

Reasoning with specifications containing method calls in JML and first-order provers

David R. Cok
B65 MC01816

Eastman Kodak Company, Research & Development Laboratories
Rochester, NY 14650-1816
USA

cok@frontiernet.net

ABSTRACT

Allowing method invocations in program specifications increases modularity and comprehensibility and is as important in specifications as it is in the program itself. However, method invocations do not map neatly into the first-order logics that are often used for assuring the correctness of specifications. One problem is translating specifications in a way that acknowledges the potential for exceptional behavior. The ESC/Java2 tool has been able to achieve a practical translation of method invocations within the design constraints of its parent tool, ESC/Java. Furthermore, the techniques used are applicable to other specification constructs such as quantifiers and model variables.

1. INTRODUCTION

Research and practical capability in program verification is advancing significantly with clearer semantics, evolution of languages and tools, and experience with industrial-scale software systems. Using method calls in specifications provides a level of abstraction and conciseness that promotes reading, writing and understanding specifications and will likely assist in their automated verification as well. However, method calls in specifications have not been widely supported and have unclear semantics in the light of potential exceptional behavior. This paper discusses the implementation of an extension to the static program checker ESC/Java that allows the use of method calls in specifications, with a discussion of the difficulties caused by the possibility of exceptions or non-terminating behavior. With that accomplished, several other programming language constructs can also be handled by the underlying prover. The approach described here is applicable to any source code translator interfacing with a prover that operates in a generic first-order logic (as opposed to a logic specifically designed to handle programming language constructs).

The solution described in this paper was implemented and tested using the Java Modeling Language, the ESC/Java2 project, and the Simplify prover, which are briefly described in sections 2-4. Section 5 describes a solution for translating method calls and issues arising from exceptional termination, with an example in the Appendix. Applications to other specification language features are presented in section 6. The paper ends with descriptions of some future work and of related work and conclusions in sections 7 and 8.

2. THE JAVA MODELING LANGUAGE

The Java Modeling Language (JML) has by now been described in several publications [1, 13, 14] and that full description will not be repeated here. The discussion in this paper can be illustrated using simple preconditions and postconditions.

- A **requires** keyword followed by a predicate declares a precondition for a routine.
- A **ensures** keyword followed by a predicate declares a normal postcondition for a routine.
- A **signals** keyword declares a postcondition that holds if the routine exits with the given exception.
- A **diverges** keyword declares a condition that holds if the routine never terminates.

Specifications are included in the text of a Java program by placing them in specially formatted comments, as shown in the figures. The syntax of the specification predicates follows Java closely. It excludes any operations that have side effects, such as the increment operator `++`. Other operations, such as arithmetic and comparison operators, have the same syntax and semantics as in Java. In particular, specification predicates may include method calls, if those methods are designated **pure**; tools supporting JML can then check that the implementations of **pure** methods have no side-effects.

3. ESC/JAVA2

The ESC/Java2 tool [5], an extension of ESC/Java [7, 8], implements the translation of Java programs and JML specifications into a target logic. ESC/Java was a pioneering tool in the application of static program analysis and verification technology to annotated Java programs. The tool and its built-in prover operate automatically with reasonable performance. The program annotations needed are easily read, written and understood by those familiar with Java and are partially consistent with the syntax and semantics of the separate Java Modeling Language (JML) project [1, 16]. Consequently, the original ESC/Java was a research success and was also successfully used by other groups for a variety of case studies [9].

The ESC/Java2 project extends ESC/Java and its long-term utility by addressing a number of issues.

- ESC/Java2 fully parses current JML and Java 1.4, so it is compatible with the variety of tools that now work with JML specifications.
- ESC/Java2 checks more of JML than did ESC/Java. For example, frame conditions were not checked in ESC/Java, but errors in frame conditions could cause

the prover to reach incorrect conclusions. ESC/Java also lacked the ability to use methods in annotations, limiting the annotations to statements only about low-level representations.

- ESC/Java2 provides ongoing distribution and maintenance. As companies were bought and research groups disbanded, there was no continuing development or support of ESC/Java; the tool was untouched for over two years and its source code was not available.

The engineering goals of ESC/Java were to be automatic and useful in finding bugs and violations of program specifications. It was not designed to be complete nor entirely sound. ESC/Java2 has continued that spirit, though some unsound aspects have been corrected.

It is important to note that ESC/Java translates the Java source code into a single-assignment guarded command language [6]. It does not incorporate an explicit notion of program state in the logical structure that represents the program. This simplifies the logic and makes it much easier to reason about variables that remain unchanged; however, it also adds difficulty to handling loops and to the translation of method calls used in annotations.

