Extended Static Checking in JML4: Benefits of Multiple-Prover Support

Perry R. James, Patrice Chalin
Dependable Software Research Group
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
{perry,chalin}@dsrg.org

ABSTRACT

The implementations of many seemingly simple algorithms are beyond the ability of traditional Extended Static Checking (ESC) tools to verify. Not being able to verify toy examples is often enough to turn users off of the idea of using formal methods. ESC4, the ESC component of the JML4 project, is able to verify many more kinds of methods in part because of its use of novel techniques which apply multiple theorem provers. In particular, we present Offline User-Assisted ESC (OUA-ESC), a new form of verification that lies between ESC and Full Static Program Verification (FSPV), that allows users to control the level of completeness of the tool. ESC4's improved performance should encourage greater use of static verification.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Programming by contract, Correctness proofs*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*

Keywords

extended static checking, static verification, theorem provers, Java Modeling Language, JML4, ESC, ESC4

1. INTRODUCTION

Extended Static Checking (ESC) tools, such as ESC/-Java2 [13], provide a simple-to-use, compiler-like interface that points out common programming errors by automatically checking a class's implementation against its specification. ESC/Java2 is the de facto ESC tool when working with the Java Modeling Language (JML) [21], however, it has some important limitations. For example, it is unable to verify the assertions shown in Figure 1:

- Specifications with numeric quantifiers. (Due to their second-order nature, numeric quantifiers can be a challenge to support.)
- Specifications with method calls, even when the contract is merely `ensures true`.
- Quantified expressions are not allowed everywhere that expression are.

assert 6 == (**product int** i; 0 < i && i < 4; i);
assert isAlwaysTrue(3);
assert (E_1 ? E_2 : (**forall int** i; i > 0; i != 0))
| | (E_3 && (**forall int** i; i > 0; i != 0));

Figure 1: Assertions that ESC/Java2 cannot verify

In fact, the last of these limitations, as well as others, exists even in more modern ESC tools like Spec#'s Boogie program verifier [7]. We believe that ESC4 is the first ESC tool that can verify all of these.

In this paper, we report on work done to overcome these and other limitations. Our work has been carried out within the context of ESC4 [11], the ESC component of the JML4 project [10]. One of the main goals of ESC4 is to build a better ESC tool for Java programs specified using JML [21]. In addition to overcoming the previously mentioned limitations, ESC4 has other enhancements. Most notably, it introduces a new category of static verification, called Offline User-Assisted ESC (OUA-ESC), that falls between the fully automated classical ESC and interactive Full Static Program Verification (FSPV).

In the next section we report some examples of methods that ESC4 is able to verify but that ESC/Java2 cannot. Section 3 provides an overview of ESC4 including a brief description of the architectural components and overall verification process used to support multiple provers. OUA-ESC is described in Section 4. Related Work is described in Section 5. Conclusions are presented in Section 6.

2. ESC4 ENHANCEMENTS

In this section we describe some of the enhancements made to ESC4. In general, we do so by presenting examples of specifications that ESC/Java2 is unable to verify[1] but that ESC4 can handle. The mechanisms by which the verification is made possible are also described. In particular, we explain the following enhancements: support for

[1]Most are also beyond the capabilities of Boogie, as will be explained in Section 5.

```
/*@ requires n >= 0;
  @ ensures  \result ==
  @ (\product int i; 1 <= i && i <= n; i); */
public static int factorial(int n) {
    return (n == 0)
           ? 1
           : n * factorial(n - 1);
}
```

Figure 2: Arithmetic quantified expression

- numeric quantifiers,
- method calls in specifications, and
- quantified expressions anywhere a boolean expression is allowed.

In addition, we also briefly comment on ESC4's ability to

- handle the automated verification of specifications involving non-linear arithmetic and
- report assertions that are provably false.

2.1 Arithmetic quantifiers

Besides existential and universal quantifiers, JML supports the generalized numeric quantifiers `\sum`, `\prod`, `\min`, `\max`, and `\num_of`. Like the logical quantifiers, these have one or more bound variables, an optional expression limiting the range of these variables, and a body expression. An operator is folded into all values the body expression can take on when the range expression is satisfied. As expected, the expression

```
(\sum int i; 3 < i & i < 7; i)
```

evaluates to `15` i.e., `4 + 5 + 6`.

