with testing the guard before the second iteration. Although this is a straightforward approach and avoids the need for loop invariants, it is unsound.

Object Invariants. When checking the implementation of a method, ESC/Java2 assumes initially that all allocated objects satisfy their invariants. However, on checking a call to a method, ESC/Java2 imposes a weaker condition on the caller. This is that all actual parameters of the call and all static fields that are in scope are shown to satisfy their invariants, but not every object in existence. Since more is assumed than is proven, this is unsound.

In addition, when ESC/Java2 checks the body of a routine `r`, it does not consider all invariants but only a heuristically chosen "relevant" subset. If an invariant is deemed irrelevant during the checking of a routine that calls `r`, yet deemed relevant during the checking of `r`, then the invariant will not be checked (even for parameters) at the call site. However, it will nonetheless be assumed to hold initially during the verification of `r`. Conversely, ESC/Java2 might consider some invariant to be irrelevant to `r`, yet relevant to a caller. In this case, ESC/Java2 will not check that the body of `r` preserves the invariant. Nonetheless, it will assume, while checking the caller, that the invariant is preserved by the call.

Modification Targets. When reasoning about a call to a routine, ESC/Java2 assumes that the routine modifies only its specified modification targets (as given in `modifies` and/or `also_modifies` pragmas). However, when checking the implementation of a method, ESC/Java2 does not check that the implementation modifies only the specified targets.

Multiple Inheritance. When checking a method `m` of a class `C`, which inherits from `A` and `B`, ESC/Java2 assumes that the preconditions of `m` in `A` and `B` hold. However, if a routine `r` contains a call to `m` from an object of dynamic type `C` and static type `A`, then ESC/Java2 will only check the preconditions of `m` in `A`. This is unsound.

Ignored Exceptional Conditions. ESC/Java2 ignores cases where instances of unchecked exception classes (e.g., `OutOfMemoryError`, `StackOverflowError`, `ThreadDeath`, `SecurityException`) might be thrown either synchronously or asynchronously, except by explicit throw statements in a routine body being checked or in accordance with the throws clauses of routines called by a routine being checked.

Static Initialisation. ESC/Java2 does not perform extended static checking of static initialisers and initialisers for static fields. Neither does it check for the possibility that they do not give rise to errors such as null dereferences, nor does it check that they establish or maintain static or object invariants.

Class paths and `.spec` *files.* When a *.spec* file exists on the class path, ESC/Java2 chooses the specifications to check in an unsound manner. If ESC/Java2 is run on `A.java` where `A.spec` also exists, only the specifications in `A.java` are used. If ESC/Java2 is run on `B.java`, which contains calls to methods in `A.java`, then only the specifications in `A.spec` are used.

Initialisation of Fields declared `non_null`. In the case where a field is declared `non_null`, it may arise that ESC/Java2 uses the existence of a `modifies` pragma in the constructor (or in the specifications of a method called from the constructor) to *assume* that this field is indeed set to a non-null value. However, the `modifies` pragma simply declares what *can* be modified. It does not ensure that the field *is* modified. Therefore, this assumption is unsound.

Quantifiers and Allocation. When T is a reference type, specification expressions of the forms (`\forall T t; ...`) and (`\exists T t; ...`) quantify over allocated instances of T. If a method allocates new objects but is not annotated with a postcondition containing an occurrence of `\fresh` or `\old`, ESC/Java2 may infer unsoundly that some property holds for all allocated objects after completion of a call, when the property may in fact not hold for objects allocated during the call.

2.1.3 Theorem Prover

Finally, there are areas of unsoundness in Simplify, the main theorem prover currently used by ESC/Java2 [4]. Our work identifying issues with Simplify and warning the user about such will need to be repeated with each new theorem prover that is being added to ESC/Java2. Currently, partial support exists for PVS [10], the SMT-LIB [11] provers Sammy [6] and Harvey [1], and the new CVC3 (a merge of CVC Lite [2] and Sammy), and Coq [5].

Search Limits in Simplify. Simplify sometimes fails to prove the validity of an input formula or provide a counterexample. Such failures happen in a number of different ways. These scenarios are typical of many automated first-order provers.

- **Time Limits.** The first way Simplify can fail is it can simply not find a proof or a (potential) counterexample for the verification condition for a given routine within a set time limit. In this case, ESC/Java2 issues no warnings for the method, even though it might have issued a warning if given a longer time limit. If Simplify reaches its time limit after reporting one or more (potential) counterexamples, then ESC/Java2 will issue one or more warnings, but perhaps not as many warnings as it would have issued if the time limit had been longer.
- **Limit the Number of Warnings.** There is also a bound on the number of counterexamples that Simplify will report for any conjecture, and thus on the number of warnings that ESC/Java2 will issue for any routine. Thus many warnings "early" in a method can result in missing (possibly more serious) problems "later" in the method.
- **Universal Quantifiers.** Additionally, Simplify has problems dealing with (universal) quantifiers. When reasoning about universal quantifiers, Simplify frequently needs "triggers" to guide skolemization. A set of heuristics are used to help guide proof search, but they are not guaranteed to be sound. In particular, Simplify can miss seemingly "obvious" proofs because it moves down a branch of the proof tree and is unable to backtrack properly.

  These kinds of failures are witnessed in practise because first-order assertions are usually directly translated into first-order terms in verification conditions. Thus, while the quantifiers used in ESC/Java2's object logics are "well-triggered," user quantifiers are not. This type of failure must be communicated to the user in a natural manner, so rather than showing a mysterious failure from the prover, ESC/Java2 indicates that the user's specifications are overly-rich for the current prover and suggests trying other provers.

Prover Failures. Simplify, like many complex programs, also occasionally crashes. When Simplify fails, it is not sufficient to just hide the crash from the user and report back an incomplete verification, but instead it must try to characterise the failure so that the user can take remedial action by either rewriting specifications or using a different prover.

Arithmetic. The Simplify theorem prover, like many Nelson-Oppen inspired provers [9], includes a decision procedure for linear rational arithmetic based on the simplex algorithm. If integer operations in Simplify's simplex module result in overflows, they will silently be converted to incorrect results. Likewise, if non-linear arithmetic is used in assertions, then Simplify's arithmetic subsystem is not sound. Thus, when potential overflow or non-linear arithmetic expressions are detected by the system, an appropriate warning must be issued.

Other provers that use decision procedures, particularly new SMT-LIB provers, have exactly the same kind of behaviour and require the same kind of warnings. Unfortunately, characterising such prover limitations, especially in the presence of multiple interacting decision procedures, requires intimate knowledge of the prover's design and construction and is sometimes more art than science.

2.2 Forms of Incompleteness

This section presents the areas of incompleteness in ESC/Java2, each classified according to the underlying cause.

2.2.1 Semantics

Many sources of incompleteness in ESC/Java2 stem from the fact that we do not fully capture the semantics of Java and JML in the tool.

Floating-Point Numbers. The semantics for floating-point operations in ESC/Java2 are currently extremely weak. They are not strong enough to prove `1.0 + 1.0 == 2.0` or even `1.0 != 2.0`.

Strings. The semantics for strings are also quite weak. They are strong enough to prove `"Hello world" != null`, but not strong enough to prove the assertion `c == 'l'` after the assignment `c = "Hello world".charAt(3)`. Also, Java's treatment of string concatenation is not accurately modeled by ESC/Java2.

New, rich, verification-centric specifications of `java.lang.String` are being written to correct this issue. To accomplish this goal, the new specifications heavily directly leverage the sequence theories supported by modern first-order provers. This work was

halted when the new specifications pushed the boundaries of Simplify's capability to reason about sequences too far. Thus, the work is on-hold until CVC3 is integrated.

Unspecified Java APIs. Not all of the classes in the Java libraries have full JML specifications. Therefore, reasoning about calls to methods of these classes is incomplete.

Type Disjointness. According to the rules of the Java type system, if two distinct classes S and T are not subtypes of each other, then S and T have no non-`null` instances in common. The mechanism that ESC/Java2 uses to model the Java type system is sufficient to enforce this disjointness for explicitly-named types, but not for all types (e.g., the dynamic element types of array variables).

Arithmetic Overflow. In order to reduce the likelihood of arithmetic overflow occurring in the prover, ESC/Java2 treats all integer literals of absolute magnitude greater than 1000000 as symbolic values whose relative ordering is known but whose exact values are unknown. Thus, ESC/Java2 can prove the assertions `2 + 2 == 4` and `2000000 < 4000000` but not `2000000 + 2000000 == 4000000`.

Reflection. The semantics for reflection is extremely limited. For example, ESC/Java2 can determine that `Integer.class` is a non-`null` instance of `java.lang.Class`, but not that it is distinct from `Short.class`, or even that it is equal to `Integer.TYPE`.

2.2.2 Verification Methodology

The verification methodology used in ESC/Java2 is also unsound for a number of reasons.

Modular checking. The use of modular checking causes ESC/Java2 to miss some inferences that might be possible through whole program analysis. When translating a method call `E.m(...)`, ESC/Java2 uses the specification of `m` for the static type of `E`, even if it is provable that the dynamic type of `E` at the call site will always be a subtype that overrides `m` with a stronger specification.

2.2.3 Theorem Prover

The verification conditions that ESC/Java2 gives to the Simplify theorem prover are in a language that includes first-order predicate calculus (FOPC) (with equality and uninterpreted function symbols) along with some (interpreted) function symbols of arithmetic.

Since the true theory of arithmetic is undecidable, Simplify is necessarily incomplete. In fact, the incompleteness of Simplify's treatment of arithmetic goes well beyond that necessitated by Gödel's Incompleteness Theorem. In particular Simplify has no built-in semantics for multiplication, except by constants. Also, mathematical induction is not supported.

In addition, FOPC is only semi-decidable. That is, all valid formulas of FOPC are provable, but any procedure that can prove all valid formulas must loop forever on some invalid ones. Naturally, it is not useful for Simplify to loop forever, since ESC/Java2 issues warnings only when Simplify reports (potential) counterexamples. Therefore, Simplify will sometimes report a (potential) counterexample C, even when it is possible that more work could serve to refute C, or even to prove the entire verification condition.

3. WARNING SYSTEM

Clear user feedback is important in any tool that performs static analysis. Given the potential soundness and completeness pitfalls discussed in Section 2, a warning system for such stumbling blocks would be extremely beneficial, especially to new or inexperienced users.

This section presents such a warning system that has been implemented as an extension to ESC/Java2. We describe how constructs, in Java and JML, that ESC/Java2 treats in an unsound or incomplete manner are detected. In addition, we provide examples of the warnings that are emitted.

3.1 General Detection Methodology

We wish to detect many different kinds of contextual soundness and completeness issues. Also, many of these issues exist across code paths within ESC/Java2. As we now support, or are now working on support for, two calculi (weakest precondition and strongest postcondition), the use of an optional dynamic single assignment translation, three different logics, and five different provers, this means that we have at least seventy different code paths for verification. Thus, our detection methodology needs to be reusable across different parameterisations.

Therefore, we decided to implement each detection algorithm as an independent, type- and assertion-aware visitor that walks the fully resolved, typed, and annotated abstract syntax tree (AST).

For a given execution of ESC/Java2 with warnings enabled, each relevant visitor runs in sequence. The visitors are implemented as pure classes, so they do not affect the state of the AST.

Many of these visitors are simply performing type- and assertion-aware pattern matching on fragments of the AST. For example, to detect the use of large integer literals in arithmetic expressions, all the visitor must detect are AST fragments involving binary expressions, checking for one of a finite set of Java binary operators, recursively searching each operator's subexpressions for large Java integer literals.

Some visitors must be more complex, as they involve AST subtrees that are not obviously directly related in the tree. For example, we must examine all the invariants of an entire type hierarchy (including all inherited interfaces) if we wish to check the structure of relevant invariants for a given context.

3.2 ESC/Java2 Soundness Warnings

In the soundness warning system, there are three categories for constructs about which ESC/Java2 does not reason soundly. These are:

1. Constructs that produce warnings in warning user mode.
2. Constructs the produce warnings only in a special verbose warning mode.
3. Constructs that do not yet produce warnings.

The constructs that produce warnings in a special verbose warning mode occur too frequently to emit soundness or incompleteness warnings in a normal warning mode. Consequently, there is also a *Verbose Warning Mode* that emits warnings for all constructs that ESC/Java2 treats in an unsound or incomplete manner.

3.2.1 Warning User Mode

Currently, the following constructs emit soundness warnings in the *Warning User Mode*: *Unsound Pragmas*, *Static Initialisation*, *String Concatenation*, *Specification Inheritance*, *Quantifiers* and *Allocation* and *Search Limits* in Simplify.

This set of constructs has been chosen for *Warning User Mode* as they are relatively easy to detect while not occurring so frequently that the warnings displayed to the user would be overwhelming.

The following is an example of the clear and terse warning emitted in the case where the tool detects the initialisation of a static field on line 15 of a class called `Test.java`:

```
Test.java:15 Warning: ESC/Java2 does not
perform extended static checking of static
initialisers.

    static int a = 1;
    ^
```

3.2.2 Verbose Warning Mode

The additional constructs in this mode are: *Loops*, *Object Invariants* and *Arithmetic Overflow*.

As it is a *verbose* mode, the warning messages emitted also give extra information to the user. This includes an extended explanation of the unsoundness and a pointer towards a source of more information including a direct citation to the relevant documentation.

An example of a warning in this user mode is where the tool detects the a loop on line 36 of a class called `Loop.java` is:

```
Loop.java:36: Warning: ESC/Java2 does not
consider all possible execution paths
through a loop.

    for(int i=0, i<n; i++){
    ^

It considers only those that execute at
most one complete iteration, plus testing
the guard before the second iteration.

This is unsound.

To make ESC/Java2 consider more
iterations, use the -loop option.

More information can be found in Section
2.4.3 and Appendix C.0.1 of the ESC/Java2
User Manual.
```

This kind of warning behaviour, one that directly cites relevant detailed documentation, is inspired by Eiffel Software's EiffelStudio IDE which cites relevant sections of Meyers's "Eiffel the Language" and "Object-Oriented Software Construction" texts.

3.2.3 Unimplemented Constructs

Finally, there are some constructs that do not yet emit soundness warnings. These are: Ignored Exceptional Conditions, Constructor Leaking, Initialisation of Fields Declared `non_null`, Class paths with *.spec* files and Shared Variables.

These constructs are an open problem, in part because we must start relying upon more than just syntactic and lightweight semantic information (i.e., types) to reason about them. It may be necessary to do first-order reasoning to detect some of these scenarios.

3.3 ESC/Java2 Completeness Warnings

In the completeness warning system, the same three categories apply for constructs about which ESC/Java2's reasoning is incomplete.

3.3.1 Warning User Mode

Currently, the following constructs emit completeness warnings in the *Warning User Mode*: *Large Numbers*, *Reflection* and *Bitwise Operators*.

This set of constructs has been chosen for *Warning User Mode* as they are relatively easy to detect while not occurring so frequently that the warnings provided to the user would be overwhelming.

The following is an example of the clear and terse warning emitted where the tool detects the use of the left shift bitwise operator on line 87 of a class called `Bitwise.java`:

```
Bitwise.java:87: Warning: The semantics
of the left shift operator is incomplete.

    int_a << 2;
    ^
```

3.3.2 Verbose Warning Mode

The constructs for which warnings are emitted in this mode are: *Floating-Point Numbers*, *Strings* and *Arithmetic Overflow*.

The last warning to be given in *Warning User Mode* is to remind the user of the inherent incompleteness of Simplify. This warning states:

```
The theorem prover used by ESC/Java2,
Simplify, is necessarily incomplete.
This is due to the undecidability and
semi-decidability of some of the under-
lying theories used by Simplify.
```

Note that the warning message is parameterisable across prover names.

As with the soundness warnings, extra information is given to the user in *Verbose Warning Mode*. An example of such a warning is where the tool detects the use of floating-point numbers on line 64 of a class called `Decimals.java` is:

```
Decimals.java:64: Warning: The semantics
of floating-point operations are
incomplete.

    double d = 1.0 + 2.0;
    ^

They are not strong enough to prove
1.0 + 1.0 == 2.0 or even 1.0 != 2.0.

For more information, please see Appendix
C.1.1 of the ESC/Java2 User Manual.
```

3.3.3 Unimplemented Constructs

Finally, there are some constructs that do not yet emit completeness warnings. These are *Type Disjointness* and *Modular Checking*.

4. FUTURE WORK

The most obvious piece of further work to be carried out is the extension of the soundness and completeness warning system to cover more scenarios.

The extensions presented in this paper are ones that should be enabled by default in ESC/Java2. At present, it is only an option that can be switched on. Users that are aware of the myriad of options available in ESC/Java2 are those that are experienced in using the tool. These programmers are probably well-aware of the soundness and completeness issues with the tool. So how do we make the tool more user friendly, especially for beginners, without inundating them with excessive feedback?

One solution lies in the evolution of ESC/Java2 from a command line tool into one element of an Integrated Verification Environment (IVE). The authors are part of the EU MOBIUS Project[3] and are responsible with others for the development of such an IVE. In such a system, the level of feedback to the user will be configurable, allowing the user to fine-tune the information s/he receives. The environment will also highlight or underline pieces of code that are not reasoned about soundly or completely by ESC/Java2. This allows the user to made aware of such warnings without being forced to read through them all in the process of verification.

Currently all of these visitors, their specifications, and associated unit tests are hand-written. Given the complexity of the tool and aforementioned growing number of critical code paths through the tool, we believe that *generating* the visitors is a wise next step. We plan on defining a formal language in which one can specify the soundness and completeness limitations of various subsystems and generating the appropriate visitors with specifications, much like we already generate the Java and JML AST classes in ESC/Java2.

Likewise, to better support the rich warning messages discussed in Section 3.2.2, we plan on refining the ESC/Java2 architecture into a new version, integrated with the Mobius IVE, using a literate programming-style [8].

Finally, we imagine that some of the more complex situations we wish to check will necessitate the use of a prover to perform logical reasoning.

5. CONCLUSION

We have presented an extensions to the ESC/Java2 tool that provides useful feedback to the user during the verification process. Indeed, user friendliness of static analysis tools is an area that requires more research. It is one of the complaints of first-time users of ESC/Java2 that the feedback offered by the tool is hard to clearly understand and often overwhelming. One step has now been taken in improving this situation, but more are required.

6. ACKNOWLEDGMENTS

This work is being supported by the European Project Mobius within the frame of IST 6th Framework, national grants from the Science Foundation Ireland and Enterprise Ireland and by the Irish Research Council for Science, Engineering and Technology. This paper reflects only the authors' views and the Community is not liable for any use that may be made of the information contained therein.

7. REFERENCES

[1] Alessandro Armando, Silvio Ranise, and Michael Rusinowitch. A rewriting approach to satisfiability procedures. *Journal of Information and Computation*, 183(2):140–164, June 2003.

[2] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *CAV*, Lecture Notes in Computer Science. Springer–Verlag, 2004.

[3] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer*, Feb 2005.

[4] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[3]The Mobius Project: http://mobius.inria.fr/

[5] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. *The Coq Proof Assistant User's Guide*. INRIA, Rocquencourt, France, rapport techniques 154 edition, 1993.

[6] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04 (Boston, Massachusetts)*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer–Verlag, 2004.

[7] Joseph R. Kiniry and David R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer–Verlag, Jan 2005.

[8] Donald E. Knuth. *Literate Programming*. Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information, 1992.

[9] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[10] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer–Verlag.

[11] SMT-LIB: The satisfiability modulo theories library. http://goedel.cs.uiowa.edu/smtlib/.

Early Detection of JML Specification Errors using ESC/Java2

Patrice Chalin

Dept. of Computer Science and Software Engineering,
Dependable Software Research Group, Concordia University
1455 de Maisonneuve Blvd. West, Montréal
Québec, Canada, H3G 1M8
chalin@cse.concordia.ca

ABSTRACT

The earlier errors are found, the less costly they are to fix. This also holds true of errors in specifications. While research into Static Program Verification (SPV) in general, and Extended Static Checking (ESC) in particular, has made great strides in recent years, there is little support for detecting errors in specifications beyond ordinary type checking. This paper reports on recent enhancements that we have made to ESC/Java2, enabling it to report errors in JML specifications due to (method or Java operator) precondition violations and this, at a level of diagnostics that is on par with its ability to report such errors in program code. The enhancements also now make it possible for ESC/Java2 to report errors in specifications for which *no* corresponding source is available. Applying this new feature to, e.g., the JML specifications of classes in `java.*`, reveals over 50 errors, including inconsistencies. We describe the adjustment to the assertion semantics necessary to make this possible, and we provide an account of the (rather small) design changes needed to realize the enhancements.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—programming by contract; D.3.3 [Programming Languages]; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs.

General Terms

Documentation, Design, Languages, Theory, Verification.

Keywords

Java Modeling Language, JML, Extended Static Checking, Precondition Errors, Specification Debugging.

1. INTRODUCTION

It is well appreciated that the earlier in a product's lifecycle that an error is detected, the less costly it will be to correct. This has motivated considerable research in the area of Static Program Verification (SPV) so that today, a variety of approaches and tools are becoming available to developers. In fact, developers can already make routine use of tools that effectively eliminate certain classes of error. One promising technological approach to SPV is Extended Static Checking (ESC) [11]. ESC tools, like ESC/Java2 [7] and Spec#'s Boogie [8], offer fully automated checking of code against specifications. Despite the fact that automation is achieved at the expense of completeness and/or soundness, in practice, the tools are still quite effective at revealing coding errors.

Unfortunately, there is an important lacuna: SPV tools offer no support for the detection of errors in specifications beyond conventional type checking. But writing error free specifications is just as hard as (if not harder than) writing correct code, hence tool assistance would be welcome. Specifications containing errors can cause problems: e.g., consider a situation where a method *m* has a specification (contract) *c* containing errors, then

1. A developer can *waste time* trying to get an ESC tool to prove that *m* satisfies *c*. Anyone who has used a verification tool is likely to have had this experience; i.e. being convinced that the specification (or theorem) is correct, one persists in trying to get the verifier to agree, only to realize, in all humility, that the tool was right, and that the specification was in error.
2. The ESC tool *is* able to prove that *m* satisfies *c*. This merely delays the discovery of the error (in *both* the specification and the implementation) until a later lifecycle phase. As a result, the error will be more expensive to correct.
3. In the worst case, *c* is inconsistent. Hence, any invocation of *m* in the code would amount to asserting falsehood, from which the verifier can trivially prove anything. For example, ESC/Java2 would be able to prove the assertion following a call to *m*:

```
m();
//@ assert 0 == 1;
```

   One might remark that: "if c is inconsistent, would ESC/Java2 not report an error in any attempt to prove that the implementation of *m* satisfies *c*?" Yes, but this assumes that the source for m is available, which is not the case, .e.g., for third party libraries distributed in binary form.

In all three situations mentioned above, any assistance provided in the early detection of specification errors would avoid loss of time or the increased cost associated with fixing the error at a later

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006). November 10-11, 2006, Portland, Oregon, USA.

```
public class MyUtil {
 //@ ensures \result ==
 //@   java.lang.Math.min(a1.length, a2.length);
 public static int minLen(int[] a1, int[] a2);

 //@ requires n <= a.length;
 //@ ensures \result ==
 //@   (\sum int i; 0 <= i && i < n; a[i]);
 public static int sumUpTo(int[] a, int n);
}
```

Figure 1. Interface specification, `MyUtil.jml`.

stage. Note that the errors reported by ESC tools can be partitioned into two categories:

- errors due to a *precondition violation*, be it for operators of the Java language or class constructors and methods. Common examples of the former include null pointer exceptions and array index-out-of-bound errors.
- *correctness* issues—when a constructor or method implementation fails to meet its specification.

There is an order to this categorization since it only makes sense to discuss correctness issues once precondition errors have been resolved. While it is not possible for ESC tools to identify correctness errors in specifications[1], it can be done for precondition violations.

Building upon our earlier work [4, 5], this paper reports on a recent **feature enhancement**—called **definedness checking**—that we have made to ESC/Java2, enabling it to report errors in *specifications* due to *precondition violations* at a level of diagnostics that is on par with its ability to report such errors in program code. (This work is actually being done as a first step in a two part enhancement plan, the second of which—support for *consistency checking*—will be the subject of a subsequent publication.)

To our knowledge, ESC/Java2 is the first static program verification tool to offer such definedness checking. Hence, e.g., ESC/Java2 now diagnoses in specifications, just as easily as in source code, one of the most common programming errors, null pointer exceptions (NPEs). Since specifications are often created by the same developers who write the corresponding code, NPEs in specifications are just as likely to occur.

Another important related enhancement made to ESC/Java2 includes its ability to **report errors in specifications** for which **no** corresponding **source** is available (recall that ESC/Java2 formerly only checked *source code* relative to its interface specifications). This key enhancement now permits checking of the comprehensive collection of public library API specifications shipped with ESC/Java2. Identifying and correcting bugs in API specifications is significant since it can positively impact all developers who make use of them—and, as we shall illustrate in Section 2.2, errors in API specifications can have serious consequences. The enhancement also enables better support for those development groups who follow the practice of writing interface specifications prior to writing code.

The remaining sections are organized as follows. Section 2 presents examples of specifications that at first appear to be correct, or that have been in use for several years now, and yet contain serious flaws including inconsistencies. The examples serve to motivate the addition of definedness checking to ESC/Java2 since prior to this enhancement, the tool was theoretically *incapable* of detecting such errors. We explain the nature of this incapacity in Section 3 by briefly describing the former logical underpinnings of the tool (inherited from the Java Modeling Language) as well as the new assertion semantics that make definedness checking possible. Section 4 explains the basic mode of operation of ESC/Java2 by decomposing it into processing stages. This allows us to explain how support for definedness checking required changes to only one of the processing stages. In Section 5, we answer the question, "better diagnostics, but at what cost?" Related work is discussed in Section 6, while we offer conclusions and mention future work in Section 7.

[1] It might be possible in the Java Modeling Language (JML) since it supports specification refinement, but this is a seldom used feature which is in fact not common to the languages used by ESC tools.

```
public class PairSum {
  public static int pairSum(int[] a, int[] b) {
    int n = MyUtil.minLen(a, b);
     // Commutativity of addition allows us to use sumUpTo twice ...
    return MyUtil.sumUpTo(a, n) + MyUtil.sumUpTo(b, n);
   }
  public static void main(String[] args) {
    int[] a = null;
    int sum =  pairSum(a, a);
   }
}
```

Figure 2. `PairSum` class (using `MyUtil`).

2. MOTIVATING EXAMPLES

2.1 MYUTIL/PAIRSUM

As a first example, consider the following scenario. Assume that a friend, who is a formal methods aficionado, provides you with a copy of her `MyUtil` class. Of course, being sympathetic to the cause, she also provides you with the interface specification given in Figure 1. The utility class provides two methods, one that returns the minimum length of its two argument arrays, and the other which returns the sum of the integer elements of an array, up to, but not including a given index.

Eager to make use of the functionalities of `MyUtil`, you write a method that will compute the pair wise sum of two arrays, up to the length of the shorter of the two arrays. See Figure 2. Invoking ESC/Java2 on `MyUtil.jml` and `PairSum.java` yields no error messages, and yet execution of `PairSum.main()` raises a null pointer exception. We will defer until Section 3.2 a technical discussion explaining why ESC/Java2 "believes" that no exceptions should have been raised by `PairSum`. For now, suffice it to say that rerunning ESC/Java2 with definedness checking enabled, easily reports:

```
MyUtil: minLen(int[], int[]) ...
-------------------------------------------------
MyUtil.jml:3: Warning: Possible null dereference
  //@   java.lang.Math.min(a1.length, a2.length);
                           ^
-------------------------------------------------
   [0.062 s 12135232 bytes]  failed
```

What is the source of the problem? Intuitively we can understand that the specifications of `minLen()` and `sumUpTo()` are in a sense, incomplete. E.g., the method contract of `minLen()` does not prevent it from being called with `null` arguments, and yet under such circumstances, the interpretation of the postcondition does not make sense due to precondition errors.

2.2 API SPECIFICATIONS FOR JAVA.UTIL.*

```
/*@  public normal_behavior
  @   requires a != null;
  @   assignable a[fromIndex..toIndex-1];
  @   ensures (\forall int i;
  @              fromIndex < i && i < toIndex;
  @              a[i-1] <= a[i]);  // (*)
  @   ... // more ensures clauses here
  @ also
  @  public exceptional_behavior
  @   requires a == null || fromIndex > toIndex
  @         || fromIndex < 0 || toIndex > a.length;
  @   assignable \nothing;
  @   signals_only NullPointerException, IllegalArgumentException,
  @               ArrayIndexOutOfBoundsException;
  @   signals (NullPointerException) a == null;
  @   signals (IllegalArgumentException) fromIndex > toIndex;
  @   signals (ArrayIndexOutOfBoundsException) fromIndex < 0;
  @   signals (ArrayIndexOutOfBoundsException)
  @             a != null && toIndex > a.length;
  @*/
public static void
   sort(int[] a, int fromIndex, int toIndex);
```

Figure 4. `java/util/Arrays.refines-spec`.

Somewhat disgruntled, you decide not to use `MyUtil` and instead favor the more reliable `java.util.*` classes. Thankfully, ESC/Java2 comes with API specifications for these classes, among others.

Unfortunately, other problems arise as well. Consider the code given in Figure 3. The `ArraysBug` class contains three methods that exercise the functionality of the `java.util.Arrays.sort()` methods. The contract for `ArraysBug.m0()` states that the only behavior which `m0()` can have is to return a null pointer exception. Following a common ESC idiom, we have added an "`assert false`" statement at the end of the method body to indicate that flow control should never reach that point. In this example though, such a statement is superfluous since the contract of `m0()` mandates that it always return exceptionally—i.e., an `exceptional_behavior` case implicitly adds an "`ensures false`" clause. Similarly, the contract for `m1a()` states that calling it

```
public class ArraysBug {
 //@ public exceptional_behavior
 //@ signals_only NullPointerException;
 void m0() {
   java.util.Arrays.sort( (int[]) null );
   //@ assert false; // this point is never reached
 }

 //@ public exceptional_behavior
 //@ signals_only ArrayIndexOutOfBoundsException;
 void m1a() {
   java.util.Arrays.sort(new int[]{1,2}, -1, 99);
 }

 //@ public behavior
 //@ ensures false;
 //@ signals_only ArrayIndexOutOfBoundsException;
 //@ signals (Throwable) false;
 void m1b() {
   java.util.Arrays.sort(new int[]{1,2}, -1, 99);
 }
}
```

Figure 3. `ArraysBug.java`.

```
/*@ public normal_behavior
  @  ensures -1 <= \result && \result <= 9;
public static model pure int digitVal(char ch)
{
  if (!java.lang.Character.isDigit(ch)) {
   return -1;
  } else {
   int val = ch;
   // Determine the base (0 value) depending on the type of digit ...
   if (val <= 0x06F9 || val >= 0x0E50)
     base = val & 0xFFF0;
   else
     base = ((int)(val - 6) & 0xFFF0) | 0x0006;
  // convert to a value between 0 and 9 inclusive
  return (int)(val - base);
 }
} @*/
```

Figure 5. Model method defined in `java/lang/Character.jml`.

always raises an index out of bounds exception. Finally, the contract for `m1b()` is inconsistent—i.e. while it has an implicit precondition of *true*, *both* its normal and exceptional postconditions are unsatisfiable.

While ESC/Java2 can prove the correctness of `m0()` and `m1a()`, it is also able to prove `m1b()`! Since, the contract of `m1b()` is unimplementable, the only way in which ESC/Java2 can "prove" that the body of `m1b()` satisfies it, is if the specification of `java.util.Arrays.sort(int[],int,int)` is inconsistent. An excerpt of the specification of `java.util.Arrays.sort(int[],int,int)` is given in Figure 4. The method contract has only two specification cases. What is the source of the problem this time? With definedness checking enabled, we find that ESC/Java2 is *unable* to prove that the array element access at (*) is within the bounds of the array. Inspection of the contract reveals that this is because the first specification case has no requires clause placing bounds on `fromIndex` or `toIndex`. Adding as a precondition, the obvious constraints on these two parameters, allows ESC/Java2 to prove that `ArraysBug.m1b()` cannot meet its specification. (For lack of space we do not discuss the nature of the inconsistency here, we merely note that the added requires clause guards the particular call made to `sort()` by `m1b()` from the source of the inconsistency.)

2.3 Other API Specification Errors

Performing definedness checks on all of the `java.*` API specifications reveals about 50 errors related to potential null pointer exceptions and array out of bounds errors—since these are the only checks currently implemented, we anticipate that more errors will be found as we increase the definedness coverage of the tool. (Use of definedness checking also exposed a bug in ESC/Java2's handling of specification inheritance—cf. bug#430.)

ESC/Java2 also reports bugs in the implementation of model methods such as the one given in Figure 5. Asking ESC/Java2 for counter examples eventually allows us to deduce that `digitVal()` will fail to satisfy its postcondition for `ch` in the small range of $4970 \leq$ `ch` ≤ 4975.

We believe that the examples given in this section clearly illustrate the benefits of the new definedness checking that has been added to ESC/Java2.

3. SUPPORTING DEFINEDNESS CHECKING

3.1 BACKGROUND

ESC/Java2 can analyze Java source files annotated with specifications written in the Java Modeling Language (JML). At a minimum, JML can be seen as an extension to Java that adds support for Design by Contract (DBC) [22, 28], though it has more advanced features—such as specification only class attributes, support for frame axioms, and behavioral subtyping—that we believe are essential to writing complete interface specifications [6].

In the spirit of DBC, JML specifications are expressed via *program assertions* embodied in class invariants, as well as constructor and method contracts expressed using pre- and post-conditions. In the next section, we describe the logical semantics of JML assertions. This will enable us to explain why ESC/Java2 was unable to prove that the `PairSum` program would cause exceptions to be generated at runtime.

3.2 JML'S CLASSICAL ASSERTION SEMANTICS

As is common in Behavioral Interface Specification Languages (BISLs) like JML, assertions are traditionally interpreted as formulae in a classical two-valued logic in which partial functions are modeled by underspecified total functions [3]. Hence, when a partial function $f : A \to B$ with domain $D \subseteq A$ is applied to a value v outside of D, then $f(v)$ is nonetheless assumed to have *some* value in B, though we do not know which value it is.

Returning to the `MyUtil`/ `PairSum` example of Section 2.1, we can now understand that under such a semantics, `minLen(null, null)` has the (well-defined[2]) value of `null.length`—whatever particular `int` value it might be. While ESC/Java2 is checking the body of the `pairSum()` method, it assumes that the local variable `n` gets assigned the value of `null.length`. Next, ESC/Java2 checks that the precondition of `sumUpTo()` is satisfied. Recall that the precondition is: `n <= a.length`. Since `a` is `null` and `n` is equal to the value of `null.length`, the expression reduces to *true*, hence the precondition holds. As a consequence, ESC/Java2 has no errors to report.

3.3 NEW ASSERTION SEMANTICS BASED ON STRONG VALIDITY

Backed by a survey of industrial software developers [3], we recently proposed a new logical foundation for JML in which partiality is modeled directly [4, 5] rather than approximated via under-specification [12]. While we will not go into the details here, in essence, we proposed that a JML assertion be considered valid iff it is both

- *defined*, and
- true.

Hence, assertion failure can result either from undefinedness or evaluation to false. It is useful to distinguish between these two cases of assertion failure in practice, as we will explain in the next subsection. Technically speaking, this newly proposed definition of assertion validity is what Konikowska *et al*. call *strong validity* [19]. This is in contrast to *classical validity*, currently adopted by all BISLs, including JML. Key to the definition of strong validity is the so-called "is-defined" operator which we will describe in Section 3.5 after a short remark about blame assignment.

[2] It is well-defined *relative* to the classical assertion semantics of JML.

3.4 RESPONSIBILITY / BLAME ASSIGNMENT

The disciplined use of assertions in the context of Design By Contract (DBC) [28, 29] also naturally gives rise to the concept of *responsibility assignment*. Hence, for example, the client of a method has the responsibility of ensuring that the method's precondition holds before invoking it. In return, when a method is called under these circumstances, it commits to respecting its postcondition. When an assertion fails, we can assign blame to the party that did not fulfill its responsibilities: if the precondition is violated then the client is to blame, and if the postcondition is violated then the method implementation is to blame.

Adoption of an assertion semantics based on strong validity gives rise to another kind of responsibility that comes to rest upon the *specifier*: he or she must ensure that the assertions written in contracts are always defined. This becomes a proof obligation on the part of the specifier, not much different from normal proof obligations which are an integral part of model-based specification approaches that define operations by means of pre- and post-conditions: e.g. satisfiability obligations in VDM [15, §5.3] and Z [32].

Thus, for example, upon failure of a precondition, we have two cases: if the precondition is undefined then we blame the specifier, otherwise as before, blame falls upon the client code. Similar remarks can be made for postconditions.

3.5 THE "IS-DEFINED" OPERATOR

Strong validity relies on the notion of an "is-defined" operator, $D(e)$, which is true iff the expression e is defined, i.e. it does not contain the application of a partial function to a value outside its domain. For example, $D(3/x)$ would be equivalent to $x \neq 0$.

When applied to an expression consisting of a constant or a variable, D is *true*. For a strict function f having arity n and precondition p, we have

$$D(f(e_1, \ldots, e_n)) = D(e_1) \wedge \ldots \wedge D(e_n) \wedge p(e_1,\ldots,e_n)$$

Note that by a function we mean any operator or method used in an assertion expression—such methods are required to be `pure` in JML [24]. As can be seen from the preceding definition, a strict function yields undefined whenever any of its arguments is undefined. Here are examples for division and (non-conditional) conjunction:

$$D(e_1 / e_2) = D(e_1) \wedge D(e_2) \wedge e_2 \neq 0$$
$$D(e_1 \,\&\, e_2) = D(e_1) \wedge D(e_2)$$

In order to ensure that D remains computable, we require that a function not contain, directly or indirectly any recursive applications of itself in the statement of its precondition [14, §9.3].

The non-strict (i.e. conditional) operators of most programming languages consist of conditional conjunction, conditional disjunction and a ternary (McCarthy) conditional operator. All three can be written in terms of the latter so it is sufficient to define D for this operator:

$$D(e_1 \,?\, e_2 : e_3) = D(e_1) \wedge (e_1 \Rightarrow D(e_2)) \wedge (\neg e_1 \Rightarrow D(e_3))$$

Given that "$e_1 \parallel e_2$" can be written as "e_1 ? *true* : e_2", and "e_1 && e_2" as "e_1 ? e_2 : *false*" it follows that

$$D(e_1 \parallel e_2) = D(e_1) \wedge (\neg e_1 \Rightarrow D(e_2))$$
$$D(e_1 \,\&\&\, e_2) = D(e_1) \wedge (e_1 \Rightarrow D(e_2))$$

D can also easily be defined over quantifiers—examples are provided by Konikowska for Kleene and McCarthy quantifiers [19].