4. SIMPLIFY

ESC/Java2 and ESC/Java translate program source code into guarded commands, then into a single-assignment representation, and finally into verification conditions. These conditions are passed to an accompanying prover that judges them to be valid or invalid and may produce counterexamples to demonstrate invalidity. The prover used is Simplify [19, 22], which accepts verification conditions expressed in a first-order logic (including universal and existential quantification) with equality and untyped total functions, extended with a simple theory of arithmetic.

Simplify implements interacting decision procedures that cooperate to assess the satisfiability of a set of input formulas. In the context of a programming language, the prover has knowledge of numerical and boolean values and operators on those values. A base set of axioms describes the behavior of arrays, types, and the subtype relationship. Object identity corresponds to a simple equality relationship among untyped, uninterpreted constants. Object fields are modeled as arrays: a field named `f` is considered an array, and a field reference `o.f` is translated as the array reference `f[o]`. Other constructs are modeled as function terms.

5. TRANSLATING METHOD CALLS

5.1 Method calls within annotations

The problem at hand is to translate expressions containing method calls into the target logic described above. Information about the behavior of a method resides in the specifications of the method being called. Thus, we need to translate the specifications of the called method in a manner similar to that used to translate the calling expression.

In some cases there exists an expression whose value is the result of the method call. For instance, if a method's specifications have a postcondition of the form

`ensures \result == ... ;`

then one could extract such an expression, at least under the preconditions for which the postcondition holds. (The JML symbol `\result` represents the result returned by a method.) That expression could then be substituted for the method

call itself, after appropriate substitution of actual for formal arguments. This procedure does not work in general however. There may not be such an expression available. There may be more than one such expression available, requiring a prescient choice of the best one to substitute for the method call. In addition, the expression being substituted may contain other method calls that will themselves require substitution; the substitution procedure may not terminate if there is any recursive use of method calls in the annotations. Even without recursion, the depth of rewriting can create very large verification conditions (easily consuming 256MB on the ordinary but realistic sets of specifications contained in the JML library, in experiments with ESC/Java2).

Inlining the implementation of the called method is another approach. This can result in large, unwieldy verification conditions and does not work in the presence of recursion. It also can lose natural relationships between identical subexpressions and complicates the logical predicates of a specification with the imperative constructs of a method body. Also, we would like to be able to reason about uses of methods without needing their implementations.

The approach most appropriate to modular reasoning in the context of ESC/Java2 is to define a new uninterpreted function in the logic corresponding to each pure method used in a specification. A naive translation of methods to function terms would translate a method call of `sort()` into a function term with no arguments, namely, `(sort)`. This procedure encounters the following complications.

- As will be obvious to any Java programmer, the argument list of the method must include the receiver object (`this`), if the method is not static.¹ Thus a method call `sort()` is translated as `(sort this)`. This allows a natural distinction between the method calls `sort()` and `a.sort()`. These are translated into the terms `(sort this)` and `(sort a')`, where `a'` is the translation of the programming language expression `a`. Reasoning about aliasing is naturally handled as well, since if it is established, for example, that `(EQ this a')`, then `(EQ (sort this) (sort a'))` will immediately follow.
- Secondly, the method implementation may use fields of the receiving object that are not listed in the argument list. The values of instance fields may be considered to be implicitly included via the `this` argument, but their values then depend on the current program state.
- Most importantly, the semantics of equality among function terms is not appropriate to the reference semantics of an object-oriented language such as Java. Two function terms in the logic are equal if they have the same function symbol, the same number of arguments, and the arguments are pairwise equal. This definition of equality is fine for the immutable values of Java's primitive types, but not for reference values. Reference values referring to the same object in different program states will test equal even though their internal states may be different, since the logic used does not contain a global memory model.

The translation procedure adopted here is to include as an argument of the method a value indicating the program state in which the method is being evaluated. Remember that a method used in an annotation must be pure, that

¹Not quite as obviously, functions representing constructors of inner classes must also include a reference to the enclosing class as an argument.

is, it must not change the program state. Consequently the pre- and post-conditions are evaluated in a common state. The state constant is uninterpreted; that is, it is not used in any context other than to distinguish different program states for different method calls. With this procedure we can maintain the single-assignment mechanism adopted by ESC/Java, without introducing a full memory model into the logic, but still utilize a first-order logic for proof obligations. Having a representation of explicit state would enable a more concise translation, since then the assumptions about the behavior of methods could be universally quantified by a state variable. However, that would make for a more complex logic and in any case would be a different design than that adopted by ESC/Java and extended by ESC/Java2.