ESC/Java2 uses Simplify [15] as its underlying Automated Theorem Prover (ATP). ESC4, however, makes simultaneous use of a range of ATPs, currently Simplify, CVC3 [1], and Isabelle/HOL [24]. ESC/Java2 translates numeric quantified expressions as uninterpreted constants since Simplify is unable to cope with them. ESC4 does so as well for Simplify and CVC3, but since Isabelle is able to work with higher-order expressions, ESC4 faithfully translates all quantified expressions for it. As a result, many methods that use them can be verified automatically.

Figure 2 shows how compactly the specification for the factorial function can be expressed. Without numeric quantifiers it would be difficult[2] to express such a contract.

Numeric quantified expressions are translated into one of two forms, depending on the syntactic form of the range. If explicit bounds can be determined, then the expression is translated into an Isabelle function call that is very amenable to use in automatic verification. For example,

`(\sum int i;` a `<= i & i <=` b`;` $E(i)$`)`

would be translated into the Isabelle expression

$$sum\ a\ b\ (\lambda i.E(i))$$

[2]If not impossible, given that it is unclear what the semantics of recursive method contracts are in JML.

```
fun sum_helper ::
     "nat ⇒ int ⇒ (int ⇒ int) ⇒ int"
where
  "sum_helper 0          lo body = 0"
| "sum_helper (Suc n) lo body =
   (body (int n + lo)) + (sum_helper n lo body)"

fun sum :: "int ⇒ int ⇒ (int ⇒ int) ⇒ int"
where
"sum lo hi body =
       sum_helper (nat (hi - lo + 1)) lo body"
```

Figure 3: Definition of `sum` from Isabelle UBP

The Universal Background Predicates (UBPs) [19] in ESC4 are prover-specific collections of definitions that provide the semantics of Java and JML. The definition of `sum`, the summation function, from the UBP for Isabelle is shown in Figure 3. This UBP also contains some lemmas for dealing with -1 and `nats`, which are needed e.g., in loop invariants. Note that the range is shown as having type `int ⇒ int`, which is a function from `int` to `int`. This function is formed as a lambda expression whose single bound variable is that of the quantified expression.

When ESC4 cannot determine a numeric range, Isabelle's set-comprehension notation [24, §6.1.2] is used, which allows the translation to capture the full meaning of the original JML expression. Hence,

`(\sum int i;` $R(i)$`;` $E(i)$`)`

would be expressed as the Isabelle expression

$$\Sigma\ \{\ E(i)\ |\ i\ .\ R(i)\ \}$$

While lemmas containing the set-comprehension form are not easily discharged automatically, Section 4 describes how verification conditions (VCs) can be manually discharged.

2.2 Method calls in specifications

There has been much discussion in the literature about the handling of method calls in the context of Behavioral Interface Specification Languages (BISL) like JML (e.g., [22, 12, 14]) but, to our knowledge, ESC4 is the first ESC tool for JML that reasons about method calls in specifications.

Instead of relying on techniques that require separate encoding of the called methods in the various ATPs as function symbols and axioms, our initial implementation of methods is very simple. We inline a called method's pre- and post-conditions. While this approach does not scale to work with very large nesting of methods, it is sound and has worked well with code of the size shown here.

JML restricts methods called from specifications to those that are `pure`, (i.e., that do not have side effects). Calls to pure methods whose return type is a non-reference type can be replaced with an assertion of the called method's precondition (for purposes of definedness checking [9]) and assumption of its postcondition. If the called method's contract contains method calls, there is no need to check that the second-level preconditions hold, since they would be checked when processing the called method itself.