An example of an assertion expression that is both classically valid and strongly valid is

```
x == 0 || 3/x == 3/x
```

because $D(x == 0 \parallel 3/x == 3/x)$

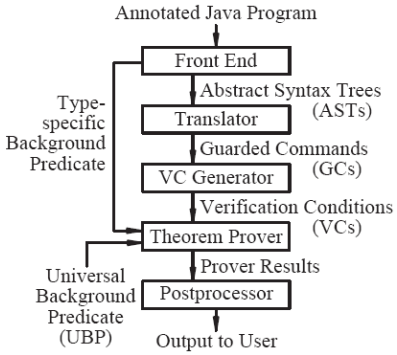


Figure 6. ESC/Java2Pipeline Architecture (excerpt from [11]).

$= D(x == 0) \ \wedge \ (\neg(x == 0) \Rightarrow D(3/x == 3/x))$

$= true \ \wedge \ (x \neq 0 \Rightarrow x \neq 0)$

which is *true*. In contrast, the expression

```
3/x == 3/x
```

is classically valid, but not (strongly) valid because $D(3/x == 3/x)$ is $x \neq 0$.

While the adoption of a new logical foundation for JML may seem like a big change, as we shall see in the next section, it is straightforward to implement.

4. ESC/JAVA2 REDESIGN

4.1 ESC/JAVA2 CONCEPTUAL ARCHITECTURE

Before explaining the implementation of the new semantics we begin by reviewing ESC/Java2's overall conceptual architecture (essentially an instance of pipes-and-filters [31]). The main processing stages are shown in Figure 6. Input to the tool consists of one or more JML annotated Java source files or pure JML interface specification files. The source(s) are parsed. Checking in ESC/Java2 is modular and this manifests itself already at the next stage; i.e., on a per class basis, each method in turn is translated into a Guarded Command (GC) program [26]. Each such program entirely captures the proof obligations related to establishing the correctness of the method in question, relative to its specification. In particular, this means that calls made inside the method body are represented by an inlined version of the contract of the called method.

GCs are then converted into verification conditions (VCs) which are fed to a fully automated theorem prover. Currently ESC/Java2 (and Spec#'s Boogie) make use of Simplify [9]. Noteworthy efforts have been deployed in the past two years so that new backends (e.g., CVC3) should be available for use with ESC/Java2 before year's end [17]. As indicated in the diagram, the prover is also provided with a Universal Background Predicate containing an axiomatization of concepts true of all Java programs, and a Type-specific Background Predicate which, as the name implies, axiomatizes concepts that are specific to the type (class or interface) being processed.

If the prover is able to discharge a method's VCs, then we consider the method implementation to be correct. If all of a class' VCs are met, then the class is said to meet its specification. As usual, while the theory is fairly straightforward, the pragmatics (which we will briefly touch upon in Section 4.3), complicate matters somewhat. E.g., significant extra machinery is needed to allow for meaningful post-processing of a prover's output especially when the prover is *unable* to discharge a VC. Accurate and meaningful error reporting is essential. Further details concerning the processing performed by ESC/Java2 can be found in [11].

4.2 SUPPORTING THE NEW SEMANTICS

Changes to ESC/Java2 in support of the new semantics were confined to the "Translation to GC" stage. The creation of a guarded command program for a given method actually occurs in two steps: the method is first translated into a "sugared GC" language, before subsequently being "desugared" into the following primitive GC language [26]:

```
C ::= Id := Expr
    | ASSUME Expr
    | ASSERT Expr
    | C ; C'
    | C □ C'
```

The commands represent: assignment, primitive assume and assert commands, sequential and alternative composition. In the latter case (involving an application of the box operator), the composite command behaves either like its first operand or its second operand, with the choice being non-deterministic. Note that in the present discussion, we are disregarding (Java) exception processing since it would unnecessarily complicate the presentation of the new semantics.

The two-staged GC translation process allows more flexibility in, e.g., selectively enabling or disabling the various kinds of checks to be performed. Controlling which checks to perform can be done globally (e.g. via a command line arguments), or even on a line by line basis of the input source.

We will describe the implementation of the new semantics in terms of the translation of JML specification constructs into the primitive GC language. As can be expected, the translation will make extensive use of the is-defined operator, *D*, of Section 3.4. We begin with the most basic of the JML assertions, namely inline assert and assume statements.

4.2.1 *INLINE ASSERTIONS*

JML `assert` and `assume` statements can appear in constructor and method bodies as well as static initialization blocks. Under the new assertion semantics, such statements are translated into a sequence of two guarded commands: the first asserts that the given predicate is defined, then follows the `assert` or `assume` command proper. For example,

```
⟦ASSERT R⟧  =  ASSERT ⟦D(R)⟧ ;
               ASSERT ⟦R⟧
```

This follows naturally from the definition of strong validity. Note that with this approach, it is no longer relevant that the given assertion expression, *R*, contain partial functions or not. This is because interpretation of *R* is guarded by an `ASSERT` of the definedness condition of *R*; hence all occurrences of partial functions will be to values inside their domain.

Other JML constructs use assertions as basic building blocks, and hence our adapted translation of a single assert statement into a pair of guarded commands, will be a recurring theme.

4.2.2 *BASIC METHOD CONTRACTS*

Under the current semantics of JML, the translation of a method with precondition *P*, body *B* and postcondition *Q* is handled as follows[3]:

[3] $\{P\}B\{Q\}$ is the compact and familiar Hoare-triple syntax.

```
⟦{P}B{Q}⟧ = ASSUME ⟦P⟧ ;
            ⟦B⟧ ;
            ASSERT ⟦Q⟧
```

Under the new semantics we have:

```
⟦{P}B{Q}⟧ = ASSERT ⟦D(P)⟧ ;
            ASSUME ⟦P⟧ ;
            ⟦B⟧ ;
            ASSERT ⟦D(Q)⟧ ;
            ASSERT ⟦Q⟧
```

The new GCs are underlined.

4.2.3 *Class Invariants*

In those cases where a (non-`helper`) method belongs to a class *C* having invariant *I*, we get:

```
⟦{P}B{Q}⟧ = ASSERT ⟦D(I(this))⟧ ;
            ASSUME ⟦∀ o:C . I(o)⟧ ;
            ASSERT ⟦D(P)⟧ ;
            ASSUME ⟦P⟧ ;
            ⟦B⟧ ;
            ASSERT ⟦D(Q)⟧ ;
            ASSERT ⟦Q⟧ ;
            ASSERT ⟦D(I(this))⟧ ;
            ASSERT ⟦∀ o:C . I(o)⟧ ;
```

Upon entry to the method and on exit from the method, the invariants of all instances of class *C*, including `this`, must hold. The invariant definedness need only be checked relative to one instance of *C*, choosing `this` is most convenient. (While it is known that JML's semantics of invariants is unsound, we provide a compatible definition under the new semantics—finding a sound and effective solution to this problem is still an active area of research [6].)

4.2.4 *Checking In The Absence of Source Files*

Having accurate specifications for public library APIs is essential to the working developer. Lack of specifications discourages use of the tools. On the other hand, flawed specifications can be useless at best, dangerously misleading at worst. As was illustrated in Section 2.2, use of a public API method having an inconsistent specification will always result in the (false!) impression of correct code.

Since such libraries are often only available in binary form, practically all of the given library specifications had been subject to no more than type checking, and an occasional manual design review. As was pointed out earlier, ESC/Java2 was originally designed to check the correctness of source code (i.e. an implementation) relative to a given specification. As we have demonstrated earlier, performing basic definedness checks (and eventually consistency checks) can be quite useful.

Given a method specification for which no method body is available, we generate a GC of the following form:

```
⟦{P}_{Q}⟧ = ASSERT ⟦D(I(this))⟧ ;
            ASSUME ⟦∀ o:C . I(o)⟧ ;
            ASSERT ⟦D(P)⟧ ;
            ASSUME ⟦P⟧ ;
            ⟦return _⟧ [] ⟦throw new Exception()⟧ ;
            ASSERT ⟦D(Q)⟧ ;
            ASSERT ⟦D(I(this))⟧ ;
```

where we take as a bogus body, one that can either return (an unspecified return value, if such a value is needed) or raise an exception.

4.3 Accurate Error Reporting

As in most software applications, particularly compilers, providing accurate and helpful error reporting usually requires considerable extra effort beyond the processing of "normal" input.

The explanation, in the previous subsections, of the translation into GCs had conceptual clarity as a main objective. In this section, we briefly describe the extra processing required to enable ESC/Java2 to report specification errors, pin-pointing their source, as accurately as could be expected of a modern compiler—i.e., accurately identifying the cause of the error (such as Division by Zero) as well as the line number and character position of the problematic partial operator.

`ASSERT` commands can have associated labels which the backend prover uses when reporting VC proof failures. Conceptually, a label *L* (containing a file id, line number and character position) would be reported by the prover if it were unable to prove *E* in:

```
ASSERT Label(L, E);          // (1)
```

Unfortunately, if *E* is a complex expression, we might be unable to tell which subterm of *E* is to blame. Finer grained error reporting can be obtained by decomposing (1) into an *expanded* GC program, more refined, though equivalent in effect to the original single assert command.

Of concern to us here are expressions consisting of definedness predicates. Recall that for a strict function *f* having arity *n* and precondition *p*, we have

$$D(f(e_1, \ldots, e_n)) = D(e_1) \wedge \ldots \wedge D(e_n) \wedge p(e_1,\ldots,e_n)$$

The expanded GC program for $D(f(e_1, \ldots, e_n))$, is defined as

```
E⟦D(f(e1, …, en))⟧ = E⟦D(e1)⟧ ;
                     ... ;
                     E⟦D(en)⟧ ;
                     ASSERT Label(L, ⟦p(e1,…,en)⟧ )
```

where *L* is a label generated from the location associated with *f*. For the conditional operator, recall that

$$D(e_1 \,?\, e_2 : e_3) = D(e_1) \wedge (e_1 \Rightarrow D(e_2)) \wedge (\neg e_1 \Rightarrow D(e_3))$$

The expanded GC form would be:

```
E⟦D(e1 ? e2 : e3)⟧ = E⟦D(e1)⟧ ;
                     { ASSUME ⟦e1⟧ ;
                       E⟦D(e2)⟧ ;
                       []
                       ASSUME ⟦¬e1⟧ ;
                       E⟦D(e3)⟧ ;
                     }
```

While conditional conjunction and disjunction are simplifications of the ternary conditional operator, it is important not to eliminate the box operator from the expanded form. Recall that, e.g., "e_1 && e_2" is equivalent to "$e_1 \,?\, e_2 : \mathit{false}$", thus we have

```
E⟦D(e1 && e2)⟧ = E⟦D(e1)⟧ ;
                 { ASSUME ⟦e1⟧ ;
                   E⟦D(e2)⟧ ;
                   []
                   ASSUME ⟦¬e1⟧ ;
                 }
```

5. Better Diagnostics, At What Cost?

The treatment of definedness conditions given here is quite similar to the type-correctness conditions (TCCs) of the PVS theorem prover [33].

One of the main objections to using a definition of assertion validity that takes definedness into account is that it is likely to contribute to making the already sizeable verification conditions even larger. As a consequence, it is believed that this would lead to ESC tools being able to prove fewer methods correct. Like in PVS, we expected most definedness conditions to be easily discharged since they are, by their very nature, much smaller and simpler than the assertion expressions they guard.

Our experiences show that the overhead is not perceptibly significant, though such experiences are preliminary since we have as yet to implement all planned definedness checks. It is still worth noting that, e.g., in processing 90 KLOC of code, we have yet to come across a (correct) method that could not be proven with definedness checking enabled and yet could be proven correct otherwise. Addition of the remaining definedness checks will be completed shortly, after which a more rigorous assessment of the cost (in time and memory consumption) of definedness checking will be in order. In the advent that the overhead would indeed be prohibitive, then ESC/Java2 could imitate PVS and allow users to check definedness conditions separately.

It is interesting to note that the arrival of new ESC/Java2 prover backends like CVC3, which directly support three-valued logics and partiality, will eliminate having definedness checks factored out as separate "side" conditions. (Of course, it remains to be seen if such provers can rival their classical counterparts.)

6. Related Work

To our knowledge, the enhancements we have made to ESC/Java2 are a first of its kind and this mainly because all other static program verification systems (e.g. [1, 2, 8, 27, 34]) are based on a classical definition of assertion validity.

As was mentioned earlier, adoption of strong validity allows us to extend the usual Design by Contract responsibility/blame matrix—attributed to software components (clients and/or service providers) [29]—to assigning responsibility/blame to *specifiers*, in ensuring that contract assertions are always defined. Findler *et al.* also assign responsibility/blame to specifiers but only relative to the conformance of subclass contracts to the constraints of behavioral subtyping [10]. We note that in JML, subclasses automatically inherit their supertype contracts and hence naturally enforcing behavioral subtyping [21].

Approaches to assertion semantics based on strong validity, and hence using a definedness operator, have been advocated by other authors for some time now. The most fundamental works being that of Hoare and He, in their "*Unifying Theories of Programming*" [14], as well as Konikowska's "*Two Over Three: A Two-Valued Logic for Software Specification and Validation Over a Three-Valued Predicate Calculus*" [18]. We invite the reader who is interested in a more detailed discussion of these two approaches in relationship to our work to consult [5].

Leino also makes use of a "Defined" operator in the formal semantics of his Ecstatic language [25], but this operator is only applied to expressions appearing in general program statements rather than assertions. Morris provides a semantics for non-deterministic expressions and also makes use of an "is well-defined" operator (Δ) [30]. Morris' operator is more general in that $\Delta(E)$ not only holds when E is not undefined but also when it is *deterministically* defined. Like Leino, Morris does not apply definedness to the semantics of assertion expressions.

Of course, "definedness" is also an elementary concept in VDM's three-valued Logic of Partial Functions (LPF). The "is-defined" operator is written as Δ. One of the claimed advantages of LPF is that specifiers should seldom have to refer to Δ when conducting proofs of VDM specifications [16]. While a three-valued logic like LPF has a natural correspondence with RAC assertion semantics, unfortunately, there are no provers supporting LPF (although the *Overture* initiative might change that [20]).

7. Conclusions And Future Work

The focus of current static program verification (SPV) tools is, somewhat naturally, on *source code* bugs. Little support beyond well-formedness and type checking is offered for the static "debugging" of specifications. This is mainly due to reliance on an assertion semantics based on classical validity: under such a definition there are no partial functions, and hence, in a sense, no precondition errors to report. In this paper we have demonstrated how an SPV tool like ESC/Java2 can easily be extended to support definedness checking of assertion expressions. Only one of the multiple processing stages of ESC/Java2 needed to be enhanced; hence, in particular, the change was made while preserving the same classical prover backend.

ESC/Java2's new definedness checking seems to add marginal computational overhead while, in our opinion, offering a significant debugging capability for specifications. In fact, having applied definedness checking to the `java.*` API specifications shipped with ESC/Java2 revealed over 50 errors, one of which lead to the identification of an inconsistent method specification.

The enhancements that we have presented can be applied to other SPV tools, such as Spec#'s Boogie verifier [8].

We will continue to extend the scope of ESC/Java2's definedness checking. In particular, one of the next milestones is the addition of support for the checking of method preconditions at the point of a method call in an assertion expression. Following this, we plan on conducting a rigorous assessment of the impact on time and resource requirements due to the extra load of definedness checks. The start of this empirical assessment is likely to coincide with the availability of CVC3 as a new prover backend for ESC/Java2. Since CVC3 has direct support for partiality, it will be interesting to determine if the overhead of checking definedness conditions as "*side*-conditions" (when considered in the context of a classical prover) can be reduced or eliminated.

Finally, this will lead us to stage two of our planned enhancements to ESC/Java2, namely the addition of consistency checking of constructor and method contracts.

Acknowledgments

We gratefully acknowledge George Karabotsos' contribution to the implementation of definedness checking in ESC/Java2. The research reported here was, in part, supported by NSERC of Canada under grant 261573-03.

References

[1] J. Barnes, *High Integrity Software: The Spark Approach to Safety and Security*. Addison-Wesley, 2003.

[2] L. Burdy, A. Requet, and J.-L. Lanet, "Java Applet Correctness: A Developer-Oriented Approach". *Proceedings of the International Symposium of Formal Methods Europe*, vol. 2805 of *LNCS*. Springer, 2003.

[3] P. Chalin, "Logical Foundations of Program Assertions: What do Practitioners Want?" *Proceedings of the Third International Conference on Software Engineering and Formal Methods (SEFM'05)*, Koblenz, Germany, September 5-9. IEEE Computer Society Press, 2005.

[4] P. Chalin, "Reassessing JML's Logical Foundation". *Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP'05)*, Glasgow, Scotland, July, 2005.

[5] P. Chalin, "De-risking the Verifying Compiler Project: Recovering Soundness", Dependable Software Research Group, Department of Computer Science and Software Engineering, Concordia University, ENCS-CSE-TR 2005-009, 2006.

[6] P. Chalin, J. Kiniry, G. T. Leavens, and E. Poll, "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2". *Fourth International Symposium on Formal Methods for Components and Objects (FMCO'05)*, 2005.

[7] D. R. Cok and J. R. Kiniry, "ESC/Java2: Uniting ESC/Java and JML". In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean editors, *Proceedings of the International Workshop on the Construction and Analysis of Safe,*

Secure, and Interoperable Smart Devices (CASSIS'04), Marseille, France, March 10-14, vol. 3362 of *LNCS*, pp. 108-128. Springer, 2004.

[8] R. DeLine and K. R. M. Leino, "BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs", Microsoft Research, Technical Report, 2005.

[9] D. L. Detlefs, G. Nelson, and J. B. Saxe, "A Theorem Prover For Program Checking", Compaq SRC, Research Report 159, 2002.

[10] R. B. Findler and M. Felleisen, "Contract Soundness for Object-Oriented Languages". *16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*, Tampa Bay, FL, USA, October 14 - 18. ACM Press, 2001.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java". *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, June, vol. 37(5), pp. 234-245. ACM Press, 2002.

[12] D. Gries and F. B. Schneider, "Avoiding the Undefined by Underspecification", in *Computer Science Today: Recent Trends and Developments*, vol. 1000, J. v. Leeuwen, Ed.: Springer-Verlag, 1995, pp. 366-373.

[13] J. Grundy, "Predicative programming--A survey". *International Conference Formal Methods in Programming and Their Applications*, Novosibirsk, Russia, June 28 – July 2. Springer, 1993.

[14] C. A. R. Hoare and J. He, *Unifying Theories of Programming*. Prentice Hall, 1998.

[15] C. B. Jones, *Systematic Software Development using VDM*, 2nd ed. PHI, 1990.

[16] C. B. Jones and C. A. Middelburg, "A Typed Logic of Partial Functions Reconstructed Classically", *Acta Informatica*, 31(5):399-430, 1994.

[17] J. R. Kiniry, P. Chalin, and C. Hurlin, "Integrating Static Checking and Interactive Verification: Supporting Multiple Theories and Provers in Verification". *Proceedings of the International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, Zürich, Switzerland, October 10-13, 2005.

[18] B. Konikowska, "Two Over Three: A Two-Valued Logic for Software Specification and Validation Over a Three-Valued Predicate Calculus", *Journal of Applied Non-Classical Logics*, 3:39-71, 1993.

[19] B. Konikowska, A. Tarlecki, and A. Blikle, "A Three-valued Logic for Software Specification and Validation". *Second VDM Europe Symposium. VDM - The Way Ahead (VDM'88)*, Dublin, Ireland, September. Springer, 1988.

[20] P. G. Larsen and N. Plat, "Introduction to Overture". *First Overture Workshop*, Newcastle upon Tyne, UK, July, 18, 2005.

[21] G. T. Leavens, "JML's Rich, Inherited Specifications for Behavioral Subtypes", Department of Computer Science, Iowa State University, Ames, Iowa. USA, TR #06-22, 2006.

[22] G. T. Leavens and Y. Cheon, "Design by Contract with JML", www.jmlspecs.org, 2006.

[23] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, "How the design of JML accommodates both runtime assertion checking and formal verification", *Science of Computer Programming*, 55(1-3):185-208, 2005.

[24] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin, "JML Reference Manual", http://www.jmlspecs.org, 2006.

[25] K. R. M. Leino, "Ecstatic: An object-oriented programming language with an axiomatic semantics". *Fourth International Workshop on Foundations of Object-Oriented Languages*, January, 1997.

[26] K. R. M. Leino, J. B. Saxe, and R. Stata, "Checking Java programs via guarded commands", COMPAQ SRC, Palo Alto, CA, SRC Technical Note 1999-002. 21 May 1999, 1999.

[27] C. Marché, C. Paulin-Mohring, and X. Urbain, "The Krakatoa tool for certification of Java/JavaCard programs annotated in JML", *Journal of Logic and Algebraic Programming*, 58(1-2):89-106, 2004.

[28] B. Meyer, "Applying Design by Contract", *Computer*, 25(10):40-51, 1992.

[29] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice-Hall, 1997.

[30] J. M. Morris, "Non-deterministic expressions and predicate transformers", *Information Processing Letters*, 61(5):241-246, 1997.

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

[32] J. M. Spivey, *The Z Notation: A Reference Manual*. Prentice-Hall, 1989.

[33] SRI International, "The PVS Specification and Verification System", http://pvs.csl.sri.com.

[34] J. van den Berg and B. Jacobs, "The LOOP compiler for Java and JML". In T. Margaria and W. Yi editors, *Proceedings of the Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, vol. 2031 of *LNCS*, pp. 299-312. Springer, 2001.

Experiments in the use of τ-simulations for the components-verification of real-time systems

Françoise Bellegarde, Jacques Julliand, Hassan Mountassir, Emilie Oudot

LIFC - Laboratoire d'Informatique de l'Université de Franche-Comté
FRE CNRS 2661
16, route de Gray,
25030 Besançon Cedex, France
Ph:(33) 3 81 66 66 51, Fax:(33) 3 81 66 64 50
{bellegar,julliand,mountass,oudot}@lifc.univ-fcomte.fr

ABSTRACT

We present a verification framework exploiting τ-simulations as a way to preserve local linear properties checked on the components of real-time systems. Therefore, we consider a component-based modeling of real-time systems. Their properties are expressed in a timed logic, MITL (Metric Interval Temporal Logic).
For component-based models, traditional verification techniques are generally applied to the complete composed model, even if some properties only concern some components of the system, if not only one. We show that it is possible to check such local linear properties (safety as well as liveness) on the components they concern, and then to ensure their preservation using τ-simulation relations. We show the interest of the method by applying it on two real-time systems examples and by comparing the results with traditional verification techniques.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods,Model-checking*; I.6.5 [**Simulation and modeling**]: Model Development—*Modeling methodologies*

General Terms

Verification, Experimentation

Keywords

τ-simulation, integration of components, timed systems, preservation of linear-time properties

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006), November 10–11, 2006, Portland, Oregon, USA.

1. MOTIVATIONS

Component-based modeling is a modeling method that receives more and more attention. In particular, timed systems are often modeled this way. First, it consists in decomposing the system into a set of sub-systems, called components. The complete model is obtained by putting all the components together, thanks to some parallel composition operator. To ensure their correctness, such models have global requirements to meet, i.e., requirements about the behaviour of the complete model, as well as so-called local requirements. Local requirements are properties concerning the components (or subsets of components) of the system.
Model-checking is a verification method that can be used in order to verify these properties on the model. For both kinds of properties, the verification is performed on the global model. However, this method is generally not applicable on large-sized systems, due to the exponential blow-up of the state space.
We propose a verification method for local linear properties of real-time systems modeled in a compositional framework, by taking advantage of the modeling process. We propose to model systems incrementally, by integration of components: instead of building once the complete model, components (or assembling of components) are integrated step by step to the others, and local properties are checked on these components (or assembling of components) before integration. Model-checking is still applicable since the size of each component / assembling of components is small enough.
To ensure that locally established properties continue to hold after integration, we propose to use timed τ-simulations, i.e., simulation relations extended to handle timing aspects and internal activity of the models. Indeed, it is known that in the untimed case, classic τ-simulations preserve linear safety properties. Divergence-sensitive and stability-respecting τ-simulations [12] also handle the preservation of linear liveness properties. For that matter, this last kind of τ-simulation is already used in other incremental modeling processes, such as the refinement of B event systems [1], which preserves LTL properties [9].
In [5], we defined a timed τ-simulation adapted for timed systems and showed that it preserves safety (un)timed properties. Moreover, we showed that it is well-adapted with integration of components made with the classic parallel com-

position operator ||. That is, given two components A and B, A τ-simulates $A||B$ and B τ-simulates $A||B$. In terms of preservation of properties, this means that linear safety properties of A and B are preserved *for free* when A and B are merged together. The timed τ-simulation is also compatible with the operator ||: given a third component C, if B τ-simulates A, then $B||C$ τ-simulates $A||C$. Compatibility allows to benefit of the compositionnality property: given components A, B, C and D, if B τ-simulates A and D τ-simulates C, then $B||D$ τ-simulates $A||C$. Thus, this first timed τ-simulation has nice properties w.r.t. parallel composition. However, as it only preserves safety properties, we extended it into a divergence-sensitive and stability-respecting (DS) timed τ-simulation to preserve a larger spectrum of properties, such as liveness or bounded liveness properties. We proved in [5] this new relation preserves all properties which can be expressed by the linear-time logic MITL [4].

In this paper, we aim at showing the interest of the verification method we propose by applying it to the verification of the local properties of two timed systems. In both examples, we compare the application of our method with the classic verification method, consisting in verifying directly all the properties on the complete model. The first example is a production cell, made up of at least seven components. We focus in particular on the local properties of one of these components, and perform the verification locally on this component before verifying the preservation thanks to the timed τ-simulations. This example shows that the cost of this kind of verification (local verification and preservation checking) is lower than the cost of the classic verification method, in terms of computation times. The second example is a well-known protocol, the CSMA/CD protocol, composed of a medium and at least two senders. With this example, we show that one of the main properties of the protocol can be checked by only considering the smallest possible number of senders, i.e. two, instead of verifying it with a greater number of senders. Moreover, in both examples, we identify cases where the classic verification method is not applicable while the method we propose is.

The paper is organized as follows. In section 2, we present the model we use for modeling timed systems – timed automata [3], the logical formalism MITL used to formulate their properties and the parallel composition || used for the integration of components. In section 3, we recall the definitions and results of [5], where we defined timed τ-simulations and gave its properties w.r.t. the preservation of MITL properties and the compatibility with parallel composition. Section 4 shows the interest in practice of the verification method we propose, by applying it to two examples of timed systems and comparing the results with the classic verification method. Finally, section 5 sums up the results presented in the paper and plans the future works.

Related works. Numerous works have been devoted to the study of timed simulation relations and their preservation ability. A time-abstracting simulation has been studied in [13], where an algorithm to check the relation is proposed. However, timed properties are not preserved by this relation. A timed simulation was defined in [17]. The authors showed that the problem of verifying the existence of this timed simulation is EXPTIME. But, internal activity is not handled by this relation. The closest notion of simulation, in comparison with our timed τ-simulation, is the timed ready simulation of [14]. Internal activity is also considered. As our (DS) timed τ-simulation, this relation also handles the preservation of safety properties, but does not preserve other kind of properties, such that liveness properties. To our knowledge, there is no previous definition or use of simulation relations handling internal activity as well as timing constraints, and preserving safety and, especially, liveness properties.

2. MODELING TIMED SYSTEMS

Timed automata [3] are amongst the most studied formal models for timed systems. They are classical finite automata with real-valued variables called clocks modeling the time elapsing.

2.1 Clock valuations and clocks constraints

Let X be a set of clocks. A clock valuation over X is a function $v : X \to R^+$ assigning to each clock in X a real value. For $\delta \in \mathbb{R}^+$, the valuation $v + \delta$ is obtained by adding δ to the value of each clock. Given $Y \subseteq X$, and a valuation v over X, the dimension-restricting projection of v on Y [18], written $v\rfloor_Y$, is a new valuation containing only the values in v concerning the clocks in Y. The *reset operation* on v of the clocks in Y, written $[Y := 0]v$, creates a valuation obtained from v by setting to zero all clocks in Y, and leaving the values of other clocks ($\in X \backslash Y$) unchanged. A clock constraint over X is a predicate of the form

$$g ::= x \sim c \mid g \wedge g \mid \texttt{true} \text{ where } x \in X, c \in \mathbb{N} \text{ and } \sim \in \{<, \leq, =, \geq, >\}$$

The set of all clock constraints over X is called $\mathcal{C}(X)$. We say that a valuation v satisfies a constraint $x \sim c$, written $v \in x \sim c$, if $v(x) \sim c$. Note that a clock constraint over X defines a X-polyhedron. The *reset operation* defined on valuations can be extended straightforwardly to polyhedra. The *backward diagonal projection* of a polyhedron g defines a new polyhedron $\swarrow g$ s.t. $v' \in \swarrow g$ if $\exists \delta \in \mathbb{R}^+ \cdot v' + \delta \in g$.

2.2 Timed automata (TA)

Syntax. Let *Props* be a set of atomic propositions. A timed automaton A over *Props* is a tuple $\langle Q, q_0, \texttt{Labels}, X, T, \texttt{Invar}, L\rangle$ where Q is a finite set of locations, q_0 is the initial location, *Labels* is a finite set of names of actions and X is a finite set of clocks. $\texttt{Invar} : Q \to \mathcal{C}(X)$ is a function associating to each location a clock constraint called its invariant. The invariant of a location defines the time progress condition for this location. $L : Q \to 2^{Props}$ is the labelling function for the locations. $T \subseteq Q \times \mathcal{C}(X) \times \texttt{Labels} \times 2^X \times Q$ is a finite set of transitions. Each transition can reset clocks and is equiped with a clock constraint called its guard, defining *when* the transition can be taken. We write a transition e as a tuple (q, g, a, λ, q') where q and q' are respectively the source and target location of the transition, g is its guard, a its label and λ the set of clocks to be reset by the transition. In the rest of the paper, we use the notations $\texttt{source}(e)$ and $\texttt{target}(e)$ for q and q', $\texttt{guard}(e)$ for g, $\texttt{label}(e)$ for a and

$\texttt{reset}(e)$ for λ.

Semantics. The semantics of A is an infinite graph where states[1] are pairs (q, v) composed of a location q of A (the discrete part of the state) and a clock valuation v s.t. $v \in \texttt{Invar}(q)$. The initial state is the pair (q_0, v_0) where v_0 is the clock valuation assigning 0 to each clock in X. The transitions of the graph are either discrete transitions or time transitions:

- discrete transitions: given a transition $e = (q, g, a, \lambda, q')$ of A, $(q, v) \xrightarrow{e} (q', v')$ is a discrete transition of the graph if $v \in g$ and $v' \in \texttt{Invar}(q')$. The valuation v' is obtained from v by resetting the clocks in λ. We call (q', v') a discrete successor of (q, v),
- time transitions: $(q, v) \xrightarrow{\delta} (q, v+\delta)$ is a time transition of the graph, for $\delta \in \mathbb{R}^+$, if $v + \delta \in \texttt{Invar}(q)$. We call $(q, v + \delta)$ a time successor of (q, v).

Runs. A run $\rho = (q_0, v_0) \xrightarrow{\delta_0} (q_0, v_1) \xrightarrow{e_0} (q_1, v_2) \xrightarrow{\delta_1} (q_1, v_3) \xrightarrow{\delta_2} (q_1, v_4) \xrightarrow{e_1} (q_2, v_5) \cdots$ of a TA A is a path in its semantic graph. Note that consecutive time transitions are not concatenated. We note $\Gamma(A)$ the set of runs of A. A run ρ is *non-zeno* if time can progress along ρ without upper bound. We write $\texttt{time}(\rho)$ for the total time elapsed during ρ. If $\texttt{time}(\rho) = \infty$ then ρ is called non-zeno.

2.3 Properties of timed systems

MITL (Metric Interval Temporal Logic) [4] is a logical formalism allowing to express linear timed properties. It can be viewed as an extension of the (untimed) linear logic LTL [16], where each temporal operator used in the formulas is constrained by a non singular interval with integer bounds (a singular interval is of the form $[a, a]$, i.e., it is closed and the left and right bounds are equal). MITL formulas are defined inductively by the following grammar:

$$\varphi ::= ap \mid \neg\varphi \mid \phi \vee \psi \mid \phi \, \mathcal{U}_I \, \psi$$

where ap is an atomic proposition and I is a non singular interval with integer bounds. Other classical temporal operators can be defined: $\Diamond_I \varphi = true \, \mathcal{U}_I \varphi$ (eventually φ) and $\Box_I \varphi = \neg\Diamond_I \neg\varphi$ (always φ).

MITL properties are interpreted over runs of a timed automaton. Intuitively, a property $\phi \, \mathcal{U}_I \, \psi$ holds on a run if, when ϕ is met, ϕ holds until ψ is true. Moreover, ψ must hold at a time t within the time interval I following the moment when ϕ was true. An MITL property is satisfied by a timed automaton if it holds on each run of the automaton.

2.4 Integration of components

Consider a timed system composed of a set of components A_1, A_2, $\cdots$, A_n, each one modeled by a TA. Integration of components is a type of incremental modeling, which consists in first considering one component, for instance A_1. Then, the other components are successively added to A_1, until obtaining the complete system. We consider that this integration is achieved by using the classic parallel composition operator $||$. This composition is defined as a synchronized product where the synchronizations are done on actions with the same label, while other actions interleave. Formally, consider two TA $A_i = \langle Q_i, q_{0_i}, \texttt{Labels}_i, X_i, T_i, \texttt{Invar}_i, L_i \rangle$, for $i = 1, 2$, s.t. $X_1 \cap X_2 \neq \emptyset$. The parallel composition of A_1 and A_2, written $A_1 || A_2$, creates a new TA which set of clocks is $X_1 \cup X_2$ and which labels are the set $\texttt{Labels}_1 \cup \texttt{Labels}_2$. The set Q of locations consists of pairs (q_1, q_2) composed of a location of each A_i. $\texttt{Invar}((q_1, q_2))$ is defined as $\texttt{Invar}(q_1) \wedge \texttt{Invar}(q_2)$ and $L((q_1, q_2))$ is the set $L(q_1) \cup L(q_2)$. The initial location is the pair (q_{0_1}, q_{0_2}). The set of transitions T is given by the three following rules:

Synchronization: $$\frac{(q_1,g_1,a,\lambda_1,q_1') \in T_1 \ , \ (q_2,g_2,a,\lambda_2,q_2') \in T_2}{((q_1,q_2),g_1 \wedge g_2,a,\lambda_1 \cup \lambda_2,(q_1',q_2')) \in T}$$

Interleaving: $$\frac{(q_1,q_2) \in Q \ , \ (q_1,g_1,a,\lambda_1,q_1') \in T_1 \ , \ a \notin \texttt{Labels}_2}{((q_1,q_2),g_1,a,\lambda_1,(q_1',q_2)) \in T}$$
$$\frac{(q_1,q_2) \in Q \ , \ (q_2,g_2,a,\lambda_2,q_2') \in T_2 \ , \ a \notin \texttt{Labels}_1}{((q_1,q_2),g_2,a,\lambda_2,(q_1,q_2')) \in T}$$

Incremental modeling, and in particular integration of components, is a way to cope with the complexity of the verification when large-sized model are treated. Indeed, it could allow to verify local properties of the components at each step of the modeling, i.e., at each integration of components, instead of performing the verification directly on the complete system. However, this reasoning is valid only if the integration preserves the properties already checked.

3. EXPLOITING SIMULATIONS TO PRESERVE LOCAL PROPERTIES

We showed in [5] that timed τ-simulations are sufficient conditions for the preservation of MITL properties during incremental modeling, and in particular, integration of components. That is, if a component τ-simulates the whole system w.r.t. some τ-simulation relation, then already established properties of the component are preserved after integration in its environment. We defined in [5] two τ-simulation relations: one dealing with safety MITL properties (called a *timed τ-simulation*)[2], and the other to handle all MITL properties (*divergence-sensitive and stability-respecting timed τ-simulation*).

3.1 Timed τ-simulations

Consider a TA A_1 that has to be integrated in an environment E. We note $A_2 = A_1 || E$ the result of the integration. Consider also that A_1 and E have a common subset of labels of actions – the synchronized actions –, i.e., $\texttt{labels}_{A_1} \cap \texttt{labels}_E \neq \emptyset$. Let us rename by τ all the labels of the own actions of the environment (the non-common labels of E) and call them τ-actions.
The *timed τ-simulation* $\mathcal{S}$ between the semantic graphs of A_2 and A_1 is characterized by the following points: if A_2 can make an action of A_1 after some amount of time (that is, either a synchronized action or an own action of A_1), then

[1] In the rest of the paper, we directly call these states, the states of A, instead of the states of the semantic graph of A.

[2] Note that, although *deadlock-freedom* properties can be classified as safety properties, we do not consider them like this, but as a separate class of properties. Thus, this kind of properties are not preserved by the timed τ-simulation.

A_1 could also do the same action after the same amount of time (clauses 1 and 2 of Definition 1), and own actions of E (τ-actions) stutter (clause 3).

DEFINITION 1 (TIMED τ-SIMULATION $\mathcal{S}$). *Let* $A_1 = \langle \mathrm{Q}_1, \mathrm{q}_{0_1}, \mathtt{Labels}_1, \mathrm{X}_1, \mathrm{T}_1, \mathtt{Invar}_1, \mathrm{L}_1 \rangle$ *and* $A_2 = \langle \mathrm{Q}_2, \mathrm{q}_{0_2}, \mathtt{Labels}_1 \cup \{\tau\}, \mathrm{X}_2, \mathrm{T}_2, \mathtt{Invar}_2, \mathrm{L}_2 \rangle$ *be two TA such that* $X_1 \subseteq X_2$. *S_1 and S_2 are the respective set of states of A_1 and A_2. The relation $\mathcal{S}$ is included in $S_2 \times S_1$. We say that $(q_2, v_2)\mathcal{S}(q_1, v_1)$ if $v_2\rfloor_{X_1} = v_1$ and*

1. *Strict simulation:*
$(q_2, v_2) \xrightarrow{e_2} (q_2', v_2') \wedge \mathtt{label}(e_2) \in \Sigma_1 \Rightarrow \exists (q_1', v_1') \cdot$
$((q_1, v_1) \xrightarrow{e_1} (q_1', v_1') \wedge \mathtt{label}(e_1) = \mathtt{label}(e_2) \wedge (q_2', v_2')\ \mathcal{S}\ (q_1', v_1'))$.
2. *Time transitions:*
$(q_2, v_2) \xrightarrow{\delta} (q_2, v_2') \Rightarrow$
$\exists (q_1, v_1') \cdot ((q_1, v_1) \xrightarrow{\delta} (q_1, v_1') \wedge (q_2, v_2')\ \mathcal{S}\ (q_1, v_1'))$.
3. *Stuttering:*
$(q_2, v_2) \xrightarrow{e_2} (q_2', v_2') \wedge \mathtt{label}(e_2) = \tau \Rightarrow (q_2', v_2')\ \mathcal{S}\ (q_1, v_1)$.