As uninterpreted values, the state constants serve simply to distinguish the values produced by different instances of method calls in annotations. In each case the single-assignment translation step ensures that each field and variable used in the pre- and postconditions of the method is translated with its current value in that state. Fields that are not mentioned in a frame condition (**assignable** or **modifies** clause) are presumed to be unchanged.

Functions in Simplify's logic are total. If, as is common, the JML specifications for a method are partial, the new function introduced by this translation will be underspecified. This is consistent with how partiality is handled elsewhere in JML.

However, there is an additional difficulty: method implementations are not necessarily guaranteed to terminate normally, returning a value. This affects how the method should be translated and is discussed in the following section.

5.2 Handling abnormal termination

In JML, a method's **ensures** postcondition states that (under the given precondition) if a method terminates normally, then the given predicate holds; the **signals** postcondition states that if the method terminates with the given exception, then its predicate holds. In JML's semantics, if a method terminates with an exception or does not terminate at all, the result value is undefined. Thus, in order to reason about the use of a method call in an annotation, we must know when a method does terminate. That is, the assumption we need to generate for a method has the form

$$(\forall \text{forall } args; \text{normalReturn}(args) ==> \text{normalPostconditionHolds}(args)).$$

Consider the code fragment of Fig. 1. Since the **diverges** predicate is **false**, we know that the method will always terminate. Similarly, the **signals** clause states that if $!(o == \text{null})$ then the method will not terminate exceptionally. Hence if $!(o == \text{null})$ the method will terminate normally; consequently the behavior of the method is defined by the assumption

$$(\forall \text{forall Example } o; !(o == \text{null}) ==> (o != \text{null} ==> \text{valueOfI}(\text{state}, o) == o.i)).$$

In general, with predicates for the **signals** and **diverges** clauses, the generated assumption has the form

$$(\forall \text{forall } args; ! (\text{signalsPredicate}(args) \vee \text{divergesPredicate}(args)) ==> (\text{precondition}(args) ==> \text{postconditionHolds}(args))).$$

However, what if the user omits the **signals** clause, as in Fig. 2? The default for an absent **signals** clause is **true**, meaning that there is no restriction if the method terminates exceptionally. The corresponding assumption is

$$(\forall \text{forall Bad } o; \text{false} ==> (\text{valueOfI}(\text{state}, o) == o.i)).$$

```
class Example {
    public int i;
}
public class Good {
    /*@ ensures o!=null ==> \result == o.i;
        diverges false;
        signals (Exception e) o == null; */
    //@ pure
    static public int valueOfI(Example o);

    //@ ensures valueOfI(o) > 0;
    public void init(Example o);
}
```

Figure 1: A class with a specification that includes normal, abnormal and non-termination conditions.

```
public class Bad {
    /*@ ensures \result == o.i; */
    /*@ pure */
    static public int valueOfI(Example o);

    //@ ensures valueOfI(o) > 0;
    public void init(Example o);
}
```

Figure 2: An inadequately specified method. Method `valueOfI` may throw an exception for any argument.

This assumption is trivially true and says nothing that defines the behavior of the `valueOfI` method.

It is not uncommon for a method's specifications to omit the specification of exceptional behavior. The specification writer is simply stating that as long as the method (or those it calls) do not throw exceptions, the result will satisfy the given postcondition. However, if a method that is used in an annotation does not provide **signals** and **diverges** clauses, the effect will be more significant. In that case, any combination of method arguments might result in non-normal termination. Thus the returned value of the method is undefined for any argument combination, and consequently no conclusion about the behavior of the method will be able to be drawn. Fortunately the result of omitting the **signals** clause will be that the postcondition of the `init` method (in the example here) will not be able to be established, rather than, say, silently stating that the method meets its specifications. However the naive specification writer might be puzzled at this behavior without some warning that the generated assumptions are trivially satisfied. In effect, for a method that is used in an annotation, a specification that omits a statement of exceptional and divergent behavior is too weak to be useful.

One might take the approach that a method used in an annotation is expected to terminate normally, at least for the preconditions under which it is called (referred to as *implicit specification of exceptions* below). However, this is equivalent to assuming JML's `normal.behavior` or

signals (Exception) false;

when a **signals** clause is missing. In contrast, the usual JML semantics is that a missing **signals** clause is equivalent to

`signals (Exception) true;`

A better approach (called *explicit specification of exceptions*) would require that any method used in an annotation have a specification for its exceptional and divergent behavior, as is shown in Fig. 1. The result of the method in question will be undefined if the method does not terminate normally. Thus the specification must at least be detailed enough to preclude exceptional or divergent behavior under the circumstances in which the method is called. One can do this by stating the conditions under which any **Exception** will not occur. If there is a predicate only for one particular exception type, there is still the possibility that for any argument some other exception might be thrown. A simple specification idiom might be that methods used in annotations only have normally terminating behavior (for the preconditions in which they are used in a specification). This sort of specification severely limits the behavior of any subtypes. This approach has the advantage of a clear semantics that is consistent with the current definition of JML; it has the disadvantages that specifications for methods used in annotations must be detailed (more so than nearly all specifications already written) and that those specifications are more constraining on subtypes.