Figure 4 shows a method, `testIsCd`, whose in-line asserts call a method, and even that called method's specification contains method calls. ESC4 is able to correctly handle all

```
public class CommonDivisor {

    //@ requires b >= 0 & a > 0;
    //@ ensures \result == (b % a == 0);
    //@ pure
    public static boolean divides(int a, int b) {
        return (b - (b/a)*a) == 0;
    }

    //@ requires c > 0 & a >= 0 & b >= 0;
    //@ ensures \result <==>
                  divides(c, a) & divides(c, b);
    //@ pure
    public static boolean
            isCommonDivisor(int c, int a, int b) {
        return divides(c, a) & divides(c, b);
    }

    public static void testIsCd() {
        //@ assert isCommonDivisor( 2,  4,  6);
        //@ assert isCommonDivisor( 3,  4,  6);
    }
}
```

Figure 4: Method calls in specifications

three of these methods. That is, an error is reported only for the second call to `isCommonDivisor` (because 3 is not a divisor of 4).

2.3 Restoring First-Class Status of Quantified Expressions

As show in the third assertion in Figure 1, ESC4 allows quantified expressions to appear in conditional expressions. Simplify makes a strong distinction between formulas and terms, and permitting quantifiers only as formulas. The implementation in Simplify's UBP of conditional expressions, including conditional conjunctions and disjunctions (i.e., `&&` and `||`), requires that their subexpressions be terms. ESC4 uses an only slightly modified version of ESC/Java2's UBP for Simplify [19], so its use of Simplify has the same restriction. The other ATPs used by ESC4 do not have this limitation, as their input languages provide support for conditional expressions. This permits ESC4 to treat quantified expressions as first-class expressions.

2.4 Non-linear arithmetic

Figure 5 shows a JML-annotated method that computes the cube of its integer parameter using only shifts (multiplication by 2) and additions [20]. This method is an interesting example for verification because it is far from obvious that its body respects its simple contract.

Without adequate tool support for static verification, extensive testing would be needed to build confidence in the method's correctness. ESC/Java2 is unable to verify this method because its underlying ATP, Simplify, is unable to reason about non-linear arithmetic. This is true of most ATPs. Instead of relying on a single theorem prover, ESC4 simultaneously uses a range of theorem provers (see Section 3.2). For our Cube example, all of the VCs can be discharged by at least one of the ATPs, so ESC4 is able to verify that the method is correct.

```
//@ requires x > 0;
//@ ensures \result == x * x * x;
public int cube(int x) {
    int a = 1;
    int b = 0;
    int c = x;
    int z = 0;
    //@ maintaining a == 3*(x-c) + 1;
    //@ maintaining b == 3*(x-c)*(x-c);
    //@ maintaining z == (x-c)*(x-c)*(x-c);
    //@ decreasing c;
    while (c > 0) {
            z += a + b;
            b += 2*a + 1;
            a += 3;
            c--;
    }
    //@ assert z == x * x * x;
    return z;
}
```

Figure 5: Computing x^3 with shifts and additions

2.5 Reducing Prover Overhead

Isabelle's power comes at the cost of it being slower than the other ATPs used. It is not uncommon for it to take 10 times as long as Simplify to process a VC, but it is able to discharge whole classes of VCs that Simplify cannot. Even though the other ATPs are faster than Isabelle, they are much slower than simple manipulations of in-memory data structures or simple checks of the file system. ESC4 uses several techniques to help offset the theorem provers' cost by eliminating unnecessary invocations of them.

One of these is keeping a persisted cache of VCs that have previously been discharged. Before sending a VC to any of the ATPs, the system checks if the VC cache already contains it. If so, it is discharged immediately. If it is not found but a prover is able to show that it holds true, then it is added to the cache. Isabelle is currently the last prover in our prover chain. If it is not able to discharge a VC then some information is left in the file system that indicates this situation. If this indication is present, then none of the theorem provers is able to prove it, and we can immediately return this failure status.

Another novel technique is used to keep from having Isabelle waste time trying to discharge a VC that is easily proved false. Before invoking Isabelle, a faster ATP is used to try to prove its negation (or rather, the negation of the original assert). For example, if the original VC has the form $(p \longrightarrow q)$ then ESC4 tries to show $(p \longrightarrow \neg q)$. If this modified VC can be shown to be true then the original VC must be false, and this extra information can be reported to the user. It is often useful to know that an assertion *is* false rather than just that the theorem prover was unable to prove it true.