We say that A_1 τ-simulates A_2 w.r.t. $\mathcal{S}$, written $A_2 \preceq A_1$, if $s_{0_2}\ \mathcal{S}\ s_{0_1}$, where s_{0_1} and s_{0_2} are the respective initial states of A_1 and A_2.

This τ-simulation only preserves safety properties. To preserve also liveness properties, it has to be (1) *stability - respecting*: the integration of A_1 in E must not create deadlocks in comparison with A_1 (all the deadlocks appearing in $A_1||E$ must already exist in A_1), and (2) *divergence-sensitive*: the own actions of the environment E must not have the possibility to take the control forever, i.e., $A_1||E$ must not contain non-zeno runs composed of an infinite sequence of successive τ-actions. Such a run is called a non-zeno τ-cycle, and a TA containing such a run is called τ-divergent.

The predicate `free`. To deal with the non-introduction of deadlocks during the integration, we use the predicate `free` introduced in [18]. Informally, $\mathtt{free}(q)$ computes the set of valuations of the states with discrete part q that can let some time pass and then take a discrete transition (i.e., non-blocking states):

$$\mathtt{free}(q) = \bigcup_{e \in \mathtt{out}(q)} \swarrow (\mathtt{guard}(e) \cap ([\mathtt{reset}(e) := 0]\mathtt{Invar}(\mathtt{target}(e))))$$

where $\mathtt{out}(q)$ is the set of discrete transitions leaving from q.

Non-zeno τ-cycles. Let A be a TA where some labels of actions are renamed by τ. We say that A does not contain any non-zeno τ-cycles (and thus that A is not τ-divergent) if:

$\forall \rho, k \cdot (\rho \in \Gamma(A) \wedge \mathtt{time}(\rho) = \infty \wedge k \geq 0 \Rightarrow$
$\exists k', e \cdot (k' \geq k \wedge (\rho, k') \xrightarrow{e} (\rho, k' + 1) \wedge \mathtt{label}(e) \neq \tau))$.

We restrict the timed τ-simulation of Def. 1 by adding conditions imposing divergence-sensitivity and stability-respect. This Divergence-sensitive and Stability-respecting (DS) timed τ-simulation is defined as follows.

DEFINITION 2 (DS TIMED τ-SIMULATION $\mathcal{S}_{ds}$). *Consider two TA* $A_1 = \langle \mathrm{Q}_1, \mathrm{q}_{0_1}, \mathtt{Labels}_1, \mathrm{X}_1, \mathrm{T}_1, \mathtt{Invar}_1, \mathrm{L}_1 \rangle$ *and* $A_2 = \langle \mathrm{Q}_2, \mathrm{q}_{0_2}, \mathtt{Labels}_1 \cup \{\tau\}, \mathrm{X}_2, \mathrm{T}_2, \mathtt{Invar}_2, \mathrm{L}_2 \rangle$, *such that* $X_1 \subseteq X_2$ *and A_2 is not τ-divergent. S_1 and S_2 are the respective set of states of A_1 and A_2. The relation $\mathcal{S}_{ds}$ is included in $S_2 \times S_1$. We say that $(q_2, v_2)\mathcal{S}_{ds}(q_1, v_1)$ if*

$$(q_2, v_2)\mathcal{S}(q_1, v_1) \text{ and } v_2 \notin \mathtt{free}(q_2) \Rightarrow v_1 \notin \mathtt{free}(q_1)$$

We say that A_1 τ-simulates A_2 w.r.t. $\mathcal{S}_{ds}$, written $A_2 \preceq_{ds} A_1$, if $s_{0_2}\ \mathcal{S}_{ds}\ s_{0_1}$.

3.2 Properties of timed τ-simulations

We give in this section the main propositions and theorem showing the interest of the timed τ-simulations for the incremental modeling of timed systems. Indeed, the timed τ-simulation is well-adapted to the parallel composition operator $||$ (Proposition 1). Moreover, this operator does not add τ-divergence if the environment in which a component is integrated is not τ-divergent (Proposition 2). Theorem 1 is the main result, showing that the DS timed τ-simulation preserves MITL properties. Proofs can be found in [5].

PROPOSITION 1. *Let A, B, C and D be TA. Then, we have the following:*

1. $A||B \preceq A$,
2. *if* $A \preceq B$ *then* $A||C \preceq B||C$,
3. *if* $A \preceq B$ *and* $C \preceq D$ *then* $A||C \preceq B||D$.

PROPOSITION 2. *Consider two TA A and B s.t. some actions of B are renamed by τ. If B is not τ-divergent, then neither is $A||B$.*

THEOREM 1. *Let φ be a MITL property, A and B be TA. If* $A \models \varphi$ *and* $B \preceq_{ds} A$, *then* $B \models \varphi$.

Therefore, in the context of integration of components, the preservation of local safety properties can be obtained for free, since parallel composition is compatible with the timed τ-simulation. The preservation of local linear liveness properties of a component can be ensured only by checking that its integration in an environment does not create deadlocks and that this environment is not τ-divergent.

REMARK 1. *TCTL [2] is a branching-time logic which is often used to express timed properties. However, TCTL properties are not preserved by the DS timed τ-simulation. Indeed, even in the untimed case, simulation relations can not handle the preservation of branching-time properties. This kind of properties gives the possibility to add existential or universal quantifiers in the formulas. In particular, properties including existential quantifications, such as reachability properties, are not preserved (note that each MITL formula is implicitly preceded by a universal quantification, meaning that the property must hold on all runs). One often need to use bisimulation relations to ensure the preservation of such properties, but they are not appropriate to formalize incremental modeling, and thus, integration of components.*

3.3 Checking the simulations

To check the DS timed τ-simulation, we developped the tool VESTA[3]. The structure of the tool was guided by the structure of the tool OPEN-KRONOS [18], allowing an easy connection to the OPEN-CAESAR platform [11]. It takes as input two TA A_1 and A_2, where A_2 represents the integration of A_1 in an environment E, i.e. $A_1||E$.

[3]available at: `http://lifc.univ-fcomte.fr/~oudot/VeSTA`

The tool performs a joint on-the-fly depth-first search on symbolic graphs representing the two TA (the so-called *simulation graphs* [18]), and checks if $A_2 \preceq_{ds} A_1$. This verification is in $\mathcal{O}((|Z_1| + |\longrightarrow_1|) \times (|Z_2| + |\longrightarrow_2|))$, where Z_i and $\longrightarrow_i$, for $i = 1, 2$, are respectively the set of states and transitions of the simulations graphs of each A_i. If the verification fails, the tool returns a diagnostic. This diagnostic consists in a trace of A_2 containing a state which does not satisfy the relation, and the corresponding trace in A_1. To check if A_2 is τ-divergent, we use the tool PROFOUNDER [19], which can in particular detect non-zeno cycles. Thus, we use it to detect non-zeno τ-cycles first in E, then in A_2 if τ-cycles are detected in E.

4. EXPERIMENTS

We present in this section examples on which we lead experiments to show the interest of timed τ-simulations. For each case study, the verification of the properties was achieved with the tool KRONOS [20]. KRONOS is a verification tool for timed systems which performs TCTL model-checking [2], in particular for component-based models (indeed, KRONOS can compute the parallel composition of TA). TCTL is a logical formalism that allows to express branching-time properties. It can be seen as the timed extension of the untimed logic CTL [8, 10]. To our knowledge, there is no tool performing MITL model-checking. Thus, we focused on linear-time properties that can also be expressed in TCTL to lead the verification with KRONOS. The verification of the simulations was done with VESTA.

4.1 Production Cell

The production cell case study was developed by FZI (the Research Center for Information Technologies, in Karlsruhe) as part of the Korso project. The goal was to study the impact of the use of formal methods when treating industrial applications. Thus, this case study was treated in about thirty different formalisms. We treat it with timed automata, as it was in [7].

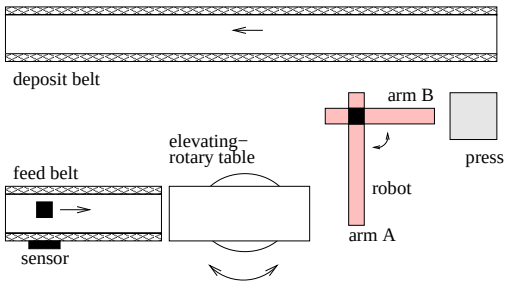


Figure 1: The Production Cell

Modeling. The cell consists of six devices: a feed-belt and a deposit-belt, from which pieces to be treated arrive and are evacuated, a sensor detecting the arrival of the pieces, an elevating rotary table, a two-arms robot and a press (see Fig. 1). The sensor, on the feed-belt, detects when a piece passes in front of it and sends a signal to the robot to inform that a piece is going to be available. When the piece arrives at the end of the feed-belt, it is transfered to the table which goes up while turning until being in a position where the piece can be taken by the arm A of the robot. The robot turns 90° so that the arm A can put down the piece on the press, where it is processed and then transported by the arm B to the deposit belt.
In the following, we focus on local properties concerning the robot. Thus, let us give details about its behaviour: when a piece is available on the table, the robot picks it up and moves to the press so that its arm B is in front of the press. If there is a piece on the press, the arm B takes it, otherwise the robot goes on turning to place the arm A in front of the press to put its piece down. Next, if the arm B is full, the robot goes to the deposit belt to unload its piece (and then goes back to the table), else it goes to an intermediary position, called wait position. From this wait position, the robot can either go back to the table to pick a new piece up or to the press to get a processed piece.
The cell is modeled by at least seven components, one for each device, and one or several pieces. Each component is modeled by a TA. These TA can be found in [7]. The complete model is obtained by making the parallel composition of all these components. The timing constraints of the plant are shown in Fig. 2. We give in Fig. 3 the size, in terms of number of states and transitions, of the simulation graphs of each component.

Device	Description	Time
Robot	moves to press	5
Robot	turns 90°	15
Robot	moves to deposit belt	5
Robot	from deposit belt to table	25
Robot	from deposit belt to wait pos.	22
Robot	from press to wait pos.	17
Robot	from wait pos. to table	3
Robot	from wait pos. to press	2
Robot	at wait pos.	2
Feed Belt	piece moves to sensor	3
Feed Belt	piece moves to table	1
Table	raises and turns	2
Table	returns and turns	2
Press	presses a piece	22-25
Press	ready for a new piece	18-20
Deposit Belt	evacuates a piece	4

Figure 2: Timing constraints for the production cell

Component	Robot	Press	Feed belt	Dep. Belt
States/Trans.	39/40	7/7	6/6	4/4
Component	Table	Sensor	Piece	Complete Model
States/Trans.	6/6	2/2	7/7	1655/2395

Figure 3: Size of the simulation graphs of each component of the production cell

Verification. To ensure that the modeling is correct, there are several properties to check. We focus on the local properties of the components, and in particular on the local properties concerning the robot. Here is a non-exhaustive list of dynamic properties to check on the robot component. Properties 1 and 2 are safety requirements, properties 3 and 4 are liveness requirements and properties 5 to 7 are bounded-response requirements:

(1) When the robot is in wait position, its two arms are empty,

(2) The robot is not waiting in front of the table if the arm A is full,

(3) If there is a piece on arm B, the robot will eventually go to the deposit belt,

(4) If there is a piece on arm A, the robot will eventually go to the press,

(5) When the robot is in front of the deposit belt, then it goes back to the table within 25 time units (t.u.) if there are no pieces on the press,

(6) When the robot is in front of the deposit belt, then it goes to the wait position within 22 t.u. if there is a piece on the press,

(7) When it is in wait position, either the robot goes to the press within 2 t.u. to unload it or it goes back to the table within 3 t.u. to pick a new piece.

The following liveness property concerns the correct interaction between the robot and the press :

(8) If arm A is full then the press will eventually be free.

We used two approaches to verify these properties on the plant. As a first approach, we verified the properties in a classic way, directly on the global model, with only one piece. As a result, we obtained that all the properties hold on this model of the plant. The second approach consisted in verifying the properties locally, i.e., on the robot component for properties 1 to 7, and on the composition $robot||press$ for property 8. Here again, the verification succeeded. Next, to guarantee the preservation of these properties, first when integrating the robot with the press, then when integrating the composition $robot||press$ with the rest of the components, we used the DS timed τ-simulation. We used our tool VeSTA to check it for both cases, and obtained as a result that the verification was successful. Thus, properties are preserved.

Fig. 4 gives the results of the comparison of the two approaches in terms of verification times (in seconds). We can see that, even on this small example, the second approach only needs 0,57 sec. of computation time to ensure that the properties hold on the cell, whereas the classic approach consumes 19,58 sec.
Note that, in both approaches, we focused on a global system which contains only one piece. The reason is the following. First, in the case of the second approach, adding other pieces to the global system does not affect the results of the preservation. As the component piece already exists in the global system and that there are no synchronizations between the pieces, no new deadlocks can appear while adding a new piece. Indeed, the system can behave like it did with only one piece, or synchronize with the new piece. In this last case, as the environment of the pieces could synchronize with one piece without introducing deadlocks, then it will synchronize with the new pieces in the same way, thus without introducing deadlocks. On the other hand, in the first approach, adding pieces considerably increase the computation time for the verification of liveness or bounded-response properties. Indeed, even with few pieces, the memory needed to perform classic verification of such properties is too large for the verification to be run to completion.

4.2 CSMA/CD protocol

The CSMA/CD protocol (Carrier Sense, Multiple Access with Collision Detection protocol) [15] is used in broadcast networks with a single channel, to which many stations try to access. The protocol solves the problem of how to assign the use of the channel to one of the stations and allows to detect collisions when two stations try to send data simultaneously.

Modeling. The protocol works as follows: a station tries to send a data. If the channel is busy, it waits some amount of time and then retries to send. Otherwise, it begins to send its data. If two or more stations begin to send a data simultaneously, and thus a collision arises, these stations detect the collision and wait some amount of time to begin retransmitting the data.
At least two TA are used to model the protocol: one per station, called sender, and a medium. These TA are shown in Fig. 5. Transitions cd_i represent a collision detection by the i^{th} sender, and cd_M models this detection by the medium. The complete model of the protocol can contain several senders. For n senders, the synchronizations are the following: for $a \in \{begin, busy, end\}$, there is a synchronized transition $a_i||{a_M}^4$, for $i \in [1..n]$. Moreover, the cd transitions of each TA are synchronized, i.e., in the parallel composition, there is a transition $cd_M||cd_1||\cdots||cd_n$.

Verification. A main property of the protocol is that *whatever the number of stations, if a collision occurs between two stations i and j, $i \neq j$, both detect it within 26 t.u.* This is a bounded-response property, written in MITL by:

$$\Box(transm_i \wedge transm_j \Rightarrow \Diamond_{\leq 26}(coll\_detected_i \wedge coll\_detected_j)).$$

This property holds in the case of a modeling with only two senders S_1 and S_2. The verification with KRONOS takes less than 0.001 seconds of computation time. We want to ensure that adding other senders does not alter the result of the verification. That is, we want that, with a number $n > 2$ of senders, if S_1 and S_2 transmit their data simultaneously, they still detect the collision. To preserve this bounded-liveness property, we have to check that

$$S_1||S_2||S_3||\cdots||S_n||Medium \preceq_{ds} S_1||S_2||Medium.$$

As the senders S_i are not τ-divergent, and with Proposition 1, we only have to check that the addition of $S_3, \cdots, S_n$ does not add deadlocks to $S_1||S_2||Medium$.
As in the production cell example, there already exist two senders in the system. Thus, when adding a new sender, the synchronizations of this sender with the medium (actions *begin*, *end* and *busy*) can take place as they did with only two senders. Thus, no deadlocks can be introduced by these synchronizations. The main difference with the production

[4]Note that, for more simplicity, we extend here the notation || initially defined on timed automata, to transitions. Thus, $a||b$ denotes the transition resulting of the synchronization of transitions a and b.

Property	Global Verification	Local Verification	Preservation checking
Prop. 1	0.01	< 0.001	
Prop. 2	0.01	< 0.001	
Prop. 3	0.98	< 0.001	
Prop. 4	15.79	0.04	0.05
Prop. 5	0.68	< 0.001	
Prop. 6	0.48	< 0.001	
Prop. 7	0.7	< 0.001	
Prop. 8	0.93	0.02	0.46
Total	19.58	0.06	0.51

Figure 4: Production cell : local and global verification times (in seconds)

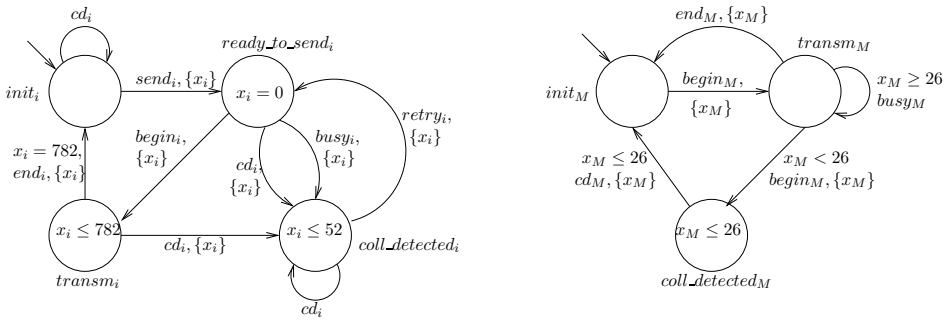


Figure 5: Timed automata for the ith sender and the medium of the CSMA/CD protocol

cell example is that there exists synchronizations between all the components of the system, i.e., all the senders and the medium (action *cd*). That is, when two senders detect a collision and try to take action *cd* between the locations $\texttt{transm}_\texttt{i}$ and $\texttt{coll\_detected}_\texttt{i}$, all other senders must allow them to detect the collision and thus, must allow them to take this transition. If not, a new deadlock occurs, since the synchronization can not take place, whereas it could be taken with two senders. However, we see that, at each location of the TA of the senders, the transition *cd* appears. This means that whenever a collision must be detected by two senders (i.e., taking transition *cd* between the locations $transm_i$ and $coll\_detected_i$), the other senders also have the possibility to take a transition `cd`, allowing the two first senders to detect the collision. No deadlocks can appear while adding new senders, and thus, the property is preserved whatever the number of senders may be.
We compared our approach with a classic verification approach, and tried to verify this property, for instance, for two senders S_1 and S_2, using the tool KRONOS:

- Up to six senders (S_1 to S_6), the property can be checked successfully. The computation times for the verification changed from less than 0,5 seconds (three senders) to more than 57 minutes (six senders). These computations times take into account the time consumed to make the parallel composition of all the components and the verification time. Note that, in the last case, the verification time takes about 30 seconds, while the construction of the composition takes almost 57 minutes.

- For seven senders or more, the construction of the TA resulting of the parallel composition of all the components takes a considerably long time. For instance, we aborted the construction for seven senders after ten hours of computation without results. Thus, as the composition could not be obtained, it was impossible to perform the verification of the property.

5. CONCLUSION AND FUTURE WORKS

We aimed at treating the problem of the verification of linear-time properties, expressed in MITL, of timed systems. We considered timed systems modeled in a compositional framework, and focused on the verification of local properties of the components.

In [5], we proposed to use timed τ-simulations as a way to verify these properties at a lower cost. The main thesis is that a local linear property can be checked only on the component it concerns, instead of on the complete composed system, and that the preservation of the property when integrating the component in its environment can be checked by means of timed τ-simulation relations. More precisely, we defined two such relations: a timed τ-simulation handling the preservation of linear safety properties, and a divergence-sensitive and stability-respecting timed τ-simulation for the preservation of all MITL properties, in particular liveness properties.

In this paper, we studied the impact in practice of this methodology. We applied it to two timed systems, and compared the results with the ones obtained by a direct verification on the complete system. It turns out that, in terms of computation times, the methodology appears to be more efficient. Moreover, both examples show the interest of the methodology when a system S is susceptible to include an indeterminate number n of components $C_{i,i=1..n}$ of the same

"kind", i.e., $S = E||C_1||\cdots||C_n$. In this case, we say that S has n as a parameter. For instance, the parameter of the production cell is the number of pieces on the cell, while in the CSMA/CD protocol, it is the number of senders. In such a parametrized system, the interesting cases are when the verification can be performed with a small fixed number l of such components, that is $S_l = E||C_1||\cdots||C_l$, implying the preservation of the properties on $S_m = E||C_1||\cdots||C_m$, $\forall m \geq l$, under some simple conditions. As the parallel composition is compatible with the timed τ-simulation, it is enough that the conditions ensure that adding the $C_i s$ does not introduce any deadlocks. For instance, for the production cell example, the condition is that there are no synchronizations between the pieces. Thus, an interesting work is to determine such simple conditions.

Another work direction would be to study the compatibility of the timed τ-simulations with other composition operators, in particular those preserving deadlock-freedom, such as the one presented in [6]. Indeed, this analysis would allow to evaluate the interest of the methodology we propose in a more general framework, i.e., also for systems using composition paradigms which differ from the classic parallel composition one.

6. REFERENCES

[1] J.-R. Abrial. Extending B without changing it (for developing distributed systems). In *1st Conference on the B method*, pages 169–190, Nantes, France, November 1996.

[2] R. Alur, C. Courcoubetis, and D. Dill. Model-Checking in Dense Real-time. *Information and Computation*, 104(1):2–34, 1993.

[3] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[4] R. Alur, T. Feder, and T. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43:116–146, 1996.

[5] F. Bellegarde, J. Julliand, H. Mountassir, and E. Oudot. On the contribution of a τ-simulation in the incremental modeling of timed systems. In *Proceedings of the 2^{nd} International Workshop on Formal Aspects of Component Software (FACS'05)*, volume 160 of *Electronic Notes in Theoretical Computer Science*, pages 97–111, Macao, Macao, October 2005. Elsevier.

[6] S. Bornot, J. Sifakis, and S. Tripakis. Modeling Urgency in Timed Systems. In *COMPOS'97*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[7] A. Burns. How to verify a safe real-time system: The application of model-checking and timed automata to the production cell case study. *Real-Time Systems Journal*, 24(2):135–152, 2003.

[8] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[9] C. Darlot, J. Julliand, and O. Kouchnarenko. Refinement Preserves PLTL Properties. In *Proceedings of 3^{rd} International Conference on B and Z Users (ZB'03)*, volume 2651 of *Lecture Notes in Computer Science*, pages 408–420, Turku, Finlande, June 2003. Springer-Verlag.

[10] E. Emerson and J. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of the 14^{th} ACM Symp. Theory of Computing (STOC'82)*, pages 169–180, San Francisco, CA, USA, May 1982.

[11] H. Garavel. OPEN/CAESAR: An Open Software Architecture for Verification, Simulation and Testing. In B. Steffen, editor, *Proceedings of 1^{st} International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, Lisboa, Portugal, March 1998.

[12] R. v. Glabbeck. The Linear Time - Branching Time Spectrum II ; The semantics of sequential systems with silent moves. In *Proceedings of 4^{th} international Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81, Hildesheim, Germany, august 1993. Springer-Verlag.

[13] M. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of the 36^{th} IEEE Symposium on Foundations of Computer Science*, pages 453–462, 1995.

[14] H. Jensen, K. Larsen, and A. Skou. Scaling up Uppaal : Automatic verification of real-time systems using compositionnality and abstraction. In *Proceedings of the 6^{th} international symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'00)*, pages 19–30, London, UK, 2000. Springer-Verlag.

[15] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE Transactions on Software Engineering, Special Issue on Real-Time Systems*, 18(9):794–804, September 1992.

[16] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18^{th} IEEE Symposium on Foundations Of Computer Science*, pages 46–77, 1977.

[17] S. Tasiran, R. Alur, R. Kurshan, and R. Brayton. Verifying Abstractions of Timed Systems. In *Proceedings of the 7^{th} Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 546–562, Pisa, Italy, 1996.

[18] S. Tripakis. *The analysis of timed systems in practice.* PhD thesis, Universite Joseph Fourier, Grenoble, France, December 1998.

[19] S. Tripakis, S. Yovine, and A. Bouajjani. Checking Timed Büchi Automata Emptiness Efficiently. *Formal Methods in System Design*, 26(3):267–292, May 2005.

[20] S. Yovine. Kronos: A verification tool for real-time systems. *Journal of Software Tools for Technology Transfer*, 1(1/2):123–133, October 1997.

JML-based Verification of Liveness Properties on a Class in Isolation*

Julien Groslambert
Université de Franche-Comté - LIFC - CNRS
16 route de Gray
25030 Besançon cedex
France
groslambert@lifc.univ-fcomte.fr

Jacques Julliand
Université de Franche-Comté - LIFC - CNRS
16 route de Gray
25030 Besançon cedex
France
julliand@lifc.univ-fcomte.fr

Olga Kouchnarenko
Université de Franche-Comté - LIFC - CNRS - INRIA CASSIS
16 route de Gray
25030 Besançon cedex
France
kouchna@lifc.univ-fcomte.fr

ABSTRACT

This paper proposes a way to verify temporal properties of a Java class in an extension of JML (Java Modeling Language) called JTPL (Java Temporal Pattern Language). We particularly address the verification of *liveness* properties by automatically translating the temporal properties into JML annotations for this class. This automatic translation is implemented in a tool called JAG (JML Annotation Generator). Correctness of the generated annotations ensures that the temporal property is established for the executions of the class in isolation.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Software/Program Verification—*Temporal Property Verification*

General Terms

Verification, Security

Keywords

Liveness Properties, Class in Isolation, JML, Automatic Annotation Generation

1. INTRODUCTION

Recently, significant progresses have been made in the field of smart card application verifications. The development of the Java Modeling Language project[1] (JML) is a part in these results [6]. The JML project defines a specification language which is syntactically and semantically close to Java, thus making specifications more accessible to Java programmers. JML allows adding to the Java class traditional formal annotations like method pre- and postconditions and class invariants. However, it is difficult to directly specify complex dynamic properties in JML, like temporal properties [15] that are often needed to express the security policies that the Java implementation has to ensure. For example, no JML clause permits easy expression of the followings properties

$$\begin{array}{c} \textit{“After the invocation of the method } \texttt{m} \\ \textit{the property } P \textit{ must be satisfied in all the states”}. \qquad (S) \\ \textit{“After the non-exceptional termination of the method } \texttt{m} \\ \textit{a state where } P \textit{ holds must inevitably} \\ \textit{be reached in the future”}. \qquad (L) \end{array}$$

Following [15], Property S is a safety property, expressing that *something bad* – a state where $\neg P$ holds after the invocation of `m` – must never happen whereas L is a liveness property, expressing that, under certain conditions – an invocation of `m` –, *something good* – a state where P holds – must *inevitably* happen in the future.

A key concept for reasoning modularly about safety properties is the notion of *class invariant*, i.e., a predicate over the variables of the class that must hold along all the history of all the instances of the class. Reasoning in a modular way about invariant consists in: (a) considering only the class *in isolation* [1], i.e., without regarding the program using the class, we show that the constructor of the class establishes the invariant and that each method of the class preserves it. (b) showing that under certain conditions on the program, the invariant is still satisfied.

We propose to address the expression and the verification of liveness properties following the same approach. Therefore, this paper particularly focuses on the first task, i.e., the verification of *liveness* properties in isolation.

For that, we provide in the paper (1) an execution path semantics of a Java Class in isolation and a semantics of the main JML annotations; (2) a primitive liveness operator Loop, inspired from [9], for expressing liveness properties; (3) a way to verify on a class in isolation liveness properties expressed with the Loop primitive, by generating JML annotations ensuring their satisfaction; (4) a tool, called JAG [10] (for JML Annotation Generator) implementing an automatic translation of liveness properties into standard JML annotations that ensure the satisfaction of these prop-

*Research partially funded by French Research ACI *Geccoo*.
[1]See `http://www.jmlspecs.org`.

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006), November 10–11, 2006, Portland, Oregon, USA.

erties. Translation of liveness properties is done through the primitive `Loop` operator.

We particularly focus on Java card applet. So, this paper focus on single Java Class in isolation and the problem of inheritance and sub-typing is not addressed.

The paper is outlined as follows. Section 2 quickly presents JML on an example. Section 3 presents the semantics framework of the paper. In particular, Section 3.1 defines a semantics for the class C in isolation whereas Section 3.2 gives the semantics of JML main annotations. Section 4 presents the verification of liveness properties on a class in isolation, through appropriate annotation generation. Section 5 presents the JAG tool, implementing this automatic generation of annotations. Section 6 concludes and presents the perspectives of future work.

Nota Bene: The proofs of propositions and theorems are not included in this paper, but can be found in [14].

2. OVERVIEW OF JML AND EXAMPLE

JML (Java Modeling Language) [16] is a specification language especially tailored for Java applications. Originally, JML was proposed by G.T. Leavens and his team; the development of JML is now a community effort. JML has been successfully used in several case studies to specify Java applications, and notably to specify smart card applications, written in Java Card [6, 13]. JML is developed following the Design by Contract approach [21], where classes are annotated with class invariants and method pre- and post-conditions. The predicates are written as (side-effect-free) boolean Java expressions, extended with specific constructs. Specifications are written as Java comments marked with a `@`, i.e., annotations follow `//@` or are enclosed between `/*@` and `@*/`. Below, we give a brief introduction to the main constructs of JML, by means of the example in Fig.1.

The class `Buffer` works as follows. A method `storeData()` personalizes the application by setting the length of the transaction. One can initialize a new transaction with the method `begin()`, creating a new temporary `buffer`. Then, a `write()` method fills the modifications in the temporary `buffer`, that is validated, i.e., assigned to the attribute `status`, by an invocation of `commit`. It is also possible to abort the transaction (`abort()`).

Figure 1 presents the main JML annotations on the simple example of a buffer. It shows a declaration of a class `invariant`, denoting a predicate that has to hold before and after every method call, i.e., in so-called JML *visible* states. History `constraint`s allow expressing a relation between the pre- and post-state of all methods. Pre-state values of expressions are denoted by the JML keyword `\old`. Using the clause `for`, one may specify the list of the methods for which the history constraint must be satisfied. When this clause is omitted, the constraint must hold for all the methods of the class. The clause `requires` denotes the precondition of the method, i.e., a predicate that must be true when the method is called. A postcondition is expressed with an `ensures` clause. A method may terminate exceptionally by throwing an exception and satisfying the exceptional postcondition (`signals` clause). The `assignable` clause gives the list of variables that can be potentially modified by the method. A method without side-effect is denoted by the keyword `pure`. The specification of a method can also contain a `diverges` clause (not displayed in this example). If the predicate of a `diverges` clause of a method m is satisfied by the pre-state of m, then the execution of m *may* not terminate. Otherwise the method *must* terminate. By default, the JML `diverges` clause is set to `false`. A method with a `helper` modifier *may* not preserve the invariant. JML also introduces its own variables – declared with the keyword `ghost`. A special `set` annotation exists to assign their value.

In the rest of the paper, given a method m, we denote by `requires`(m) (resp. `diverges`(m)), the predicate of the `requires` clause of m (resp. the `diverges` clause). The correctness of a Java class C w.r.t. a JML annotation $\mathcal{A}$, denoted $C : \mathcal{A}$ in the rest of the paper, can be established by model-checking [24] or by a prover (B or Coq) via a proof obligation generator (Jack[8] or Krakatoa [20]).

3. A JAVA EXECUTION SEMANTICS

Linear temporal properties [23] have a semantics over infinite executions of a program. So, to express such properties in terms of equivalent JML annotations, we need a path semantics of the JML. This semantics is presented in this section.

3.1 Class in Isolation Semantics

In this section, we define, in term of *execution paths* the semantics of a class C *in isolation*. A class C is the description of a set of objects I_C, called the instances of C.

Definition 1 (Java Class) *A* class C *is a tuple* (V_C, M_C) *where* V_C *is the set of attributes of the class,* M_C *is the set of methods of the class. Within the set* M_C *of methods, we consider the following particular subsets:*

- $Cons_C \subseteq M_C$ *the set of constructors of the class.*
- $Helper_C \subseteq M_C$ *the set of helper methods of the class.*
- $\mathcal{PM}_C$ *the set of progress (side-effect) methods. The set of non-progress (pure) methods is denoted by* $\overline{\mathcal{PM}_C}$. *Notice that* $M_C = \mathcal{PM}_C \cup \overline{\mathcal{PM}_C}$.

Notice that, as said before, we do not consider in this paper the problem of inheritance. Intuitively, an *execution path* of a Java statement is the sequence of memory states reached during the execution of the statement. The structure of the memory model is not in the scope of the paper, but has been formally specified in [26] or [20]. Intuitively, it is composed of a *heap*, mapping variables to memory addresses and a *store*, mapping addresses to their values. JML predicates are pre/post predicates, therefore they are evaluated on two memory states. Let $P \in \mathcal{P}red$ be a JML predicate and let s_{pre} and s_{cur} be two memory states, we denote by $(s_{pre}, s_{cur}) \models P$ the satisfaction of P at these memory states. The values of the variables with an `\old` are in s_{pre} and the others are in s_{cur}. If P does not contain variables relating to a preceding state, i.e., no `\old`, then we simply denote $s_{cur} \models P$. Given a state s and a variable `a`, s(`a`) returns the value of `a` in the state s. The Java memory contains also an *execution stack* [18]. As in Logozzo [19], we do not explicitly use the *execution stack*, *heap* and *store* but we assume to have *special variables* to observe it.

Definition 2 (Special Variables)
Let s *be a memory state, we assume to have the following special variables:*

```
public class Buffer {
  int len;
  byte [] status;
  byte [] buffer;
  int position = 0;
  boolean perso = false;
//@ ghost boolean trDepth = false;

//@ invariant position >= 0;

/*@ constraint position > \old(position)
  @ for write;
  @*/

/*@ normal_behavior
  @   requires perso == false;
  @   requires l > 0;
  @   assignable len,perso;
  @*/
void storeData(int l){
  len = l;
  perso = true;
}

 /*@pure@*/ byte [] getStatus(){
  return status;
}

 /*@pure@*/ int getBufferLess(){
  return len - buffer.length;
}

/*@ normal_behavior
  @   requires trDepth == false;
  @   requires perso == true;
  @   assignable buffer;
  @ also
  @ exceptional_behavior
  @   requires perso == false;
  @   assignable \nothing;
  @   signals (Exception e) true;
  @*/
void begin() throws Exception{
  if (perso == false) {
    throw new Exception();
  }
  buffer = new byte[len];
  //@ set trDepth = true;
}

/*@ normal_behavior
  @ requires trDepth == true;
  @ requires perso == true;
  @ assignable status,position;
  @*/
void commit(){
  status = buffer;
  position = 0;
  //@ set trDepth = false;
}

/*@ normal_behavior
  @ requires trDepth == true;
  @ requires perso == true;
  @ assignable position;
  @*/
void abort(){
  position = 0;
  //@ set trDepth = false;
}

/*@ normal_behavior
  @ requires trDepth == true;
  @ requires perso == true;
  @ requires position < len;
  @ assignable position;
  @ assignable buffer[position-1];
  @ ensures position <= len;
  @ ensures position == \old(position)+1;
  @*/
void write(byte b){
  buffer[position] = b;
  position++;
}

}
```

Figure 1: Class `Buffer`

- $s(\mathsf{excp})$, *a flag denoting that an exception has been thrown.*
- $s(\mathsf{stackHeight})$, *the height of the execution stack.*
- $s(\mathsf{curMethod})$, *the current method (the method on the top of the stack).*
- $s(\mathsf{curInstance})$, *the pointer on the current object.*

Intuitively, we define a path of a Java statement T as a sequence of states that are reached during the execution of T. For that, we assume a transition relation $\rightarrow$ associated to each Java statement T. Readers can find an example of such a relation in [12] for a sequential Java core.

Definition 3 (Java Statement Path) *Let s_0 be a Java state and let T be a Java statement, the* path *of T, denoted $s_0[T]$ is either:*

- *if T terminates, the* ***finite sequence*** *σ of states $s_0, s_1, s_2, \ldots, s_n$ such that $\forall i.(0 \leq i < n \Rightarrow (s_i \rightarrow s_{i+1}))$; or*
- *if T diverges, the* ***infinite sequence*** *σ of states $s_0, s_1, s_2, \ldots$ such that $\forall i \geq 0.(s_i \rightarrow s_{i+1})$.*

In the rest of the paper, we denote by s_i the i^{th} state of the execution path σ. We denote by ϵ the empty path and given an execution path σ the *path suffix* σ_i denotes the infinite path $s_i, s_{i+1}, s_{i+2} \ldots$ and the *path segment* σ_i^j denotes the finite path $s_i, s_{i+1}, s_{i+2} \ldots s_j$. A predicate P without `\old` holds on σ_i and σ_i^j if $s_i \models P$ as in LTL logic [23]. Given a finite path $\sigma = s_0, \ldots, s_n$ and another path $\sigma' = s_a, s_b, \ldots$, the sequential composition of σ and σ', denoted σ, σ' is equal to the path $s_0, \ldots, s_n, s_a, s_b, \ldots$.

As explained in Section 1, we aim at verifying that a property of a class C is satisfied by any instances of C and by any programs using the class C.

Therefore, we introduce a class in isolation semantics, handling all potential executions of all instances of C.

As explained in Sect. 2, JML considers only *visible* states, i.e., states before the invocation of a non-helper method or after the termination of a non-helper method. Therefore, we define the notions of pre- and post-states for a method m as follows.