5.3 The translation procedure

The translation, then, consists of the following steps:

- Select a unique function identifier for each method declaration in the program. Overriding method declarations have different identifiers than those of the methods they override.
- Define a unique state constant (distinct from all other constants) for each unique program state within a calling method's implementation. A new state is created after every operation with a side-effect. In practice, state constants are only needed for those points in a program where an annotation containing a method call occurs.
- Where a method call is used in an annotation expression, translate that method invocation into the logic as a function term. Use the unique identifier for the method as the function name (based on the static type of the receiver expression at the point of call). Include as arguments the translations of (a) the state constant for this program state, (b) the receiver object (if the method is not static), and (c) each of the actual arguments of the method call.
- The specifications of the called method must be turned into assumptions.
 - They are first desugared by combining preconditions and postconditions into stand-alone implications of the form ([21] describes the details):


```
ensures precondition ==> postcondition; .
```
 - Recalling the discussion of exceptional postconditions above, we use as the composite predicate the expression


```
( !signalsPredicate && !divergesPredicate ) ==>
  ( precondition ==> postcondition ) .
```
 - Any instance of `\result` is replaced by an instance of the function term, with formal names for its arguments.
 - The expression is enclosed in a universal quantification over its formal parameters.

Thus (in a class named `Z`)

```
requires i != 0;
signals (Exception) false;
diverges false;
ensures \result == i+1;
/*@ pure */ public int next(int i);
```

in a state with state constant `stateX` creates the assumption

```
(\forall Z object; (\forall int i;
  i != 0 ==> next(stateX,object,i) == i + 1)) .
```

Since values (e.g. of fields) are not extracted out of a program state, there is no quantification over the state constant. Instead the assumption above is repeated with a different state constant in each context where the method is called and any free variables are translated in the context of that call. Also, recall that since the method being used in the annotation must be pure, the preconditions, diverges conditions, signals conditions, and normal postconditions are all evaluated in the same state.

- In order to connect the use of a method call in the program with its use in an annotation, an implicit postcondition is added that equates the result of the method to the term representing the method (e.g. `ensures \result == m(...)`). This adds a corresponding assumption about the result of a method call in the program source code.
- If the called method has no specifications, then no other assumptions are introduced describing the behavior of the method. This will limit the conclusions that can be drawn. The only connection between the value of a method call in one program state and the value in another program state is the definition of the value through the method's specifications.
- JML allows annotations to appear in the body of a method as well; `assert` statements are one example. These are translated in the same way as postconditions; they simply use the appropriate state constant. Since loops are partially unrolled by ESC/Java, they can be handled without additional special treatment.

If there is more than one instance of the same method call within a given program state, those calls are translated in the same way, enabling the prover to identify their return values as equal (even in the absence of specifications). If a method call in the postcondition occurs within an argument of `\old`, indicating it is to be evaluated in the pre-state, then it will be translated using the state constant for the pre-state.

Appendix A contains a discussion of the details of an actual example translation.

6. APPLICATION TO OTHER ANNOTATION CONSTRUCTS

With translation of method calls to an underlying first-order logic enabled in ESC/Java2, several other specification constructs can be readily translated and used in static checking as well. These are described briefly in this section.

6.1 Constructor calls

Constructor calls can be treated as calls of static methods. That is, they do not depend on an implicit `this` argument. If they are constructors of a Java inner class, they will depend on an implicit argument representing an instance of the enclosing class. Since some of the arguments may be

reference values, the translated function must also have a state constant as an argument. The assumptions about the constructed value are formed from the specifications of the constructor declaration.

Constructor calls are different than method calls in that they dynamically allocate new objects on the heap. Thus the result is a reference value not equal to any previous reference value. ESC/Java2 (following ESC/Java) provides axioms concerning allocation that ensure this behavior, but those are beyond the scope of this paper.