3. OVERVIEW OF ESC4

Like other ESC tools, ESC4 converts an abstract syntax tree (AST) first to a Guarded Command (GC) program and then to a Verification Condition (VC). The VC is a logical expression in first-order logic, and if it can be shown to always be true (e.g., by an ATP) then the method is said to have been successfully verified. Otherwise, errors are reported that include the assertions that could not be verified.

3.1 Generation of VCs

JML4[10] is a version of the Eclipse JDT [16] that, among other things, produces a JML-decorated AST that can be used by various tools, which currently include rudimentary Runtime Assertion Checking (RAC) and Full Static Program Verification (FSPV). ESC4takes as its input a fully resolved and checked AST produced by JML4. ESC4 is implemented as a compiler stage between flow analysis and code generation. If the compiler's front end finds any errors in a class then ESC4 does not process it. The AST for each method is converted first to a GC program using techniques based on those described by Barnett and Leino [6]. This approach allows for the straightforward translation of while loops and other control-flow structures to an acyclic control-flow graph. Dynamic Single Assignment is used to removed any side effects, but unexpected difficulties were encountered with conditional expressions. Method calls are replaced with their specifications, although doing so requires quite a bit of care because of the special handling required by quantified expressions and conditional expressions.

Using a weakest-precondition calculus, this passive, acyclic graph that represents an entire method and its specification is converted to a single VC.

3.2 Discharging VCs

ESC4 uses a configurable `ProofCoordinator` to discharge VCs. This allows different prover strategies to be easily implemented. Currently, the first strategy tried is to prove the entire VC using Simplify. If this fails, the VC is broken into a set of sub-VCs, the conjunction of which is equivalent to the original VC. This is done by recognizing that VCs are sequences of implications and conjunctions in which the atomic conjuncts are assertions in the GC program and implications are assumptions. The implications are distributed over the conjunctions to form a set of implications. We have proved this decomposition sound.

Once the method's VC has been split into sub-VCs, each is given in turn to Simplify, CVC3 and Isabelle. By far, Simplify is the fastest of these three, when it is able to discharge a VC, and it is the first that ESC4 uses.

Isabelle, which is more commonly used as an interactive theorem prover, is used as an ATP by having it use a hard-coded proof strategy. Isabelle is much slower than the other two provers, but this is compensated for by its being able to discharge many VCs that other ATPs cannot. Section 4 describes another way that the power of Isabelle is used in ESC4.

Since ESC4 is run every time that a method is saved and successfully compiled, it is important that it be as quick as possible. To help with this goal and to eliminate redundant calls to the theorem provers, once a VC has been proven, it is stored in a cache. This cache is consulted before calling any of the ATPs. Also, the `ProofCoordinator` leaves some information in the file system so that it can determine that Isabelle was previously unable to discharge a VC. This eliminates invocations of Isabelle that are known will fail.

4. OFFLINE USER-ASSISTED EXTENDED STATIC CHECKING

4.1 Overview

ESC4 introduces a novel form of static verification that lies somewhere between the fully automatic classical ESC (which is incomplete) and, interactive and complete FSPV. We call this kind of verification Offline User-Assisted ESC (OUA-ESC). OUA-ESC enables a developer to take advantage of the full power of Isabelle as an interactive theorem prover to discharge a VC that cannot be discharged automatically. Once the proof has been written, ESC4 makes use of it during subsequent compilation cycles, enabling Isabelle to act as an ATP over the user-supplied proof. Hence, OUA-ESC allows JML4's static verifier to take advantage of the full power of Isabelle—in conjunction with user-supplied proofs—as one of its automatic theorem provers, thus increasing the ESC4's completeness to the same level as that achievable by FSPV.

Figure 7 shows the proof for a VC that cannot be discharged automatically. Part of ESC4's error message is the name of the theory file corresponding to the assertion that could not be verified. Work is underway to offer an Eclipse QuickFix that will automatically open the file in an Eclipse ProofGeneral perspective [4]. This file imports the Universal Background Predicate (UBP), a collection of theorem and function definitions that can be used in proofs. This is followed by a lemma that states the VC that could not be discharged. The proof given by ESC4 is "`oops`", which causes Isabelle to stop processing the lemma and ignore it. As an optimization, if the theory file is found to still contain the proof "`oops`" then Isabelle is not invoked.

