

Reasoning with specifications containing method calls and model fields

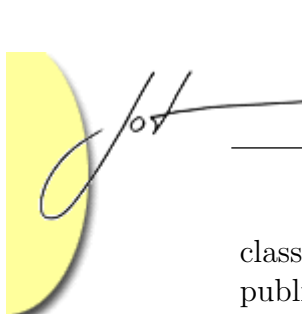
David R. Cok, B65 MC01816, Eastman Kodak Company, Research & Development Laboratories, Rochester, NY 14650-1816, USA, cok@frontiernet.net

Allowing abstraction in program specifications increases modularity and comprehensibility and is as important in specifications as it is in the program itself; two such abstraction mechanisms are method invocations and model fields. 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. Furthermore, translating model fields into verification conditions exposes the non-trivial interactions between frame conditions and the model representations. The ESC/Java2 tool has been able to achieve a practical translation of method invocations and model fields within the design constraints of its parent tool, ESC/Java. Furthermore, the techniques used are applicable to other specification constructs such as generalized quantifiers.

1 INTRODUCTION

Research and practical capability in program verification is advancing significantly with the help of clearer semantics, evolution of languages and tools, and experience with industrial-scale software systems. One research thread in program verification combines (a) specifications indicating the intent of a program with (b) tools that check those specifications either dynamically at run-time or statically without needing to execute the program. 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. ESC/Java [9, 10] is a successful, publicly available, static checker, but it does not allow such abstraction. This paper discusses the implementation of an extension to ESC/Java, called ESC/Java2 [7, 8], 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, 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.

Model fields [6, 17], though not present in Java itself, also provide a useful abstraction. They can be used to model the behavior of interfaces and abstract



classes that do not have an implementation and they can be used to separate the public and private aspects of a class's specification. However, model fields exhibit properties both of fields and of methods; the interactions of these properties lead to complications and potential logical inconsistencies.

The solutions described in this paper were implemented and tested using the Java Modeling Language[1], the ESC/Java2 tool, and its back-end theorem prover, Simplify[25, 28]. The Java Modeling Language, described in section 2, is a specific and fairly rich language for expressing specifications about Java programs; for example, it supports both method calls in annotations and model fields. ESC/Java2, introduced in section 3, is a tool that checks these specifications against the program's implementation. Section 4 describes Simplify, the theorem prover used by ESC/Java2 to check the various logical conditions corresponding to the specifications. Section 5 describes a solution for translating method calls and issues arising from exceptional termination, with an example in the Appendix. The issues and implementation of model fields are described in section 6. Section 7 discusses applications to other specification language features and some outstanding issues. The paper ends with descriptions of some future work, related work and conclusions, and acknowledgements in sections 8-10.

2 THE JAVA MODELING LANGUAGE

The Java Modeling Language (JML) has by now been described in several publications [1, 16, 17] and that full description will not be repeated here. The discussion in this paper can be illustrated using simple preconditions and postconditions.

- A **behavior** keyword introduces a *specification case*, which is a set of annotations controlled by a conjunction of one or more preconditions.
- A **requires** keyword followed by a predicate declares a precondition for a method.
- A **ensures** keyword followed by a predicate declares a postcondition for a method that holds when the method terminates normally.
- A **signals** keyword declares a postcondition that holds if the method exits with the given exception.
- A **diverges** keyword declares a condition that must be true of the program state at the time the method is called (the pre-state) if the method never terminates.
- A **assignable** keyword declares a frame condition, namely those fields that may be assigned to (or modified) by a method.
- The **pure** modifier indicates a method that has no side-effects on the program state.
- The `\result` identifier denotes the return value of a method.
- The `\old` identifier denotes an expression evaluated in the pre-state of a method.
- The **model** modifier indicates a method or field that is defined in annotations



only.

- A **represents** clause indicates a constraint for the value of a model field.
- The **in** and **maps** clauses define additional contents of a datagroup corresponding to a model field.
- An **assert** statement indicates an annotation predicate that is to be verified at a given point in the program text.

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. Fig. 1 shows a class with specifications containing method invocations. In it the public specifications of public methods use pure public methods and not private implementation details. The combination of JML's visibility modifiers, pure methods, datagroups, and model methods and fields allows a separation of public information from private implementation detail.

3 ESC/JAVA2

The ESC/Java2 tool [7, 8], an extension of ESC/Java [9, 10], 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 decision procedure 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, 19]. Consequently, the original ESC/Java was a research success and was also successfully used by other groups for a variety of case studies [11, 12].