6.2 Array allocation

Translating expressions such as `new int[9]` that allocate new arrays is quite straightforward. These expressions do not depend on the current state nor on any implicit receiver argument. Consequently a single function whose arguments include the dimensions of the array and the type of its elements is all that is needed. Just as for constructors, axioms regarding allocation are required, so that the value produced by a new array expression is known to be different than any previously produced reference value. Axioms about the dimensions, type, and initial values of the array are also needed. ESC/Java included such a function and axioms in its built-in axiom set, as does ESC/Java2.

6.3 Quantified expressions

Besides the usual universal and existential quantified expressions, JML also defines the quantifiers `\min`, `\max`, `\sum`, `\prod`, and `\numof`. For example, the value of the expression `(\min int i; i <= 0 && i < 10; p(i))` is the smallest value of `p(i)` for `i` in the given range.

The translation of each of these consists of syntactically replacing the expression with a skolemized function call (whose name is unique) and introducing appropriate assumptions about the function. Implicit receiver and state arguments are also needed, as described previously. If the quantifier is within the scope of another quantified expression, there will also need to be function arguments for any bound variable used in the replaced expression.

One must also introduce assumptions concerning the value of this introduced function, corresponding to the value of the original quantified expression. For example, the assumptions associated with

```
( \min decl; range-predicate; expr)
```

are

```
( \exists decl; range-predicate) ==>
```

```
( \exists decl; range-predicate && MIN(...) == expr)
```

and

```
( \forall decl; range-predicate ==> MIN(...) <= expr),
```

with suitable universal quantification and where `MIN(...)` is replaced by the actual skolem function call expression.²

6.4 Model variables

Model variables are declarations of fields within annotation comments that are not treated as fields of the object. Rather a model variable is associated with a representation, typically in terms of the internal state of the object. The model variable may be used in annotations as an abstract representation, or *model*, of some quantity related to the ob-

ject at hand. For example, Java's `java.util.Collection` interface, which has no implementation, might nonetheless declare a model variable such as

```
public model instance non_null JMLObjectBag
    theCollection;
```

(using one of JML's mathematical library classes). In this example, any implementation of the `Collection` interface is modeled with a field of type `JMLObjectBag`. Then, even in the absence of an implementation of the method, a routine such as `isEmpty()` could use the model variable in the specification

```
ensures \result <==> theCollection.isEmpty();
```

Model variables in JML may have a functional representation, a predicate representation, or no representation at all. Work on translation of model variables in JML for the LOOP tool occurred concurrently with the work in this paper and is discussed in [3]. We came to a similar solution and offer some additional observations here.

6.4.1 Functional representations

JML denotes a functional representation by the syntax

```
//@ represents x <- expression ;
```

There is a specific value, provided by the expression, for the model variable (in a given program state). This expression could be simply substituted for occurrences of the model variable, as stated by Breunese and Poll [3]. However, this is successful only in simple cases. If there is heavy use of model variables, the nested substitutions can be quite deep. Furthermore, JML allows multiple redundant representations, requiring a choice of which to use. Finally, direct or mutual recursion would prohibit simple substitution. In this implementation in ESC/Java2, representing functional model variables by method calls was the better solution.

6.4.2 Predicate representations

JML also allows the values of model variables to be specified with a predicate representation:

```
//@ represents x \such.that predicate ;
```

In this case the predicate does not necessarily give an executable expression for the model variable and may not even constrain the model variable to a single value. As Breunese and Poll point out, if there are no possible values satisfying the predicate, inconsistency in the generated assumptions can result, if appropriate care is not taken.

In this case, we represent the model variable by a method call, with state and receiver arguments as discussed above; the `\such.that` predicate becomes an assumption.

6.4.3 Model variables with no representation

A model variable, particularly in an abstract class or interface, may have no representation at all. It may be used in the specification of various methods, but its representation would be supplied by derived classes that implement the interface. Fig. 3 shows an example of such an interface. In this situation, the model variables are still translated as method calls, but now there are no assumptions generated from represents clauses. Instead, only the pre- and post-conditions of methods whose specifications mention the model variable provide information about the behavior of the variable.

Representation-less model variables do pose a challenge in the translation and checking of method bodies. If the model variable is **assignable** for a given method, then its value may change in the course of execution of the body of that method. But without a representation, there is no way to

²The value of the quantified expression when the range predicate is empty is not handled here. JML currently defines this as the largest value of the type of the quantified expression (for `\min`, and the smallest such value for `\max`). It might also be considered as undefined.

```

public interface NoRep {
    //@ public model String outputText;
    //@ invariant outputText != null;

    /*@ assignable outputText;
       ensures outputText.equals(
           \old(outputText) + s); */
    public void print(String s);
}

```

Figure 3: The specification and code for the interface `NoRep`, demonstrating a model variable with no representation.

reason about its value at points within the body.³

6.4.4 Unmodified model variables

There is a special case of using model variables that allows a simplification in their translation. If a model variable is not implicitly or explicitly mentioned in a method’s **assignable** clauses⁴, then its value may not change during the course of the body of the method. Thus for that method’s body and the checking of its pre- and post-conditions, the model variable may be treated as a constant. The value of the model variable can be determined or constrained by the value of its representation evaluated in the pre-state (or any state) of the method.