With the addition of this ability to make use of arbitrarily complex proof techniques, ESC4 is able to discharge any VCs that are produced, limited only by the capabilities of Isabelle and the skills and needs of the user. If the user does not want to manually discharge a VC, the lemma file can still prove useful, as it contains a trace of the method from the precondition through the body to an assertion that is reported as not holding.

Isabelle's automatic simplification commands, while not enough to fully prove the lemmas, are usually able to reduce the original, quite large, lemma to subgoals that show only the missing facts that would allow the proof to go through. Often these smaller forms are

- obviously true and just require a little manipulation for Isabelle to recognize their truth,
- obviously false and lead to a search in the VC for clues to the error in the source code, or
- surprisingly false and lead to the modification of specifications to allow the necessary information to be available.

The last case is most apparent when assumptions are missing, such as method preconditions or loop invariants.

Both of the automatic Isabelle commands `QuickCheck` and `refute` can provide counterexamples that are often helpful in determining where the code or specification is incorrect by pointing out why Isabelle thinks the lemma is false.

```
public class IntSqrt {

    //@ requires x >= 0;
    //@ ensures  \result * \result <= x;
    //@ ensures  x < (\result + 1) * (\result + 1);
    public static int sqrt(int x) {
        if (x == 0)
            return 0;
        if (x <= 3)
            return 1;

        int y = x;
        int z = (x + 1) / 2;

        //@ maintaining z > 0;
        //@ maintaining y > 0;
        //@ maintaining z == (x / y + y) / 2;
        //@ maintaining x < (y + 1) * (y + 1);
        //@ maintaining x < (z + 1) * (z + 1);
        //@ decreasing y;
        while (z < y) {
                y = z;
                z = (x / z + z) / 2;
        }
        return y;
    }
}
```

Figure 6: Calculating the integer square root

Learning just a little bit about Isabelle and ProofGeneral is enough to allow the gathering of a lot of useful information about an undischarged VC. In addition to the normal simplification procedures, the 2008 release of Isabelle [2] includes the `sledgehammer` command [25], which can automatically search for some slightly more sophisticated proofs.

Once a proof for a VC has been accepted by Isabelle, it can be used by ESC4 on future runs.

4.2 Example of OUA-ESC

As an example, consider the method in Figure 6[3], which computes the integer square root using Newton's method. The ATPs used by ESC4 are able to discharge 14 of the 19 sub-VCs generated for this method. For each of the remaining 5 sub-VCs, a separate `.thy` file was output. The first corresponds to the loop invariant `x < (y + 1) * (y + 1)` not holding on entry to the loop. If a proof is given, such as the one shown in Figure 7, then ESC4 will be able to discharge the corresponding sub-VC during the next compilation cycle.

To prove this lemma, we opened its file in ProofGeneral and started by applying `auto`. A single subgoal was left, which we copied and pasted as a separate helper lemma above the main lemma. If we can prove this helper then Isabelle would be able to prove the main lemma using its simplification procedures.

As we often do, we first try to find a proof for helper lemmas using Isabelle's `sledgehammer` command [25] to find a Metis proof. Metis [3] is an ATP that tries to prove lemmas

[3]This is a variant of an example given in the Why distribution [5].

```
theory IntSqrt_sqrt_1
imports UBP
begin

lemma helper: "(0::int) < x ==> x < (x + 1) * (x + 1)"
  by (metis add1_zle_eq eq_iff_diff_eq_0 int_one_le_-
iff_zero_less linorder_not_less mult_less_cancel_left2
order_le_less_trans order_less_le_trans pordered_ring_-
class.ring_simps(27) ring_class.ring_simps(9)
zadd_commute zle_add1_eq_le zle_linear zless_add1_eq
zless_le)

lemma main: "([| (True & ((x::int) >= (0 :: int)));
(~ ((x::int) = (0 :: int))); (~ ((x::int) <= (3 ::
int))); ((y::int) = (x::int)); ((z::int) =
(((((x::int)) + ((1 :: int)))) div ((2 :: int))))|]
==> ((x::int) < (((((y::int)) + ((1 :: int)))) *
((((y::int)) + ((1 :: int)))))))"
  apply (auto)
  apply (simp add: helper)
done

end
```

Figure 7: A proof for a VC from the code in Figure 6

using a filtered list of theorems. Isabelle's `sledgehammer` can be thought of as a relevance filter that finds such a list. When `sledgehammer` is able to find a proof, it can be copied and pasted from the ProofGeneral's response buffer directly into the proof script. If `sledgehammer` is unable to find a proof for a helper lemma, it may still be provable manually by providing an explicit proof. The Isar language [26] provides a structured way of expressing proofs in a form that is similar to pen-and-paper proofs but that can still be checked by Isabelle. We comment more about the relevance of Isar in the next subsection.