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 or model fields 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.

```
public class MethodAnnotations {

    private final int[] array = new int[20];
        //@ in state; maps array[*] \into state;
    //@ private invariant array != null;

    private int size; //@ in state;

    //@ public model JMLDataGroup state;

    //@ private behavior
    //@ ensures \result == array.length;
    //@ signals (Exception) false;
    //@ public model pure int capacity();

    //@ private behavior
    //@ ensures \result == size;
    //@ signals (Exception) false;
    public /*@ pure @*/ int size() { return size; }

    //@ public invariant size() >= 0 && size() <= capacity();

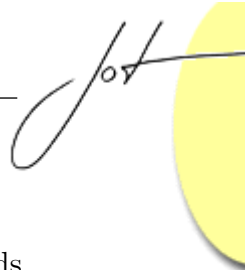
    //@ public behavior
    //@ requires size() < capacity();
    //@ assignable state;
    //@ ensures size() == \old(size()+1);
    //@ signals (Exception) false;
    public void push(int i) {
        array[size++] = i;
        //@ assert size() == \old(size())+1;
    }

    //@ public behavior
    //@ requires size() > 0;
    //@ assignable state;
    //@ ensures size() == \old(size()-1);
    //@ signals (Exception) false;
    public int pop() {
        return array[--size];
    }
}
```

Figure 1: An example class with specifications containing method invocations.

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 find all specification errors or to fully represent Java semantics. ESC/Java2 has continued that spirit, though some unsound aspects have been corrected.

It is important to note that ESC/Java does not incorporate an explicit notion of program state in the logical structure that represents the program. Instead, a single-assignment calculus represents new values of program variables by new identifiers in the logic. This simplifies the logic and makes it much easier to reason about variables that remain unchanged; however, it also adds difficulty to the translation of loops and 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 theorem prover that may judge them to be valid or invalid and may produce counterexamples to demonstrate invalidity. The tool used is Simplify [25, 28], 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 and cooperating decision procedures for some subdomains of first-order logic, along with heuristics to handle quantification, in order 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]`. The various operators and built-in functions of Java are modeled as function terms.

5 TRANSLATING USES OF METHOD CALLS IN ANNOTATIONS

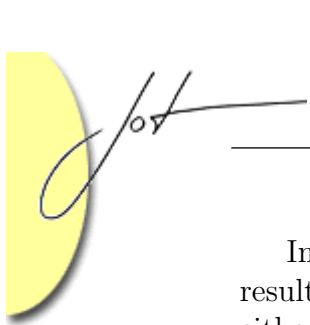
Method calls within annotations

The problem at hand is to translate specification 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

$$\text{ensures } \backslash\text{result} == \dots ;$$

then one could extract such an expression, at least under the preconditions for which the postcondition holds. 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).



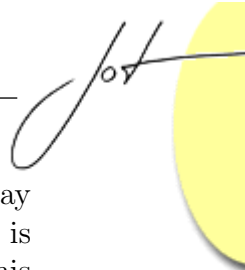
Inlining the implementation of the called method is another approach. This can result in large, unwieldy verification conditions and does not work in the presence of either recursion or overriding methods. It also can lose natural relationships between identical subexpressions and complicates the logical 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 and to the implementation inherited from ESC/Java 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()` in Java is translated into a term of the logic as $(\text{sort } \text{this})$. This allows a natural distinction between the method calls `sort()` and `a.sort()`. These are translated into the terms $(\text{sort } \text{this})$ and $(\text{sort } a')$, where a' is the translation of the programming language expression `a`. Reasoning about aliasing is naturally handled as well: if it is established, for example, that $(EQ \text{ this } a')$, then $(EQ (\text{sort } \text{this}) (\text{sort } a'))$ 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 by way of 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 adds a state argument to the function representing a method; the value used for that function argument is a unique constant indicating the program state in which the method is being evaluated. It would be equivalent to use a different name for the new function (rather than the same name with a different state constant as argument) in each new context in which it was being invoked. Remember that a method used in an annotation must be pure; that is, it must not change the program state, at least in ways that are observed by

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



the program.² Consequently the pre- and post-conditions of the pure method may be 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. 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 from state to state.

JML specifications are often partial. That is, the behavior of the method is not specified for all combinations of input. They may also be partial in the sense that there is no value of the type of the result for the given arguments (such as for division by 0). This is the case for conditions in which the method does not terminate or terminates with an exception.

On the other hand, functions in Simplify's logic are total. The logical term introduced as the translation of a pure method will simply be undefined for those function arguments for which the JML specification does not normally terminate or the result is not specified. This is consistent with how partiality is handled elsewhere in JML [18]. However, this partiality of specification affects how the method should be translated and the deductions that can be inferred from its logical representation; this problem is discussed in the following section.

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 normally. That is, the assumption we need to generate for a method has the form

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

²Which changes are considered observable and how to control what is observed are matters of current research. JML currently does allow allocations of new objects to be performed by pure methods; the state issues regarding those allocations are already handled within the logic of ESC/Java by a separate 'allocation' state variable.

```

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) throws IllegalArgumentException;

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

```

Figure 2: A class with a specification that includes normal, abnormal and non-termination conditions.

in which $normalReturn(args)$ is true precisely when the method always returns normally for the given arguments. For those argument combinations for which the method might possibly not return normally, the result must be undefined.