Definition 4 ***(Pre-, Post-, Matching Post- and Inner-state of a Method)*** *Let σ be a path, let s_i be its i^{th} state (with $i > 0$) and let m be a method. We say that s_i is a* pre-state *of m, denoted $s_i \models \mathsf{pre}(m)$ if:*

$$s_i(\mathsf{curMethod}) = m \quad \textit{and}$$
$$s_i(\mathsf{stackHeight}) = s_{i-1}(\mathsf{stackHeight}) + 1$$

s_i is a post-state *of m, denoted $s_i \models \mathsf{post}(m)$ if:*

$$s_i(\mathsf{curMethod}) = m \quad \textit{and}$$
$$s_i(\mathsf{stackHeight}) - 1 = s_{i+1}(\mathsf{stackHeight})$$

Given a pre-state s_i of m, a state s_j, where $j > i$ is its matching post-state *s_j, denoted $s_j \models \mathsf{post}_{s_i}(m)$, if:*

$$s_j \models \mathsf{post}(m) \ \wedge \ s_j(\mathsf{stackHeight}) = s_i(\mathsf{stackHeight})$$
$$\wedge \forall k.(i < k < j \Rightarrow (s_k(\mathsf{stackHeight}) \geq s_i(\mathsf{stackHeight}))).$$

Let i be the index of the pre-state of m, let j be the index of its matching post-state, a state s_k is an visible inner-*state of m, denoted $s_k \models \mathsf{inner}_{s_i}(m)$ if $i \geq k \geq j$. If m does not terminate, s_k is an* inner-*state if $i \geq k$.*

Then, we define the notion of *visible states* following [16].

Definition 5 (Visible States)
Given a Java execution path σ, the state s_i is a visible state *for an instance i_C of the class C, denoted $s_i \models \mathsf{visible}(i_C)$ either if*

- *s_i is the post-state of a* constructor *of i_C,*

$$s_i \models \mathsf{post}(m) \ \wedge \ m \in Cons_C \ \wedge$$
$$s_i(\mathsf{curInstance}) = i_C, \textit{or}$$

- s_i *is the pre-state of a* non-helper *method invoked on* i_C*,*

$$s_i \models \mathit{pre}(m) \ \wedge \ m \in M_C \ \wedge \ m \notin Cons_C \ \wedge \ m \notin Helper_C \ \wedge \ s_i(\mathsf{curInstance}) = i_C, \text{ or}$$

- s_i *is the post-state of a* non-helper *method invoked on* i_C*.*

$$s_i \models \mathit{post}(m) \ \wedge \ m \in M_C \ \wedge \ m \notin Cons_C \ \wedge \ m \notin Helper_C \ \wedge \ s_i(\mathsf{curInstance}) = i_C.$$

Therefore, for easily reasoning about JML, we define the notion of *visible state execution path* as follows:

Definition 6 (Visible State Abstraction and Visible State Execution Path) *Let* σ *be an execution, we define the* visible state abstraction *for an instance* i_C*, denoted* $\mathsf{vsa}_{\mathsf{i}_C}(\sigma)$*, by:*

- $\mathsf{vsa}_{\mathsf{i}_C}(\epsilon) = \epsilon$
- *if* $s_0 \models \mathsf{visible}(\mathsf{i}_C)$ *then* $\mathsf{vsa}_{\mathsf{i}_C}(\sigma) = s_0, \mathsf{vsa}_{\mathsf{i}_C}(\sigma_1)$ *else* $\mathsf{vsa}_{\mathsf{i}_C}(\sigma) = \mathsf{vsa}_{\mathsf{i}_C}(\sigma_1)$.

Given a Java statement S*, we define the visible state execution path of* S *on a state* s_0*, denoted* $s_0[S]_{\mathsf{i}_C}$*, as follows:*

$$s_0[S]_{\mathsf{i}_C} =_{def} \mathsf{vsa}_{\mathsf{i}_C}(s_0[S])$$

Notice that the visible state abstraction hides:

- The details of the C method's body execution.
- The invocations of helper methods.
- The invocations of methods of other classes (both methods of other classes invoked by C and by the environment of C).

Let Σ be a set of paths, the visible state abstraction of Σ w.r.t. an instance i_C, denoted $\mathsf{vsa}_{\mathsf{i}_C}(\Sigma)$ is defined as the set of all the abstractions of the paths of Σ, i.e.,

$$\mathsf{vsa}_{\mathsf{i}_C}(\Sigma) = \{\mathsf{vsa}_{\mathsf{i}_C}(\sigma) | \sigma \in \Sigma\}.$$

Then, we define the semantics of an instance i_C in isolation, denoted Σ_{i_C}. The semantics of i_C captures all the potential executions of i_C. So, it is intuitively the set of all paths Σ_{i_C} starting at invocation of a constructor creating i_C, followed by an arbitrary number of invocations on i_C of the methods of C within their preconditions.

Definition 7 (Instance Semantics) *Let* i_C *be an instance of* $C = (V_C, M_C)$*, we denote* Σ_{i_C} *the set of executions iteratively defined as follows:*

- $\epsilon \in \Sigma_{\mathsf{i}_C}$.
- *Let* s_0 *be a state, let* $m \in Cons_C$*. If* $s_0 \models \texttt{requires}(m)$ *then*

$$s_0[(m, \mathsf{i}_C)]_{\mathsf{i}_C} \in \Sigma_{\mathsf{i}_C}.$$

- *Let* $\sigma \in \Sigma_{\mathsf{i}_C}$ *be a finite execution and* s_n *be its last state. Let* s_{n+1} *be a state such that* $\forall v \in V_C.(s_n(v) = s_{n+1}(v))$*. Let* m *such that* $m \in M_C \wedge m \notin Cons_C \wedge m \notin Helper_C$*. If* $s_{n+1} \models \texttt{requires}(m)$ *then:*

$$(\sigma, (s_{n+1}[(m, \mathsf{i}_C)]_{\mathsf{i}_C})) \in \Sigma_{\mathsf{i}_C}.$$

The class semantics of a class C is defined as the set of all executions of its instances.

Definition 8 (Class semantics) *Given a class* C*, let* I_C *be the set of instances of* C*. We define* Σ_C*, the semantics of the class* C*, by:*

$$\Sigma_C =_{def} \bigcup_{i_C \in I_C} \Sigma_{i_C}$$

3.2 JML Semantics

To express temporal properties by JML annotations, we need an execution semantics of JML annotations. To our knowledge, JML semantics has been given in terms of wp-calculus (see for example [20]), but never in terms of properties of the execution paths. We propose in this section a semantics for the `invariant` clauses, `constraint` clauses and a `behavior` specification.

Definition 9 (Path Execution Semantics of JML annotations) *Given a set of executions* Σ_C *of a class in isolation, the* path execution semantics of JML annotations *is displayed in Fig. 2.*

The semantics is given w.r.t. the definition in [16]. It must be understood as follows[2].

- `Invariant`: The invariant must be satisfied by each visible state.
- `Constraint`: For the body of each method included in the `for` clause, the constraint must hold between the pre-state and the post-state, but also between all visible states that arise during the execution of the method, i.e., all *inner* states of the method.
- `Behavior` method specification: each specification of a method can be desugared as a `behavior` specification [16]. This JML specification is interpreted on a path as follows. i If the predicate P of the `requires` clause is satisfied by the pre-state of the method m, that implies:
  - If the predicate D of the `diverges` is satisfied on the pre-state, then if the method terminates, i.e., the method has a post-state, the predicate Q of the `ensures` clause must be satisfied between the pre-state and the post-state if it is a normal termination ($s_j(\mathsf{excp}) = false$). Otherwise, i.e., if it is an exceptional termination, the predicate R must be satisfied.
  - If the predicate D of the `diverges` is not satisfied on the pre-state, then the method *must* terminate, i.e., the method *must* have a post-state. Moreover, if it is a normal termination the predicate Q must be satisfied, and the predicate R must be satisfied otherwise.

  Notice that in each case, only attributes within the list A of the `assignable` clause can be modified ($\forall a.(a \in V_S \wedge a \notin A \Rightarrow (s_i(a) = s_j(a)))$).

[2]The definitions of `constraint` and `assignable` are proposed accordingly to the semi-formal description in [16]. Notice that, for technical reasons, an alternative semantics of these clauses has been implemented in some tools

```
//@ invariant I;
```
$\equiv_{def}$ $\forall \sigma \in \Sigma_C \ . \ \forall i \geq 0 \ . \ \sigma_i \models I$

```
//@ constraint H for M;
```
$\equiv_{def}$ $\forall \sigma \in \Sigma_C \ . \ \forall i \geq 0 \ . \ \forall m \in M.$
$(s_i \models \mathsf{pre}(m) \Rightarrow$
$(\forall k_1, k2.(i \leq k_1 < k_2$
$\wedge s_{k_1} \models \mathsf{inner}_{s_i}(m) \wedge s_{k_2} \models \mathsf{inner}_{s_i}(m) - \Rightarrow$
$(s_{k_1}, s_{k_2}) \models H)))$

```
/*@ behavior;
  @ requires P;
  @ diverges D;
  @ assignable A;
  @ ensures Q;
  @ signals
  @ (Exception e) R;
  @*/
m()
```
$\equiv_{def}$ $\forall \sigma \in \Sigma_C \ . \ \forall i \geq 0 \ .($
$((\sigma_i \models (P \wedge \neg D) \wedge \sigma_i \models \mathsf{pre}(m)) \Rightarrow$
$\exists j > i.($
$(s_j \models \mathsf{post}_{s_i}(m) \wedge s_j(\mathsf{excp}) = false$
$\wedge (s_i, s_j) \models Q$
$\wedge \forall a.(a \in V_S \wedge a \notin A \Rightarrow (s_i(a) = s_j(a))))$
$\vee$
$(s_j \models \mathsf{post}_{s_i}(m) \wedge s_j(\mathsf{excp}) = true$
$\wedge (s_i, s_j) \models R$
$\wedge \forall a.(a \in V_S \wedge a \notin A \Rightarrow (s_i(a) = s_j(a)))))$
$\wedge$
$((\sigma_i \models (P \wedge D) \wedge \sigma_i \models \mathsf{pre}(m)) \Rightarrow$
$\forall j > i.($
$(s_j \models \mathsf{post}_{s_i}(m) \wedge s_j(\mathsf{excp}) = false \Rightarrow$
$(s_i, s_j) \models Q$
$\wedge \forall a.(a \in V_S \wedge a \notin A \Rightarrow (s_i(a) = s_j(a))))$
$\wedge$
$(s_j \models \mathsf{post}_{s_i}(m) \wedge s_j(\mathsf{excp}) = true \Rightarrow$
$(s_i, s_j) \models R.$
$\wedge \forall a.(a \in V_S \wedge a \notin A \Rightarrow (s_i(a) = s_j(a))))))))$

Figure 2: Path execution semantics of JML annotations

4. LIVENESS PROPERTIES VERIFICATION

This section deals with the verification of liveness properties on the execution semantics Σ_C of a class C. For that, we presents in Section 4.1 a *liveness primitive operator* Loop. Under a *progress hypothesis* on the environment presented in Section 4.2, the satisfaction of the Loop operator can be ensured by appropriates JML annotations. This result is established in a theorem given in Section 4.3.

4.1 The Loop Primitive

In this section, Q denotes a JML predicate, M denotes a subset of $\mathcal{PM}_C$, and V denotes a JML expression returning an integer. The $\mathsf{Loop}(Q, V, M)$ primitive is satisfied by an execution if, after any states of the execution satisfying Q, a state where $\neg Q$ holds must *eventually* be reached. Besides the predicate Q marking the loop entry condition we also require, to prove the termination of the loop, a variant V and a set of methods $M \subseteq \mathcal{PM}_C$.

Definition 10 (Loop Primitive) $\mathsf{Loop}(Q, V, M)$[3] *holds on an execution σ, written $\sigma \models \mathsf{Loop}(Q, V, M)$, if*

$$\forall i.((i \geq 0 \wedge \sigma_i \models Q) \ \Rightarrow \ (\exists j.j > i \wedge \sigma_j \models \neg Q)).$$

If σ is a finite execution of length n, $\sigma \models \mathsf{Loop}(Q, V, M)$ if

$$\forall i.((0 \leq i \leq n \wedge \sigma_i \models Q) \ \Rightarrow \ (\exists j.i < j \leq n \wedge \sigma_j \models \neg Q)).$$

Notice that the variant V and the set M of methods do not appear in the above expression since they are only used to generate the appropriate proof obligations for the termination of the loop. For the verification of the Loop operator, finite executions are viewed as infinite executions by infinitely repeating the last state of the execution. The infinite extension of a finite execution is the following.

Definition 11 (Infinite Extension of Finite Execution) *Let σ be a finite execution $s_0, s_1, s_2 \ldots, s_n$. We extend it to the infinite sequence σ' such that σ' is $s_0, s_1, s_2 \ldots, s_n, s_n, \ldots$.*

The infinite extensions of finite executions are suitable for verifying the Loop primitive.

Lemma 1 *Let σ be a finite execution, σ' be the infinite extension of σ. We have*

$$\sigma' \models \mathsf{Loop}(Q, V, M) \quad \Leftrightarrow \quad \sigma \models \mathsf{Loop}(Q, V, M)$$

Notice that Σ_C contains all *potential* executions of instances of C. We address the verification of a particular subset of Σ_C that satisfies a *progress hypothesis*.

4.2 Progress Hypothesis PH

Using the semantics of LTL [23], Hypothesis $PH(Q, M)$ is expressed by the LTL operators G^∞ ("almost everywhere") and F^∞ (*"infinitely often"*).

Intuitively, $\mathsf{G}^\infty P$ means that after a finite number of states, the property P holds forever. The semantics of G^∞ is the following

$$\sigma_i \models G^\infty P \equiv_{def} \exists j \geq i.(\forall k.(k \geq j \Rightarrow \sigma_k \models P)).$$

Given a predicate P, the formula $F^\infty P$ means that at any state of the execution, there always exists a future state verifying P. Formally,

$$\sigma_i \models F^\infty P \equiv_{def} \forall j \geq i.(\exists k.(k \geq j \wedge \sigma_k \models P)).$$

In order to verify $\Sigma_C \models \mathsf{Loop}(Q, V, M)$, we need to assume progress of the environment, i.e., the environment invokes

[3]Notice that Loop(Q,V,M) semantics corresponds to LTL property $\mathsf{GF}\neg Q$.

the methods of the subset M of the *progress* methods. Two behaviors of the environment are allowed:

- The environment calls methods in M infinitely often.
- The environment performs a finite number of invocations of methods in M until a state i such that any state of σ_i satisfies $\neg Q$.

Therefore, we define the progress hypothesis $PH(Q, M)$ as follows.

Definition 12 (Progress Hypothesis $PH(Q, M)$)

$$(\mathsf{G}^{\infty}\neg Q) \vee (\mathsf{F}^{\infty}\mathsf{pre}(M)) \qquad \text{(PH(Q,M))}$$

where $\mathsf{pre}(M)$ *denotes the predicate* $\bigvee_{m\in M} \mathsf{pre}(m)$.

We denote $\Sigma_{C/PH(Q,M)}$ the subset of executions of Σ_C satisfying $PH(Q, M)$.

Definition 13 (Class under $PH(Q, M)$) $\Sigma_{C/PH(Q,M)}$ *is the set of execution defined as follows.*

$$\Sigma_{C/PH(Q,M)} = \{\sigma | \sigma \in \Sigma_C \wedge \sigma \models PH(Q, M)\}.$$

In the next section we show how to use appropriate JML annotations for establishing that $\Sigma_{C/PH(Q,M)} \models \mathsf{Loop}(Q, V, M)$.

4.3 Annotations for the Loop operator

Verification of the Loop primitive is quite similar to a termination proof, since we have to show that as long as Q it must always be possible to invoke a method of M and methods in M must decrease a well founded variant V. Here we propose proof obligations – inspired from [9] – expressed as JML annotations. These proof obligations guarantee the satisfaction of the Loop primitive by an execution satisfying the hypothesis $PH(Q, M)$.

Let $\mathsf{Loop}(Q, V, M)$ be the Loop primitive. Let $\mathcal{A}_{1-5}$ be the following set of JML annotations.

`//@ invariant` V `>= 0;` $(\mathcal{A}_1)$

`//@ constraint` Q `==>` V `< \old(`V`) for` M`;` $(\mathcal{A}_2)$

`//@ constraint` Q `==>` V `<= \old(`V`) ;` $(\mathcal{A}_3)$

`//@ invariant` Q `==>` $\bigvee_{m\in M} \mathsf{requires}(m)$ $(\mathcal{A}_4)$

`//@ invariant` Q `==>` $\bigwedge_{m\in M_C} (\mathsf{requires}(m)$ `==> !`$\mathsf{diverges}(m))$`;` $(\mathcal{A}_5)$

Intuitively, $\mathcal{A}_{1-5}$ could be understood as follows.

$\mathcal{A}_1$ The variant V actually is greater than zero, i.e., it is an expression over a well-founded set.

$\mathcal{A}_2$ As long as Q holds, when a method in M is executed, the variant V must decrease. It ensures the progress when the environment satisfies $PH(Q, M)$ (livelock-freeness).

$\mathcal{A}_3$ As long as Q holds, when a method of C is executed, the variant V must not increase.

$\mathcal{A}_4$ As long as Q holds there always should be a method in M that might be called, i.e., its precondition holds. This ensures the deadlock-freeness of the system.

$\mathcal{A}_5$ As long as Q holds, all callable methods must not diverge. This ensures the non-divergence of the system.

Hypothesis $PH(Q, M)$ is the disjunction of $(\mathsf{F}^{\infty}\mathsf{pre}(M)$ and $\mathsf{G}^{\infty}(\neg Q)$, therefore, for each of these hypothesis, we show respectively in Lemma 2 and Lemma 3 that, assuming that the code of the class is correct w.r.t. the annotations $\mathcal{A}_{1-5}$ $(C : \mathcal{A}_{1-5})$, the satisfaction of Loop (Q, V, M) is established on Σ_C.

Lemma 2 *If* $C : \mathcal{A}_{1-5}$ and $\sigma \in \Sigma_C$ and $\sigma \models (\mathsf{F}^{\infty}\mathsf{pre}(M))$ *then* $\sigma \models \mathit{Loop}(Q, V, M)$.

Lemma 3 *If* $C : \mathcal{A}_{1-5}$ and $\sigma \in \Sigma_C$ and $\sigma \models \mathsf{G}^{\infty}(\neg Q)$ *then* $\sigma \models \mathit{Loop}(Q, V, M)$.

A consequence of Lemma 2 and Lemma 3 is the following theorem.

Theorem 1

If $C : \mathcal{A}_{1-5}$ then $\Sigma_{C/PH(Q,M)} \models \mathit{Loop}(Q, V, M)$.

An interesting property is obtained when $M = \mathcal{PM}_C$. In this particular case, Hypothesis $PH(Q, M)$ is not only sufficient, but also necessary.

Proposition 1 *When* $M = \mathcal{PM}_C$, *given* $\sigma \in \Sigma_C$, $\sigma \models \mathit{Loop}(Q, V, M)$ *and* $C : \mathcal{A}_{1-5}$ *imply that* $\sigma \models PH(Q, M)$.

We now show how liveness properties (expressed here in JTPL) can be embedded into a Loop primitive.

5. JML ANNOTATION GENERATOR TOOL

The generation of annotations for safety properties in [25] and of liveness properties presented in Sect. 4 is implemented in a tool, called JAG (for JML Annotation Generator) [10].

The JAG tool takes as an input a formula expressed in JML Temporal Pattern Logic (JTPL), first introduced in [25]. A JTPL formula is a combination of JML predicates, *events* and temporal operators. Using JTPL formulae, one can express, on the example of the `Buffer` (see Fig. 1 Sect. 2), the following properties:

1. After the invocation of `storeData` (**after** `storeData` **called**), the variable `perso` is **always** `true`, expressed in JTPL as follows.

   after `storeData` **called always** `perso;` (S)

2. After starting a transaction, i.e., the normal termination of the method `begin` (**after** `begin` **normal**), a state where `trDepth` is `false` must eventually be reached.

   after `begin` **normal eventually** `!trDepth`
   under variant `getBufferLess()`
   for `begin, commit, abort, write .` (L)

Notice that in Property L, the event is `begin` **normal** and not `begin` **called** since a buffer transaction starts only when the method `begin` terminates normally. Notice also that since Property L is a liveness property, the user gives a variant and a set of progress methods with the JTPL clause `under_variant ... for`.

The result of the translation of Properties S and L is displayed in Fig. 3.

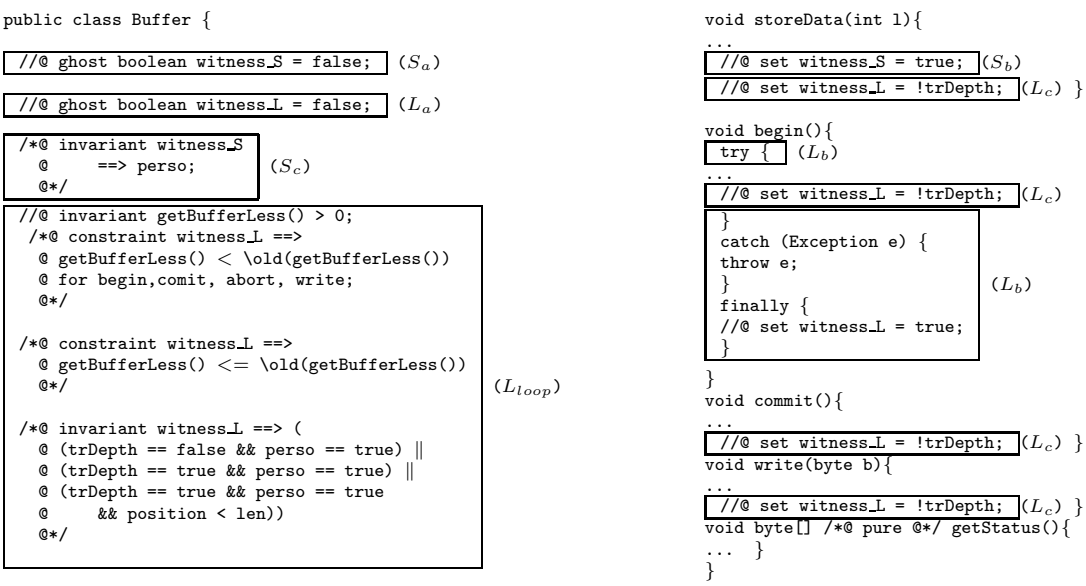


Figure 3: Buffer with generated annotations

1. First, JAG generates a *ghost* boolean variable for observing the occurrences of the events of the temporal properties. These ghost variables are assigned w.r.t. the events occurring in the formula.

   Example 1 (Ghost Variables Generation for S) *The* `ghost` *variable* `witness_S`*, corresponds to the event* `storeData` ***called*** *of S. It is initially declared with the value* `false` *(see Annotation S_a in Fig. 3) and it is set to* `true` *when the method* `storeData` *is called (see annotation S_b). So, in each state after the event* `storeData` ***called****, the value of the ghost variable* `witness_S` *is* `true`*, i.e.,* `witness_S` *is true exactly with the scope of the property.*

   Example 2 (Ghost Variables Generation for L) *The* `ghost` *variable* `witness_L`*, corresponding to the event* `begin` ***normal*** *of the temporal property L is also declared with the value false (Annotation L_a in Fig. 3). The ghost variable* `witness_L` *is assigned using a* `try...catch...finally` *statement (see annotation L_b). Notice that, in case of exception, the caught exception is re-thrown, the execution does not go into the finally block, the reader can see that* `witness_L` *is set to* `true` *only when* `begin` ***normal*** *occurs. The* `ghost` *variable* `witness_L` *is set to* `false` *again by adding to each method a set statement (annotation L_c).*

2. Second, it generates an invariant to ensure the satisfactions of a safety property.

   Example 3 (Invariant Generation for S) *The invariant for S is displayed in Fig. 3 (annotation S_c). It means that when the variable* `witness_S` *is true, i.e., after the first occurrence of* `storeData` ***called****, the predicate* `(perso == true)` *must be* `true` *- the definition of Property S.*

3. Finally JAG translates each liveness property into a `Loop` primitive and generates the corresponding JML annotations.

   Example 4 (Generation of annotations for L) *The JML primitive corresponding to L is*

   `Loop(witness_L, getBufferLess(), {begin, commit, abort, write})`

   The corresponding annotations are displayed in Fig. 3 (see Annotations L_{loop}).

   Notice that, since no method of `Buffer` diverges, Annotation $\mathcal{A}_5$ does not appear.

The tool is able to keep the trace of the generated annotations, i.e., it is possible, given a generated annotation, to find the original intermediate primitive and the original temporal property. Since the generated output file contains standard JML annotations, it can be used with other JML tools [7] to validate or prove the temporal formulae. In particular, we have successfully used it for the following purposes.

- **Verification of the correctness of the Java code w.r.t. the JML annotations** with the proof obligation generators Jack [8] and Krakatoa [20].
- **Validation of a JML model** with JML-TT [5];
- **Formal verification of a JML model** with the JML2B method [2];
- **Test generation and Runtime Assertion Checking** with the test generators Tobias [17], Jartege [22] and JML-TT [4].

Test generation and Runtime Assertion Checking using JAG has been studied on a industrial Javacard application [3].

6. CONCLUSION AND FUTURE WORKS

This paper presents a way to verify liveness properties on Java classes in isolation by generating appropriate JML annotations. This requires that the user specifies a variant for the verification of a Loop primitive to which liveness properties are reduced. The generated JML annotations are verified (or validated) with any tools handling JML. The JAG tool implements this translation. It has been used for several toy examples and a Java Card Electronic Purse Specification (over 500 lines of JML).

To the best of our knowledge, this is the first attempt to verify liveness properties for potentially infinite-state systems using a translation into JML. For finite state systems, liveness properties expressed in LTL are usually verified automatically by model checkers such as SPIN [11]. For infinite state systems, model checking is used on liveness preserving abstractions.

Currently we are working on extensions of JAG to other temporal properties. In particular, we currently address the verification of properties expressed by Büchi automata. Then, the Büchi acceptance condition is checked using Loop primitives introduced in this paper. The second challenge is, assuming that a liveness is established on the class in isolation, to provide techniques for verifying that the (single- or multi-threaded) environment effectively satisfies $PH(Q, M)$.

Acknowledgment: We like to thank Marieke Huisman for her interesting and helpful comments and suggestions to improve this work. Further, we acknowledge all the anonymous referees for their corrections, comments and advices.

7. REFERENCES

[1] J. Andronick, B. Chetali, and O. Ly. Using Coq to verify Java Card Applet Isolation Properties. In *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'2003)*, 2003.

[2] F. Bouquet, F. Dadeau, and J. Groslambert. Checking JML specifications with B machines. In *ZB'05*, volume 3455 of *LNCS*, pages 435–454. Springer-Verlag, 2005.

[3] F. Bouquet, F. Dadeau, J. Groslambert, and J. Julliand. Safety property driven test generation from JML specifications. In *FATES/RV'06*, LNCS, pages 225–239. Springer-Verlag, 2006. To appear.

[4] F. Bouquet, F. Dadeau, and B. Legeard. Automated Boundary Test Generation from JML Specifications. In *FM'06*, volume 4085 of *LNCS*, pages 428–443. Springer-Verlag, 2006.

[5] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. JML-Testing-Tools: a symbolic animator for JML specifications using CLP. In *TACAS'05 Tool session*, volume 3440 of *LNCS*, pages 551–556. Springer, 2005.

[6] C-B. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. Formal methods for smart cards: an experience report. *Sci. Comput. Program.*, 55(1-3):53–80, 2005.

[7] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An Overview of JML Tools and Applications. In *FMICS 03*, volume 80 of *ENTCS*, pages 73–89. Elsevier, 2003.

[8] L. Burdy, A. Requet, and J.-L. Lanet. Java Applet Correctness: a Developer-Oriented Approach. In *FM'03*, number 2805 in LNCS, pages 422–439. Springer, 2003.

[9] R.M. Burstall. Program Proving as Hand Simulation with a Little Induction. *Information Processing*, pages 308–312, 1974.

[10] A. Giorgetti and J. Groslambert. JAG: JML Annotation Generation for Verifying Temporal Properties. In *FASE*, LNCS, pages 373–376. Springer, 2006.

[11] G.J. Holzmann. The Model Checker SPIN. In *IEEE Trans. on Software Engineering*, volume 23-5, pages 279–295, 1997.

[12] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA*, volume 34(10), pages 132–146. ACM, 1999.

[13] B. Jacobs, C. Marché, and N. Rauch. Formal Verification of a Commercial Smart Card Applet with Multiple Tools. In *AMAST'04*, number 3116 in LNCS, pages 21–22. Springer, 2004.

[14] O. Kouchnarenko, J. Groslambert, and J. Julliand. JML-based Verification of Liveness Properties on a Class. Technical Report RR2006-7, LIFC, 2006.

[15] L. Lamport. Proving the Correctness of Multiprocess Programs. In *IEEE Transactions on Software Engineering*, volume 3(2), pages 125–143, 1977.

[16] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D.R. Cok, and J. Kiniry. JML Reference Manual. Department of Comp. Science, Iowa State University. Available from http://www.jmlspecs.org, 2003.

[17] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron. Filtering TOBIAS Combinatorial Test Suites. In *FASE 2004*, volume 2984 of *LNCS*, pages 281–294. Springer-Verlag, 2004.

[18] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1997.

[19] F. Logozzo. Class Invariants as Abstract Interpretation of Trace Semantics. *Computer Languages, Systems and Structures*, 2005.

[20] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/Java Card programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.

[21] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd rev. edition, 1997.

[22] C. Oriat. Jartege: A Tool for Random Generation of Unit Tests for Java Classes. In *SOQUA 2005*, volume 3712 of *LNCS*, pages 242–256. Springer-Verlag, 2005.

[23] A. Pnueli. The Temporal Logic of Program. In *18th Ann. IEEE Symp. on foundations of computer science*, pages 46–57, 1977.

[24] Robby, E. Rodríguez, M. Dwyer, and J. Hatcliff. Checking Strong Specifications Using an Extensible Software Model Checking Framework. In *TACAS 2004*, volume 2988, pages 404–420. Springer, 2004.

[25] K. Trentelman and M. Huisman. Extending JML Specifications with Temporal Logic. In *AMAST'02*, number 2422 in LNCS, pages 334–348. Springer, 2002.

[26] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A Type-Theoretic Memory Model for Verification of Sequential Java Programs. In *WADT*, volume 1827 of *LNCS*, pages 1–21. Springer, 1999.

Using Resemblance to Support Component Reuse and Evolution

Andrew McVeigh, Jeff Kramer and Jeff Magee
Department of Computing
Imperial College
London SW7 2BZ, United Kingdom
{amcveigh, jk, jnm}@doc.ic.ac.uk

ABSTRACT

The aim of a component-based approach to software is to allow the construction of a system by reusing and connecting together a number of existing components. To successfully reuse a component, alterations generally need to be made to it, particularly if the abstraction level is high. However, existing usage of a component means that it cannot be altered without affecting the systems that reuse it already. This leads to a dilemma which frustrates the goals of the compositional approach to reuse.

To help resolve this dilemma, we introduce the resemblance construct, allowing a new component to be defined in terms of changes to a base component. This allows us to effectively alter a base component for reuse, without affecting the existing definition or any users of the component. We use an example to show how this and other constructs ameliorate the reuse problems of complex, possibly composite, components.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Languages*; D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*; D.2.10 [**Software Engineering**]: Design—*Representation*

General Terms

Design, languages

Keywords

Architecture, components, composition, reuse, modelling

1. INTRODUCTION

When taking a compositional approach to system construction, a composite component can be created by composing and connecting together a number of other components. Each of the constituent components of the composite may either be composite themselves or leaf components which have no further decomposition [20, 9]. Complex subsystems, and even entire systems can be represented as composites which can then be reused as parts of other systems. The aim is to assemble systems from increasingly higher-level components, offering a compelling approach to construction and reuse. In practice, however, a number of issues frustrate this goal.

To set the context, consider that a system is constructed from both existing components, and new components developed specifically for that architecture. Existing components are obtained from a component provider, or taken from an existing system. It is unlikely that changes can be made to an existing component specifically to accommodate a new system, as existing usage in other environments places constraints on what can be changed. To be successfully reused, however, existing components generally require alterations before they can be integrated into a new architecture [6].

This situation leads to a dilemma: components cannot be reused without changes, but existing usage heavily constrains any changes. The more complex or higher-level a component is, the less is the likelihood that it will be suitable for reuse in an unaltered form. This situation is closely related to the abstraction problem [5]: components are more valuable when they represent higher-level abstractions targeted at a particular domain, but this specificity limits their reuse. This is particularly a problem with composite components as they hide their constituent components and abstractions.

In order to examine this dilemma more closely, a reuse scenario from an existing system is presented. By analysing this situation, we form a set of requirements that a solution must meet in order to address the identified issues.

From these requirements we develop the concept of *resemblance*, which is an inheritance-like construct for components. This allows us define a new component in terms of changes to a, possibly composite, base component. The key is that the changes are held in the new component, and do not affect the base definition. Combined with a small number of other constructs, we demonstrate how this ameliorates the reuse problems. We further show how the constructs can also help with component evolution, acting as a type of decentralised configuration management (CM) system.

The rest of the paper is organised as follows. We begin by presenting the component model as general background for the discussion and to establish terminology. A simplified example of a component reuse problem from a working system is shown, leading to a conceptual view of the problem. We then introduce the resemblance and other constructs and show how a component can

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006), November 10-11, 2006, Portland, Oregon, USA.

be altered for new requirements, without losing the link back to the original definition. We conclude with a discussion of related work which contrasts this work with architecturally-aware CM systems, and product family approaches.

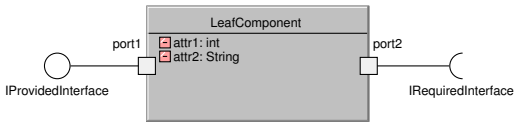
2. THE COMPONENT MODEL

In keeping with Darwin [10] and UML2 [12], we define a component as an instantiable, class-like construct which explicitly describes the interfaces that it provides and requires. An interface represents a collection of methods defining a service and may inherit from other interfaces. Interfaces can only be provided or required via ports, and each port has a name and may be indexed. Ports serve to name the role of interfaces as services offered or required by a component.

A component may have attributes, which can only be of primitive type. These present a view, or projection, on the internal state of the component.

Components are either leaf or composite, where a leaf component cannot be further decomposed and is associated directly with an implementation in (currently) Java.

Figure 1 shows a leaf component with two attributes and two ports. The graphical representation is a UML2 composite structure diagram where a provided interface is shown as a circle, and a required interface is shown as a semi-circle. Note that the leaf is directly associated with a Java implementation class.



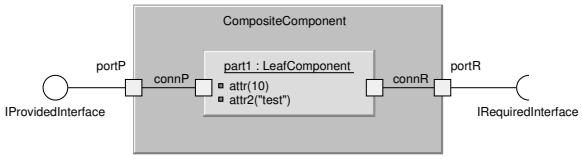

```
component LeafComponent
   describes com.example.JavaLeafComponent
{
   attributes:
      int attr1; String attr2;
   ports:
      port1 provides IProvidedInterface;
      port2 requires IRequiredInterface;
}
```

Figure 1: Definition of a leaf component

The textual definition in the lower half of the figure is from the Backbone architecture description language (ADL). This experimental language has been defined, as part of this work, in order to demonstrate the concepts in this paper, and to also explore the use of UML2 as an ADL. We have developed a prototype Backbone interpreter which can assemble a system from the ADL representation and the Java implementation of the leaf components.

Although Backbone has been designed around the UML2 component meta-model, it bears more than a passing resemblance to Darwin. This presumably reflects the influence that Darwin, ROOM [15], ACME [3] and other ADLs have had on the UML2 specification.

A composite component (figure 2) can additionally contain a number of component instances, each of which is shown as a box within the component. These instances are called *parts* in UML2 terminology. Each part has a name (part1), and a component type (LeafComponent) which is the component it is an instance of. Further, a part can define slots, which hold values for the attributes of the component type e.g. attr(10). The parts of a composite represent its initial configuration and state.




```
component CompositeComponent
{
   ports:
      portP provides IProvidedInterface;
      portR requires IRequiredInterface;
   parts:
      LeafComponent part1
         set attr1(10), attr2("test");
   connectors:
      connP joins portP to port1@part1;
      connR joins portR to port2@part1;
}
```

Figure 2: Definition of a composite component

Ports are wired together using connectors (connP, connR). In UML2, connectors represent little more than an aliasing of two different ports [4].

3. MOTIVATING EXAMPLE

This example is based on a reuse problem experienced when extending our graphical modelling tool. This tool is being developed as part of this work to provide an environment to support the concepts outlined in this paper.

As we work through the example, we use it to distill four requirements that a solution to the component reuse problems must address.

3.1 Context

Company X is a component provider that produces components for constructing graphical drawing tools. The major component is a composite called CDrawing, which represents a drawing framework.

Also available are a set of components which can draw various complex shapes when used with the framework. One such component is CPostItNote, which displays a small note surrounded by a border as shown in figure 3.

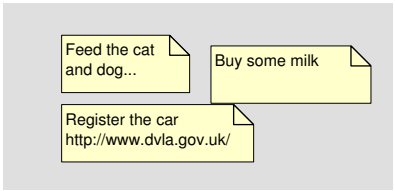
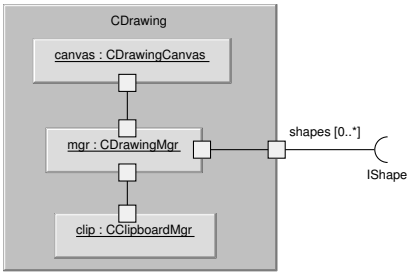


Figure 3: Post-it notes displayed in a drawing

X sells these components to third parties, providing an ADL representation for each component, along with the Java interface definitions, but not the Java implementation source code. X maintains the components using its own CM systems, and periodically releases new versions, which aim for backwards compatibility. A major aim has been to make the drawing components as reusable

as possible. However, due to the large number of customers using the components, changes cannot be specifically introduced for one customer's system.

The definition of CDrawing is shown in figure 4. It is a composite component with parts to handle clipboard functionality and a drawing canvas. An indexed port (shown by a multiplicity of [0..*] which denotes a lower bound of 0 and an unlimited upper bound) is used to hold the list of shapes, which are used for drawing the display. This is shorthand for a set of ports: shape[0], shape[1] and so on.




```
component CDrawing
{
   ports:
      shapes[0..*] requires IShape;

   parts:
      CDrawingMgr mgr;
      CDrawingCanvas canvas;
      CClipboardMgr clip;

   connectors:
      shapes joins shapes to shapes@manager;
      canvas joins surface@canvas to view@manager;
      clip joins board@clip to view@manager;
}
```

Figure 4: Definition of CDrawing

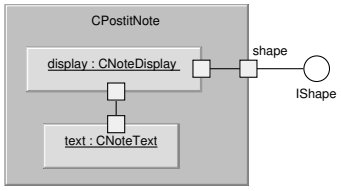
The composite CPostitNote component (figure 5) is designed to work with the framework by providing the IShape interface. The CNoteDisplay part handles the display of text on the screen and the word wrapping. The plain text is stored in the CNoteText part.