4.3 Discharging Helper Lemmas

Being able to provide both Metis and Isar proofs fits nicely into an iterative development cycle. Initially, developers may only be interested in knowing that the lemma holds true, without concern for the readability or maintainability of the proof. This is especially true during active development, as the lemmas needed for a method may change often. Also, Metis proofs may be unstable over time, as the heuristics that the Metis prover uses to find a proof from a list of theorems could be subject to change. Furthermore, providing an explicit Isar-style proof may give insights into the problem domain.

In our example, a Metis proof was found for the helper that we could copy and paste into the theory file. Executing the proof of this particular helper lemma takes a long time, but once it and the original lemma are proved inside ESC4, the sub-VC is stored in the method's VC cache and Isabelle is not asked to prove it again.

4.4 Issues

While the VCs stored in the theory files are not very user-friendly, future work includes making them more palatable. Until then, simply having the ProofGeneral parse the lemma causes it to be pretty printed as the single subgoal to be discharged. This causes unnecessary typing information to be removed, and the structure of the expression is shown through proper indentation.

Information about expressions' source-code positions in the generated VCs. This position information is used for two purposes: for error reporting and for making identifiers unique. Unfortunately, having position information in the VCs is a major source of brittleness of both the VC cache and the OUA-ESC process. With it, adding even a single character to the source file would cause the text of the cache entry or generated lemma to change. To avoid this, we plan to remove position information whenever possible from lemmas in both the VC cache and the lemmas sent to Isabelle. This will not cause a problem with error reporting because only VCs that are true are stored in the cache and because we use the problems that are indicated by Simplify to provide error reporting. Making identifiers unique, can be partially addressed by only including the position information if the same identifier is used more than once in a given sub-VC (e.g., if two quantifiers' bound variables share the same name). A further optimization would be to replace an absolute position with a relative position (so, e.g., the two aforementioned bound variables would be suffixed with `_1` and `_2` instead of their character positions).

4.5 Summary

Offline User-Assisted ESC provides users who are willing to put in the effort of developing proofs with a way to verify much more code than classical ESC, which relies solely on ATPs, can. This benefit is provided without the burden of forcing users who are not willing (or able) to generate the needed proofs with doing anything extra. The end result is an overall verification technique that offers a level of completeness in verification that is proportional to the effort the end user is willing to invest (and this is usually proportional to the criticality of the code).

5. RELATED WORK

5.1 ESC/Java and ESC/Java2

ESC/Java2 [13] is the successor to the earlier ESC/Java project [19], the first ESC tool for Java. ESC/Java's goal was to provide a fully automated tool to point out common programming errors. The cost of being fully automated and user friendly required that it be–by design–neither sound nor complete. Soundness was lost by not checking for some kinds of errors (e.g., arithmetic overflow of the integral types is not modeled because it would have required what was felt to be and excessive annotation burden on its users). ESC/Java provides a compiler-like interface, but instead of translating the source code to an executable form, it transforms each method in a Java class to a VC that is checked by an ATP. Reported errors indicate potential runtime exceptions or violations of the code's specification. "The front end produces abstract syntax trees (ASTs) as well as a type-specific background predicate for each class whose routines are to be checked. The type-specific background predicate is a formula in first-order logic encoding information about the types and fields that routines in that class use" [19]. The ESC/Java2 project first unified the original program's input language with JML before becoming the platform developed by many research groups.

5.2 Spec#, VCC, and HOL-Boogie

Spec# is Microsoft's extension to C# for supporting verified software [7]. It is composed of

- the Spec# programming language, a superset of C# enriching it with support for Design by Contract
- the Spec# compiler, which includes an annotated library, and
- the Boogie static verifier, which performs ESC.