Consider the code fragment of Fig. 2. Since the `diverges` predicate is `false`, we know that the method `valueOfI` will always terminate. Similarly, the `signals` clause states that if `!(o == null)` then the method will not terminate exceptionally.³ Hence if `!(o == null)` the method will terminate normally; consequently the behavior of the method is defined by the assumption

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

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

$$(\forall \text{args} ; !(signalsPredicate(args) \ || \ divergesPredicate(args)) \implies (precondition(args) \implies postconditionHolds(args))).$$

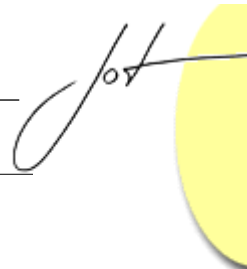
However, what if the user omits the `signals` clause, as in Fig. 3? The default for an absent `signals` clause is `true`, meaning that there is no restriction if the method terminates exceptionally. The corresponding assumption is

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

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, as in Fig. 3. The specification writer is simply stating that as long as the method (or those it calls) does 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

³JML only specifies exceptional behavior for those cases in which an object of type (or subtype of) `java.lang.Exception` is thrown. In particular, there are no assertions about behavior if a `java.lang.Error` is thrown. Other non-Exception `Throwable` objects are also ignored.



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

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

Figure 3: An inadequately specified method. Method `valueOfI` may throw an `IllegalArgumentException` exception for any value of the argument.

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*). 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`.

An alternate approach⁴ is that a method declared `pure`, which must be the case for methods used in annotations, must always terminate normally, for any precondition, in addition to having no side effects. This avoids the problem with exceptional termination by definition, but significantly limits the Java methods that might otherwise be considered `pure`.

Another 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. 2. 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 no `Exception` will 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); using JML's `normal_behavior` specifications would accomplish this. This approach has the advantage of a clear semantics that is consistent with the current definition of JML; it has the disadvantage that specifications for methods used in annotations must be more detailed than

⁴under discussion on the `jmlspecs-interest` mailing list at http://sourceforge.net/mail/?group_id=65346.

they might otherwise be written.

The requirement of extra detail is mitigated by the following recent evolution of JML's semantics. The default `diverges` clause is `diverges false;`. That is, an omitted `diverges` clause is interpreted as requiring termination. The defined default for an omitted `signals` clause is `signals (Exception) true;`, which allows an Exception to be thrown. On the other hand, ESC/Java⁵ uses the `throws` clause of the method declaration to define its default. ESC/Java and ESC/Java2 will only allow those exceptions to be thrown that are explicitly listed in the `throws` clause, not even permitting additional unchecked exceptions such as a `RuntimeException`. Thus if a method declaration has an empty `throws` clause, then (for ESCJava(2)) the default `signals` clause is `signals (Exception) false;`; this is consistent with the desired behavior above, namely, a tight restriction on non-normal termination of the method. Recent discussions⁶ have proposed that JML adopt this same default behavior. In that case the *explicit specification of exceptions* will also be the natural default.

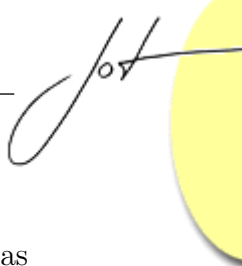
The translation procedure

The translation, then, consists of the following steps:

- Select a unique function identifier for each method declaration in the program. A family of overriding method declarations have the same identifier. One could use distinct identifiers for different overriding methods and switch among them based on the dynamic type of the receiver argument. It appears, however, that using the same identifier for all members of an overriding family of methods and then adding tests on the dynamic type into each specification's preconditions is a simpler design.
- 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. 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 about the behavior of the method. In the case of a family of overriding and overridden methods, the specifications must have an additional antecedent of the form `(this instanceof T)` for the type T in which the specification appears.
 - They are first desugared by combining preconditions and postconditions into stand-alone implications of the form ([27] describes the details):

⁵as I was reminded by a referee

⁶since the original workshop presentation, on the `jmlspecs-interest` mailing list



`ensures precondition ==> postcondition; .`

- Recalling the discussion of exceptional postconditions above, we use as the composite predicate the expression

$$(\text{!signalsPredicate} \ \&\& \ \text{!divergesPredicate}) \implies (\text{precondition} \implies \text{postcondition}).$$

- 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 the specification (in a class named `Z`)

```

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

```

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

$$(\forall \text{all } Z \text{ object}; (\forall \text{all } \text{int } i; i \neq 0 \implies \text{next}(\text{stateX}, \text{object}, i) == i + 1)) .$$

Since values (e.g. of fields) are not extracted out of a memory model 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; 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.