3.2 Reuse Scenario

Company Y now wishes to reuse the CDrawing and CPostitNote components to construct a desktop tool for taking notes. For this task, the clipboard is not needed and Y wishes to omit this facility to minimise the size of the application. In addition, CPostitNote must support hyperlinks and the CDrawing component must support changing the zoom level. Although this is a simple scenario, it shares many of the characteristics of real-world reuse situations.

Y clearly must make changes in order to reuse the existing components, leading to our first requirement:

Alter It should be possible to alter a component to allow it to be reused into a new system. The changes required may be extensive.




```
component CPostitNote
{
   ports:
      shape provides IShape;

   parts:
      CNoteDisplay display;
      CNoteText text;

   connectors:
      shape join shape to shape@display;
      text joins data@text to text@display;
}
```

Figure 5: Definition of CPostitNote

In our scenario, Y contacts X and suggests that X makes the changes, or at least provide variation points to make the incorporation of the features possible. However, the provider does not wish to alter the components, as this would require a major change for existing customers. In addition, if this courtesy was extended to all reusers, the architecture would quickly descend into a generic mess with variation points for every conceivable option.

This leads to our next requirement:

NoImpact Alterations to a component for reuse must not impact existing users of the component. Further, the alterations should not impose an obligation on the provider to accept or even know about the changes.

At any rate, the alterations required for reuse are often specific to the new application, and cannot easily be generalised for incorporation into a generic component. In this sense, the alterations fall into the same category as glue code which often has to be written to adapt a component for reuse in a new context. Like glue code, the alterations belong with the system where the component is being reused, not with the original component definition.

Subsequently, Y performs an analysis and decides that its requirements could be met by omitting (or stubbing out) the CClipboardMgr part from CDrawing, upgrading the CNoteDisplay part from CPostitNote and introducing a zoom manager component into CDrawing. Graphically, this would look as shown in figure 6 (changes highlighted).

As a consequence of the analysis, Y decides to make the changes directly to the components themselves. However, a further obstacle is that X has only released their components in a binary form in order to protect their intellectual property. This leads to the next requirement:

NoSource The reuse approach should work even if the full source code of the implementation is not available.

3.3 Evolution Scenario

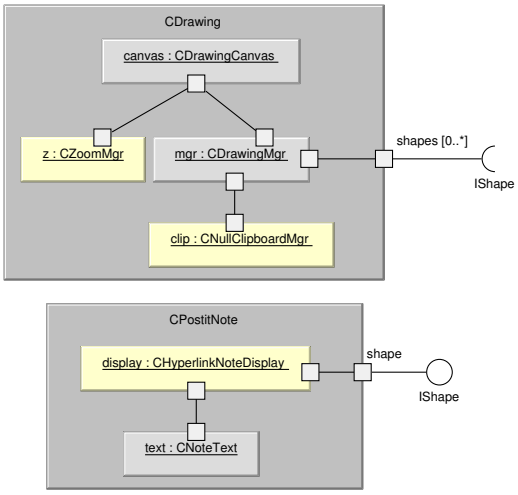


Figure 6: The architecture with Y's changes

Suppose that Y is somehow able to reuse the components for its product, incorporating the changes as described in figure 6. X then issues a new release, upgrading the CDrawing component to use the new CFastDrawingCanvas component, providing improved performance. Clearly, Y wishes to incorporate this improvement into its reuse of CDrawing, leading to a requirement that any reuse solution should not cut off a component from its natural upgrade path from the provider. This effectively rules out copy and paste as a reuse mechanism.

Upgrade It should be possible for a reuser to accept an upgrade to a component, even if that component has been altered for reuse.

4. ADL CONSTRUCTS FOR COMPONENT REUSE AND EVOLUTION

From the analysis of the requirements, constructs have been developed and integrated into the Backbone ADL. The constructs are *resemblance*, *redefinition* and *stratum*.

4.1 Analysing the Requirements

Requirements *Alter* and *NoImpact* appear to be in direct conflict. The provider and other reusers do not have to accept or even know about changes to the component, and yet alterations must still be allowed in order to facilitate reuse.

This situation can be resolved by holding any alterations to a component separately from its original definition. By keeping these alterations with the system that is reusing the component, no-one else will be impacted by, or even aware of, the changes. *Upgrade* further suggests that changes should be held in such a way as to allow them to be analysed and combined with future upgrades of the component. This suggests keeping the alterations explicitly as differences or deltas, rather than storing the entire altered component.

Alter indicates that we need the ability to modify any aspect of a component to facilitate reuse, including interface definitions. This blurs the line between modification for reuse, and the evolution of a component. Such a facility will allow upgrades to also be delivered as a set of differences, distilling the *Upgrade* requirement into the ability to merge two different sets of alterations.

Finally, the requirements imply that we need a way to group related definitions together to differentiate between an existing system and a new system.

The resemblance, redefinition and stratum constructs have been developed in response to the above analysis. Resemblance allows one component to be defined in terms of alterations to a base component, such that the base definition is not affected. Redefinition allows the definition of an existing component to be altered or evolved, and coupled with resemblance allows the new definition to be phrased in terms of alterations to the old definition. Stratum provides a package-like mechanism for grouping a related set of definitions.

4.2 Using Resemblance to Express Change

The resemblance construct allows one component to be defined in terms of changes to another. This is an inheritance-like construct for components, but it does not imply a subtype relationship between components in the way that inheritance usually does between classes [16], as features can be added or removed.

A component can indicate that it resembles a base component, by providing a list of changes in terms of renaming, adding, replacing or deleting elements from the base. For instance, we can form CNewDrawing in terms of CDrawing, thereby altering it for reuse:

```
component CNewDrawing resembles CDrawing {
  replace-parts:
    CNullClipboardMgr clip;
  parts:
    CZoomMgr z;
  connectors:
    zoom joins zoom@z to
      surface@canvas; }
```

This component definition does not perturb the original definition, and does not affect any existing usages of it.

4.3 Using Strata to Control Dependencies

The stratum construct exists to group definitions and control their dependencies. A stratum is a package-like concept which groups a set of related component and interface definitions. It indicates which other stratum are visible for these definitions to refer to through dependency relations. To facilitate strata reuse, circular strata dependencies are not allowed.

To simplify the tracking of dependencies and the analysis of how strata can be combined to create a system, we have restricted the concept to being non-hierarchical. In other words, a stratum cannot contain another stratum. The only valid relationship between stratum is a dependency.

A system is constructed by indicating which strata will be included and in what order. For instance, if CDrawing is in stratum Base and CNewDrawing is in stratum Extension, then a strata load list of {Extension, Base} will cause Base to be loaded into the interpreter, followed by Extension.

4.4 Using Redefinition to Evolve Components

It is not always sufficient to reuse a component by declaring a new component that resembles it. When a component is used in an existing architecture, and a wide-ranging change is required, the original component definition may need to be altered. Redefinition provides a way to alter the definition of the component, but still keep the differences in a separate stratum so that the revised definition is only visible to those systems which include the stratum.

To redefine the CDrawing component, we can use redefinition and resemblance together. The redefinition allows the replacement of an existing definition, and resemblance allows the new definition to be expressed in terms of differences to the previous definition.

```
redefine-component CDrawing
  resembles [previous]CDrawing
{
  replace-parts:
    CNullClipboardMgr clip;
  parts:
    CZoomMgr z;
  connectors:
    zoom joins zoom@z to
      surface@canvas; }
```

Redefinition can also be used without resemblance, in order to wrap and adapt a component. For instance, we can redefine CDrawing to include the old definition as a part which is then delegated to in the new definition.

```
redefine-component CDrawing
{
  ports:
    shapes[0..*] requires IShape;
  parts:
    [previous]CDrawing old;
  connectors:
    delegator joins shapes to
      shapes@old; }
```

If the redefinition is in the Extension stratum and the original definition in Base, then the load list of {Extension, Base} will include the alterations. If, however, another client does not wish to use the changes, Extension is simply omitted from the load list. Conceptually, the changes are applied at start-up time to effect the alterations.

Further, using this construct, a provider can issue updates to a component and release this as another stratum. Suppose that X releases an updated form of CDrawing in a stratum called Update, where CFastDrawingCanvas has replaced the original CDrawingCanvas part.

```
redefine-component CDrawing
  resembles [previous]CDrawing
{
  replace-parts:
    CFastDrawingCanvas canvas; }
```

We can include both sets of alterations above by using the load list of {Update, Extension, Base}. The base definition is loaded, and then modified by the inclusion of the redefinition in the Extension stratum. Finally, the definition is again modified by the redefinition in the Update stratum.

4.5 Summary of Approach

The relationship between the constructs is shown in figure 7, where component definitions are shown as small boxes within a stratum. Stratum are loaded in the reverse order of the load list, and each successive stratum has the ability to alter any definitions in lower strata via redefinition.

Redefinition is shown as an arrow from an upper to a lower stratum, allowing alterations to be made to a definition in a lower stratum. Resemblance is shown as an arrow from a lower to an upper stratum, allow a definition in a stratum to reuse and alter a definition from a lower stratum without perturbing the original definition. Even though the system is loaded from bottom to top, the eventual view of the system is from the top down.

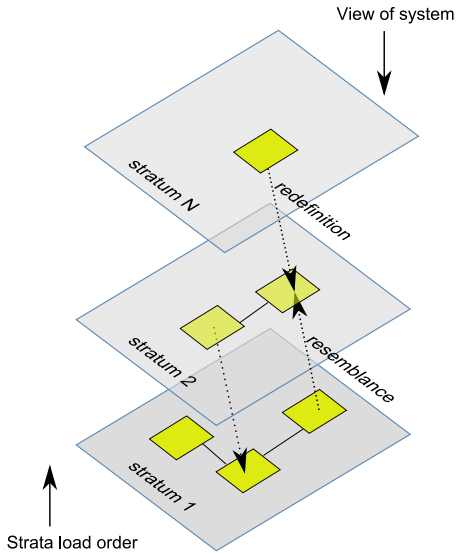


Figure 7: Conceptual view of constructs

Resemblance and redefinition support *Alter* and *NoImpact* by allowing extensive alterations to be made to a component without impacting any existing usages. As explained previously, support for *Upgrade* relies on the ability to combine multiple redefinitions of a single component. This can result in name collisions and other issues, analogous to the problems experienced by the use of multiple inheritance [16]. This situation also occurs when combining two independently developed systems that redefine the same component in a common stratum.

Currently when two redefinitions cannot be merged automatically due to overlap, manual alteration of one of the redefinitions is required. In this case, the replace, add and delete facilities seem rather uncompromising. See the section on further work, detailing possible solutions to this problem.

NoSource is partially supported, as long as the Backbone definitions and interface definitions are provided (even if the implementation code is not). The range of alterations for leaf components is then restricted to adaptation through decoration [2] or outright replacement. It is still possible to freely alter composite components, as they only have a Backbone expression.

Backbone further supports *Alter* by allowing interfaces to be redefined also, and tracking the possible leaf components which also

need to be redefined to support this. At an implementation level, this relies on the Java facility where a definition in one JAR file can supersede or hide the definition in another.

The approach integrates well with existing CM systems. Backbone programs are textual and can be controlled like any set of source files. The stratum and resemblance mechanisms address the concerns about either holding the entire architectural configuration of a system in a single file, or having to scatter the configuration across many files [13]. The definitions within a stratum are held in files, and each file can hold a number of Backbone definitions and redefinitions. This allows related alterations to be grouped and controlled in a simple and straight forward manner.

5. RELATED WORK

A number of approaches have been previously proposed that deal with many of the requirements presented. Amongst other mechanisms, parameterisation is used in Koala to capture options supported by a component [19]. This approach only supports planned variation which conflicts with the *Alter* requirement. This can also result in a combinatorial explosion of options if the parameters of the constituent parts of a composite are also exposed.

Koala and other approaches allow for variation in an architecture [20, 18] to be expressed through variation points. These capture possible component variants at predefined points in an architecture. This is referred to *variation over space*. The points must be planned in advance and designed into a system, which mitigates against this technique for the reuse of existing components which must remain unchanged.

In current product family approaches, if deep modifications or new variation points are required for an existing component these must be introduced by forming a new revision of the component. This is known as *variation over time*, and any unplanned changes require perturbing the original definition violating many of the requirements. Further, repeated introduction of variation points can quickly create complex and generic architectures which are difficult to reuse and reason about.

The introduction of variation points and the general evolution of architectures has been made more feasible through systems like Mae which have integrated CM and architectural concepts [18, 13]. This approach provides an overarching CM system which understands architectural and evolutionary concepts and can support the creation of variants. This approach assumes that all components are available via a unified and consistent CM system, which is not feasible in an environment with many (possibly commercial) component providers. Further this does not solve the need to create many variation points to satisfy those wanting to reuse the components, eventually leading to a complex, very generic architecture which deeply violates the *NoImpact* requirement.

ROOM includes a notion of inheritance which allows for additive and subtractive changes to be specified against actors [14]. A ROOM actor is analogous to a (composite) component with its own thread of control. No formal model of this language has been constructed, and the inheritance facility is not suitable for redefinition, evolution or arbitrary change.

Architectural reconfigurations have previously been used to alter the architecture of a running system, using the property of quiescence to discern when a component can be upgraded [8]. In contrast, the approach presented here provides an intuitive modelling construct for these types of changes, and applies the concepts to the specification and reuse of components. In theory, it is possible to utilise the work on quiescence to effect architectural changes at runtime also.

C2SADEL [11] is a variant of the C2 ADL [17], supporting component specifications through the explicit declaration of state along with pre and post-conditions that indicate changes to that state. This system addresses evolution using a type-based taxonomy of components and connectors and supports configuration evolution, but does not feature composite components. The approach is supported by a modelling environment called DRADEL.

In terms of component technologies, a number of approaches allow for the selective updating of components in a system. COM [1], for example, uses indirection and a registry-based approach to allow one component to instantiate another without having direct knowledge of the exact component type that will be used. Through this mechanism, it is possible to update only some of the components in a system, assuming that the updated components support at least the old interfaces. In contrast, Backbone is focussed on modelling and reasoning about changes to the architecture of a system. The outcome of these changes can eventually be expressed as a set of component updates, which could be realised using the mechanism of the COM component technology.

6. CURRENT STATUS

The interpreter, jUMbLe modelling tool and Alloy model for Backbone are available at the following location: http://www.doc.ic.ac.uk/~amcveigh/backbone.html

6.1 The Backbone Interpreter

An interpreter for Backbone has been developed in order to experiment with the language. This fully supports the resemblance, redefinition and stratum concepts. Note however that in the current interpreter, redefinition automatically presumes resemblance from the base component, as opposed to the examples presented earlier in 4.4.

The interpreter is written in Java, and uses reflection to instantiate and connect components at startup time. A strata load list is supported.

In recent use, it became apparent that names of elements in Backbone programs are being used for two purposes: human understandability and logical identity. E.g. a component specifies that it resembles another component by using its name. Unfortunately, support for renaming interferes with the concept of identity. As a result, it has been decided to explicitly separate the two concepts. The identity will be assigned as a globally unique identifier.

For instance, when defining a component, both the identity and name will be used (identity/name). However, when referring to an element, only the identity is required. This ensures that the identity remains the same, even if the element's name changes. The following definition shows how the code listing in 4.4 might look under this scheme.

```
redefine-component C0012/CDrawing
  resembles [previous]C0012
{
```

```
replace-parts:
  CNullClipboardMgr P009/clip;
parts:
  CZoomMgr P023/z;
connectors:
  zoom joins PT001@P023 to
    PT002@P010; }
```

Clearly, assigning and working with identifiers places a large burden on a designer. However, this is not an issue with a graphical approaches to modelling, which explicitly separate the two concepts. For instance, a dependency relation between component A and B is not linked via the name of the components, but by their logical identities. Changing the names will not affect the relation.

6.2 Graphical Modelling with Backbone

In order to support modelling with Backbone, we have developed a prototype UML2 modelling tool called jUMbLe. A key focus of the approach is to completely hide the textual language (including logical identities), and allow designers to work directly with UML2 composite structure diagrams. The tool allows the creation of composite structure diagrams and package (stratum) diagrams.

The next step is to support the resemblance construct in the modeller. The aim is to allow the designer to alter a component by deleting and adding parts, and have the tool record the changes explicitly.

6.3 Formal Model of Backbone

A formal model of Backbone has been created in Alloy [7]. Alloy is a formal language based on a combination of predicate logic and relational algebra. Specifications can be model checked for counter-examples within a finite state space.

The current Alloy model does not support resemblance, although this is being added. The aim is to show that two redefinitions of the same component can lead to potential conflict. This model will further be used to verify that any solution to this conflict ameliorates the problem.

7. CONCLUSIONS AND FURTHER WORK

From one perspective, resemblance provides a compelling modelling construct which allows an inheritance-like concept to be applied to components at all levels, including the architectural level. It makes it possible to derive other components from a base component, with changes to reflect new requirements, supporting a more incremental approach to system construction. This partially addresses the abstraction problem, as highly specific components can be altered to be reused in a new context. This is useful for internal reuse within a system, as well as for reusing existing components from providers.

By providing uniform reuse and evolution support, the constructs prevent the need to compulsively make components intended for reuse more generic. Unplanned changes can be catered for at the time when the change is required, rather than requiring a costly and sometimes unused upfront investment.

From another perspective, resemblance and the supporting constructs provide a decentralised form of version control, which integrates well with existing CM systems. This offers a multi-authority approach to change control, and allows the changes to be held where the component is reused, rather than were the component is initially defined. Either CM revisions or redefinition can be used for modelling variation over time, and resemblance combined with redefinition can be used for modelling variation over space. Alterations are managed by the team that desires the changes rather than the provider of the component, allowing the original component to retain a coherent architectural vision.

There is a potential conflict between Backbone and a CM system when dealing with variation over time. Ideally, alterations will be specified using redefinitions, even for provider-supplied component upgrades, as this allows better reasoning about the combination of changes. However, it is not possible to keep specifying deltas indefinitely in this way, so a utility is provided which can compress multiple redefinitions into one new definition. We call this process *baselining* in keeping with the terminology of CM system. We are also investigating the possibility of constructing reverse redefinitions from a baseline, which preserve the characteristics of previous definitions.

As explained in 4.5, multiple redefinitions of a single component present a problem when incompatible or overlapping alterations are specified in two independent strata. We are currently pursuing two approaches to resolve this situation. The first approach is based around graph transformations. This involves expressing alterations using an extensible set of transformation patterns. We aim to construct the patterns to limit or resolve any interference between redefinitions although we anticipate the need to for human guidance in some cases.

The second approach is more declarative, where we allow behavioural specifications to be registered with each component. These specifications describe the effect that the component is designed to achieve, in terms of the message protocols of the constituent components. An existing architecture can then be analysed in conjunction with a behavioural specification for a new architecture, with the aim of automatically determining the alterations required to effect the new specification.

8. REFERENCES

[1] D. Box. *Essential COM*. Addison-Wesley Professional, 1997.

[2] E. Gamma, R. Helm, R. Johnson, and V. J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[3] D. Garlan, R. Monroe, and D. Wile. Acme: an architecture description interchange language. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 7. IBM Press, 1997.

[4] M. Goulo and F. Abreu. Bridging the gap between acme and uml 2.0 for cbd. In *Specification and Verification of Component-Based Systems (SAVCBS 2003)*, pages –, 2003.

[5] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley; 1st edition (August 16, 2004), 2004.

[6] U. Holzle. Integrating independently-developed components in object-oriented languages. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 36–56. Springer-Verlag, 1993.

[7] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[8] J. Kramer and J. Magee. The evolving philosophers problem - dynamic change management. *Ieee Transactions on Software Engineering*, 16(11):1293–1306, Nov. 1990.

[9] J. Kramer, J. Magee, and M. Sloman. Configuration support for system description, construction and evolution. In *Proceedings of the 5th international workshop on Software specification and design*, pages 28–33, Pittsburgh, Pennsylvania, United States, 1989. ACM Press.

[10] 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, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[11] N. Medvidovic, D. Rosenblum, and R. Taylor. A language and environment for architecture-based software development and evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 44–53, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[12] OMG. Uml 2.0 specification. *Website*, http://www.omg.org/technology/documents/formal/uml.htm, 2005.

[13] R. Roshandel, A. Van Der Hoek, M. Mikic-Rakic, and N. Medvidovic. Mae—a system model and environment for managing architectural evolution. *ACM Trans. Softw. Eng. Methodol.*, 13(2):240–276, 2004.

[14] B. Selic, G. Gullekson, and P. Ward. Inheritance. In *Real-Time Object-Oriented Modeling*, volume First, pages 255–285. Wiley, 1994.

[15] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[16] A. Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996.

[17] R. Taylor, N. Medvidovic, M. Anderson, E. Whithead Jr., and J. Robbins. A component- and message-based architectural style for gui software. In *Proceedings of the 17th international conference on Software engineering*, pages 295–304, Seattle, Washington, United States, 1995. ACM Press.

[18] A. van der Hoek, M. Mikic-Rakic, R. Roshandel, and N. Medvidovic. Taming architectural evolution. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–10, Vienna, Austria, 2001. ACM Press.

[19] R. van Ommering. Mechanisms for handling diversity in a product population. In *ISAW-4: The Fourth International Software Architecture Workshop*, 2000.

[20] R. van Ommering. Building product populations with software components. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 255–265, New York, NY, USA, 2002. ACM Press.

Simplifying Reasoning about Objects with Tako

Gregory Kulczycki and Jyotindra Vasudeo
Virginia Tech, Falls Church, VA 22043
{gregwk, vasudeo}@vt.edu

ABSTRACT
A fundamental complexity in understanding and reasoning about object-oriented languages is the need for programmers to view variables as references to objects rather than directly as objects. The need arises because a simplified view of variables as (mutable) objects is not sound in the presence of aliasing. Tako is an object-oriented language that is syntactically similar to Java but incorporates alias-avoidance techniques. This paper describes the features of the Tako language and shows how it allows programmers to view all variables directly as objects without compromising sound reasoning. It discusses the benefits of such a language, including its use as an instructional tool to help teach students how to reason formally about their code.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Contructs and Features – *abstract data types, polymorphism, control structures*.

General Terms
Design, Education, Languages, Verification.

Keywords
Alaising, Semantics, Java, Tako

1. INTRODUCTION
References are pervasive in popular object-oriented languages. They permit efficient data assignment and parameter passing of non-trivial objects and are used to implement object identity. However, the need to reason about references and the aliasing that results from their use in such languages has frustrated students, programmers and formalists alike. As a result, significant research has focused on alias control techniques and alias-avoidance techniques for object-oriented languages [15].

Alias control techniques typically involve extending common object-oriented languages with annotations to ensure that certain types of aliasing do not occur [3][9][14][26]. They strive to conform as much as possible with a traditional style of object-oriented programming. Therefore, potentially aliased objects are still the norm, while alias-controlled objects are the exception.

In contrast, alias avoidance techniques typically involve a fundamental change to traditional object-oriented languages by replacing reference assignment—the primary cause of aliasing—with alternatives that do not introduce aliasing, such as value copying [3], destructive read [25], or swapping [12]. These approaches are also referred to as ones that use *unique references*, because in the implementation of languages that use them, each object must have exactly one reference to it. Despite the names *alias control* and *alias avoidance*, nearly all approaches to object aliasing—including ours—permits aliasing to some degree. In alias avoidance techniques, however, potential aliasing is the exception rather than the rule.

A common theme in languages that use alias control and even most alias avoidance techniques is that sound reasoning forces their semantics to be referenced-based. Variables that denote objects are viewed as mere references into a global heap, and method calls modify the heap abstraction rather than the abstract values of the variables (because the abstract *values* of the variables, according to the semantics, are *references*).

The language described in this paper, Tako, is different in this respect. It is intended to facilitate a simple value-based semantics called clean semantics [19] that has the following properties: (1) the state space is comprised of variables whose abstract values are objects rather than references, and (2) the effect of a method call is restricted to the abstract values of the variables involved: the arguments to the call and any relevant globals.

The key benefit of this approach is that it greatly simplifies reasoning about objects. Representing the state space abstractly is straightforward whether programmers are tracing through their code or reasoning about it symbolically. The fact that Tako supports a simple and sound view of the program state makes it particularly useful as an educational tool for introducing students to formal reasoning. From the perspective of object-oriented programming, a drawback of Tako is that it does not conform to some of the paradigms of traditional object-oriented programming. Despite this, Tako, like Java, contains all of the features traditionally found in object-oriented languages, such as classes, inheritance, and polymorphism.

Tako is essentially a redesign of Java that incorporates the alias-avoidance techniques found in Resolve. The Resolve language [30][31] is an integrated programming and specification language intended to support full, heavyweight program verification. For years, various universities including Ohio State, Clemson, and Virginia Tech have offered courses in which variants of Resolve have been used to introduce both undergraduate and graduate students to formal reasoning. Resolve has many features that facilitate formal verification, but as designers of Tako, we are primarily interested in the alias-avoidance features of Resolve and whether they can be successfully and independently applied to a traditional object-oriented language such as Java.

Section 2 of this paper introduces the features of Tako, with emphasis on how it differs from Java. Section 3 describes how

SAVCBS 2006, November 10-11, 2006, Portland, Oregon, USA.

Tako supports clean semantics and facilitates reasoning, using examples similar to those we have used in courses at Virginia Tech. Section 4 discusses related work and future directions.

2. OVERVIEW AND FEATURES

As illustrated in Figure 1, a Tako stack implementation is syntactically similar to a Java stack implementation. They declare the same variables, they have similar methods, and, with one exception, they use the same keywords.

2.1 Data Assignment

The most important difference between the classes in Figure 1 is that wherever Java uses reference assignment—the main source of aliasing—Tako uses alternative data assignment mechanisms. As in the Resolve language, Tako uses swapping as its primary means of data assignment. In the body of the push method, the Java code assigns the object in *contents*[*top*] to *x* by copying its reference. But the Tako code uses a swapping operator (:=:) to swap the values of *contents*[*top*] with *x*. A call to Java's push method creates an alias between the incoming object *x* and the top element of the current stack; a call to Tako's push method transfers *x*'s object to the top of the stack, and replaces it with some unspecified but valid object of its type.

Swapping is described as simultaneous assignment in [17] and is proposed as an alternative to both reference and value copying in [12]. Swapping is a constant-time operation because a compiler can implement it by swapping object references; swapping preserves unique references, so programmers can reason about it as if object values are swapped. Swapping never creates aliases. For efficiency, a compiler can implement small objects such as integers and booleans directly as objects while implementing non-trivial objects using references.

Swapping is symmetric—it requires both objects to be of the same type. For assigning a type to a supertype, Tako provides an initializing transfer operator (←). The initializing transfer operation $s \leftarrow c$ assigns circle *c* to shape *s* and initializes *c* to a new circle object. The approach does not introduce aliasing, but it does require the creation of a new object, and—because Tako's underlying implementation guarantees unique references—it ultimately requires memory from an old object to be reclaimed. Therefore, initializing transfer is less efficient than swapping.

2.2 Function Assignment

Both the swap statement and the transfer statement require a variable on either side of the operator. For assigning the result of an expression evaluation to a variable, Tako provides a function assignment operator (:=), as in the *new* expression that initializes the *contents* array in the stack's constructor in Figure 1. The same operator is used to copy values from one variable to another, as in the statement *MAX* := *n*.

Java methods return references rather than values, potentially introducing aliasing. For example, the Java *pop* method is a side-effecting function that returns a reference to *contents*[*top*], so that the assignment *x* = *s*.*pop*() causes two references to point to the same object. Tako avoids this problem with the introduction of a distinguished *result* variable that always holds the return value of a non-void method. The result variable is automatically initialized at the beginning of the method, and whatever object value it holds at the end of the method is returned. Since the result variable goes out of scope at the end of the method, we know that any returned reference (to the result variable's object) will continue to be unique. The syntax ***return*** ⟨*expression*⟩ is not permitted in Tako, though the *return* keyword may be used alone to denote that the program should return from the method with the current *result* value.

Consider the following Tako *pop* method written as a side-effecting function.

```
public Object pop() {
    top--;
    result :=: contents[top];
}
```

The result variable gets a new initial Object as soon as the

```
public class BddStack {
    private final int MAX;
    private Object[] contents;
    private int top;

    public Stack(int n) {
        MAX := n;
        contents := new Object[MAX];
    }
    public void push(Object x) {
        assert depth() < MAX;
        contents[top] :=: x;
        top++;
    }
    public void pop(Object x) {
        assert depth() > 0;
        top--;
        x :=: contents[top];
    }
    public int depth() {
        result := top;
    }
}
```

```
public class BddStack {
    private final int MAX;
    private Object[] contents;
    private int top;

    public Stack(int n) {
        MAX = n;
        contents = new Object[MAX];
    }
    public void push(Object x) {
        assert depth() < MAX;
        contents[top] = x;
        top++;
    }
    public Object pop() {
        assert depth() > 0;
        top--;
        return contents[top];
    }
    public int depth() {
        return top;
    }
}
```

Figure 1. A Tako bounded stack implementation (left) compared to a Java bounded stack implementation (right)

method is invoked, as if the first statement were *Object* ***result*** := ***new*** *Object*(). After the swap operation, *result* holds the object originally held by *contents*[*top*] and *contents*[*top*] holds the newly created object originally held by *result*. Thus, the method returns the object originally at the top of the current stack. Internally, it places a new initialized object in the cell of the *contents* array that previously held the stack's top element.

The function assignment operator can also be used to copy objects. The compiler expects the function assignment operator to have a variable on the left-hand side and an expression on the right-hand side. If it does not find a variable on the left-hand side, it will report an error. However, if it finds a variable rather than an expression on the right-hand side, it will check to see if a *replica* method has been implemented for the variable's type. If so, it will call the replica method for the variable's object; if not, it will report that no replica method could be found. Thus, the compiler considers the statement $s := t$ to be special syntax for $s := t.replica()$. The replica method is intended to be used for small objects where copying is the preferred form of data assignment, such as integers and booleans. In principle, though, any Tako class can be extended with a replica method.

2.3 Parameter Passing

Parameter passing in Java is accomplished by copying the references of the arguments to the formal parameters without copying them out again. This approach is problematic for a language that intends to facilitate value semantics, because the semantics that describe this form of parameter passing are difficult to formulate without introducing the notion of reference. Java parameter passing cannot be viewed as in-out because assignments to formals inside the body are not reflected in the actuals. It cannot be viewed as copying object values in only, because in Java you *can* update an argument's conceptual object value as long as you do not change its reference value.

Tako avoids this difficulty by fully supporting in-out (also called value-result) parameter passing. Conceptually, programmers can reason about in-out parameter passing as if object values are copied into the method, the method is executed, and object values are copied out again. However, a compiler can implement in-out parameter passing efficiently by copying references to the formal parameters, executing the method, and copying references back out, which is what C# does with *ref* parameters.

The effects of in-out parameter passing with references and in-out parameter passing with objects are semantically equivalent whenever the arguments are not aliased [11]. Since Tako avoids aliasing, in-out parameter passing is an appropriate choice. Note that Tako uses in-out parameter passing by default for all parameters, even the current *this* parameter, which cannot be a *ref* parameter in C#.

Using in-out parameter passing gives programmers the option of writing certain methods as procedures (i.e., void methods) rather than side-effecting functions. For example, the Tako *pop* method is a procedure whereas the Java *pop* method must be written as a side-effecting function. A function is free of side-effects if its execution does not change the program state. Keeping functions free from side-effects simplifies reasoning about programs that include conditions, as in if and while statements.

One aliasing problem that in-out parameter passing does not solve is the repeated argument problem [19]. As long as parameter passing is implemented by copying references—whether in-out or in-only—aliasing can be introduced when arguments are repeated in a call. For example, the call $q.append(q)$ introduces aliasing between the implicit formal parameter *this* and the explicit formal parameter in the body of the append method. The call $a[i].append(a[j])$ does the same when i and j are equal.

We are currently exploring alternative designs for handling repeated arguments. One option is to throw a runtime error when repeats occur, another is to initialize the second and subsequent repeats, as described in [19]. In either case, the compiler will warn programmers when arguments are potentially repeated.

Tako includes an *eval* parameter mode that indicates that a function is expected for evaluation. The *eval* mode is often used for small types such as integers and booleans. As with function assignment, if a variable a is given where a function is expected, the compiler will translate it as *a.replica*(). If no replica function is found, the compiler will report an error. Since the result of an expression evaluation is always a new object, repeated arguments do not pose a problem for eval parameters.

A potential problem with in-out parameter passing and Java-like inheritance concerns the passing of subtypes. If *c* has type *Cat* and *d* has type *Dog*, what should the effect be of "*s.push*(*c*); *s.pop*(*d*);"? If we blindly permit this, it results in the dog variable d holding a cat object, causing a type violation. Some object-oriented languages that support the conceptual equivalent of in-out parameters (such as C#) do not allow programmers to pass subtype objects to them. In Tako, this is not an option—particularly since Tako does not yet support generics. The stack class in Figure 1 would be of little value if we were restricted to populating the stack with objects of type *Object*.

One option is that when a parameter is transferred back, an implicit cast is done. If the cast cannot be made, a runtime error occurs. Due to the poor performance of runtime casts in Java, this solution, though adequate, is not the most efficient, and we are currently exploring alternatives.

2.4 Initialization

As in the case of the initializing transfer operator, Tako sometimes requires the compiler to automatically create new, initialized objects. This is done for newly declared variables as well. In Java, the declaration "*Circle c*;" does not initialize *c*, and if *c* is not assigned to before it is used, a compile-time error occurs. In Tako, the declaration *Circle c* is interpreted by the compiler as *Circle c* := ***new*** *Circle*().

Types that do not have a default constructor get initialized to null. Tako tries to facilitate a view of all variables as objects, so including null values in the language may seem odd. Technically, a value semantics can accommodate null values by introducing them as distinguished "object" values. Specifications are complicated when null values are permitted [7]. However, types derived from interfaces cannot have constructors, and some classes, such as the bounded stack class in Figure 1, effectively require parameters in their constructors. Therefore, in the current version of Tako, types may be nullable or non-nullable, based on whether a default constructor is provided for the class. Non-nullable classes are encouraged because they simplify reasoning. Null pointer exceptions do not occur with non-nullable classes.

2.5 Pointer Component

One of the primary motivations for Tako is the simplification of object-based reasoning through alias avoidance. By providing

efficient alternatives to data assignment and parameter passing that avoid aliasing, Tako supports the construction of typical classes as pure value types, allowing programmers to reason about variables directly as objects. There are circumstances, however, in which aliasing cannot be avoided without sacrificing efficiency. For example, references and aliasing are needed to implement all the methods in the list component in constant time. For the efficient implementation of lists and other typically linked data structures, Tako provides a pointer component that models a system of linked locations. Each location holds data (objects) and has a fixed number of links to other locations. Position variables reside at the various locations. In the following Position interface, k is the number of outgoing links at each location.

```
interface Position {
  static final int k;
  public void takeNew();
  public void moveTo(Position p);
  public void redirectLink(int k, Position p);
  public void followLink(int k);
  public void swapContents(Object x);
  public boolean isWith(Position p);
  public boolean isAtVoid();
}
```

Presenting pointers in the form of a class has the advantage that programmers can reason about pointers the same way they reason about any other object. No special proof rules are needed for pointers, and no universal heap structure needs to be included in the semantics. Programmers can maintain a sound view of position variables as abstract object values just as they can with any other variable in Tako.

```
allocate p;        p.takeNew();
  p -> q;            p.moveTo(q);
  p -> q^PREV;       p.moveTo(q); p.follow(PREV);
  p^NEXT -> q;       p.redirectLink(NEXT, q);
  p <-> q;           p :=: q;
  p *:=: s;          p.swapContents(s);
```

Figure 2. Special syntax for position objects

The Position interface provides a way for programmers to view pointers in a value-based reasoning environment. However, the compiler cannot implement a Position class as it can other classes because position variables must not only provide the functional benefits of pointers but the performance benefits as well. For example, the call *p.moveTo*(*q*) moves *p* to *q*'s location, effectively resulting in *p* and *q* being aliases. Although the programmer can reason about the statement as a method call, the Tako compiler will implement it by copying a single reference.

With the help of the special syntax shown in Figure 2, the implementation of linked data structures using Tako pointers has a relatively straightforward translation into Java, as illustrated in Figure 3. The Tako pointer component shares many similarities with Resolve's *Location_Linking_Template*, whose specification and reasoning in terms of clean semantics is detailed in [20].

3. VALUE-BASED REASONING

This section describes the properties of the value-based reasoning system that Tako facilitates and presents an example of specification and verification using Tako.

3.1 Clean Semantics

The value-based semantics for Tako should have the following two properties. First, the state space should be based on the object values of variables. Specifically, at any point in the program, the state consists of the abstract object values of the currently defined programming or conceptual variables. Conceptual variables are similar to model variables or data groups [8][22], and they are used to help model the program state. An example of their use is given in the mathematical model of the reference-based stack component below. The second property of our semantics is a frame property [6]. It states that the portion of the state space that can be modified by a method call is restricted to certain syntactically discernible variables—the arguments to the call and any global (static) variables listed in the *affects* clause of the method's declaration.

Together, the *variable-based* property and the *effects-restricted* property define the behavior of a clean semantics as given in [19]. This notion was proposed as a syntactic yet formalizable way to capture the notion of localized reasoning about operation

```
public class LinkedList {
    class Node is Object^(NEXT);
    private Node head, pre, last;
    private int left_length, right_length;

    public LinkedList() {
        allocate head;
        pre -> head;
        last -> head;
    }

    public void insert(Object x) {
        Node post, new_pos;
        post -> pre^NEXT;
        allocate new_pos;
        new_pos *:=: x;
        pre^NEXT -> new_pos;

        ...
```

```
public class LinkedList {
    class Node {
        Node next = null;
        Object contents = new Object();
    }
    private Node head, pre, last;
    private int left_length, right_length;

    public LinkedList() {
        head = new Node();
        pre = head;
        last = head;
    }

    public void insert(Object x) {
        Node post, new_pos;
        post = pre.next;
        new_pos = new Node();
        new_pos.contents = x;
        pre.next = new_pos;

        ...
```

Figure 3. A portion of a linked list implementation using Tako (left) compared to one using Java (right)

invocations. The potential for aliasing in a programming language complicates reasoning and makes clean semantics harder to achieve. However, a system may permit aliasing and still conform to clean semantics, as with Tako's pointer component.