6.5 Exceptional behavior

Constructor calls in annotations have the same problems with exceptional behavior as do method calls and they can be handled in the same way using the conventional specifications. However, quantified expressions and model variables both utilize expressions that may throw exceptions and neither have the syntax that method declarations have to specify the conditions under which exceptional behavior may or may not happen. How to handle exceptional behavior in these cases remains an unresearched question.

7. FUTURE WORK: IMMUTABLE VALUES

The complication of introducing state constants as additional arguments is a result of the underlying logic using uninterpreted values for reference quantities in the programming language. Since these reference values refer to mutable objects, one must retain a state value to indicate which state of the object is meant. If all of the arguments were primitive type values such as integers and booleans, then a state value would not be needed. These values of non-reference types are immutable: if two values compare equal, they will always have the same properties in any program state.

Some reference types are also immutable. For example, values of `java.lang.String` that compare equal (with `==`) will always have identical properties even in future program states. One requirement for the values of a type to be immutable is that no method of the type modify the internal state of the object; this condition is assured to hold if all methods of the type and any subtype are **pure**. However, it

³JML’s **in** clause, not discussed here, does provide some information on which other fields of the class contribute to the value of a model variable and consequently at which program points the value of a model variable might change.

⁴including not specified by a JML datagroup

is also necessary for immutability that the internal representation not contain mutable objects and that the representation not be exposed in a way that the internal state could be modified by some external means.

Reasoning with immutable objects is potentially simpler and more efficient than with typical mutable objects. JML includes a library of classes representing mathematical concepts useful in specifying classes [15]; they are heavily used in the specification of JML code and sample classes and in JML’s specifications of Java classes. Checking these specifications might be more straightforward if it could be established that instances of these classes are immutable. For this to be possible, we need a set of sufficient conditions for immutability that can be statically checked, a proof of soundness regarding immutability, and a demonstration by a working implementation of the utility of immutable classes in program verification.

8. CONCLUSIONS

There are by now several tools that statically check specifications against source code by logical reasoning. Java is a common but not the exclusive source language. The target logics and the accompanying provers vary widely: for example, Krakatoa [17] uses the Coq proof assistant, Jive [18] and LOOP [10, 12] use PVS [20], KeY [2] uses OCL and its own prover, and JACK [4] interfaces with Atelier B, Simplify, Coq and PVS.

It is also typical to carefully specify the mapping of the semantics of the source language into the target logic. However, we know of no published treatment describing the mapping of the specification language, particularly of method calls, into logical assertions. The LOOP tool has a comprehensive representation of Java’s memory model and the program translation and all work in PVS focuses directly on this model. The LOOP tool permits one to specify and reason about specifications that use pure methods. To do so, one either uses the specifications alone, in a manner similar to that which is described in this paper, or one uses the implementations of the methods and symbolically executes them within PVS. The latter approach is implied, for example, in [11], though it notes that the semantics of method invocations in specifications is still unclear. Similarly, Krakatoa defines all logical predicates in the context of a global heap; it also introduces a new assumption to encapsulate the behavior of pure methods. KeY allows simple query functions that do not cause exceptional behavior.

Though there are similar aspects among these approaches, the solution used by ESC/Java2 for translating method calls demonstrates a straightforward translation in the context of a general purpose first-order logic and prover. In doing so it maintains the design philosophy and usefulness of the original ESC/Java tool, while adding the capability of using method calls in annotations. The discussion above also illustrates the complexities of handling potentially non-normally terminating functions in a specification language. It appears that the tools above that handle method calls all implicitly use the implicit specification of exceptions of section 5.2. ESC/Java2 has been successfully using this approach in its recent alpha releases and is in the early stages of experimentation with the preferred explicit specifications.

9. ACKNOWLEDGMENTS

Thanks to Joseph Kiniry for comments on an early version

of the paper and for some material on LOOP. Kiniry also is a partner in the support, maintenance and development of ESC/Java2. Thanks also to Gary Leavens for comments that improved the discussion overall, as well as to the reviewers.