- A pure Java method `m` may be used in a Java program and in an annotation. A program fragment such as

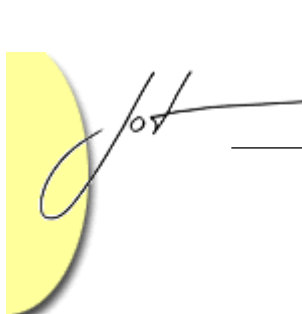
```

    int i = m();
    //@ assert i == m();

```

should be provable (as long as `m` is presumed to terminate normally and be deterministic), even without specific specifications about `m`'s result. In order to connect such use of a method call in an annotation with its use in the program, an implicit postcondition could be added to the method's specification that equates the result of the method to the term representing the method (e.g. `ensures \result == m(...)`). This would add a corresponding assumption to the verification conditions about the result of a method call in the program source code. This assumption must not be used when the result is a fresh, newly allocated object. The usefulness of this assumption in other cases is being evaluated.

- 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, because 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 pre- and 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 that 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.

An issue: are pure methods deterministic?

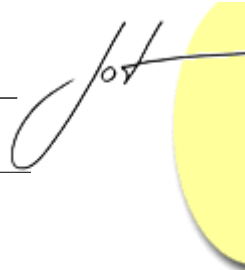
Using method invocations within annotations raises another issue: what is the relationship between the results of two different invocations (in annotations) of a pure method that occur within the same program state?⁷ That is, can we assume that `assert m() == m();` will succeed?

The value of a method such as `m()` is constrained by the method's postconditions but may not be constrained to a single value. Assuming that the assertion above will be true (if the value of `m()` is defined) corresponds to assuming that the method execution is deterministic and not dependent on hidden volatile variables. In most specification situations it would be quite difficult to specify straightforward behavior without this assumption. ESC/Java2 currently presumes such deterministic behavior for pure methods; it is easy to model non-determinism for non-pure methods. JML does not have syntax to distinguish deterministic and non-deterministic methods.

This distinction is only relevant to those methods used in annotations, since we want assertions such as the one above to be valid in annotations. Reasoning about Java methods themselves is performed using only the method's explicit specifications.

One situation in which `m() == m();` ought to be false is when `m()` returns a newly allocated object. The two invocations ought to produce two distinct instances of the appropriate type. Since the program state does change because of the method call, at least in that the heap is changed, one can debate whether `m()` ought to be considered pure. JML currently allows this fairly common situation to be considered pure; various functions (e.g. `toString()`) that return `String` objects are good

⁷Thanks to Peter Müller and Erik Poll for emphasizing the importance of determinism in using methods in specifications; additional discussion of this point and of possible JML syntax is present in the Jml-interest mailing group archives.



```
abstract class AbstractList {

    //@ model public int size; //@ in isEmpty;
    //@ model public boolean isEmpty;
    //@ public represents isEmpty <- (size == 0);
    //@ public invariant size >= 0;

    //@ assignable isEmpty;
    //@ ensures isEmpty;
    abstract public void clear();

    //@ assignable size; // but isEmpty is not assignable
    //@ ensures size <= \old(size);
    //@ ensures \old(size) > 0 ==> size > 0;
    abstract public void shrink();
}

class ListImpl extends AbstractList {

    //@ pure
    private int length() { ... }

    protected void clear() { ... }

    protected void shrink() { ... }

    //@ private represents size <- length();
}

public class Client {

    //@ requires list != null;
    //@ modifies list.size;
    public void m(AbstractList list) {
        list.shrink();
    }
}
```

Figure 4: An example of the use of model fields.

examples of methods that are pure except for allocation and would not satisfy the above equality.

6 TRANSLATING USES OF MODEL FIELDS IN ANNOTATIONS

Syntax

The syntax related to the declaration and use of model variables is shown in Fig. 4. Within annotations, field declarations can appear as `model` declarations. These fields simply represent values that are constrained by expressions or predicates given in `represents` clauses. A `represents` clause may appear in either the class or interface declaring a model field or in a subclass or subinterface.

Model field names may appear in assignable clauses. In that case the name stands for a set (a *datagroup*) containing field designations and the contents of other datagroups. A field may be declared as a member of a datagroup using the `in` or `maps` (not shown) declarations; datagroups may be extended in subtypes. In Fig. 4,

the `size` model field is a member of the `isEmpty` datagroup, as well as automatically being a member of the `size` datagroup.

The specification of the method `AbstractList.clear()` declares that the method may change `isEmpty` and in fact ensures that its value is `true` when the method terminates. Since `size` is a member of the `isEmpty` datagroup, it also might be modified in the execution of `clear()`. In contrast, `AbstractList.shrink()` may modify `size` but may not modify `isEmpty` (so the second `ensures` clause for `AbstractList.shrink()` is redundant).