The Spec# system is among the most advanced ESC tools currently available.

We translated the JML code in Sections 1 and 2 into Spec# and tested them with version 1.0.20411.0 (11 April 2008) under Visual Studio 2008. We were surprised at the results. Of the three assertions in the introduction, the first was correctly handled, while the third caused the IDE to throw an exception. Only the example using method calls in specifications (see Figure 4) was handled the same way as in ESC4. When processing the example that showed ESC4's ability to indicate that a subexpression is provably false, Spec# is only able to detect that the assertion is violated. That is, it is unable to identify the offending subexpression or to indicate that the expression is definitely false.

Leino and Monahan [23] report a way to handle arithmetic quantifiers using ATPs. This addition was enough to allow the simple example in the first assertion in Figure 1 to be verified, but not the example in Figure 2.

Böhme, Leino, and Wolff [8] report on HOL-Boogie, an extension of Isabelle/HOL that can be used in place of the Z3 ATP in the regular Boogie toolchain. Currently it is used in the VCC toolchain, but the Spec# system could be modified to make use of it. This addition would have the possibility of allowing Spec# to verify most of the examples presented in this paper, although the verification would require manual proofs, as there is no indication that Isabelle is used as an ATP.

HOL-Boogie does not provide proof-status feedback to the IDE (i.e., VisualStudio). Our approach does provide such feedback, with the goal of being able to do the proofs within the JML4 IVE, thus delivering a more satisfying user experience.

Like ESC4, HOL-Boogie supports splitting a method's VC into sub-VCs, which it then tries to discharge using Z3 and Isabelle/HOL. Unlike ESC4, this splitting is done by Isabelle, which itself makes calls to Z3. Any user-supplied proofs are *not* used to discharge the corresponding sub-VCs.

5.3 Caduceus

Caduceus [18] is a tool for the static verification of C programs annotated with contracts similar to those of JML. Its output is the input language of the Why tool [17], which produces VCs that are discharged using automatic and interactive theorem provers. These include Simplify, CVC3, and Isabelle/HOL that are used by ESC4 as well as PVS, Coq, Z3, and others.

Like ESC4, Caduceus treats quantified expressions as first-class expressions. In addition to function calls being allowed in specifications, a construct called *predicates* allows the definition of specification-only functions.

The ATPs used by the Why tool cannot reason about numeric quantifiers or non-linear arithmetic, so code that makes use of them could not be verified automatically. The interactive provers may allow for them.

6. CONCLUSION AND FUTURE WORK

In this paper we presented several examples of code that ESC4 is able to verify and that other commonly used ESC tools are not. ESC4 can verify code that uses numeric quantifiers. Unlike other ESC tools, ESC4 does not limit quantified expressions to being the only expression in an assertion. Also, we believe that ESC4 is the first ESC tool for JML that can verify code that uses method calls in specifications or that uses non-linear arithmetic. It is able to do all of these things because it uses a variety of ATPs, where one is able to compensate for the weaknesses of the others. VCs that can be proved to be false are reported as such to users. Those that are proved true are cached to eliminate unnecessary invocations of the theorem provers. We have given an introduction to Offline User-Assisted ESC, which we believe will be a powerful addition to static verification.

ESC4 is a quickly evolving research platform. Even though there are things it can do that ESC/Java2 cannot, there is much more that ESC/Java2 can do that ESC4 does not yet do. To close this gap, we are continuing to flesh out ESC4's capabilities to more fully support Java and JML.

At the same time, we are working on performance and usability issues. Of specific interest is to make OUA-ESC easier and more intuitive to use by implementing QuickFixes and integration with the ProofGeneral plugin for Eclipse. Performance gains can easily be made by making the interface to the theorem provers more efficient. The underlying JDT compiler has recently been made multithreaded, and we aim to soon have ESC4 to be able to also take advantage of multicore systems. In a similar vein, we hope to soon start work on a distributed version of the prover back-end to take advantage of non-local computing resources.