10. 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] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 2003. To appear.
- [3] C.-B. Breunese and E. Poll. Verifying JML specifications with model fields. In *Formal Techniques for Java-like Programs. Proceedings of the ECOOP'2003 Workshop*, pages 51–60, 2003. Technical Report 408, ETH Zurich.
- [4] L. Burdy and A. Requet. JACK: Java applet correctness kit. In *Proceedings, 4th Gemplus Developer Conference*, Singapore, Nov. 2002.
- [5] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. Technical report, University of Nijmegen, 2004. NIII Technical Report NIII-R0413.
- [6] Unpublished reports about ESC/Java's design are available online at <http://www.research.compaq.com/SRC/esc/design-notes/>.
- [7] C. Flanagan, K. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata, et al. The Extended Static Checker for Java, 1999. See <http://research.compaq.com/SRC/esc/>.
- [8] 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.
- [9] E.-M. Hubbers, M. Oostdijk, and E. Poll. Implementing a Formally Verifiable Security Protocol in Java Card. In D. Hutter, G. Müller, W. Stephan, and M. Ullmann, editors, *Proceedings of the First International Conference on Security in Pervasive Computing*, volume 2802 of *Lecture Notes in Computer Science*, pages 213–226. Springer-Verlag, 2004. March 12–14, 2003, <http://www.dfki.de/SPC2003/>.
- [10] B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 58:61–88, 2004.
- [11] B. Jacobs, J. Kiniry, and M. Warnier. Java Program Verification Challenges. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, volume 2852 of *Lecture Notes in Computer Science*, pages 202–219. Springer, Berlin, 2003.
- [12] B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer, 2001.
- [13] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.
- [14] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06t, Iowa State University, Department of Computer Science, Dec. 2002. See www.jmlspecs.org.
- [15] 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. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects: First International Symposium, FMCO 2002, Leiden, The Netherlands, November 2002, Revised Lectures*, volume 2852 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2003.
- [16] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106. ACM, Oct. 2000.
- [17] C. Marché, C. Paulin, and X. Urbain. The Krakatoa tool for JML/Java program certification. Available at <http://krakatoa.lri.fr>, 2003.
- [18] J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description. Available at <http://softech.informatik.uni-kl.de/old/en/publications/jive.html>, 2000.
- [19] C. G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, Stanford, CA 94035, 1980. Available from University Microfilms.
- [20] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. Henzinger, editors, *Computer Aided Verification*, number 1102 in *Lecture Notes in Computer Science*, pages 411–414. Springer, 1996.
- [21] A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report 00-03d, Iowa State University, Department of Computer Science, July 2003.
- [22] Unpublished reports about Simplify are available online at <http://research.compaq.com/SRC/esc/Simplify.html>.

APPENDIX

A. AN EXAMPLE

This section shows an example translation of some simple code contrived to show the translation concisely. The guarded commands of Fig. 4 are a subset of the commands generated by the translation of the method `m` in the code of Fig. 5. Though the guarded commands are shown in the internal language used within ESC/Java, the outlines of the translation are apparent.