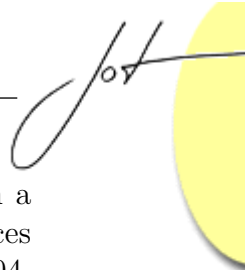
Interaction of frame conditions and model field representations

The implementation of model fields, pure methods, and checking of frame conditions in JML and ESC/Java has brought to the fore current problems in JML with the interactions between frame conditions and model fields. As shown in the example, model fields may be listed in frame conditions, either explicitly or implicitly as members of a listed datagroup, just as Java fields may be. If a model field is contained in the frame condition, its value may change during the execution of the method; if it is omitted from the frame condition, its value must not change during the execution of the method.

However, depending on the representation of the model field, its value may be affected by the values of fields that are not members of its datagroup and may not even be members of the class hierarchy that contains the model field. Changes to such fields may change the abstract value of the model field; the methods causing such changes may not even be aware of the existence of the model field in question and consequently will not list it in its `modifies` clause.

On the other hand, if a model field is specified as modifiable by the body of a method, we need to be able to reason about its changing value at each assignment or method call within the body of the method. A method call is accompanied by its own specifications. If those state the effect of the method call on the value of the model field, appropriate reasoning can be performed; if the method call's specifications say nothing about the value of the model field after the call, then the resulting value of the model field will understandably be arbitrary. An assignment is analogous to a method call without specifications. The effect of an assignment on a concrete field is well known: only the concrete field assigned to is modified and all others are unchanged. But without some restriction on the model fields that might be affected by assignment to a particular concrete field, the value of a model field after the assignment is undetermined.

These issues, considering both soundness and modularity, have been discussed in more detail and addressed in previous and ongoing work, though the implementation of those results in JML and its related tools is not complete. Müller's thesis[22] and a subsequent paper with Poetzsch-Heffter and Leavens[23] propose a universe type system and a concept of relevant locations to control visibility and use of concrete and model fields and to assign responsibility for reasoning about their modification. Implementation of the universe type system is underway in JML. It may also be possible to use JML's syntax for defining datagroups to help specify relationships



between fields and model values they affect; restrictions on the fields allowed in a **represents** clause may be desirable and even necessary.⁸ Two other relevant pieces of work have appeared since the original workshop presentation of this paper in 2004, that on Spec# [3] and further work on universe types to provide encapsulation that controls aliasing [24].

Verification conditions for model fields

JML and ESC/Java were designed to do *modular* specification and checking. Any representation clause for a particular model field may not be visible within the modules at hand; if it is visible, JML provides two forms: a functional representation and a predicate representation. Work on translation of model variables in JML for the LOOP tool occurred concurrently with the work in this paper and is discussed in [4]. That paper also translates model field representations as invariants but does not discuss the interaction with frame conditions outlined above.

Functional and predicate 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 field, as stated by Breunese and Poll [4]. 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. It is simpler to translate such a representation in the same way as a predicate representation, generating an appropriate invariant for each **represents** clause.

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

```
//@ represents x \such_that predicate ;
```

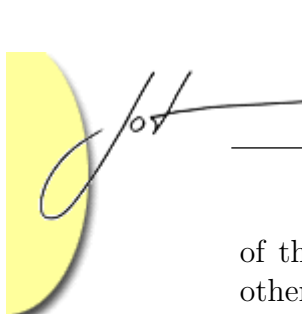
In this case the predicate naturally implies an invariant; but it does not necessarily give an executable expression for the model variable and may not even constrain the model variable to a single value. As Breunese and Poll point out, if there are no possible values satisfying the predicate, inconsistency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

Non-assignable model fields

If a model variable is neither implicitly nor explicitly mentioned in a method's **assignable** clauses⁹, then its value may not change during the course of the body

⁸Private communications with Gary Leavens and Clyde Ruby related to Ruby's in-progress thesis.

⁹including not specified by a JML datagroup



of the method. Such a non-assignable model field is translated in the same way other fields are, as a simple variable or an array reference. No new variable names are needed as the program state changes since the value of the field is presumed not to change. If the model field has a **represents** clause, that representation implies an assumption about the value of the model field. We include that assumption in the pre-state assumptions. The assumption ought to be true in all states throughout the method's execution. We currently do not check that this is true; a future enhancement should verify this presumption since otherwise inconsistent assumptions may be introduced, as discussed above.

Assignable model fields

If the model field is assignable within a method, then one must allow for different values at different states in the course of the method's execution. In ESC/Java's current translation scheme, this is easier to manage using a translation of the model field as a function term, with state and receiver arguments; this corresponds to the translation that would occur for a method call with no arguments, as discussed in section 5.