3.2 Simple Stack Specification

This section describes the specification and reasoning for a Tako stack component. The stack component is a typical Tako component because it specifies a single mathematical model for its type and does not require any conceptual state variables to describe its behavior. We have used this example and others like it to introduce graduate students to formal reasoning in courses not normally associated with formal methods, such as software engineering, and theory of algorithms. The software engineering course covers general topics, but approximately the last 25% of the course focuses on component-based software engineering and formal methods. Tako code is used to illustrate key principles. The algorithms course uses the Cormen et al. text [10], whose latest edition puts greater emphasis on demonstrating the correctness of algorithms and includes discussions on loop invariants. Their "proofs" of correctness are typical mathematical proofs given in natural language. We occasionally augment these proofs using formally specified Tako components and demonstrate formal correctness through a symbolic reasoning table, like the one described below.

```
import spec.MathString;
public interface Stack {
   model MathString;
   initialization ensures
        this = EMPTY_STRING;
   public void push(Object x);
     ensures this = <#x> o #this;
   public void pop(Object x);
     requires |this| > 0;
     ensures #this = <x> o this;
   public int depth();
     ensures this = #this and result = |this|;
}
```

Figure 4. A specification for a Tako stack

Figure 4 gives a specification for an unbounded Tako stack. The *model* clause indicates that an object of type Stack is modeled as (has a conceptual value of) a mathematical string of objects. The clause does not specify a variable name as the current stack *this* is implied. A string is similar to a sequence except that it is not indexed. The *initialization ensures* clause gives the behavior of the default (no-argument) constructor, which in this case guarantees that an initial stack will be empty.

A Java stack specified in JML (the Java Modeling Language [21]) would likely be modeled using a JMLObjectSequence. JML provides three different sequences depending on weather the sequence holds object types (references to objects), equals types (non-clonable objects), or value types (object values). Variables in Tako always denote object values, so all mathematical structures hold value types, eliminating the need for such a distinction.

A hash mark (#) is used only in an ensures clause—it denotes the incoming value of a variable. The expression $\langle e \rangle$ is a unary string containing the element e, and the symbol o denotes string concatenation. So the ensures clause for *push* states that the current stack is equal to the string containing the original value of x concatenated with the original stack value. Notice that the outgoing value of x is left unspecified. The method is specified this way partly because of the new paradigm for component construction in Tako. If we specified that the outgoing value of x was the same as the incoming value of x ($x = \#x$) we would effectively force the implementer to make a deep copy of x. When we don't specify how x changes, the reasoning system guarantees only that x contains a valid value of its type.

Only actual parameters and global variables listed in an *affects* clause may be modified by a method, so Tako does not provide a *modifies* or *assignable* clause as JML does. None of the stack methods modifies any global state variables, so none of them have an affects clause.

In the *depth* method, the keyword ***result*** denotes the return value. Also, since the current stack (***this***) is considered the first parameter to the call, our frame property states that its value *may* be modified. But we do not want the function to have side-effects, so we must explicitly state in the ensures clause that it is *not* modified.

Once students are introduced to formal specification, they practice reading the specifications by tracing through code. The following tracing table (Table 1) gives an example. Students are given variable values for state 0 and asked to fill in the other states. The assertion x = ?? indicates that x is a valid but unspecified value of its type.

Table 1. Tracing table for simple stack code

St	Facts
0	s = ⟨ 4, 5 ⟩ **and** t = ⟨ 7, 8, 9 ⟩ **and** x = 3
t :=: s;	
1	s = ⟨ 7, 8, 9 ⟩ **and** t = ⟨ 4, 5 ⟩ **and** x = 3
t.push(x);	
2	s = ⟨ 7, 8, 9 ⟩ **and** t = ⟨ 3, 4, 5 ⟩ **and** x = ??

3.3 More Complex Stack Specification

The fact that the potential for aliasing does not exist greatly simplifies our ability to represent and understand the state space. Consider the tracing table in Table 2 for similar code with the swap statement replaced by an assignment. If this were Java code, we would know that the object value of s in state 2 is ⟨ 3, 4, 5 ⟩ since s and t point to the same object. But if we assume that variables denote strict object values we will conclude that s = ⟨ 4, 5 ⟩ is unchanged, making the simple value-based specification unsound in the presence of aliasing.

Table 2. Problematic tracing table for code with aliasing

St	Facts
0	s = ⟨ 4, 5 ⟩ **and** t = ⟨ 7, 8, 9 ⟩ **and** x = 3
t = s;	
1	s = ⟨ 4, 5 ⟩ **and** t = ⟨ 4, 5 ⟩ **and** x = 3
t.push(x);	
2	s = /* *what goes here?* */ **and** t = ⟨ 3, 4, 5 ⟩ **and** x = ??

We can remedy this by giving the stack a specification that accounts for aliasing, such as the one in Figure 5. Here, stack variables are modeled as locations, and the conceptual variable

obj maps locations to mathematical strings (conceptual stack objects). Like to model variables in JML, conceptual variables do not correspond to programming objects, but they are necessary for reasoning about the component. The conceptual variable *obj* is a global variable, so it must be listed in the affects clause of any method (such as push) that potentially modifies its value.

With this specification, we can reason soundly about the stack component in the presence of aliasing, as shown in Table 3. However, even this specification is an oversimplification of object-oriented logics as it does not account for the effects of (future) inheritance.

```
import spec.MathString;
import spec.Location;

public interface Stack {

   var obj: Location → MathString;
   model Location;

   public void push(Object x);
      affects obj;
      ensures this = #this and
         obj(this) = <#x> o #obj(this) and
         ∀r: Stack, if r ≠ this then
            obj(r) = #obj(r);
```

Figure 5. Portion of a reference-based stack specification

We can simplify the appearance of the specification in Figure 5 to something resembling Figure 4 by indicating—as JML does—that all variables have reference semantics. This approach, however, will not simplify the states in our tracing table. There is no rule that can tell us how to transition from the state "s = ⟨ 4, 5 ⟩ **and** t = ⟨ 4, 5 ⟩ **and** x = 3" through *s.push*(*x*), to the next state, without telling us whether *s* and *t* are aliased.

Table 3. Sound tracing table for stack code with aliasing

St	Facts
0	s = @47 **and** t = @53 **and** x = 3 **and** contents = { @47 ↦ ⟨ 4, 5 ⟩, @53 ↦ ⟨ 7, 8, 9 ⟩ }
t = s;	
1	s = @47 **and** t = @47 **and** x = 3 **and** contents = { @47 ↦ ⟨ 4, 5 ⟩, @53 ↦ ⟨ 7, 8, 9 ⟩ }
s.push(x);	
2	s = @47 **and** t = @53 **and** x = 3 **and** contents = { @47 ↦ ⟨ 3, 4, 5 ⟩, @53 ↦ ⟨ 7, 8, 9 ⟩ }

3.4 Reasoning about Stack Reverse

Consider the specification and implementation for the stack reverse method in Figure 6. The method has no precondition, and the postcondition states that the mathematical string that models the stack will be reversed. The implementation pops elements one at a time from the current stack and pushes them onto a temporary stack. Before the method returns, the current stack is swapped with the temporary stack. We want to reason about the correctness of this implementation with respect to its specification, so we include an invariant for the loop. The decreasing clause allows us to prove that the loop terminates.

A tracing table for the reverse method is given below. It demonstrates that when the current stack has a value of ⟨ 3, 4 ⟩, the reverse method will change its value to ⟨ 4, 3 ⟩, satisfying the postcondition of the reverse method.

Once students are comfortable with tracing tables, we introduce them to symbolic reasoning [13][30]. A symbolic reasoning table for the reverse procedure is given in Table 5. For each state, the table shows a path condition, facts, and obligations. The path condition must hold for the program to enter the specified state, the facts tell us what we know about the values of the variables in that state, and the obligations tell us what needs to be true before we can move to the next state.

```
public void reverse()
        ensures this = REV(#this);
    {
        Stack temp;
        Object x;
        while (this.depth() != 0)
           decreasing |this|;
           maintaining REV(temp) o this = #this;
        {
           this.pop(x);
           temp.push(x);
        }
        this :=: temp;
    }
```

Figure 6. Specification and implementation of stack reverse

In general, obligations come from preconditions of called operations and facts come from their postconditions. For example, the requires clause of the pop method indicates that the stack must be non-empty. All variables are indexed with the current state, so in state 1 we have an obligation to show that $|\textbf{\textit{this}}_1| > 0$. The pop method ensures that the old value of the stack is equivalent to the new value of the *x* parameter concatenated with the new value of the stack. So in state 2 we have the fact that $\textbf{\textit{this}}_1 = \langle x_2 \rangle$ o $\textbf{\textit{this}}_2$. We also know that $temp_2 = temp_1$ in accordance with the frame property.

Table 4. Tracing table for stack reverse method

St	Facts
0	**this** = ⟨ 3, 4 ⟩ **and** temp = ⟨ ⟩ **and** x = 0
while (**this**.length() != 0) {	
1	**this** = ⟨ 3, 4 ⟩ **and** temp = ⟨ ⟩ **and** x = 0
this.pop(x);	
2	**this** = ⟨ 4 ⟩ **and** temp = ⟨ ⟩ **and** x = 3
temp.push(x);	
3	**this** = ⟨ 4 ⟩ **and** temp = ⟨ 3 ⟩ **and** x = ??
// **this**.length() != 0	
1′	**this** = ⟨ 4 ⟩ **and** temp = ⟨ 3 ⟩ **and** x = ??
this.pop(x);	
2′	**this** = ⟨ ⟩ **and** temp = ⟨ 3 ⟩ **and** x = 4
temp.push(x);	
3′	**this** = ⟨ ⟩ **and** temp = ⟨ 4, 3 ⟩ **and** x = ??
} // **this**.length() = 0	
4	**this** = ⟨ ⟩ **and** temp = ⟨ 4, 3 ⟩ **and** x = ??
this :=: temp;	
5	**this** = ⟨ 4, 3 ⟩ **and** temp = ⟨ ⟩ **and** x = ??

The facts in state 0 come from the precondition (if any) of the method you are trying to prove correct, and the obligations in the last state come from the method's postcondition.

A reasoning table for code that involves a loop is slightly more complex than one that does not. It effectively breaks up the table into three separate sub-tables dedicated to proving the following three properties: *initialization* – the invariant is true when the loop first begins; *maintenance* – if the invariant holds at the beginning of the *n*-th iteration, it also holds at the end of the (*n*+1)-th iteration; and *termination* – the invariant and the negation of the while condition allow you to prove what you want to prove (in our case, the postcondition of the reverse method).

Table 5. Symbolic reasoning table for stack reverse method

St	P Cond	Facts	Obligations
0		Object.**is_init**(x_0) **and** Stack.**is_init**($temp_0$) **and** **this**$_0$ = #**this**	|**this**$_0$| ≠ 0 ⇒ REV($temp_0$) o **this**$_0$ = #**this**
while (**this**.length() != 0) {			
1	|**this**$_1$| ≠ 0	REV($temp_1$) o **this**$_1$ = #**this** **and** x_1 = ??	|**this**$_1$| > 0
this.pop(x);			
2	|**this**$_1$| ≠ 0	**this**$_1$ = ⟨x_2⟩ o **this**$_2$ and $temp_2$ = $temp_1$	
temp.push(x);			
3	|**this**$_1$| ≠ 0	**this**$_3$ = ⟨x_2⟩ o **this**$_2$ **and** x_3 = ?? **and** $temp_3$ = $temp_2$	REV($temp_3$) o **this**$_3$ = #**this** **and** |**this**$_3$| < |**this**$_1$|
}			
4	|**this**$_4$| = 0	REV($temp_4$) o **this**$_4$ = #**this** **and** x_4 = ??	
this :=: temp;			
5	|**this**$_4$| = 0	**this**$_4$ = $temp_4$ **and** $temp_5$ = **this**$_4$ **and** x_5 = x_4	**this**$_5$ = REV(#**this**)

The obligation in state 0 must be discharged to prove the initialization property. Discharging the first part of the obligation in state 3 proves maintenance. And discharging the obligation in state 5 proves the termination property. Note that the second part of the obligation in state 3 comes from the decreasing clause. It must be discharged to prove that the while loop terminates.

The obligations may be discharged with a theorem prover, but they may also be simple enough for students and programmers to reason about themselves. Take, for example, the obligation in state 5. We want to prove that ***this*$_5$** = *REV*(#***this***). We know from the facts in state 5 that ***this*$_5$** = $temp_4$, so it suffices to show that $temp_4$ = *REV*(#***this***). We know from the facts in state 4 that *REV*($temp_4$) o ***this*$_4$** = #***this***, and we know from the path condition that \|***this*$_4$**\| = 0 which can only happen if $this_4$ is empty. So we know *REV*($temp_4$) = #***this***. Hence, $temp_4$ = *REV*(#***this***).

4. DISCUSSION

The difficulty of reasoning in the presence of aliasing is well known [15], and numerous techniques to control aliasing in object-based languages have been proposed [2][3][9][23]. We have used the term alias avoidance to refer to techniques that promote alias-free alternatives to reference assignment, such as the approach based on destructive read in [25] and the approach based on swapping in [12]. The term uniqueness has generally come to refer to techniques such as [25] and variations that employ the destructive read operator. They preserve unique references to objects, but they also support a borrowing mechanism that allows programmers to violate the uniqueness condition when they deem it useful. Borrowing raises the potential for aliasing and is therefore not conducive to value semantics. However, there is nothing fundamental about the destructive read operator that prevents it from being used in a language that does support value-based reasoning.

Most proposals for controlling object aliasing attempt to minimize the impact of their approach on programmers who have become accustomed to an object-oriented style of programming. While some practitioners have reported positive experiences when using a swap operator in object-based applications [16], we understand that allowing programmers to view all classes as value types is a radical departure from the traditional object-oriented paradigm. Among other things, it removes the distinction between primitive types and user-defined types; it forces programmers who want to modify an object inside a container to remove, modify, and re-insert it; and it requires programmers to rethink common design patterns whose implementations traditionally use aliasing, such as the singleton and the observer patterns. The question of whether programmers accustomed to traditional object-oriented paradigms can make the transition to object-oriented languages that support direct reasoning can only be answered empirically. The desire to answer this question is one of our primary motivations for creating the Tako compiler.

The current focus for the Tako language is its use as an educational tool to introduce students to formal reasoning. But we would like to develop it into a practical programming language that could be used alone or with Java components to develop non-trivial applications. Practical concerns include compiler optimizations, especially in the areas of automatic initialization and automatic casting. The current implementation of the Tako compiler is available at SourceForge under the name *takocompiler*. We have developed a medium-sized application (about 40 classes) in Tako that interfaces with Java Swing components. We have no immediate plans to see how other object-oriented languages might benefit from alias-avoidance, but we would consider it a worthy long-term pursuit. In-out parameter passing is easier to implement on the .NET platform than the JVM, making C# appealing for our research, and Eiffel's value-based expanded types may serve as a basis for alias avoidance.

From a research perspective, we are still exploring the impact that alias avoidance and clean semantics will have on advanced language features such as inheritance and concurrency. To this end, we hope to leverage both Resolve research and the ongoing research on specification and verification of Java-like languages. We are exploring using Tako in the context of both lightweight and full verification. The general trade-offs on verification rigor are described in [33]. The ultimate goal might be a verifying compiler that—due to the relative simplicity of Tako's semantics—could be much more automated than a comparable tool for Java, and would generate verified Java byte code.

5. ACKNOWLEDGMENTS

Our thanks to Murali Sitaraman, Bill Ogden, Bruce Weide and other members of the Reusable Software Research Group for their valuable insights into the specification and verification of object-based languages.

6. REFERENCES

[1] Abadi, M. and Leino, K.R.M. A logic of object-oriented programs. Dauchet, M. ed. In *Proceedings TAPSOFT '97*, pages 682-696. Springer-Verlag, New York, 1997.

[2] Aldrich, J., Kostadinov, V. and Chambers, C., Alias annotations for program understanding. In *Proceedings OOPSLA '02*, pages 311-330. ACM Press, 2002.

[3] Almeida, P.S., Balloon types: Controlling sharing of state in data types. In *Proceedings ECOOP '97*, pages 32-59. Springer-Verlag, New York, 1997.

[4] Baker, H.G. 'Use-once' variables and linear objects—storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, *30* (1). pages 45-52. 1995.

[5] Bokowski, B. and Vitek, J., Confined types. In *Proceedings OOPSLA '99*, pages 82-96. ACM Press, 1999.

[6] Borgida, A., Mylopoulos, J. and Reiter, R., ... And nothing else changes?: The frame problem in procedure specifications. In *Proceedings of the 15th International Conference on Software Engineering*, pages 303-314. IEEE Computer Society Press, 1993.

[7] Chalin, P. and Rioux, F., Non-null references by default in the Java Modeling Language. In *Proceedings SAVCBS '05*, pages 70-76. 2005.

[8] Cheon, Y., Leavens, G.T., Sitaraman, M. and Edwards, S. Model variables: Cleanly supporting abstraction in design by contract. *Software: Practice and Experience*, *35* (6), pages 583-589. 2005.

[9] Clarke, D.G., Potter, J.M. and Noble, J., Ownership types for flexible alias protection. In *Proceedings OOPSLA '98*, pages 48-64, ACM Press, 1998.

[10] Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C., *Introduction to Algorithms*, 2nd ed., McGraw-Hill, 2001.

[11] Gries, D. and Levin, G. Assignment and procedure call proof rules. *ACM Transactions on Programming Languages and Systems*, *2* (4), pages 564-579. 1980.

[12] Harms, D.E. and Weide, B.W. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, *17* (5). pages 424-435. 1991.

[13] Heym, W. *Computer Program Verification: Improvements for Human Reasoning*, Ph.D. thesis, The Ohio State University (1995)

[14] Hogg, J., Islands: Aliasing protection in object-oriented languages. In *Proceedings OOPSLA '91*, pages 271-285. ACM, 1991.

[15] Hogg, J., Lea, D., Wills, A., deChampeaux, D. and Holt, R. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, *3* (2). pages 11-16. 1992.

[16] Hollingsworth, J.E., Blankenship, L. and Weide, B.W., Experience report: Using Resolve/C++ for commercial software. In *Proceedings FSE '00*, pages 11-19. ACM Press, 2000.

[17] Kieburtz, R.B., Programming without pointer variables. In *Proceedings of the SIGPLAN '76 Conference on Data: Abstraction, Definition, and Structure*, ACM Press, 1976.

[18] Krone, J. *The Role of Verification in Software Reusability*, Doctoral Thesis, The Ohio State University, 1988.

[19] Kulczycki, G., Sitaraman, M., Ogden, W.F. and Weide, B.W. *Clean Semantics for Calls with Repeated Arguments*, Technical Report RSRG-05-01, Clemson University, 2005.

[20] Kulczycki, G., Sitaraman, M., Weide, B. and Rountev, N., A specification-based approach to reasoning about pointers. In *Proceedings SAVCBS '05*, pages 55-62. 2005.

[21] Leavens, G.T., Baker, A.A. and Ruby, C. JML: A notation for detailed design. Simmonds, I. ed. *Behavioral Specifications of Businesses and Systems*, Kluwer, 1999.

[22] Leino, K.R.M., Data groups: Specifying the modification of extended state. In *Proceedings OOPSLA '98*, pages 144-153, ACM Press, 1998.

[23] Meyer, B., *Object-Oriented Software Construction*, 2nd ed. Prentice Hall, 1997.

[24] Müller, P. and Poetzsch-Heffter, A. Modular specification and verification techniques for object-oriented software components. Sitaraman, M. and Leavens, G. eds. *Foundations of Component-Based Systems*, Cambridge University Press, Cambridge, United Kingdom, 2000.

[25] Minsky, N.H., Towards alias-free pointers. In *Proceedings ECOOP '96*, pages 189-209. 1996.

[26] Noble, J., Vitek, J. and Potter, J. Flexible alias protection. *Lecture Notes in Computer Science*, *1445*. pages 158-185. 1998.

[27] Ogden, W.F. *The Proper Conceptualization of Data Structures*. The Ohio State University, Columbus, OH, 2000.

[28] O'Hearn, P., Reynolds, J. and Yang, H. Local reasoning about programs that alter data structures. *Lecture Notes in Computer Science*, *2142*, 2001.

[29] Popek, G.J., Horning, J.J., Lampson, B.W., Mitchell, J.G. and London, R.L. Notes on the design of Euclid. *ACM SIGPLAN Notices*, *12* (3), poages 11-18. 1977.

[30] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W., Pike, S. and Hollingsworth, J.E., Reasoning about software-component behavior. In *Proceedings ICSR '00*, pages 266-283. Springer-Verlag, 2000.

[31] Sitaraman, M. and Weide, B.W. Component-based software using Resolve. *ACM Software Engineering Notes*, *19* (4), pages 21-67. 1994.

[32] Weide, B.W. and Heym, W.D., Specification and verification with references. In *Proceedings SAVCBS '01*. 2001.

[33] Wilson, T., Maharaj, S., and Clark, R.G., Omnibus verification policies: a flexible, configurable approach to assertion-based software verification. In *SEFM '05*. IEEE Press. 2005.

SAVCBS 2006 CHALLENGE PROBLEM SOLUTIONS

VC Generation for Functional Behavior and Non-Interference of Iterators

Bart Jacobs*
Dept. CS, K.U.Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
bartj@cs.kuleuven.be

Frank Piessens
Dept. CS, K.U.Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
frank@cs.kuleuven.be

Wolfram Schulte
Microsoft Research
One Microsoft Way
Redmond, WA, USA
schulte@microsoft.com

ABSTRACT

We propose a formalism for the full functional specification of *enumerator methods*, which are C# methods that return objects of type $IEnumerable<T>$ or $IEnumerator<T>$. We further propose a sound modular automatic verification approach for enumerator methods implemented using C# 2.0's *iterator blocks* (i.e., using **yield return** and **yield break** statements), and for client code that uses *for-each* loops. We require *for-each* loops to be annotated with special *for-each* loop invariants.

The approach prevents interference between iterator implementations and client code. Specifically, an enumerator method may read a field $o.f$ only if o is reflexively-transitively owned by an object listed in the enumerator method's **reads** clause, and the body of a *for-each* loop may not modify these objects. For example, we verify that a *for-each* loop iterating over an $ArrayList$ does not modify the $ArrayList$. Note that one may break out of a *for-each* loop at any time to perform modifications before the iteration is complete. This in effect invalidates the iteration since the *for-each* loop cannot be resumed.

We support specification of non-deterministic enumerations, infinite enumerations, and enumerations that terminate with a checked exception, but not enumerations with side-effects. We support verification of an enumerator method only if it is implemented using **yield** statements, and verification of client code only if it performs a *for-each* loop on an enumerator method call. That is, the present approach does not support explicit creation or manipulation of $IEnumerator<T>$ objects.

Our approach integrates easily with our concurrency approach (presented at ICFEM06), since both are based on read/write sets.

This approach was initially presented at FTfJP05. Please refer to this paper for related work, references, and a soundness proof.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/program verification

*Bart Jacobs is a Research Assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.-Vlaanderen).

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006), November 10–11, 2006, Portland, Oregon, USA.

General Terms

Design, verification

Keywords

Formal specification, iterators, Boogie, verification

1. PROOF RULES

1.1 Spec#

We present our specification and verification method for the Iterator pattern in the context of the Spec# programming system, an extension of C# with preconditions, postconditions, non-null types, checked exceptions, loop invariants, object invariants, and other reliability features, and accompanied by a compiler that emits runtime checks and a static program verifier backed by an automatic theorem prover.

We hope to add support for our approach to the Spec# program verifier in the future.

The program verifier works by translating the Spec# source code into a guarded command program, which is then further translated into verification conditions that are passed to the theorem prover. The following guarded commands are relevant to this presentation:

- An **assert** C; statement causes an error to be reported if the condition C cannot be shown to always hold.
- An **assume** C; statement causes the verifier to consider only those program executions which either do not reach this statement or satisfy the condition C.
- A **havoc** x; statement assigns an arbitrary value to the variable x.

1.2 Specification of enumerator methods

In our formalism, methods are categorized as regular methods or *enumerator methods*. Enumerator methods must have a return type of $IEnumerable<T>$ or $IEnumerator<T>$, for some T, and methods that have such return types are categorized as enumerator methods by default.

The syntax of an enumerator method's contract differs from that of a regular method. In addition to **requires** and **ensures** clauses, an enumerator method may provide one or more **invariant** clauses, which declare the method's *enumeration invariants*. Both the enumeration invariants and the **ensures** clauses may mention the keyword **values**, which denotes the sequence of elements yielded so far at a given point during the enumeration. The **values** keyword is of type $Seq<T>$, whose interface is given in Fig. 2. An enumeration invariant must hold at each point during an enumeration.

```
IEnumerable<int> FromTo(int b, int e)
  requires b ≤ e + 1;
  invariant values.Count ≤ e + 1 − b;
  invariant forall{int i in (0 : values.Count);
    values[i] == b + i};
  ensures values.Count == e + 1 − b;
{
  for (int x = b; x ≤ e; x++)
    invariant values.Count == x − b;
  { yield return x; }
}
```

Figure 1: Method *FromTo*

```
public struct Seq<T> {
  public int Count { get; }
  public invariant 0 ≤ this.Count;
  public T this[int index]
    requires 0 ≤ index ∧ index < this.Count;
  { get; }
  public Seq();
    ensures this.Count == 0;
  public void Add(T value);
    ensures this.Count == old(this).Count + 1;
    ensures forall{int i in (0 : old(this).Count);
      this[i] == old(this)[i]};
    ensures this[old(this).Count] = value;
}
```

Figure 2: The *Seq<T>* **type**

Fig. 1 shows an example of a method specified in our formalism.[1]

1.3 Verification of iterator methods

We verify an enumerator method that is implemented as an iterator method (i.e., a method whose body is a C# 2.0 iterator block) by translating it into a guarded command program. Consider the following method:

```
IEnumerable<T> M(p⃗)
requires P; invariant I; ensures Q;
{ B }
```

It gets translated into the following:

```
assume P;
Seq<T> values = new Seq<T>();
assert I; ⟦B⟧ assert Q;
```

where

$$\llbracket \textbf{yield return } v; \rrbracket \equiv \ values.Add(v); \ \textbf{assert } I;$$
$$\llbracket \textbf{yield break}; \rrbracket \equiv \ \textbf{assert } Q; \ \textbf{assume false};$$

That is, we verify that the enumeration invariants hold for the empty sequence, as well as after each **yield return** operation. Also, we check the postcondition at each **yield break** operation.

As a convenience, we insert I as a loop invariant into each loop in B.[2]

Applied to our *FromTo* example from Fig. 1, this yields the program of Fig. 3.

[1]We propose a more concise syntax for simple cases like this one below.

[2]These are "free of charge", i.e. they provide assumptions but do not incur proof obligations, since they are guaranteed by the **assert** statements inserted at the **yield return** statements.

```
assume b ≤ e + 1;
Seq<int> values = new Seq<int>();
assert values.Count ≤ e + 1 − b;
assert forall{int i in (0 : values.Count);
  values[i] == b + i};
for (int x = b; x ≤ e; x++)
  invariant values.Count ≤ e + 1 − b;
  invariant forall{int i in (0 : values.Count);
    values[i] == b + i};
  invariant values.Count == x − b;
{
  values.Add(x);
  assert values.Count ≤ e + 1 − b;
  assert forall{int i in (0 : values.Count);
    values[i] == b + i};
}
assert values.Count == e + 1 − b;
```

Figure 3: Guarded command program generated as part of the verification of method *FromTo* **of Fig. 1.**

1.4 Verification of *for-each* loops

Our formalism supports proving rich properties of *for-each* loops by allowing their loop invariants to mention the keyword **values**, analogously with our approach to method contracts for enumerator methods. Here, too, the keyword is of type *Seq<T>*, where T is the element type of the enumeration, and represents the sequence of elements enumerated so far.

Here is an example of a client of our *FromTo* enumerator method:

```
int sum = 0;
foreach (int x in FromTo(1, 2))
  invariant sum == SeqTools.Sum(values);
{ sum += x; }
assert sum == 3;
```

Now, consider a general *for-each* loop that uses a call of the general enumerator method M declared above as its enumerable expression:

$$\textbf{foreach } (T\ x \textbf{ in } M(\vec{a})) \textbf{ invariant } J; \ \{ \ S \ \}$$

To verify this *for-each* loop, we translate it into the following **for** loop:

```
assert P[a⃗/p⃗]; Seq<T> values = new Seq<T>();
for (;;)
  invariant I[a⃗/p⃗]; invariant J;
{
  bool b; havoc b; if (¬b) break; T x; havoc x;
  values.Add(x);
  assume I[a⃗/p⃗];
  S
}
assume Q[a⃗/p⃗];
```

This means that for our example client, the program of Fig. 4 needs to be verified.

1.5 Exceptions

Our formalism supports the specification of enumerator methods that may throw checked exceptions, and the verification of the iterator methods that implement these. Enumerator methods may provide exceptional ensures clauses, and these may mention keyword **values**. An example is in Fig. 5.

```
int sum = 0;
assert 1 ≤ 2 + 1; Seq<int> values = new Seq<int>();
for (; ; )
  invariant values.Count ≤ 2 + 1 − 1;
  invariant forall{int i in (0 : values.Count);
    values[i] == 1 + i};
  invariant sum == SeqTools.Sum(values);
{
  bool b; havoc b; if (¬b) break;
  T x; havoc x; values.Add(x);
  assume values.Count ≤ 2 + 1 − 1;
  assume forall{int i in (0 : values.Count);
    values[i] == 1 + i};
  sum += x;
}
assume values.Count == 2 + 1 − 1; assert sum == 3;
```

Figure 4: Guarded command program generated as part of the verification of the example client

```
class OneElementException : CheckedException {}
class ThreeElementsException : CheckedException {}

IEnumerable<int> Baz()
  ensures values.Count == 2;
  throws OneElementException ensures values.Count == 1;
  throws ThreeElementException ensures values.Count == 3;

int n = 0;
try {
  foreach (int x in Baz()) invariant n == values.Count;
  { n++; }
  assert n == 2;
} catch (OneElementException) { assert n == 1;
} catch (ThreeElementException) { assert n == 3; }
```

Figure 5: Enumerator methods that throw checked exceptions

```
⟦x = new C;⟧ ≡
  x = new C;
  tid.W[x] = true;
  tid.R[x] = 0;
  x.inv = false;

⟦x = o.f;⟧ ≡
  assert tid.R'[o];
  x = o.f;

⟦o.f = v;⟧ ≡
  assert tid.W'[o];
  assert ¬o.inv;
  o.f = v;

⟦read (o) S⟧ ≡
  assert tid.R'[o];
  assert o.inv;
  tid.R[o]++;
  foreach (p ∈ rep(o))
    tid.R[p]++;
  ⟦S⟧
  foreach (p ∈ rep(o))
    tid.R[p]−−;
  tid.R[o]−−;

⟦pack o;⟧ ≡
  assert tid.W'[o];
  assert ¬o.inv;
  foreach (p ∈ rep(o)) {
    assert tid.W'[o];
    assert o.inv;
  }
  foreach (p ∈ rep(o))
    tid.W[p] = false;
  o.inv = true;

⟦unpack o;⟧ ≡
  assert tid.W'[o];
  assert o.inv;
  foreach (p ∈ rep(o))
    tid.W[p] = true;
  o.inv = false;

⟦par (S1, S2);⟧ ≡
  let R = tid_par.R;
  par (⟦S1⟧, {
    tid_S2.R = R;
    ⟦S2⟧
  });
```

Figure 6: The programming methodology

1.6 Simplified alternative enumerator method contract syntax

The general syntax presented above offers the flexibility of nondeterministic specifications; that is, it allows underspecification. Also, it allows a non-constructive description, as well as exceptional termination. However, often this flexibility is not needed, and for these cases we provide a simpler syntax, as follows:

```
IEnumerable<T> M(p⃗)
  requires P;
  returns {int i in (0:C); E};
```

For verification purposes, we expand this into the general syntax as follows:

```
IEnumerable<T> M(p⃗)
  requires P;
  invariant values.Count ≤ C;
  invariant forall{int i in (0:values.Count);
    values[i] == E};
  ensures values.Count == C;
```

2. AVOIDING INTERFERENCE

As is apparent from the explanations above, the implementation and the client of an enumerator method are verified as if they executed separately. However, they in fact execute in an interleaved fashion. To ensure soundness, our method prevents each party from observing side-effects of the execution of the other party.

Specifically, an enumerator method may not write fields of any pre-existing objects. Also, an enumerator method may declare in its contract a *read set*, using a **reads** clause, and it may only read fields of those pre-existing objects that are in its read set (or that are owned by such objects). Conversely, during the enumeration, the client (i.e. the body of the *for-each* loop) may not write fields of these objects.

Here's an example of an Iterator pattern involving objects:

```
IEnumerable<int> EnumArray(int[]! a)
  reads a; returns {int i in (0:a.Length); a[i]};
{
  for (int i = 0; i < a.Length; i++)
    invariant values.Count == i;
  { yield return a[i]; }
}

int[] xs = {1, 2}; int sum = 0;
foreach (int x in EnumArray(xs))
  invariant sum == SeqTools.Sum(values);
{ sum += x; }
assert sum == 3;
```

The *EnumArray* method may read only the array, and the body of the **foreach** loop may not modify it. The exclamation mark indicates that the argument for parameter a must not be null.

To statically and modularly verify the restrictions outlined above, our method for avoiding interference between the client and the implementation of an enumerator method requires that the program be written according to a programming methodology that is an extension of the Spec# object invariants methodology with support for read-only access. First, we briefly review the relevant aspects of the Spec# methodology. Then we present our extended version.

2.1 Spec# Methodology

In order to allow the object invariant for an object o to depend on objects other than o, Spec# introduces an ownership system; the

object invariant for o may depend on o and on any object transitively owned by o. A program assigns ownership of an object p to o by writing p into a field of o declared **rep** while o is in the *unpacked* state, and then *packing* o, which brings it into the *packed* state. The packed or unpacked state of an object is conceptually indicated by the value of a boolean field $o.inv$, which is **true** if and only if o is in the packed state.

Packing object o succeeds only if object p and the other objects pointed to by o's **rep** fields are themselves already packed. Once o is packed, its owned objects may not be unpacked. *Unpacking* o again releases ownership of p and allows p to become owned by another object, or to become unpacked itself.

2.2 Programming Methodology

To understand the approach, it is useful to think of both parties in an enumeration as executing in separate threads. That is, the execution of a *for-each* statement starts the enumerator method in a new thread, executes the body of the *for-each* loop some number of times in the original thread, and then waits for the enumerator thread to finish. (We ignore for now the communication between both threads implied by the yielding of values, and the exact number of times the *for-each* loop is executed.) Note that we use the notion of threads as a reasoning tool only; we are not proposing implementing iterators using threads.

In our proposed system, each such thread t has a *write set* $t.W$ and a *read bag* $t.R$, both containing object references. The write set of a thread t contains those object that were created by t and that are not currently committed to (i.e. owned by) some other object. The read bag of t contains an object o if t currently has read-only access to o. The read bag is not a set, for technical reasons which will become clear later.

From $t.W$ and $t.R$, we derive the *effective write set* $t.W' = t.W - t.R$ and the *effective read set* $t.R' = t.W + t.R$. A thread t may read fields of any object in $t.R'$, and it may write fields of any object in $t.W'$, provided the object is unpacked.

The *for-each* statement may conceptually be thought of as being implemented in terms of a command **par** $(B_1,\ B_2)$; for parallel execution of two blocks B_1 and B_2. Execution of the **par** statement is complete only when execution of both blocks is finished. Suppose the **par** statement is being executed by a thread t_1. B_1 is executed in t_1, whereas B_2 is executed in a new thread, say t_2. The initial write set $t_2.W$ of t_2 is the empty set, and the initial read bag is equal to that of t_1.

The proposed methodology is formally defined in Fig. 6, where **tid** denotes the current thread. The last rule translates a parallel execution statement by inserting an assignment that initializes the read bag of the newly created thread $\mathbf{tid}_{S_2}$ with the read bag of the creating thread $\mathbf{tid}_{\mathbf{par}}$. The write set of the new thread remains initially empty. We use the following auxiliary definitions:

$$t.W'[o] \stackrel{\text{def}}{=} t.W[o] \wedge t.R[o] = 0 \quad t.R'[o] \stackrel{\text{def}}{=} t.W[o] \vee t.R[o] > 0$$
$$\text{rep}(o) \stackrel{\text{def}}{=} \{o.f \mid f \text{ is a } \mathbf{rep} \text{ field of } o \text{ and } o.f \neq \text{null}\}$$

The new **read** statement serves two purposes. Firstly, it allows a thread to take an object to which it has write access and make it read-only for the duration of the **read** statement, which enables it to be shared with newly created threads. Secondly, it allows a thread that has read access to an object o to gain access to o's owned objects. That is, it replaces the **unpack** and **pack** operations if only read access is required. Note: in contrast to the **unpack** and **pack** pair, **read** blocks are re-entrant; that is, it is allowed to nest multiple read block executions on the same object. This is useful e.g. when writing recursive methods. This is also the reason why we need a read *bag* instead of a read *set*.

```
assert P[a⃗/p⃗];
read (R) {
  par ({
    Seq<T> values = new Seq<T>();
    for (;;) invariant I[a⃗/p⃗]; invariant J;
    {
      bool b; havoc b; if (¬b) break;
      T x; havoc x; values.Add(x); assume I[a⃗/p⃗];
      S
    }
    assume Q[a⃗/p⃗];
  }, {
    Seq<T> values = new Seq<T>();
    assert I; [[B]] assert Q;
  });
}
```

Figure 7: Translation of the general *for-each* loop for the purpose of applying the non-interference methodology

```
int[] xs = {1, 2}; int sum = 0;
read (xs) {
  par ({
    Seq<T> values = new Seq<T>();
    for (;;)
      invariant values.Count ≤ xs.Length;
      invariant forall{int i in (0:values.Count);
        values[i] == xs[i]};
      invariant sum == SeqTools.Sum(values);
    {
      bool b; havoc b; if (¬b) break; T x; havoc x;
      values.Add(x);
      assume values.Count ≤ xs.Length;
      assume forall{int i in (0:values.Count);
        values[i] == xs[i]};
      sum += x;
    }
    assume values.Count == xs.Length;
  }, {
    Seq<T> values = new Seq<T>();
    assert values.Count ≤ xs.Length;
    assert forall{int i in (0:values.Count); values[i] == xs[i]};
    for (int i = 0; i < xs.Length; i++)
      invariant values.Count ≤ xs.Length;
      invariant forall{int i in (0:values.Count);
        values[i] == xs[i]};
      invariant values.Count == i;
    {
      values.Add(xs[i]); assert values.Count ≤ xs.Length;
      assert forall{int i in (0:values.Count);
        values[i] == xs[i]};
    }
    assert values.Count == xs.Length;
  });
}
assert sum == 3;
```

Figure 8: Translation of the array example for the purpose of applying the non-interference methodology

Consider the general *for-each* statement shown in Section 1.4. For the purpose of applying the proposed methodology, it is equivalent with the program in Fig. 7, assuming that method M has a **reads** R; clause. For the array example above, this yields the program in Fig. 8.