7. REFERENCES

[1] CVC3: An automatic theorem prover for Satisfiability Modulo Theories (SMT), 2008. Homepage at `http://www.cs.nyu.edu/acsys/cvc3/`.

[2] Isabelle, 2008. Homepage at `http://isabelle.in.tum.de`.

[3] Metis theorem prover, 2008. Homepage at `http://www.gilith.com/software/metis/`.

[4] Proof General Eclipse, 2008. Homepage at `http://proofgeneral.inf.ed.ac.uk/eclipse`.

[5] Why: software verification platform, 2008. Homepage at `http://why.lri.fr`.

[6] Barnett, M., and Leino, K. R. M. Weakest-precondition of unstructured programs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2005), ACM Press, pp. 82–87.

[7] Barnett, M., Leino, K. R. M., and Schulte, W. The Spec# programming system: An overview. In *CASSIS 2004: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, Marseille, France, March 10-14, 2004, Revised Selected Papers* (2004), G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds., vol. 3362 of *Lecture Notes in Computer Science*, Springer, pp. 49–69.

[8] Böhme, S., Leino, R., and Wolff, B. HOL-Boogie – An interactive prover for the Boogie program verifier. In *Proceedings of the 21th International Conference on Theorem proving in Higher-Order Logics (TPHOLs 2008)* (2008), LNCS 5170, Springer.

[9] Chalin, P. A sound assertion semantics for the dependable systems evolution verifying compiler. In *ICSE* (2007), IEEE Computer Society, pp. 23–33.

[10] Chalin, P., James, P. R., and Karabotsos, G. An integrated verification environment for JML: Architecture and early results. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems* (2007), pp. 47–53.

[11] Chalin, P., James, P. R., and Karabotsos, G. JML4: Towards an industrial grade IVE for Java and next generation research platform for JML. In *VSTTE '08: Proceedings of the 2008 conference on Verifyied Systems: Tools, Techiniques, and Experiments* (2008).

[12] Cok, D. R. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology (JOT) 4*, 8 (2005), 107–103.

[13] Cok, D. R., and Kiniry, J. R. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices* (2005), vol. 3362/2005 of *LNCS*, Springer Berlin, pp. 108–128.

[14] Darvas, Á., and Müller, P. Reasoning about method calls in interface specifications. *Journal of Object Technology 5*, 5 (2006), 59–85.

[15] Detlefs, D., Nelson, G., and Saxe, J. B. Simplify: A theorem prover for program checking. *J. ACM 52*, 3 (2005), 365–473.

[16] Eclipse. http://www.eclipse.org.

[17] Filliâtre, J.-C. The WHY verification tool: Tutorial and reference manual, 2008. Available at `http://why.lri.fr`.

[18] Filliâtre, J.-C., Hubert, T., and Marché, C. The Caduceus verification tool for C programs: Tutorial and reference manual, 2008. Available at `http://caduceus.lri.fr`.

[19] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (New York, NY, 2002), ACM Press, pp. 234–245.

[20] Kolman, B., and Busby, R. C. *Discrete mathematical structures for computer science (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, 1986.

[21] Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., and Chalin, P. JML reference manual, 2008. Available at `http://www.jmlspecs.org`.

[22] Leino, K. R. M. *Toward reliable modular programs*. PhD thesis, California Institute of Technology, Pasadena, CA, USA, 1995.

[23] Leino, K. R. M., and Monahan, R. Automatic verification of textbook programs that use comprehensions. In *FTfJP '07: Proceedings of the 9th Workshop on Formal Techniques for Java-like Programs* (2007).

[24] Nipkow, T., Paulson, L. C., and Wenzel, M.

Isabelle/HOL — A Proof Assistant for Higher-Order Logic, vol. 2283 of *LNCS*. Springer, 2002.

[25] Paulson, L. C., and Susanto, K. W. Source-level proof reconstruction for interactive theorem proving. In *Theorem Proving in Higher Order Logics: TPHOLs 2007* (2007), K. Schneider and J. Brandt, Eds., LNCS 4732, Springer, pp. 232–245.

[26] Wenzel, M. Isar - A generic interpretative approach to readable formal proof documents. In *TPHOLs '99: Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics* (London, UK, 1999), Springer-Verlag, pp. 167–184.