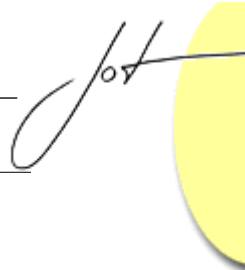
At each location where the model field is used, the assumptions generated from the **represents** clauses are also stated, so that they can be evaluated in the execution state at that point in the method's execution. This corresponds to the introduction of the post-conditions of a method invocation that is used in an annotation.

Model fields with no representation

A model field, 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. Those representations are not necessarily visible within the classes seen by a method being checked when doing modular checking of specifications. Fig. 5 shows an example of such an interface. Another example is shown in Fig. 4, in which the model field `AbstractList.size` has no representation in `AbstractList`; it is given a representation in `List` in terms of the class representation supplied in that derived class. In this situation, an assignable model field is still translated as a method call, 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.

7 APPLICATION TO OTHER ANNOTATION CONSTRUCTS

With a mechanism for translating method calls to an underlying first-order logic implemented in ESC/Java2, some other specification constructs can be readily translated and used in static checking as well. These are described briefly in this section.



```

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

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

```

Figure 5: The specification and code for the interface `NoRep`, demonstrating a model field with no representation.

Constructor calls in annotations

Constructor calls in annotations 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.

Array allocation in annotations

Translating expressions that allocate new arrays, such as `new int[9]`, 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.

Quantified expressions

Besides the usual universal and existential quantified expressions, JML also defines the generalized quantifiers `\min`, `\max`, `\sum`, `\prod`, and `\num_of`. For example, the value of the expression

$$(\min \text{ int } i; i \leq 0 \ \&\& \ i < 10; p(i))$$

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

$$(\ \backslash\min\ decl; range\text{-}predicate; expr)$$

are

$$(\ \backslash\exists\ decl; range\text{-}predicate) ==> \\ (\ \backslash\exists\ decl; range\text{-}predicate \ \&\& \text{MIN}(\dots) == expr)$$

and

$$(\ \backslash\forall\ decl; range\text{-}predicate ==> \text{MIN}(\dots) <= expr),$$

with suitable universal quantification and where $\text{MIN}(\dots)$ is replaced by the actual skolem function call expression.¹⁰

ESC/Java2 currently contains support for $\backslash\min$ and $\backslash\max$. However, the axioms for $\backslash\sum$, $\backslash\prod$, and $\backslash\text{num_of}$ are most naturally expressed recursively; reasoning about them requires induction, which is not readily supported by Simplify.

Exceptional behavior

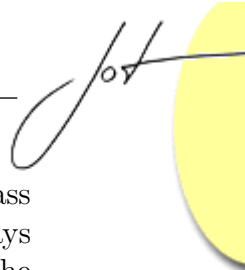
Constructor calls in annotations have the same problems with exceptional behavior as do method calls and they can be handled in the same way using the conventional specifications. However, quantified expressions and model variables both utilize expressions that may throw exceptions and neither have the 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.

8 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.

¹⁰The 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 $\backslash\min$, and the smallest such value for $\backslash\max$. It might also be considered as undefined.



Some reference types are also immutable. For example, objects of the class `java.lang.String` that compare equal (with either `==` or `.equals()`) will always have identical properties even in future program states. One requirement for the instances of a type to be immutable is that no method of the type modify the internal state of the instance; this condition is assured to hold if all methods of the type and any subtype are **pure**. However, it 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 [18]; 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.

Proof management

When creating mathematical proofs, the resulting proof will usually be simpler, shorter and more understandable if one can work entirely with more abstract concepts (e.g. a Group or Sequence) and the theorems that have been established about them. Only when necessary will a mathematician resort to expanding the definitions of these abstract concepts and assembling the proof in terms of lower-level constructions. A similar situation arises in checking the verification conditions that are produced in the course of static checking of programs. One can write specifications using abstractions such as method calls and model fields. Those have definitions in terms of their implementations within classes or perhaps in terms of objects in JML's mathematical library. All of the invariants and axioms for all of these higher and lower level constructions constitute a large set of logical conditions for a decision procedure to handle. Indeed, the introduction of abstraction mechanisms into ESC/Java2 has resulted in much larger sets of verification conditions being presented to Simplify. The problem is that Simplify simply matches terms that are equal, seeking to show unsatisfiability; it does not have heuristics to guide it to use primarily invariants concerning abstract concepts or to purposely "expand their definitions" in terms of implementation concepts. The need for such heuristics will become more important as automated checking of software programs tackles larger sets of software and seeks industrial-strength robustness in its results.