Specifying Java Iterators with JML and Esc/Java2

David R. Cok
Eastman Kodak Company
1999 Lake Avenue Rochester, NY 14650, USA
david.cok@kodak.com

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Program Verification

General Terms

Design, verification

Keywords

JML, ESC/Java2, static analysis, specification, verification

1. INTRODUCTION

The 2006 SAVCBS Workshop[1] has posed a Challenge Problem on the topic of specifying iterators. This note provides a specification in the Java Modeling Language (JML) [1, 2] for the Java interfaces *Iterator* and *Iterable* that captures the interactions between these two interfaces. An example program that uses these interfaces is checked using Esc/Java2 [3, 4, 5], demonstrating by example that the Esc/Java2 tool checks that the interfaces are used only as required by the specifications. The concluding section contains some observations on the limitations of JML for this specification task.

2. THE PROBLEM

The Challenge Problem[2] asks for a specification of the *Iterator* interface as provided in the Java programming language or its equivalent in another language. An Iterator provides an abstract mechanism for sequentially retrieving the elements of an object for which such an operation is appropriate, that is, of an *Iterable* object. There are two aspects of the behavior of an iterator.

The first is the mechanism for keeping track of which objects of the iterable collection have already been returned by the iterator and which are yet to be returned. This mechanism is dependent on the particular kind of iterable object (e.g., set, array, list, infinite sequence) and its implementation. In fact there is actually quite little one can specify about this aspect of an iterator's behavior within the *Iterator* interface itself. Space limitations preclude discussing the specification of that mechanism here.

The more interesting aspect of an iterator's behavior is the interaction among multiple iterators and with the iterable object, particularly with respect to modifications of the iterable object. In particular, the solution presented here provides specifications for three conditions: (a) an iterator may remove the object of the iterable at the current position of the iterator, but it may not remove it more than once; (b) if an iterable object is modified by its own methods, then all subsequent behavior of iterators referring to that iterable is undefined; (c) if an iterable object is modified by an iterator, then all subsequent behavior of any other iterator referring to that iterable is undefined.

Here we consider only sequential programs and provide a solution for Java 1.5 using JML. The reader is presumed to be familiar with Java and its iterator classes as well as with JML. In actuality, JML is implemented only for Java 1.4. However, the only use of Java 1.5 features here is the parameterization of the interfaces by the element type *E*, and that does not affect the discussion below. On the other hand, Java 1.4 does not have the equivalent of the *Iterable* interface, a point that is discussed further below.

3. THE JML SPECIFICATION

The proposed specifications of these two interfaces are shown in Figs. 1 and 2. A partial specification of the *Collection* interface is also shown.

The solution has the following elements:

- Because the conditions above require certain behavior *subsequent to* other behavior, a concept of time (or, more precisely, of an ordered sequence of events) is maintained in the specification by nondecreasing integer values.
- An *Iterator* maintains a reference to the *Iterable* whose contents it returns, contained in the model field `iterable`. This field is initialized at construction time (in the method `Iterable.iterator()`) and does not change thereafter, as indicated by the constraint clause.
- An *Iterator* uses the model field `iteratorTime` to keep track of when it was created or last used to modifiy the iterable. An initial value is specified by the

[1] http://www.cs.iastate.edu/~leavens/SAVCBS/2006/index.shtml.
[2] http://www.cs.iastate.edu/~leavens/SAVCBS/2006/challenge.shtml.

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006), November 10-11, 2006, Portland, Oregon, USA.

```
package java.util;
public interface Iterator<E> {
  //@ public instance model Iterable iterable;
  //@ public instance model int iteratorTime;
  //@ public instance model boolean removeOK;

  //@ initially !removeOK;
  //@ public invariant iterable != null;
  //@ public constraint iterable == \old(iterable);

  // This returns false if the parent Iterable has
  // been modified by means other than this Iterator.
  /*@ public normal_behavior
    @  ensures \result ==
    @      (iteratorTime > iterable.lastModifiedTime);
    @ public pure model boolean isValid() {
    @  return iteratorTime > iterable.lastModifiedTime;
    @ }
    @*/

  //@ public normal_behavior
  //@  requires isValid();
  /@ pure @*/ public boolean hasNext();

  //@ public normal_behavior
  //@  requires isValid() && hasNext();
  //@  assignable removeOK;
  //@  ensures removeOK;
  //@ also public exceptional_behavior
  //@  requires isValid() && !hasNext();
  //@  signals_only NoSuchElementException;
  public E next();

  /*@ public behavior
    @  requires isValid() && removeOK;
    @  assignable removeOK, iteratorTime;
    @  assignable iterable.maxIteratorTime;
    @  assignable iterable.lastModifiedTime;
    @  ensures !removeOK;
    @  ensures iterable.lastModifiedTime >
    @              \old(iterable.maxIteratorTime);
    @  ensures isValid();
    @  ensures iteratorTime <= iterable.maxIteratorTime;
    @ also public exceptional_behavior
    @  requires isValid() && !removeOK;
    @  signals_only IllegalStateException;
    @*/
  public void remove();
}
```

Figure 1: The specification of the Iterator interface.

method `Iterable.iterator()` and it is modified only by `Iterator.remove()`.

- The model field `Iterator.removeOK` indicates whether it is permissible to call the method `Iterator.remove()`. The field is initially false and is also set false upon any call of `remove`; it is set true on a call of `next`. Thus informal requirement (a) above is satisfied.

```
package java.lang;
public interface Iterable<E> {
  //@ public instance model int lastModifiedTime;
  //@ public instance model int maxIteratorTime;
  //@ initially maxIteratorTime == -1;
  //@ initially lastModifiedTime == 0;
  /*@ constraint lastModifiedTime >=
                          \old(lastModifiedTime); @*/

  //@ public normal_behavior
  //@   assignable maxIteratorTime;
  //@   ensures \result != null;
  //@   ensures \fresh(\result);
  //@   ensures \result.iterable == this;
  //@   ensures \result.isValid();
  //@   ensures maxIteratorTime >= \result.iteratorTime;
  public Iterator<E> iterator();
}

package java.util;
public interface Collection<E> extends Iterable<E> {
  // Something like the following specification
  // case must be present for any method that
  // modifies the Iterable object.
  //@ public normal_behavior
  //@   assignable lastModifiedTime;
  //@   ensures lastModifiedTime > maxIteratorTime;
  public void clear();
}
```

Figure 2: The specification of the Iterable interface and a partial specification of Collection.

- Requirements (b) and (c) above need the distinction between an Iterator's behavior being defined and not defined. This distinction is provided by the pure model method `Iterator.isValid()`. If the method returns true, the behavior is defined. The method is implemented to return true if the iterator's `iteratorTime` is larger than the corresponding iterable's `lastModifiedTime`.

- An *Iterable* maintains the "time" of its last modification in the field `lastModifiedTime`. If the *Iterable* is modified, as shown by the method `Collection.clear`, the value of `lastModifiedTime` is increased to be larger than the `iteratorTime` of any of its associated *Iterator*s. For convenience, `Iterable.maxIteratorTime` holds a value at least as large as any associated *Iterator*'s `iteratorTime`. This satisfies requirement (b) above. Note that any method in any subtype of `Iterable` that modifies the collection of elements within the Iterable (e.g., add, remove, clear) must require a specification case like that shown for `Collection.clear`.

- Requirement (c) is satisfied as follows. The specification of `Iterator.remove` requires that when called on an object *iter* (and for normal termination), the corresponding iterable's `lastModifiedTime` is increased to make all other iterators invalid, and the `iteratorTime` of *iter* itself also is increased so that *iter* is still valid.

4. STATICALLY CHECKING PROGRAMS USING ESC/JAVA2

The *Iterator* and *Iterable* interfaces do not have implementations that can be checked against specifications. However, we can check programs that use those interfaces. To do so with JML and Esc/Java2, however, we must recast the above solution in Java 1.4. For this exercise we fold the specifications from *Iterable* into Java 1.4's *Collection* interface. Then we attempt to check a number of combinations of uses of these methods, as shown in Figs. 3 and 4. Esc/Java2[3] successfully finds the incorrect uses of these methods and has no false reports on legal sequences of method calls. The problems in generating and checking the specifications were all in specification errors (not in Esc/Java2). For example, in method `m6`, if Line A is omitted, allowing aliasing between the two arguments (a common error), then Line B cannot be established: iterator `ii` will not be valid if `c==cc`.

5. OBSERVATIONS

The combination of JML and Esc/Java2 successfully specifies the Iterator example and checks uses of the interfaces in test programs. However, this exercise prompts a number of observations about the current state of JML.

5.1 Java 1.4 vs. Java 1.5

This style of solution will not work well in Java 1.4 because there is no abstract *Iterable* object. For the static checking above, we utilized the *Collection* interface as the generic iterable. However, not all iterators extend the *Collection* interface. Thus in Java 1.4 an *Iterator* can only refer to its associated object as a generic *Object*, and there is no place to put the declarations of the model fields defined above. An alternative, but messy, design is to declare a new associated `IterableData` class containing the model fields declared above in *Iterable* and used as *Iterable* is above; then we associate an `IterableData` object with each iterable `Object` by maintaining a Map from objects that would be Iterables to associated instances of `IterableData`.

5.2 Ghost field vs. Model field vs. Model method

In the specification above, various pieces of specification information are held in model fields. These might also be declared as ghost fields or model methods. Each of these choices has its disadvantages.

- Ghost fields. Iterators and Iterables are interfaces, not classes. Furthermore, they are defined in the Java library and not in user-written code. Ghost fields must be modified by JML `set` statements within the implementation of a method. In this situation, for these interfaces there is no place to put those `set` statements. This is not a problem for static checking, but runtime checking (such as with the jmlrac[2] tool) would fail to work correctly if ghost fields were used.

- Model fields. The intended use of a model field is as a means to hold an abstract representation of the state of an object; in a concrete class each model field would be provided a representation. In this case, a field such as `removeOK` does abstract part of the state of the Iterator, but that abstraction is not necessarily a representation of any concrete fields of an implementation. A typical way to provide such a concrete representation is by means of some ghost fields that essentially duplicate the model fields. The model fields work well for static checking without ghost fields and without representations. However, runtime checking would require the model fields to have some concrete representation.

- Model methods. Model methods are an alternate way of providing the functionality of a model field.[4] For example, instead of the field `removeOK`, we could have a pure, argument-less model method `removeOK()` without any implementation given. The specification of its

[3]The experiments were performed using the version in CVS HEAD as of 1 September 2006, but only using the specifications given here, not the library of system specifications provided by Esc/Java2.

[4]Model fields also have implications for data groups, which model methods do not have.

```
import java.util.Collection;
import java.util.Iterator;
public class Test {
  public void m1(/*@ non_null @*/Collection c) {
    Iterator i = c.iterator();
    i.remove();  // should FAIL
  }

  //@ signals (java.util.NoSuchElementException);
  //@ signals_only RuntimeException;
  public void m2(/*@ non_null @*/Collection c) {
    Iterator i = c.iterator();
    //@ assume i.hasNext();
    i.next();
    i.remove();  // OK
  }

  public void m3(/*@ non_null @*/Collection c) {
    Iterator i = c.iterator();
    //@ assume i.hasNext();
    i.next();
    i.remove();
    i.remove();  // should FAIL
  }

  public void m4a(/*@ non_null @*/Collection c) {
    Iterator i = c.iterator();
    //@ assert i.iteratorTime > c.lastModifiedTime;
    //@ assert i.iterable == c;
    //@ assert i.isValid();
  }

  public void m4(/*@ non_null @*/Collection c) {
    Iterator i = c.iterator();
    //@ assert i.isValid();
    c.clear();
    //@ assert !i.isValid();
  }
}
```

Figure 3: A set of test methods (in Java 1.4).

```
import java.util.Collection;
import java.util.Iterator;
public class Test2 {
  public void m5(/*@ non_null @*/Collection c) {
    Iterator i = c.iterator();
    Iterator ii = c.iterator();
    //@ assert i.isValid();
    //@ assert ii.isValid();
    //@ assume i.hasNext();
    i.next();
    i.remove();
    //@ assert i.isValid();
    //@ assert !ii.isValid();
  }

  //@ requires c != cc;     // Line A
  public void m6(/*@ non_null @*/Collection c,
                 /*@ non_null @*/Collection cc) {
    Iterator i = c.iterator();
    Iterator ii = cc.iterator();
    //@ assert i.isValid();
    //@ assert ii.isValid();
    //@ assume i.hasNext();
    i.next();
    i.remove();
    //@ assert i.isValid();
    //@ assert ii.isValid();  // Line B
  }

  public void m7(/*@ non_null @*/Collection c) {
    Iterator i = c.iterator();
    //@ assume i.hasNext();
    c.clear();
    i.hasNext();//FAILS - precondition isValid()
  }             // is not satisfied
}
```

Figure 4: Additional test methods (in Java 1.4).

result and its use on other specifications would mimic the specification and use of the model field. Static checking with such model methods is just as easy (and as hard) as when using model fields. Runtime checking has the same problems as with model fields: we need an implementation in terms of concrete or ghost fields.

One enhancement of JML that would help the above issues with runtime checking would be to provide syntax in which updates to ghost fields could be specified and compiled by a runtime checker even for methods for which the runtime checker did not compile the Java implementation of the method itself.

5.3 Specifying mutating methods

As stated earlier, the specification described here requires that all methods (of any subtype) that modify an *Iterable* object must specify that the values of `lastModifiedTime` and `maxIteratorTime` are appropriately changed. This requirement is easily forgotten. Any method that calls `remove()` will encounter those requirements in that method's specification, but other methods, such as `add`, will not. Aside from the specifications of overridden methods, there is no way within JML to require that all methods with certain properties have certain specifications without individually annotating the methods to indicate the desired property.

5.4 Specifying sequences of calls

The main limitation of JML in this context is that it provides no means to write specifications about sequences of method calls. The specification above essentially encodes two state machines: a simple one using `removeOK` and a more complicated one involving the other model fields. These machines are used to specify implicitly the behavior of sequences of method calls. However, there is no way in JML to write a specification requirement about this behavior that can be checked by some reasoning engine; in Section 3 we were only able to argue the correctness of the specifications informally. The best we can do in current JML is to write example programs and then check using a static checker that those examples are properly handled; that process, like runtime testing, does not ensure that all possible examples will behave correctly. Another common restriction is when a class has an initialization method that must be called before any other method of the class is called.

To express these conditions, JML would need to have syntax that could encode, for example, the following requirements: that two calls of `Iterator.remove` with no intervening call of `Iterator.next` must result in particular behavior; that a call of a class method not preceded by a call of the class's init method results in an exception being thrown; that a call of a particular method will render calls of another set of methods undefined. These all would require syntax enabling the expression of combinations of parameterized sequences of method calls, with options such as are found in regular expressions. In addition, we would need translation to verification conditions in an appropriate logic and suitable for a logical prover.

6. REFERENCES

[1] Many references to papers on JML can be found on the JML project website, http://www.cs.iastate.edu/~leavens/JML/papers.shtml.

[2] L. Burdy, et al. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.

[3] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. Technical report, University of Nijmegen, 2004. NIII Technical Report NIII-R0413.

[4] D. R. Cok and J. Kiniry. ESC/Java2 : Uniting ESC/Java and JML. Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system. *Lecture Notes in Computer Science*, 3362:108–128, Jan. 2005.

[5] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37, 5 of *SIGPLAN*, pages 234–245, New York, June 2002. ACM Press.

SAVCBS 2006 Challenge: Specification of Iterators

Bruce W. Weide
Department of Computer Science and Engineering
The Ohio State University
+1-614-292-1517
weide.1@osu.edu

ABSTRACT
A method for formal specification of iterators, which can be used to verify both clients and implementations, is illustrated with a *Set* abstraction as the underlying collection.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/program verification.

General Terms
Design, Verification.

Keywords
Formal specification, iterators, Resolve, verification.

1. INTRODUCTION
This short paper is a response to the SAVCBS 2006 Challenge Problem: "We invite participants to illustrate their specification and verification techniques on the problem of specifying the behavior of iterators and clients that use them". Our solution illustrates, and slightly improves on (i.e., simplifies) the iterator design and specification techniques we previously published in [8], using a *Set* abstraction with an active iterator that does not permit interleaved client modification of the elements of a *Set*. The conclusion of [8] is:

> Previously published iterator designs are unsatisfactory along several dimensions. The iterator design developed incrementally [in this paper] addresses the deficiencies of prior approaches in the following specific ways:
>
> - It is designed to support efficient implementations: neither the implementer nor the client needs to copy the data structure representing the Collection, or any of the individual Items in it.
> - Its abstract behavior (including the non-interference property) is formally specified.
> - Its implementations and clients can be verified independently, i.e., modularly in the sense of [3].
> - It can be specified as a schema for an independent generic concept that defines an iterator abstraction for arbitrary Collections, so all iterator abstractions in a system share a common interface model.

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006). November 10-11, 2006, Portland, Oregon, USA.

"Non-interference" means [8] that it "should not be permissible for a (correct) client program to iterate over a collection while interleaved operations on that collection might be changing it." Extensions and variants discussed in [8] also address issues involved when iteration over a collection modifies the collection or the items in it, when iteration might encounter items in different orders, and when iteration terminates early. None of these latter issues is explicitly discussed here.

The answers to the specific questions posed in the Challenge Problem are as follows:

- The solution is intended for use with a sequential programming language, though concurrency-hardening does not seem to pose any special problems.
- The level of annotation required (both in the contract specification and in a client program) is full behavioral specification—but no more than what is necessary and sufficient to modularly verify client correctness. In principle, it might be possible to specify or prove weaker properties with less annotation, but we see no reason to do so; this solution seems fully manageable in terms of specification and verification complexity.
- The solution is based on using a language, such as Resolve [2] or the disciplined use of C++ that we call Resolve/C++ [6], that has value semantics, with no visible references and hence no visible aliasing. We emphasize that this does not imply inefficiency compared to languages that make references manifest to the programmer. This is one of the main points of [8]—but one that we do not elaborate here except to claim the result that our design permits optimally efficient iterator implementations in the big-O sense, so introducing the reasoning complications of reference semantics would not result in efficiency improvements.
- Fully automated verification of client programs using our iterator design and specification approach is certainly possible in principle. We know how to generate mechanically the verification conditions for Resolve programs. However, there is no evidence yet to suggest that a system like Hoare's "verifying compiler", that would produce fully automatic proofs of these verification conditions, is just over the horizon. The verification conditions that arise from using our iterators are not particular difficult for humans to discharge. They do seem generally near or beyond what existing theorem provers can handle without human advice.

We regrettably have no fundamentally new observations about specifying iterators since the 1994 paper [8]. We hope the attendees at the workshop provide some additional food for thought.

2. EXAMPLE: A SET COMPONENT

Understanding our iterator design and specification requires understanding the specification of the collection over which iteration is to be done. The iterator design technique we proposed in [8] is a schema that can be used with arbitrary collections and is illustrated there with a *Queue* abstraction. Here we use for variety a *Set* abstraction: a parameterized component in which the type *Item* (of a *Set*'s elements) is a template parameter. This is its specification:

```
contract Set_Template (type Item)
  type family Set is modeled by
      finite set of Item
    exemplar s
    initialization ensures
      s = { }
  operation Add (s: Set, x: Item)
    requires
      x is not in s
    ensures
      s = #s union {#x}
  operation Remove (s: Set, x: Item,
       x_copy: Item)
    requires
      x is in s
    ensures
      x = x_copy = #x  and
      s = #s - {x_copy}
  operation Remove_Any (s: Set, x: Item)
    requires
      s /= { }
    ensures
      s = #s - {x}
  operation Is_Member (s: Set, x: Item):
       Boolean
    ensures
      Is_Member = (x is in s)
  operation Size (s: Set): Integer
    ensures
      Size = |s|
end Set_Template
```

The type specification says that a *Set* variable should be considered to have a value that is a finite mathematical set of the parameter type *Item*, and that such a variable's initial value (i.e., upon declaration) is an empty set. The operation *Add* can be used to add an element to a *Set*; the operation *Remove* to remove a particular element whose value is *x*, the removed element being returned in *x_copy*; the operation *Remove_Any* to remove and return an arbitrary implementation-determined element, which is needed for functional completeness of this component [7] in the absence of an accompanying iterator; the function operation *Is_Member* to test set membership; and the function operation *Size* to determine the number of elements. In operation specifications, the prefix "#" on a variable name in a postcondition denotes the parameter's incoming value. Function operations may not change the values of their arguments, so this fact is not specified explicitly.

Two important points must be kept in mind. First, there are no hidden references here. The simplicity of the specification is the result neither of hoping/assuming that there are no aliases (i.e., aliases really aren't possible), nor of syntactic sugar that makes *s*, *x*, *#s*, etc., simply appear to act like mathematical variables rather than the names of objects (i.e., they really do act like mathematical variables). This might surprise readers who are used to similar-looking specifications that deal with references. Second, the only other operations that are available for a *Set* type are those available for any type in Resolve: *Clear*, which resets a variable to an initial value for its type, and a swap operator ":=:" that exchanges the values of two variables of the same type [8]. Assignment ":=" is available only when the right-hand side is a call to a function operation. Readers who are unfamiliar with this style of programming under design-by-contract—value types only, built-in swapping but not variable-to-variable assignment, fully parameterized components, etc.—also might be surprised to learn that it is possible and practical to develop "real" software this way with disciplined use of C++. In fact, experience with a commercial Windows application of over 100,000 SLOC developed in this style shows that real software is not only feasible but also of notably higher quality than software built using traditional methods [4]. In other words, this proposal for how to specify iterators is not based on an unrealistic closed-world assumption; it also is not based on business-as-usual in C++ or Java.

3. CLIENT AND COMPONENT DESIGN

In our design, an iterator's abstract model value includes a string of *Item*s called *past*, which is essentially the *Item*s iterated over so far (in the order they have been processed, with the value of the *Item* currently "out" of the collection at the end of this string), a string of *Item*s called *future*, i.e., those to be iterated over in the future (in the order to be processed, unless the iteration terminates early), and a set called *original*, which is the original value of the *Set* over which iteration is being done. For the complete rationale behind this style of design, see [8]. The formal specification is on the next page (Section 5). Here is a fragment of a typical client program that iterates completely over the *Set s*:

```
Start_Iterator (i, s, x)
loop
  maintaining
    i.past * i.future =
      #i.past * #i.future  and
    <x> is suffix of i.past  and
    i.original = #i.original
  decreasing
    |i.future|
while Length_Of_Future (i) > 0 do
  Get_Next_Item (i, x)
  /* process x, with no net change to it */
end loop
Finish_Iterator (i, s, x)
```

Bracketing calls to *Start_Iterator* and *Finish_Iterator* move the elements of the original *Set s* into the *Set_Iterator i*

and back again. This prevents interference between iteration over *i* and interleaved modifications to *s*: client code that does this might be useless because the changes to *s* are lost when *Finish_Iterator* executes, but that client is not necessarily incorrect. Users of C++ or Java or other similar iterators might find this behavior unsettling. However, possible interference between interleaved modifications to a collection and iteration over it leads to informal and difficult-to-specify warnings in component libraries, such as this one for the *remove* method in the java.util package, interface *Iterator<E>* [5]:

> The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

We did not feel obligated to keep such problematic behavior regardless of its familiarity (in 2006 even more so than in 1994), opting instead for simpler, easily explicable behavior that can be specified without introducing either new specification constructs or extra-specificational warnings.

Start_Iterator records the value of *x* in the string *i.past* at the start of the loop—important to meet the precondition of the first call of *Get_Next_Item*. Parsimony, as well as a Resolve design rule urging "conservation of data", suggest that eventually *x* should have this value again after completion of *Finish_Iterator*. The loop invariant is that the concatenation of *i.past* and *i.future* does not change, that *i.original* does not change, and that the last entry in *i.past* equals *x*. Of course, there is more to the loop invariant to prove the correctness of what the client program is doing while iterating over the elements of *s*, but this is the part required to show that the iterator *i* is being used properly.

Given the stylized nature of the client code, it is easy to imagine special iteration syntax for collections, such as that now available in Java, but with a semantics that matches this common interface model [1] for iterators rather than Java's *Iterable<T>* and *Iterator<E>* interfaces.

4. REFERENCES

[1] Edwards, S.H., "Common Interface Models for Reusable Software", *Intl. J. of Softw. Eng. and Knowledge Eng. 3*, 2 (June 1993), 193-206.

[2] Edwards, S.H., Heym, W.D., Long, T.J., Sitaraman, M., and Weide, B.W., "Specifying Components in RESOLVE," *Software Eng. Notes 19*, 4 (October 1994), 29-39.

[3] Ernst, G.W., Hookway, R.J., and Ogden, W.F., "Modular Verification of Data Abstractions with Shared Realizations", *IEEE TSE 20*, 4 (Apr 1994), 288-207.

[4] Hollingsworth, J.E., Blankenship, L., and Weide, B.W., "Experience Report: Using RESOLVE/C++ for Commercial Software", *Proc. ACM SIGSOFT 8th Intl. Symp. on the Foundations of Softw. Eng.,* ACM Press, 2000, 11-19.

[5] java.util Package, *Interface Iterator <E>, remove Method Detail*, http://java.sun.com/j2se/1.5.0/docs/api/java/util/-Iterator.html, viewed 6 Oct. 2006.

[6] Resolve/C++, http://www.cse.ohio-state.edu/sce/now, viewed 6 Oct. 2006.

[7] Weide, B.W., Ogden, W.F., and Zweben, S.H., "Reusable Software Components", in *Advances in Computers*, vol. 33, M.C.Yovits, ed., Academic Press, 1991, 1-65.

[8] Weide, B.W., Edwards, S.H., Harms, D.E., and Lamb, D.A., "Design and Specification of Iterators Using the Swapping Paradigm," *IEEE TSE 20*, 8 (August 1994), 631-643.

5. APPENDIX: THE SPECIFICATION

```
contract Set_With_Iterator_Template
    enhances Set_Template
  type family Set_Iterator is modeled by (
       past: string of Item,
       future: string of Item,
       original: finite set of Item
      )
    exemplar i
    initialization  ensures
      i = (< >, < >, { })
  operation Start_Iterator (i: Set_Iterator,
       s: Set, x: Item)
    ensures
      there exists f: string of Item
        (elements (f) = #s  and
         |f| = |#s|  and
         i = (<x>, f, #s))  and
      s = { }  and
      x = #x
  operation Finish_Iterator (
       i: Set_Iterator, s: Set, x: Item)
    requires
      <x> is suffix of i.past
    ensures
      is_initial (i)  and
      s = #i.original  and
      <x> is prefix of #i.past
  operation Get_Next_Item (i: Set_Iterator,
       x: Item)
    requires
      i.future /= < >  and
      <x> is suffix of i.past
    ensures
      there exists f: string of Item
        (#i.future = <x> * f  and
         i = (#i.past * <x>, f, #i.original))
  operation Length_Of_Future (
       i: Set_Iterator): Integer
    ensures
      Length_Of_Future = |i.future|
end Set_With_Iterator_Template
```

Iterator Specification with Typestates

Kevin Bierhoff
Institute for Software Research
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
kevin.bierhoff @ cs.cmu.edu

ABSTRACT

Java iterators are notoriously hard to specify. This paper applies a general typestate specification technique that supports several forms of aliasing to the iterator problem. The presented specification conservatively captures iterator protocols and consistency rules. Two limitations of the specification are discussed.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specification—*Languages*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Design, languages, verification

Keywords

Iterator, typestate, specification, aliasing, verification

1. INTRODUCTION

The Java Collection API defines various rules for using iterators. It defines a *protocol* for accessing individual iterators. It also imposes restrictions on modifying iterated collections in order to keep iterators *consistent*. Similar rules are defined for C# enumerators.

Typestates augment the fixed type of a (mutable) object with a variable "condition" that describes the object's abstract state in its lifecycle [7]. A type system like Fugue's [4] that is based on this idea lets the programmer essentially define a state machine for each class. However, Fugue cannot fully capture iterator behavior due to its restrictions regarding aliasing and non-determinism.

This paper presents a specification of Java iterators based on a technique for typestate specifications in the presence of aliasing. The following section introduces some of the key concepts of this technique. The actual iterator specification is presented in section 3. Limitations of the specification are discussed in section 4 and section 5 concludes.

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006), November 10–11, 2006, Portland, Oregon, USA.

2. TYPESTATE SPECIFICATIONS

This section introduces a general technique for typestate specifications to the extent necessary for specifying iterators.

Hierarchical state spaces. We define orthogonal *state dimensions* with sets of mutually exclusive states [2]. The idea is to model separate aspects of object behavior separately. For example, we model Java iterators with three orthogonal dimensions (figure 1). At runtime, an iterator object will be in exactly one of the states from each dimension. The root state alive basically stands for "any state" and can be refined in an arbitrary number of dimensions. Similarly, a dimension stands for "any state" in that dimension.

States and dimensions are explicitly defined as part of an interface. For example, the next dimension depicted in figure 1 could be defined as follows.

```
states available, end refine alive as next
```

Dimensions or states do *not* correspond to implementation fields but information about fields can be *tied* to states, allowing implementation verification (see section 3.5).

Access permissions. Different variables could *alias* the same object and care must be taken to keep the "views" of those aliases onto the object consistent. Our approach is to associate variables with *access permissions* that are guaranteed to remain consistent.

A permission $perm(x, n, A)$ grants different levels of access to a part n of the state space (e.g., a state dimension) to a variable x. Permissions optionally carry additional information A about the exact state inside the part of the state space they cover (omitted otherwise). We use the following access levels.

- unique permissions guarantee that the variable is the only one that has access.
- full permissions guarantee that the variable is the only one that can change state.
- pure permissions give read-only access. There may be other pure permissions and at most one full permission around.

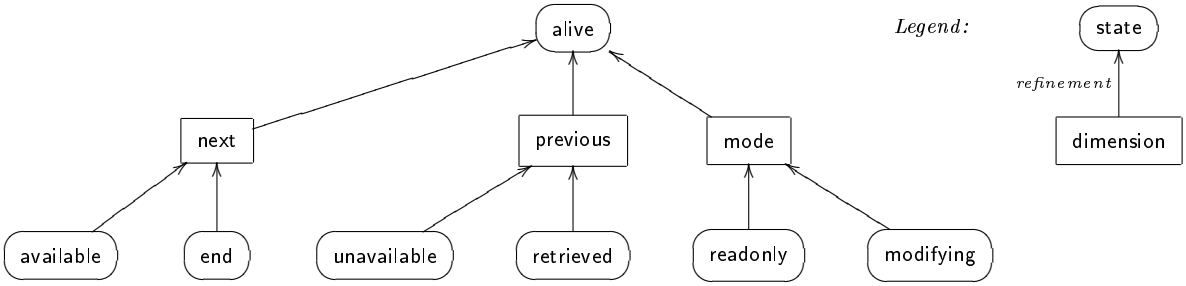


Figure 1: Iterator state space

As an example, pure(*this*, next, available) represents a pure permission on the next dimension of an (`Iterator`) receiver that is currently in the available state.

We use fractions [3] to keep track of splits. This lets us "collect" the full and all pure permissions and regain a unique permission. We omit fractions in specifications if they do not change (these permissions are universally quantified for all fractions).

Method specifications. Methods are specified with the decidable multiplicative-additive fragment of linear logic [5] (MALL). Pre- and post-conditions are separated with a linear implication ($\multimap$) and use conjunction ($\otimes$), internal choice ($\&$), and external choice ($\oplus$). We include quantifiers for receiver *this* and return value *result* to make specifications self-sufficient. In one case we explicitly quantify over fractions.

The following example specifies the `hasNext` method for Java iterators. It requires a pure permission for the receiver's next dimension. The post-condition on the right-hand side of the implication is an external choice between conjunctions. The external choice indicates that the caller has no influence on whether `hasNext` will return `true` or `false`.

$$\forall this : \texttt{Iterator}.\ \exists result : \texttt{boolean}.$$
$$\mathsf{pure}(this, \mathsf{next}) \multimap (\mathsf{pure}(this, \mathsf{next}, \mathsf{end}) \otimes result = \texttt{false}) \oplus (\mathsf{pure}(this, \mathsf{next}, \mathsf{available}) \otimes result = \texttt{true})$$

The expressiveness of linear logic specifications is similar to our earlier work based on union and intersection types [2]. Tracking permissions with linear logic ensures that permissions cannot be duplicated. This is essential for sound static verification in a permission-based approach.

The notation used in this paper is fully explicit for clarity but we envision a more practical surface notation. In particular, quantifiers are implied by standard method signatures (see figure 2). Permissions could by default apply to the receiver or the position of a permission could imply which variable it applies to [2].

3. JAVA ITERATOR SPECIFICATION

This section presents an iterator specification using the techniques introduced in the last section. We state assumptions and goals before specifying iterators and iterables. Finally, we discuss how the specification can be used in verification.

3.1 Assumptions

This specification assumes single-threaded execution. We also assume that a unique permission is needed to modify a collection directly. This can be enforced with an appropriate specification of modifying methods in the `Collection` interface (which extends `Iterable`, specified below).

3.2 Specification Goals

Goals of the presented specification include the following.

- Allow creating an arbitrary number of iterators over collections ("iterables").
- Invalidate iterators before modification of the iterated collection.
- Capture the usage protocol of Java iterators.

3.3 Specifying Iterators

The `Iterator` specification is primarily concerned with capturing the protocol for using iterators (figure 2). In order to capture the expected usage of the `hasNext` and `next` methods we introduce a state dimension next with mutually exclusive states available and end. Calling `hasNext` conceptually performs a dynamic state test on this dimension: a `true` (`false`) return value corresponds with the available (end) state. A subsequent Boolean test on the return value allows a client to deduce the state of the iterator.

Notice that we do not change state with a call to `hasNext`, expressed by only requiring a pure permission for this call. Conversely, a call to `next` can potentially change the state of the next dimension and therefore needs a full permission to the receiver. It requires the next element to be available. Which of the two states in the next dimension will apply after the call is unknown. Thus our specification enforces the characteristic alternation of calls to `hasNext` and `next`.

The specification of `remove` requires two additional state dimensions. The mode dimension characterizes iterators as readonly or modifying. This dimension is *immutable* in the sense that an iterator cannot change between these states. The `remove` method can only be called on modifying iterators. Notice how the specification preserves that state like a side condition. With regard to the other dimension, `remove` prescribes that the previous element must be retrieved in order to `remove` it, making it also unavailable. Notice that the specification for `next` changes the previous dimension to retrieved. This enforces that `remove` can be called at most once after each call to `next`. (A newly created iterator will be in the unavailable state.)

3.4 Specifying Iterables

The `Iterable` interface is used to create iterators. We define two *cases* for this method. One case creates a read-only

```
interface Iterator<c : Iterable, g : alive → Fract> {
   states available, end refine alive as next
   states unavailable, retrieved refine alive as previous
   states readonly, modifying refine alive as mode

   boolean hasNext() :
      ∀this : Iterator. ∃result : boolean.
         pure(this, next) ⊸ (pure(this, next, available) ⊗ result = true)
                          ⊕ (pure(this, next, end) ⊗ result = false)

   Object next() :
      ∀this : Iterator. ∃result : Object.
         full(this, previous) ⊗ full(this, next, available) ⊸
                  full(this, previous, retrieved) ⊗ full(this, next) ⊗ pure(result, alive)

   void remove() :
      ∀this : Iterator.
         full(this, previous, retrieved) ⊗ pure(this, modifying) ⊸ full(this, previous, unavailable) ⊗ pure(this, modifying)

   void finalize() :
      ∀this : Iterator.
         (unique(this, alive, readonly) ⊸ pure(c, alive, g)) & (unique(this, alive, modifying) ⊸ full(c, alive, g))
}
```

Figure 2: `Iterator` interface specification

iterator and divides the fraction on the receiver's permission in half. The second half is given as a pure permission to the resulting readonly iterator. The other case requires a unique permission to the receiver in order to create a modifying iterator. Only a pure permission to the receiver is retained. Notice that our iterators are parameterized with a collection and a fraction (figure 2). These parameters help describing an iterator's permission to a collection.

Calling `iterator` with a full permission will always yield a read-only iterator (because only the first case applies). When calling it with a unique permission, on the other hand, both cases could apply. Thus `iterator` conceptually returns a readonly & modifying iterator, i.e., one of the two at the caller's choice, but not both. Another call to `iterator` forces readonly while calling `remove` on an iterator forces it to be modifying. As would be expected, retrieving elements from an iterator does not force one or the other. Thus we delay the choice between these two cases until it is inevitable. A full reference to an iterable indicates the existence of read-only iterators. A pure reference to an iterable indicates a modifying iterator.

3.5 Verification

Clients that use iterators as specified above can be verified by tracking linear permissions of bound variables. If reasoning about a decidable fragment of linear logic (MALL), dependently typed objects, and splitting and coalescing of permissions can be automated then verification can proceed automatically. Capabilities like Fugue's "state predicates" [4] let us reason about correctness of iterator *implementations* as well.

The last section described how specifications of `Iterable` and `Iterator` allow creating and using iterators in the right way. The question arises, how can a collection ever be modified again after an iterator was created? And how can we create a modifying iterator after other iterators were created?

A simple variable liveness analysis can determine when an iterator is no longer needed. If a variable dies that carries a unique permission to an iterator then the iterator becomes inaccessible and is subject to garbage collection. As soon as the iterator is dead we can get back its permission to the underlying collection.

