Early Detection of JML Specification Errors using ESC/Java2

Patrice Chalin

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

ABSTRACT

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

Categories and Subject Descriptors

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

General Terms

Documentation, Design, Languages, Theory, Verification.

Keywords

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

1. INTRODUCTION

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

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

1. A developer can *waste time* trying to get an ESC tool to prove that *m* satisfies *c*. Anyone who has used a verification tool is likely to have had this experience; i.e. being convinced that the specification (or theorem) is correct, one persists in trying to get the verifier to agree, only to realize, in all humility, that the tool was right, and that the specification was in error.
2. The ESC tool *is* able to prove that *m* satisfies *c*. This merely delays the discovery of the error (in *both* the specification and the implementation) until a later lifecycle phase. As a result, the error will be more expensive to correct.
3. In the worst case, *c* is inconsistent. Hence, any invocation of *m* in the code would amount to asserting falsehood, from which the verifier can trivially prove anything. For example, ESC/Java2 would be able to prove the assertion following a call to *m*:
   ```
   m();
   //@ assert 0 == 1;
   ```
   One might remark that: "if c is inconsistent, would ESC/Java2 not report an error in any attempt to prove that the implementation of *m* satisfies *c*?" Yes, but this assumes that the source for m is available, which is not the case, .e.g., for third party libraries distributed in binary form.

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

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006). November 10-11, 2006, Portland, Oregon, USA.
Copyright 2006 ACM ISBN 1-59593-586-X/06/11 … $5.00.

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

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

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

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

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

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

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

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

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

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

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

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

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

2. MOTIVATING EXAMPLES

2.1 MYUTIL/PAIRSUM

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

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

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

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

2.2 API SPECIFICATIONS FOR JAVA.UTIL.*

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

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

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

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

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

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

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

Figure 3. `ArraysBug.java`**.**

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

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

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

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

2.3 Other API Specification Errors

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

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

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

3. Supporting Definedness Checking

3.1 Background

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

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

3.2 JML's Classical Assertion Semantics

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

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

3.3 New Assertion Semantics Based on Strong Validity

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

- *defined*, and
- true.

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

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

3.4 Responsibility / Blame Assignment

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

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

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

3.5 The "is-defined" Operator

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

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

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

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

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

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

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

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

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

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

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

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

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

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

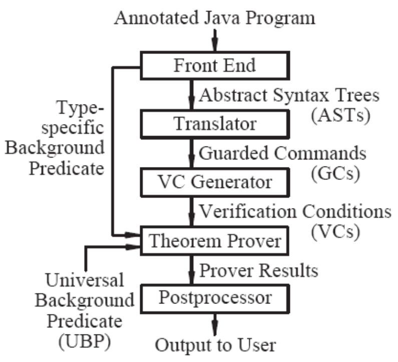

Figure 6. ESC/Java2Pipeline Architecture (excerpt from [11]).

$= D(x == 0) \land (\neg(x == 0) \Rightarrow D(3/x == 3/x))$
$= true \land (x \neq 0 \Rightarrow x \neq 0)$

which is *true*. In contrast, the expression

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

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

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

4. ESC/JAVA2 REDESIGN

4.1 ESC/JAVA2 CONCEPTUAL ARCHITECTURE

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

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

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

4.2 SUPPORTING THE NEW SEMANTICS

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

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

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

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

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

4.2.1 *INLINE ASSERTIONS*

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

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

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

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

4.2.2 *BASIC METHOD CONTRACTS*

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

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

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

Under the new semantics we have:

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

The new GCs are underlined.

4.2.3 *Class Invariants*

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

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

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

4.2.4 *Checking In The Absence of Source Files*

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

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

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

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

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

4.3 Accurate Error Reporting

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

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

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

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

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

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

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

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

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

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

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

The expanded GC form would be:

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

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

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

5. Better Diagnostics, At What Cost?

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

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

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

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

6. RELATED WORK

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

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

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

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

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

7. CONCLUSIONS AND FUTURE WORK

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

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

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

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

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

ACKNOWLEDGMENTS

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

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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