Methods and dynamic types

Method invocations in annotations are currently translated using the static type of the receiver. Sometimes more information about the receiver is known that would

```
class Super {
    //@ ensures \result >= 0;
    /*@ pure */ public int m();
}

class Sub extends Super {
    //@ also ensures \result <= 0;
    /*@ pure */ public int m();
}

public class Inheritance {
    //@ requires s != null;
    public void mm(Super s) {
        //@ assert s.m() >= 0;           // Line A
        if (s instanceof Sub) {
            Sub ss = (Sub)s;
            //@ assert ss.m() == 0;     // Line B
            //@ assert ((Sub)s).m() == 0; // Line C
            //@ assert s.m() == 0;     // Line D
        }
    }
}
```

Figure 6: An example showing the use of inherited specifications.

permit additional deductions. In Fig. 6, the assertions at lines A, B, and C can each be proved knowing only the static type of the receiver. The assertion at line D, however, requires the additional information that the dynamic type of the receiver is `Sub`; ESC/Java2 does not yet represent this information in the generated verification conditions.

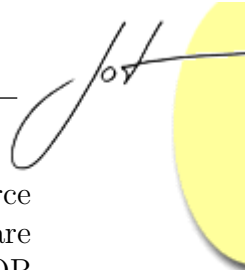
Specification language enhancements

As has been discussed above, this work with specification abstractions has pointed out areas in which specification languages such as JML may need enhancement.

- JML needs a syntax and semantics to enable control and reasoning about which field assignments might modify (and which definitely do not modify) which model field values.
- JML needs a way to distinguish deterministic from non-deterministic methods.
- Specifications in JML need to adopt a style that precludes divergent or exceptional behavior for methods and related constructs used in annotations.

9 OTHER WORK AND 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 [20] uses the Coq proof assistant, Jive [21] and LOOP [13, 15] use PVS [26], KeY [2] uses OCL and its own prover, and JACK [5] 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, definitions of the specification language are less common. The one example known to the author is the definition for the LOOP tool, which is not publicly available, of an early version of JML [15]. 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 [14], 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. 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 or default specifications.

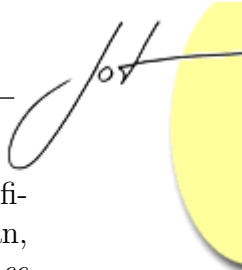
A similar approach to that used for method calls can also be used for model fields, with an optimization available for model fields that are not modifiable within the method that uses them. However, there is an outstanding issue related to reasoning about model fields: how to know in a modular fashion which model fields, if any, might have their values changed by a given assignment statement.

10 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, to Peter Müller for pointers and comments on previous work, as well as to the reviewers of both the precursor workshop (Formal Techniques for Java-like Programs, 2004) paper and this journal version. I am particularly grateful to those reviewers that took the time to provide detailed comments and suggestions.

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: Integrating object oriented design and formal verification. *Software and System Modeling*, Online First, 2004. <http://www.sosym.org>.
- [3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. Presented at CASSIS 2004. To be published., May 2004.
- [4] 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.
- [5] L. Burdy and A. Requet. JACK: Java applet correctness kit. In *Proceedings, 4th Gemplus Developer Conference*, Singapore, Nov. 2002.
- [6] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards. Model variables: Cleanly supporting abstraction in design by contract. Technical Report 03-10a, Department of Computer Science, Iowa State University, Sept. 2003. Available from <http://archives.cs.iastate.edu/>.
- [7] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. Technical report, University of Nijmegen, 2004. NIII Technical Report NIII-R0413.
- [8] 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.
- [9] 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/>.
- [10] 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.
- [11] E.-M. Hubbers. Integrating Tools for Automatic Program Verification. In M. Broy and A. Zamulin, editors, *Proceedings of the Andrei Ershov Fifth International Conference Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 214–221. Springer-Verlag, 2003. <http://www.iis.nsk.su/psi03>.



- [12] 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/>.
- [13] B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 58:61–88, 2004.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] C. Marché, C. Paulin, and X. Urbain. The Krakatoa tool for JML/Java program certification. Available at <http://krakatoa.lri.fr>, 2003.
- [21] J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description. Available at <http://softtech.informatik.uni-kl.de/old/en/publications/jive.html>, 2000.

- [22] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
- [23] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15:117–154, 2003.
- [24] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. Technical Report 424, ETH Zurich, Mar. 2005.
- [25] C. G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, Stanford, CA 94035, 1980. Available from University Microfilms.
- [26] 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.
- [27] A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report 00-03d, Iowa State University, Department of Computer Science, July 2003.
- [28] Unpublished reports about Simplify are available online at <http://research.compaq.com/SRC/esc/Simplify.html>.

ABOUT THE AUTHORS



Dr. David Cok received his Ph.D. in Physics from Harvard University and is currently Chief Technologist for the Photographic Center at the Kodak Research Lab. His research interests include image processing, image understanding, machine learning and optimization, as well as static tools for applying formal methods in industrial software development environments. He can be reached at david.cok@kodak.com.