We use the *finalizer* to specify this. A `finalize` method is defined for all Java objects and intended to be called in the process of garbage collection to release resources. We use it to release the permission to the iterated collection (figure 2; notice how the permission depends on the iterator's mode). Once released, the permission can be coalesced with any other permissions to the collection. Finalizing *all* created iterators restores the original unique permission to the collection, enabling direct modifications and creation of a modifying iterator.

4. LIMITATIONS AND COMPARISON

We identified the following two limitations of the specification presented here.

- The specification prevents the following legal use of Java iterators. A client can create two iterators and iterate over them in parallel until it decides to start modifying the collection through one of the iterators. This is legal if the other iterator is never used again. Our specification does not permit the modification because creating two iterators forces both to be read-only (see above) unless the second one is *created* after the first one dies. We are working on overcoming this problem by implicitly *changing* the iterator mode.
- As discussed above, the specification requires collections to be linear in order to be modified directly. This

```
interface Iterable {
   Iterator iterator() :
      ∀this : Iterable.
         (∀g : alive → Fract. ∃result : Iterator<this, g/2>.
            full(this, alive, g) ⊸ full(this, alive, g/2) ⊗ unique(result, alive, readonly))
       & (∃result : Iterator<this, alive ↦ 1/2>.
            unique(this, alive) ⊸ pure(this, alive, alive ↦ 1/2) ⊗ unique(result, alive, modifying))
}
```

Figure 3: `Iterable` interface specification

is stronger than one would expect; a full permission to the collection should suffice. The problem is that iterators expect collections to be immutable. We could model this with a state change of the collection (from "mutable" to "immutable"), but then we would need a dynamic test to know when the collection is mutable again. Instead we use fractions to count the number of iterators. Thus we trade states against aliasing restrictions and ease of use against flexibility in order to meet the Java specification (that does not include dynamic state tests on iterators).

The presented iterator specification uses a general technique that allows verification of iterator clients and implementations. We are aware of general techniques for functional specification (e.g. the JML [6], Spec# [1]) that rely on manual verification or automatic decision procedures but that are undecidable in general. Our technique supports certain forms of aliasing and is restricted to reasoning about typestates. The technique can capture many uses of iterators but we pay (modulo a cleverer specification) with the limitations mentioned above.

5. CONCLUSIONS

This paper presents a specification of Java iterators that may allow automatic verification of clients. The specification is conservative in that it respects the rules defined in the Java Collection API. To this end it limits aliasing of collections beyond what seems necessary and forbids a legal (albeit unusual) use of iterators.

6. ACKNOWLEDGMENTS

The author wishes to thank Ciera Christopher, Nels Beckman, and Jonathan Aldrich for helpful feedback on an earlier draft of this paper. This work was supported in part by NASA cooperative agreement NNA05CS30A, NSF grant CCF-0546550, and the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems".

7. REFERENCES

[1] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.

[2] K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. In *ACM Symposium on the Foundations of Software Engineering*, pages 217–226, Sept. 2005.

[3] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

[4] R. DeLine and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*. Springer, 2004.

[5] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[6] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev28, Iowa State University, Department of Computer Science, July 2005.

[7] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, 1986.

Reasoning About Iterators With Separation Logic

Neelakantan R. Krishnaswami
Carnegie Mellon University
neelk@cs.cmu.edu

ABSTRACT

Separation logic is an extension of Hoare logic which permits reasoning about imperative programs that use shared mutable heap structure. In this note, we show how to use higher-order separation logic to reason abstractly about an iterator protocol.

Categories and Subject Descriptors

D.2 [**Software/Program Verification**]: Correctness Proofs

General Terms

Languages, Verification

Keywords

separation logic, iterators, aliasing, challenge problem

1. JAVA STYLE ITERATORS

The iterator interface[3] in Java works roughly as follows. First, we have a mutable collection type. This type supports a number of operations, some of which like `add`-ing an element to a collection will mutate the collection, and others, like checking to see if it is `empty`, which do not modify the collection.

To get the elements of a collection, we create another mutable object called an iterator. This object has a method `next`, which returns a new element of the collection each time it is called, finally failing when there are no more elements within it.

However, both the collection and the iterator are imperative objects, and correct usage of an iterator also requires observing some additional restrictions to ensure that the state of an iterator and its underlying collection remain in sync. Specifically, a client program:

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006), November 10–11, 2006, Portland, Oregon, USA.

- may create as many iterators on a single collection as they like,
- may freely call any methods on the collection that do not change the collection's observable state (such as `empty`),
- may freely call `next` on the iterators in any order, and
- may **NOT** call `next` on an iterator after calling `add` on the underlying collection.

The general idea is that an iterator maintains a pointer into some part of the collection during its traversal, and that updating the collection may cause the iterator's reference to point to an incorrect part of the collection. So while an iterator is live, dangerous method calls to its underlying collection should be forbidden, and only safe method calls permitted.

2. SEPARATION LOGIC

Separation logic [5, 6] is an extension of Hoare logic [4] intended to simplify reasoning about aliasing with mutable data structures. We have developed a version of separation logic that permits reasoning about imperative programs in high level languages (such as Java or ML), and which uses features of higher-order logic [2] to reason abstractly about high-level aliasing behavior.

Separation logic extends the logical language of preconditions and postconditions with two new logical connectives, the separating conjunction "$A * B$", and the magic wand "$A -\!\!* B$". Intuitively, $A * B$ means that A holds in one part of the heap, and B holds in a disjoint part of the heap. This contrasts with the regular conjunction $A \wedge B$, which means that both A and B hold in the current heap. The magic wand $A -\!\!* B$ means that if you added a piece of storage which validated A to the current heap, the whole thing would validate B. (Likewise, the informal meaning of $A \supset B$ is that if A holds in the current heap, then B will too.) Finally, the separating and ordinary connectives can be freely mixed, which lets us describe quite complex aliasing behaviors.

We use the propositions of separation logic to describe the pre- and post-conditions of commands, and describe the behavior of commands with Hoare triples of the form $\{P\}\ c\ \{a : \tau.\ Q\}$. The P is the state the heap must be in before the command can be run, and Q describes the changed heap after the command finishes. Since side-effecting operations can also return values, we use the $a : \tau$ notation to name the return value in the postcondition.

$\exists$`new_coll, size, add, new_iter, next`.
$\exists coll$: ((ref list $\times$ ref $\mathbb{N}$) $\times$ seq nat $\times$ prop) $\Rightarrow$ prop.
$\exists iter$: (ref list $\times$ (ref list $\times$ ref $\mathbb{N}$) $\times$ seq nat $\times$ prop) $\Rightarrow$ prop.

$\{\top\}$ `new_coll()` $\{a : (\text{ref list} \times \text{ref } \mathbb{N}).\ \exists P.\ coll(a, [], P)\}$ and

$\forall P, c, x, xs.\ \{coll(c, xs, P)\}$
`add`(c, x)
$\{a : 1.\ \exists P'.\ coll(c, x :: xs, P')\}$ and

$\forall P, c, xs.\ \{coll(c, xs, P)\}$
`empty`(c)
$\{a : \text{bool}.\ coll(c, xs, P)\}$ and

$\forall c, xs, P.\ \{coll(c, xs, P)\}$
`new_iter`(c)
$\{a : \text{ref list}.\ iter(a, c, xs, P)\}$ and

$\forall i, c, xs, P.\ \{iter(i, c, xs, P)\}$
`next`(i)
$\{a : 1 + \text{nat}.\ iter(i, c, xs, P)\}$ and

$\forall i, c, xs, P.\ \{iter(i, c, xs, P) \supset coll(c, xs, P) *$
$coll(c, xs, P) -\!\!* iter(i, c, xs, P)\}$

Figure 1: Iterator Specification

Since a triple only specifies the behavior of a single routine, we combine triples into specifications, which are logical formulas that use triples as their atomic propositions. So we can specify an interface to a module by taking the conjunction of the triples for each operation, and then existentially quantifying over the implementations.

3. THE ITERATOR PROBLEM

3.1 Iterator Interface Specification

We give a concrete example of this idea in Figure 1, which is the specification for iterators. In this example, our overall collection type is the pair ref list $\times$ ref $\mathbb{N}$. The first field has type ref list, which is the type mutable linked lists of integers, and the second field is a pointer to a natural number. This field tracks the number of times a harmless method like `empty`, which helps illustrate the fact that the *concrete* state of an object can change even while its *abstract* state remains the same.

To describe the heap behavior, we introduce a pair of existentially quantified predicates, *coll* and *iter*. These predicates permit us to talk about the mutable state associated with collections and iterators, without revealing their concrete implementation. The predicate $coll(c, xs, P)$ asserts that the collection object c represents the abstract sequence xs, and that it is in an abstract state P. We represent abstract states using propositions, which is why we need higher-order logic. The assertion $iter(i, c, xs, P)$ asserts that i is an iterator over the collection c with elements xs and abstract state P.

For example, the specification

$$\{\top\}\ \texttt{new\_coll()}\ \{a : \tau_c.\ \exists P.\ coll(a, [], P)\}$$

states that starting from any heap, calling `new_coll` will return a new mutable list and heap structure corresponding to that list. The specification

$$\{coll(c, xs, P)\}\ \texttt{empty}(c)\ \{a : \text{bool}.\ coll(c, xs, P)\}$$

```
1   {coll(c, xs, P)}
2   letv b = empty(c) in
3   {coll(c, xs)}
4   letv i1 = new_iter(c) in
5   {iter(i1, c, xs, P)}
6   {(coll(c, xs, P) * (coll(c, xs, P) -* iter(i1, c, xs, P))}
7   letv i2 = new_iter(c) in
8   {iter(i2, c, xs, P) * (coll(c, xs, P) -* iter(i1, c, xs, P))}
9   {coll(c, xs, P)*
      (coll(c, xs, P) -* iter(i1, c, xs, P))*
      (coll(c, xs, P) -* iter(i2, c, xs, P))}
10  letv b' = empty(c) in
11  {coll(c, xs, P)*
      (coll(c, xs, P) -* iter(i1, c, xs, P))*
      (coll(c, xs, P) -* iter(i2, c, xs, P))}
12  {iter(i1, c, xs, P))*
      (coll(c, xs, P) -* iter(i2, c, xs, P))}
13  letv v = next(i1) in
14  {iter(i1, c, xs, P))*
      (coll(c, xs, P) -* iter(i2, c, xs, P))}
15  {coll(c, xs, P)*
      (coll(c, xs, P) -* iter(i1, c, xs, P))*
      (coll(c, xs, P) -* iter(i2, c, xs, P))}
16  {iter(i2, c, xs, P))*
      (coll(c, xs, P) -* iter(i1, c, xs, P))}
17  letv v = next(i2) in
18  {iter(i2, c, xs, P))*
      (coll(c, xs, P) -* iter(i1, c, xs, P))}
19  {coll(c, xs, P)*
      (coll(c, xs, P) -* iter(i1, c, xs, P))*
      (coll(c, xs, P) -* iter(i2, c, xs, P))}
20  letv _ = add(c, x) in
21  {∃Q. coll(c, xs, Q)*
      (coll(c, xs, P) -* iter(i1, c, xs, P))*
      (coll(c, xs, P) -* iter(i2, c, xs, P))}
```

Figure 2: Iterator Client

asserts that the `empty` function will return a boolean, and that it will leave the abstract state unchanged. Note that we could give a more precise specification (e.g., that the `empty` function returns true if the collection is empty and false otherwise). We choose not to in order to focus this example on aliasing.

By way of contrast, the specification for `add`

$$\{coll(c, xs, P)\}\ \texttt{add}(c, x)\ \{a : 1.\ \exists P'.\ coll(c, x :: xs, P')\}$$

says that adding an element to the collection will alter the abstract state of the object. We existentially quantify over the abstract state in the postcondition, to show we can no longer assume that P' is the same as P.

The specification for `new_iter`

$$\{coll(c, xs, P)\}\ \texttt{new\_iter}(c)\ \{a.\ iter(a, c, xs, P)\}$$

says that if we start with a collection c, then we can consume it to construct an iterator.

The `next` function has the specification

$$\{iter(i, c, xs, P)\}\ \texttt{next}(i)\ \{a : 1 + \text{nat}.\ iter(i, c, xs, P)\},$$

which says that if we have an iterator i, then `next`(i) will give us an integer or signal a failure. As with `empty`, we do not model the behavior of the iterator in any further detail — the spec could easily be refined further, but that detail would not be relevant to the issue of reasoning about aliasing. The detail that is relevant is the fact that the iterator preserves the abstract collection state P, which is

how we describe the fact that the iterator does not modify the underlying collection.

That said, a natural question is how we can create two iterators on the same collection, because the `new_iter` function transforms a $coll(c, xs)$ state to an $iter(i, c, xs, P)$ state, which means that the precondition to call `new_iter` no longer holds. This is where the *sharing axiom* comes into play – the final invariant in the specification:

$$iter(i, c, xs, P) \supset [coll(c, xs, P) * \\ coll(c, xs, P) -\!\!* iter(i, c, xs, P)]$$

is a separation logic formula that describes how to recover a collection from an iterator state. It says that if we have an iterator state $iter(i, c, xs, P)$, then that state can be viewed as two disjoint pieces, one of which is the original collection (with the invariant P maintained), and one piece that can be combined with the collection to restore the iterator.

The sharing axiom makes use of the fact we have both standard implication and separating implication available in the same logic. We use implication to reason that the same piece of state can be viewed in multiple ways, and the separating implication to reason about one isolated part of the state.

3.2 Iterator Client Usage

We can see an example of how a client would make use of this specification in Figure 2. On line 1, we see that the precondition for our program is that the variable c holds a collection. On line 4, we create an iterator i_1, consuming the collection to produce an iterator, as seen in the state on line 5. We now apply the sharing axiom on line 6 to break the iterator state into two pieces, which lets us create a second iterator bound to i_2.

The program state on line 8 contains an iterator for i_2, and some state that will let us reconstruct i_1's iterator. On line 9 we apply the sharing axiom once more, to break out the collection state again, and this lets us call `empty` on line 10.

On line 12, we use the collection and an i_1's iterator fragment to recover the precondition for calling `next`(i_1) on line 13, and then on lines 14-16, we apply the sharing axiom and combine the iterator state fragment for i_2, so that we can call `next`(i_2) on line 17.

The informal idea should be coming into focus now – we are transferring ownership of the collection between the different collections, using the sharing axiom to get a collection out of an iterator, and the deduction rule for magic wand ($A * (A -\!\!* B)$ entails B) to put it back in.

On line 18 and 19, we once again use the sharing axiom to disassemble the iterator and get back the collection, and then call `add`(c, x) on line 21. This gives us a state in which $\exists Q.\ coll(c, x :: xs, Q)$ holds. We can no longer apply the separating implication law to get a full iterator state, because we need a hypothesis of the form $coll(c, xs, P)$ to recover an *iter* state, and we don't know whether Q is the same as P. As a result, we can no longer call `next` on either i_1 or i_2 any longer, just as we desire.

So the Hoare triples and sharing axioms put us in a situation where we can create multiple iterators, and can freely call methods on the collection which don't change its abstract state, but which also enforce the property that there can be no calls to `next` after modifying the collection – and the client was able to do this without knowing anything about the details of the internal heap structure of the collection.

Interestingly, the abstract states in our spec are reminiscent of a consistency check the Java collection libraries perform. The Java libraries keep a sequence number for each collection, and update it when the collection is modified. Iterators save the sequence number when they are created, and will raise a runtime error if the underlying collection's sequence number ever differs from their saved value. With our specification, the abstract state changes whenever we call a dangerous method, and our (static) verification is kept from proceeding.

3.3 Iterator Implementation

Finally, in Figure 3, we give an example implementations for this specification. The specification is a big existential quantifier, and so our implementations are the witnesses to this existential type. For the abstract program variables (such as `next` or `empty`) we give function definitions. A collection is a linked list and a counter, and an iterator is a pointer to the interior of a list.

Most of these definitions manipulate imperative linked lists in the obvious way, but it's worth examining `empty`. A call to this function modifies the state of the collection, but in a safe way. It updates the counter, but does not modify the linked list, so iterators over the collection will not be invalidated. More elaborate examples might be something like a collection that does memoization or a splay tree that rebalances after each query. In each of these cases, the abstract state of the object does not change, even though its in- memory representation might.

We demonstrate this idea in the definitions of the existentially quantified predicates. These predicate definitions are given as functions of their input, which take in data and return propositions. The definition of the $coll(c, xs, P)$ predicate is an assertion that a collection value's first field points to an integer counter, and that its second field is a linked list representing xs, and is also in state P. The *linked_list* predicate is a recursive function on xs, which permits us to define an inductive predicate characterizing linked lists.

The *iter* predicate, for example, is an assertion stating that the iterator points to an interior pointer of the linked list, and that the predicate variable P is preserved for the whole list.

In this example, we have focused on being able to abstractly specify and reason about the imperative aspects of modules. Of course, one would also like the iterator specification to be abstract in the implementation types used for collections and iterators (here ref list $\times$ ref N), i.e., to have existential quantification over *types* to model abstract data types. We plan on addressing this in future work.

4. CONCLUSIONS

Theorem proving in higher-order logic has a long and notorious history of being very difficult to automate, and the addition of separation logic will not make this task any easier. However, different kinds of partial automation are probably feasible, and what follows are hopefully-educated, possibly-wild, guesses about the difficulty of different levels of automation.

The simplest level is just verifying that an annotated pro-

gram is actually correct with respect to our program logic. This should be quite straightforward to implement using a tactical theorem prover such as Coq or Isabelle, though we have not actually implemented this.

The next easiest task will be to automatically verify that a client program respects a given specification. The sorts of manipulations we performed in the sample client code did not make essential use of higher-order logic, since the predicate variables representing abstract states were never instantiated. Assuming this is a general pattern, checking client code should not require more know-how than checking programs that use first-order separation logic. This is still a fairly difficult problem, though substantial progress has been made with Smallfoot [1].

Automatically checking that implementations satisfy a given specification is almost certainly a much harder problem. We must construct functions that show how to realize abstract predicates (such as $coll(c, xs, P)$), and finding them can require real creativity. However, it *may* be possible to partially automate checking the function bodies given all the predicate definitions.

Finally, inferring specifications and module boundaries from programs seems completely out of reach, since the relevant abstraction boundaries simply are not evident in the code, and even skilled human programmers find identifying them a very difficult task.

However, the main purpose of this line of research is not to produce ready-to-use tools. Instead, we are trying to construct a very rich specification language capable of describing how aliasing is used as concisely and naturally as possible. Our hope is that having simple mathematical characterizations of the realistic aliasing patterns will make it easier to construct and validate more limited (and hence more automatable) methods that verify exactly and only the forms of aliasing used in well structured programs.

$$
\begin{array}{lll}
\texttt{new\_coll}\ () & \equiv & \mathsf{letv}\ counter = \mathsf{new}_{\mathbb{N}} 0\ \mathsf{in} \\
& & \mathsf{letv}\ list = \mathsf{new}_{\mathsf{list}} \mathsf{nil}\ \mathsf{in} \\
& & (list, counter)
\end{array}
$$

$$
\begin{array}{lll}
\texttt{add}(c, x) & \equiv & \mathsf{letv}\ cell = !(\mathsf{fst}\ c)\ \mathsf{in} \\
& & \mathsf{letv}\ t = \mathsf{new}_{\mathsf{list}} cell\ \mathsf{in} \\
& & c := \mathsf{cons}(x, t)
\end{array}
$$

$$
\begin{array}{lll}
\texttt{empty}(c) & \equiv & \mathsf{letv}\ cell = !(\mathsf{fst}\ c)\ \mathsf{in} \\
& & \mathsf{letv}\ \_ = \mathsf{increment}(\mathsf{snd}\ c)\ \mathsf{in} \\
& & \mathsf{listcase}(cell, \mathsf{true}, (h, t).\ \mathsf{false})
\end{array}
$$

$$
\texttt{new\_iter}(c) \equiv \mathsf{new}_{\mathsf{ref\ list}}(\mathsf{fst}\ c)
$$

$$
\begin{array}{lll}
\texttt{next}(i) & \equiv & \mathsf{letv}\ c = [!i]\ \mathsf{in} \\
& & \mathsf{letv}\ cell = [!c]\ \mathsf{in} \\
& & \mathsf{letv}\ ans = \mathsf{listcase}(cell, \mathsf{None}, \\
& & \qquad (h, t).\ \mathsf{letv}\ \_ = i := t\ \mathsf{in}\ \mathsf{Some}\ h)\ \ \mathsf{in} \\
& & ans
\end{array}
$$

$$
coll(c, xs, P) \equiv \exists n.\ \mathsf{snd}\ c \hookrightarrow n * (linked\_list(\mathsf{fst}\ c, xs) \wedge P)
$$

$$
\begin{array}{lll}
linked\_list(c, x :: xs) & \equiv & \exists c'.\ c \hookrightarrow \mathsf{cons}(x, c') * linked\_list(c', xs) \\
linked\_list(c, [\,]) & \equiv & c \hookrightarrow \mathsf{nil}
\end{array}
$$

$$
\begin{array}{lll}
seg(l, l', x :: xs) & \equiv & \exists l''.\ l \hookrightarrow \mathsf{cons}(x, l'') * seg(l'', xs) \\
seg(l, l', [\,]) & \equiv & l = l'
\end{array}
$$

$$
\begin{array}{lll}
iter(i, c, xs, P) & \equiv & \exists l, n, xs_1, xs_2. \\
& & \qquad (P \wedge (seg(\mathsf{fst}\ c, l, xs_1)\ * coll(l, xs_2))) * \\
& & \qquad i \hookrightarrow l\ * \mathsf{snd}\ c \hookrightarrow n \wedge \\
& & \qquad xs = xs_1 \cdot xs_2
\end{array}
$$

Figure 3: Iterator Implementation

5. REFERENCES

[1] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of the Fourth International Symposium on Formal Methods for Components and Objects*, Amsterdam, The Netherlands, 2001.

[2] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines and higher order separation logic. In *Proc. of ESOP 2005: The European Symposium on Programming*, pages 233–247, Edinburgh, Scotland, April 2005.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[4] C. A. R. Hoare. An axiomatic approach to computer programming. *Communications of the ACM*, 12(583):576–580, 1969.

[5] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'01)*, London, 2001.

[6] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74, Copenhagen, Denmark, July 2002. IEEE Press.

SAVCBS 2006
POSTER ABSTRACTS

Automatic Data Environment Construction for Static Device Drivers Analysis

(Extended Abstract)

Hendrik Post, Wolfgang Küchlin
University of Tübingen / Symbolic Computation Group
72076 Tübingen, Germany
{post,kuechlin}@informatik.uni-tuebingen.de

ABSTRACT

Linux contains thousands of device drivers that are developed independently by many developers. Though each individual driver source code is relatively small—≈10k lines of code—the whole operating system contains a few million lines of code. Therefore Linux device drivers offer a useful application area for modular analysis.

Our finding is that despite the precise modeling of most features of the standard systems programming language C, model checking software verification tools for C fail to provide means for modular analysis of device drivers. We inspected CBMC [2], SLAM-SDV [3], MAGIC [1], BLAST [4] and others and found that a rich additional environment model for every device driver is needed. This model must provide information on out-of-scope initialized pointers and complex data structures. We present strategies to automatically create feasible, bounded data environments for Linux device drivers instead of creating them manually. Our solution differs from general interface generation mechanisms (e.g. CUTE[5]), because is it specialised on bounded model checking of Linux device drivers written in C. Our contribution is a preprocessing step that extends the usability of CBMC for modular Linux device driver analysis.

Categories and Subject Descriptors

D.2.4 [**Software**]: Software Verification—*Model Checking*; D.4.5 [**Operating Systems**]: Reliability—*Verification*

General Terms

Verification, Experimentation

Keywords

Linux, Bounded Model Checking, Environment Modelling, Software Verification

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006), November 10–11, 2006, Portland, Oregon, USA.

1. INTRODUCTION

Software verification is commonly interpreted as the analysis of a software system concerning latent or possible errors. Though the degree of precision and the tradeoff between completeness and a low false-positive rate differs, a common pragmatic aim of verification is to find and eliminate errors. As verification requires lots of resources, it is preferably applied to systems where a stable, specified behavior of the software is of high importance.

The problem we identify with modular verification of Linux device drivers is external data representation. Data initialization and data usage are performed on multiple operating system layers. The data interface between layers includes pointers, and especially those that are subject to pointer arithmetic. Hence modular analysis faces the problem that pointers are used extensively, but exact information about their initialization or targets is commonly not available within the scope of analysis as discussed in Section 2.2.

As many tools for the analysis or even verification of C programs exist, we aim to provide preprocessing such that one of these can be used on device drivers. Our suggestion is to provide concrete targets for all externally initialized pointers, whereas all external variables of primitive data types remain unconstrained symbolic values. This approach is therefore located between pure abstract static analysis and concrete software testing.

The tools for the analysis of C programs we explicitly reviewed are CBMC[2], BLAST [4], Meta-Compilation [9], SATURN [8], MAGIC[1] and the Static Driver Verifier from the SLAM project [3]. The modular analysis tool MAGIC for example does not support dereferencing of pointers on the left side of an assignment at all. The analysis performed with BLAST in [4] has been made possible by writing a test driver. The static driver verifier from the SLAM project [3] also requires a manually created operating system model prior to analysis. The Meta-Compilation project also put an extensive amount of work into the generation of a test environment via abstracting the Linux kernel [9].

Several sophisticated tools for environment generation exist, though they target different or more general application areas and hence provide different strategies. Symstra[7] and CUTE[5] seem closest to our approach. Symstra generates environments for static analysis as we do, but its current implementation covers only programs written in Java. CUTE includes a mechanism to expand the object graph on the fly, should its analysis indicate the necessity. Its approach is a mixture of Symbolic Execution and Testing while we aim

to provide only preprocessing for model checking tools. As both tools target general applications instead of the limited area of Linux device drivers, a direct support of common Linux abstract data types was not considered. Instead of a general approach we present a minimal environment generation in order to facilitate the analysis of device drivers.

We chose the software verification tool CBMC as our analysis backend due to its extensive support of C language features as summarized in Section 2.1. We will also discuss problems when using CBMC for modular analysis.

2. VERIFICATION TOOLS AND TARGETS

2.1 CBMC

CBMC 2.1 is a bounded model checker intended to be used for the analysis of C programs and Verilog descriptions. When running in C analysis mode, it translates ANSI-C programs into propositional logic. Loops and recursion are handled by code unwinding. CBMC supports pointer arithmetic, integer operators, type casts, side effects, function calls, calls through function pointers, non-determinism, assumptions, assertions, arrays, structs, named unions, and dynamic memory. Therefore this tool is a good choice to analyze systems code written in C that makes use of these features. CBMC itself is capable of finding double-free and free-after-use errors beside bounds and pointer validity checking.

CBMC offers an extensive treatment of pointers that essentially tracks the object and the offset a pointer points to. Nevertheless two technical problems remain unsolved in the documentation. First, pointers—when dereferenced—must point to a valid object. Though this is a reasonable assumption for the runtime behavior of programs, it is not useful for modular static analysis of programs where targets of pointers are often unknown. The second shortcoming is the modeling of possible aliases between pointers to unknown targets. We illustrate both problems by a short example:

```
 1: struct person_t {
 2:   int age;
 3: } a_person;
 4: void set_age_difference(struct person_t* p1,
          struct person_t* p2, int diff) {
 5:    p2->age = p1->age + diff;
 6:    assert(p2->age == p1->age + diff);
 7: }
 8: void main() {
 9:    struct person_t* p1 = &a_person;
10:   struct person_t* p2 = &a_person;
11:   set_age_difference(p1, p2, 20);
12: }
```

The example is artificial for the sake of simplicity. In function `set_age_difference()` two structs are passed by reference. The age of the second person's record should be set to the age of the first person's record plus an age difference. Line 6 contains a check if the assignment was made correctly. The assertion is invalid because both pointers may alias each other. In this case both dereferences of the age fields are equal which violates the assertion if `diff` $\neq 0$. The two different entry points for an analysis are `main` and `set_age_difference`. Moreover CBMC allows to either enable or disable checks for invalid dereferences of pointers. The latter feature is almost undocumented. We assume that disabling pointer checks globally disables checks for `null` or uninitialized pointers. We identify four different analyses entering the module either at `main` or `set_age_difference` and either checking for invalid pointers.

1. *Entry at* ***`main`****, Pointer Checks disabled.* When CBMC starts at `main`, `p1` and `p2` are explicitely aliased and passed as a reference to `set_age_difference`. Hence dereferencing them is allowed and writing to one of the pointer targets correctly modifies the aliased pointer dereferences as well. The assertion is invalid as expected.

2. *Entry at* ***`main`****, Pointer Checks enabled.* This case leads to the same result as case 1.

3. *Entry at* ***`set_age_difference`****, Pointer Checks enabled.* This direct entry reflects the interleaving of modules within the Linux kernel where dispatch routines may be directly called from outside the module, and the environment as provided by `main` is unknown. When `set_age_difference` is analyzed with enabled pointer checks, CBMC correctly emits a warning that dereferencing the parameters in line 5 might be incorrect. This is due to the correct modeling that unconstrained pointers may be `null`. Though the result is technically correct it seems more reasonable for modular analysis to assume that the environment is correctly initialized.

4. *Entry at* ***`set_age_difference`****, Pointer Checks disabled.* Dereferencing the parameters is not a problem any more, but nevertheless the assertion is fulfilled. This is an incorrect analysis result when the scope of analysis is limited to `set_age_difference`. CBMC does not model a possible aliasing between uninitialized pointers.

Cases 3 and 4 offer significant shortcomings to device driver analysis. The impact on our work is twofold.

The user of CBMC has two choices which both lead to disadvantages: the user may globally disable pointer checks which might only be desirable for interface pointers, but not for pointers manipulated inside the module. Second, the user of CBMC faces the problem that alias relationships induced by an unknown operating system environment are not modelled or taken into account. This leads to an even less appropriate analysis. We assume that interface objects are set up correctly and hence we must perform the correct initialization before calling the entry function of the module.

2.2 Linux Device Driver Interfaces

Linux device drivers often operate on structs, each of which represents one device [6]. The generic CD-Rom device driver `cdrom.c`, for example, may service several hardware drives, each represented by one `struct cdrom_device_info`. This struct is of course passed by reference to all service routines found in the driver. The struct is partially listed in Figure 1. Devices are organized in a linked list via the field `struct cdrom_device_info * next`. When another system layer steps into the driver dispatch routines, it passes the currently serviced device via a reference to a struct. The struct itself is not initialized within any cdrom driver, but in other system layers. Invoking a verification tool on any service routine in `cdrom.c` leads to a problem. It is not evident

```
struct cdrom_device_ops {
   int (*open) (struct cdrom_device_info *, int);
   void (*release) (struct cdrom_device_info *);
   int (*drive_status)
               (struct cdrom_device_info *, int);
   ...
};
...
/* Uniform cdrom data structures for cdrom.c */
struct cdrom_device_info {
   struct cdrom_device_ops  *ops;
   struct cdrom_device_info *next;
   struct gendisk *disk;
   ...
   int speed; ...
};
```

Figure 1: The operation interface struct and the `cdrom_drive_info` struct listed from file `drivers/cdrom/cdrom.c` of Linux kernel 2.6.15.

that all references passed as a parameter are correctly initialized. Hence CBMC would correctly emit a warning that dereferencing one of them may lead to a potential `null` dereference. Though this message may be suppressed the better assumption would be to assume that the device structs are properly initialized.

In the next section we give a solution to this problem using automatically created data environments.

3. CREATING DATA ENVIRONMENTS

The general approach to create a suitable environment is to identify potentially uninitialized pointers that are either declared in the global scope of the module, or parameters to the analyzed entry point in the module. For each pointer, a fresh object can be created and the pointer is initialized by pointing to this object. This strategy is also used in Symstra[7] and CUTE[5]. This could lead to new uninitialized pointers if the object created is of a pointer type, contains fields of pointer type or is an array with elements of pointer type. In all three cases fresh objects are created for these pointers as well, up to a bounded object graph depth. This depth-bound reflects the size of a recursive data structure, e.g. a list. For binary trees this bound limits the depth of the tree. For bounded model checking this depth should be smaller than or equal to the CBMC bound for loops and recursion, as a loop iterating over all elements of a list or a recursive search for an element in a tree won't violate the unwinding assertions. Then CBMC may capture the exact behavior of functions on bounded data structures.

We identify two sources of pointers that must be initialized:

1. Pointers within the global scope of the translation unit.
2. Parameters of pointer type within the module entry function to be analyzed.

The main algorithm performs a breadth-first search on the object graph with bounded depth:

```
 1: worklist = union(global_pointers(file),
       parameters(entry_function));
 2: object stub_obj; depth = 1;
 3: new_worklist = {};
 4: while (!is_empty(worklist)
                  && depth < depth_bound) {
 5:   for each pointer p in worklist {
 6:       stub_obj = create_object_for_pointer(p);
 7:       create_assignment_for(p,stub_obj);
 8:       new_worklist = union(new_worklist,
                        pointer_members(stub_obj);
 9:   }
10:  worklist = new_worklist;
11:  new_worklist = {};
12:  depth = depth +1;
13:}
```

The search for uninitialized pointers begins with parameters of the entry function and all globally declared pointers (line 1). We do not restrict the seed set of pointers to `extern` variables (CUTE) as a routine may also rely on objects created within the module. If the global scope defines nested structs, pointer members of substructs are included recursively. Then a breadth-first search is performed until no new pointers are exhibited or a fixed depth bound is exceeded (line 4-13). The search follows the well known worklist pattern.

Some considerations complement the algorithm though they are not included in the above pseudo-code.

- In order to create a reasonable environment for pointers we face the problem that an `int *` may point to a single `int` or to an `int []`. We propose to decide whether to create an array or a single object dependent on a source code analysis that reports if the pointer is subject to any pointer arithmetic or index operation. In this case it seems reasonable to create an array of simple objects instead of a single one.
- If the depth bound terminates our algorithm, we must decide how the uninitialized pointers in the current worklist are treated. We suggest that these should be initialized to `null`.
- Common abstract data types in Linux can be created by detailed templates. The most prominent example is the definition of linked lists. List elements are then required to be structs with one field having the type `struct list_head` from `include/linux/list.h`. For each list implemented in this way, we suggest to create a bounded stub of this list with pointers pointing to the next list element. The last element terminates the list by a `null` next field. For other predefinied data types initialization can be accomplished in a similar fashion.
- Class invariants over primitive data types, e.g. sorted lists with a `int` key field, can be encoded by `assume` statements.
- The unrolling depth of each single data structure may be independently, non-deterministically chosen.

Using this strategy, aliasing occurs only when it is explicitely specified by a Linux abstract data type template. A short example of results produced by our algorithm is presented in Figure 2. For further aliasing between other arbitrary objects we offer a generic model in the next section.

```
struct cdrom_device_info_stub1;
struct cdrom_device_info_stub2;
struct cdrom_device_info_stub3;
...
void init_environment() {
// pointer initialization
   struct cdrom_device_info *
      parameter_stub = &cdrom_device_info_stub1;
   cdrom_device_info_stub1.next =
           &cdrom_device_info_stub2;
   cdrom_device_info_stub2.next =
           &cdrom_device_info_stub3;
// bound reached
   cdrom_device_info_stub3.next = null;
// call to entry point of module
   int parameter_stub2;
   open(parameter_stub,parameter_stub2);
}
```

Figure 2: Result of a manual execution of our algorithm.

3.1 Alias Modeling

In the above section we initialized all pointers by different objects. Though many device driver environments might be modeled successfully using this conservative alias policy, a finer-grained analysis might be included into our environments. A classical must / may alias analysis, or a user specified equivalent description, could specify which pointers may alias each other. These specifications may be implemented easily in a small initialization code block that can be automatically generated.

For each must alias analysis we insert a new assignment statement into the code block: we translate `p1` *must* alias `p2` into a statement `p1 = p2;` . For *may* alias relationships, we could exploit the modeling of non-deterministic data values in CBMC. The built-in CBMC function `nondet_bool()` returns either true or false. Hence we may easily translate may alias relations: `p1` *may* alias `p2` results in a statement

```
if (nondet_bool())  p2 = p1;
```

The aliasing may hence lead to an exponential number of different initialization paths.

4. SUMMARY

Many tools for model checking C code exist. Though these tools offer a sophisticated treatment of most features of the language, our finding is that they are not yet stand-alone solutions for the modular analysis of Linux device drivers. In most cases the tools have to be complemented by an extensive operating systems model or at least a test driver that invokes and initializes all necessary members in the driver. It has been described that the construction of environments needs considerable effort. In [9] the effort dominated the overall work.

The contribution of our paper is an algorithm to automatically construct simple data environments for device drivers. We have shown that these environments might improve the precision of CBMC when analysing Linux device drivers. Other solutions for interface generations exist, though they seem either dedicated to Java programs (e.g. [7]) or they aim to provide a general solution loosing the advantages the predefined Linux abstract data types. Our solution only needs a specified entry point for each module and templates for the few abstract data types in Linux.

Despite the expected advance by widening the application domain in CBMC to modular programs, we left several problems unsolved. The creation of the environment is heuristic and requires an external specification of possible alias relationships. The identification of abstract data types is heuristically inferred and must be checked and potentially corrected by the user. The same situation is given for the inference whether a pointer points to a single element or an element within an array.

Early prototypes result in hundreds of generated initialization code lines even for small list bounds of 3. CBMC was able to process the examples and find some known pointer errors that could not have been checked without our data environment.

5. REFERENCES

[1] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. In *Proc. of the 25th Int. Conf. on Software Engineering (ICSE)*, pages 385–395, Washington, DC, USA, 2003. IEEE Computer Society.

[2] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[3] V. Contributors. The SLAM Project. http://research.microsoft.com/slam/, 2006.

[4] T. Henzinger, R. Jhala, R. Majumdar, and G. SUTRE. Software verification with BLAST. In *Proc. 10th Int. SPIN Workshop (SPIN'2003), Portland, OR, USA, May 2003*, volume 2648 of *LNCS*, pages 235–239. Springer, 2003.

[5] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for c. In *Proc. of the 10th European Software Engineering Conference (ESEC/FSE-13)*, pages 263–272, New York, NY, USA, 2005. ACM Press.

[6] D. van Leeuwen, E. Anderson, and J. Axboe. *A Linux Cdrom Standard*, kernel 2.6.15 edition, March 1999. Found in Documentation/cdrom.

[7] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381, Edinburgh, UK, April 2005.

[8] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proc. of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 351–363, New York, NY, USA, 2005. ACM Press.

[9] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proc. of the 16th Int. Conf. on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 273–288. Springer Berlin / Heidelberg, 2004.