- The ASSUME statement in line (2) states the assumption that the precondition in line A of Fig. 5 holds. Note the function form used to represent the call of `Trans.p`: it has the unique name `Trans.p.7.2` and it contains a state constant, `this` parameter, and the actual arguments of the call. (The `lblneg` expression simply gives a label to a predicate for use in error messages.)

```

1) ASSUME (\forallall anytype brokenObj, c, s; boolAnd(boolOr(select(b:5.10, brokenObj),
    boolEq(Trans.p.7.2(state, brokenObj, c, s), integralEQ(select(z:3.15, c), 1))),
    is(Trans.p.7.2(state, brokenObj, c, s), \type(boolean))));
2) ASSUME (\blneg Pre:0.13.6 Trans.p.7.2(state, this, c:16.18, "A"));

3) ASSUME (\forallall anytype brokenObj<34>, c<1>, s<1>; boolAnd(boolOr(select(b:5.10, brokenObj<34>),
    boolEq(Trans.p.7.2(state, brokenObj<34>, c<1>, s<1>), integralEQ(select(z:3.15, c<1>), 1))),
    is(Trans.p.7.2(state, brokenObj<34>, c<1>, s<1>), \type(boolean))));
4) ASSERT (\blneg Assert@17.8 Trans.p.7.2(state, this, c:16.18, "B"));

5) ASSUME anyEQ(RES-18.8:18.8, Trans.p.7.2(state, this, c:16.18, "Q"));
6) ASSUME boolImplies(anyEQ(EC-18.8:18.8, ecReturn), boolOr(select(b:5.10, this),
    boolEq(RES-18.8:18.8, integralEQ(select(z:3.15, c:16.18), 1))));
7) ASSUME anyEQ(b:18.4, store(b:5.10, this, RES-18.8:18.8));

8) ASSUME (\forallall anytype brokenObj<35>, c<2>, s<2>; boolAnd(boolOr(select(b:18.4, brokenObj<35>),
    boolEq(Trans.p.7.2(state:18.6, brokenObj<35>, c<2>, s<2>), integralEQ(select(z:3.15, c<2>), 1))),
    is(Trans.p.7.2(state:18.6, brokenObj<35>, c<2>, s<2>), \type(boolean))));
9) ASSERT (\blneg Assert@19.8 boolAnd(Trans.p.7.2(state:18.6, this, c:16.18, "C"),
    Trans.p.7.2(state:18.6, this, c:16.18, "D")));

10) ASSUME (\forallall anytype brokenObj<36>, c<3>, s<3>; boolAnd(boolOr(select(b:18.4, brokenObj<36>),
    boolEq(Trans.p.7.2(state-20.8:20.8, brokenObj<36>, c<3>, s<3>), integralEQ(select(z:3.15, c<3>), 1))),
    is(Trans.p.7.2(state-20.8:20.8, brokenObj<36>, c<3>, s<3>), \type(boolean))));
11) ASSERT (\blneg Assert@21.8 Trans.p.7.2(state-20.8:20.8, this, RES:20.8, "E"));

12) ASSUME (\forallall anytype brokenObj<37>, c<4>, s<4>; boolAnd(boolOr(select(b:18.4, brokenObj<37>),
    boolEq(Trans.p.7.2(state-20.8:20.8, brokenObj<37>, c<4>, s<4>), integralEQ(select(z:3.15, c<4>), 1))),
    is(Trans.p.7.2(state-20.8:20.8, brokenObj<37>, c<4>, s<4>), \type(boolean))));
13) ASSERT (\blneg Post:15.6@22.2 boolImplies(
    boolAnd(anyEQ(ecReturn, ecReturn), Trans.p.7.2(state@pre, this, c:16.18, "A")),
    Trans.p.7.2(state-20.8:20.8, this, c:16.18, "Z")))

```

Figure 4: A subset of the guarded commands generated from the translation of the code in Fig. 5.

```

public class Trans {
  public static class C {
    public int z;
  }
  boolean b;

  @@ diverges false;
  @@ ensures b || \result == (c.z == 1);
  @@ signals (Exception) false;
  @@ pure
  public boolean p(C c, String s);

  @@ requires p(c,"A"); // A
  @@ modifies b;
  @@ ensures p(c,"Z"); // Z
  public void m(C c) {
    @@ assert p(c,"B"); // B
    b = p(c,"Q"); // Q
    @@ assert p(c,"C") && p(c,"D"); // C
    c = new C();
    @@ assert p(c,"E"); // E
  }
}

```

Figure 5: A somewhat contrived example to illustrate the translations of method calls.

- The assumption about the value of this call of `Trans.p.7.2` is provided in the ASSUME statement in line (1). It is quantified over the object and the two formal arguments of the method call. It makes the assumption that either `b` is true or the returned result is equivalent to whether the `z` field of the object `z.13.15` has the value 1; it also assumes that the type of the result is `boolean`.

The same state constant is used in lines (1) and (2).

- The ASSERT at line (4) is the translation of the `assert` statement at line B.
- The ASSUME at line (3) is the assumption associated with line (4) for the call of `Trans.p` in the `assert` statement at line B. There has been no change of state as yet, so the same state constant is used. In fact, this ASSUME is redundant with that in line (1) and could be omitted by an appropriate optimization.
- The translation of line Q generates lines 5-7. Line (5) shows the assumption that equates the value of the function term `Trans.p.7.2` to a temporary variable (`RES-18.8:18.8`), which is the result of the method call within the program; in line (6) the method specifications are applied to that variable and in line (7) it is used to create the value for the new logical variable for `b`. The Java assignment statement also causes a state change.
- Lines (8) and (9) are the translation of the `assert` statement of line C. Note that both translations use the same, new state value as well as the new value of `b`.
- Lines (10) and (11) are the translation of the `assert` statement of line E. There has been another state change and a new variable representing `c` (namely `RES:20.8`).
- Finally, lines (12) and (13) represent the postcondition. Per JML's semantics, it uses the value of `b` in the post-state, but the value of the formal argument `c` from the pre-state. In line (13), the precondition is evaluated with the pre-state state constant and the post-condition with the post-state state constant (`state@pre` and `state` are later equated).