A AN EXAMPLE

This section shows an example translation of some simple code contrived to show concisely the translation of method invocations in annotations. The verification conditions shown in Fig. 8 are a subset of the formulae generated by the translation of the method `m` in the code of Fig. 7. The verification conditions have been rewritten from the internal language used by Simplify to an equivalent, but more readable, form using the syntax of conventional first-order logic. The various numerical suffixes are appended by the translation mechanism to create related but distinct names (the names typically contain an associated line number in the program text).


```

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

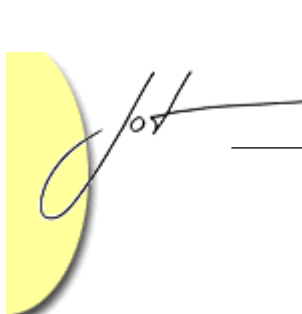
  //@ diverges false;
  //@ ensures b || \result == (c.z == 1);
  //@ signals (Exception) false;
  //@ pure
  public boolean p(C c, String s);

  //@ 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
  }
}

```

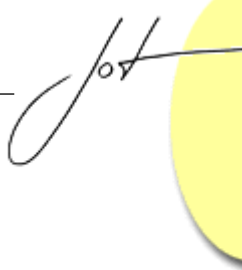
Figure 7: A somewhat contrived example to illustrate the translations of method calls.

- The ASSUME statement in line (2) of Fig. 8 states the assumption that the precondition in line A of Fig. 7 holds. Note the function form used to represent the call of `Trans.p`: it has the unique name *Trans.p* and it contains a state constant, *this* parameter, and the actual arguments of the call.
- The assumption about the value of this call of `Trans.p` 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 *c* 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* to a temporary variable (*RES.18*); this variable is the result of the method call within the program; in line (6) the method specifications are applied to that variable to state that the `ensures` predicate holds if the method returns normally; line (7) defines the



new, post-assignment, value for the variable **b** (fields are expressed as arrays in Simplify's logic). 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*).
- 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 (*state.pre*) and the post-condition with the post-state state constant (*state.20*); *state.pre* and *state* are later equated (not shown).



- 1) ASSUME $\forall brokenObj, c, s :$
 $((b.5[brokenObj] \vee$
 $(Trans.p(state, brokenObj, c, s) == integralEQ(z.3[c], 1)))$
 $\wedge is(Trans.p(state, brokenObj, c, s), \backslash type(boolean)))$
- 2) ASSUME $Trans.p(state, this, c.16, "A")$
- 3) ASSUME $\forall brokenObj34, c1, s1 :$
 $((b.5[brokenObj34] \vee$
 $(Trans.p(state, brokenObj34, c1, s1) == integralEQ(z.3[c1], 1))),$
 $\wedge is(Trans.p(state, brokenObj34, c1, s1), \backslash type(boolean)))$
- 4) ASSERT $Trans.p(state, this, c.16, "B")$
- 5) ASSUME $RES.18 == Trans.p(state, this, c.16, "Q")$
- 6) ASSUME $anyEQ(EC.18, ecReturn) \Rightarrow$
 $(b.5[this] \vee (RES.18 == integralEQ(z.3[c.16], 1)))$
- 7) ASSUME $anyEQ(b.18, store(b.5, this, RES.18))$
- 8) ASSUME $\forall brokenObj35, c2, s2 :$
 $((b.18[brokenObj35] \vee$
 $(Trans.p(state.18, brokenObj35, c2, s2) == integralEQ(z.3[c2], 1)))$
 $\wedge is(Trans.p(state.18, brokenObj35, c2, s2), \backslash type(boolean)))$
- 9) ASSERT $Trans.p(state.18, this, c.16, "C") \wedge$
 $Trans.p(state.18, this, c.16, "D")$
- 10) ASSUME $\forall brokenObj, c3, s3 :$
 $((b.18[brokenObj] \vee$
 $(Trans.p(state.20, brokenObj, c3, s3) == integralEQ(z.3[c3], 1)))$
 $\wedge is(Trans.p(state.20, brokenObj, c3, s3), \backslash type(boolean)))$
- 11) ASSERT $Trans.p(state.20, this, RES.20, "E")$
- 12) ASSUME $\forall brokenObj, c4, s4 :$
 $((b.18[brokenObj] \vee$
 $(Trans.p(state.20, brokenObj, c4, s4) == integralEQ(z.3[c4], 1))),$
 $\wedge is(Trans.p(state.20, brokenObj, c4, s4), \backslash type(boolean)))$
- 13) ASSERT $Trans.p(state.pre, this, c.16, "A") \Rightarrow$
 $Trans.p(state.20, this, c.16, "Z")$

Figure 8: A subset of the formulae generated from the translation of the code in Fig. 7. The nomenclature *brokenObj* is inherited from ESC/Java to help identify the object for which an assertion does not hold in a counterexample.