J Autom Reasoning manuscript No.
(will be inserted by the editor)

Faster and More Complete Extended Static Checking for the Java Modeling Language

Perry R. James · **Patrice Chalin**

Received: date / Accepted: date

Abstract Extended Static Checking (ESC) is a fully automated formal verification technique. Verification in ESC is achieved by translating programs and their specifications into verification conditions (VCs). Proof of a VC establishes the correctness of the program. 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). ESC is generally quite efficient, as far as verification tools go, but it is still orders of magnitude slower than simple compilation. As can be imagined, proving VCs is computationally expensive: While small classes can be verified in seconds, verifying larger programs of 50 KLOC can take hours. To help address the added cost of using multiple provers and this lack of scalability, we present the multi-threaded version of ESC4 and its distributed prover back-end.

CR Subject Classification D.2.4

Software EngineeringSoftware/Program Verification[Programming by contract, Correctness proofs]

CR Subject Classification F.3.1

Logics and Meaning of ProgramsSpecifying and Verifying and Reasoning about Programs[Mechanical verification]

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

Dependable Software Research Group
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
E-mail: {perry,chalin}@dsrg.org

```
assert 6 == (\product int i; 0 < i && i < 4; i);
assert (E1 ? E2 : (\forall int i; i > 0; i != 0));
maintaining b == 3*(x-c)*(x-c);
```

Fig. 1 Assertions that ESC/Java2 cannot verify

1 Introduction

Extended Static Checking (ESC) [1] is a fully automatic form of static analysis that provides more checks than are available from standard type checking but less than from Full Static Program Verification (FSPV) [2]. It does this by translating source code that has been annotated with specifications to Verification Conditions (VCs), which are boolean expressions in first-order logic. If the VC corresponding to a given method can be discharged then the method is correct with respect to its specification. In ESC, VCs are discharged with the help of Automated Theorem Provers (ATPs).

Extended Static Checking (ESC) tools, such as ESC/Java2 [3], provide a simple-to-use, compiler-like interface that points out common programming errors by automatically checking the implementation of a class against its specification. ESC/Java2 is the de facto ESC tool for the Java Modeling Language (JML) [4]; however, it has some important limitations. For example, it is unable to verify the assertions shown in Fig. 1, which contain (1) numeric quantifiers, (2) quantified expressions in certain positions, and (3) non-linear arithmetic.

In fact, the second of these limitations, as well as others, exists even in more modern ESC tools like Spec#'s Boogie program verifier [5]. We believe that ESC4 is the first ESC tool that can automatically 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 [2], the ESC component of the JML4 project [6]. One of the main goals of ESC4 is to build a better ESC tool for Java programs annotated with JML specifications [4]. 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).

While small classes can be verified in seconds, larger programs of 50 KLOC can sometimes take hours to verify. We believe that this is an impediment to widespread adoption of ESC: used to modern incremental software development models, developers have come to expect that the compilation (and ESC) cycles are very quick.

We also describe how ESC4 is able alleviate this problem by the following:

- We take advantage of the inherent modularity of the verification techniques underlying ESC [7] to analyze the methods in a given compilation unit in parallel. This is possible because the ESC analysis done for a given method is independent of that for any others. (Section 6.2)
- We take advantage of ESC4's proof strategies to develop distributed discharging so that non-local resources can be used to reduce the time to verify a set of classes. (Section 6.3)
- The previous two points are achieved by means of OS-independent proof services: If an executable version of a given prover is not available for a given platform, that prover can be exposed through a service and used remotely as if it were local. (Section 6.4)

While tools exist for verifying distributed and multi-threaded code, we have not found another verifier that makes use of these techniques to speed up its own analysis. We believe that ESC4 is the first fully automatic static-verification tool to do so.

In the next section we report some examples of methods that ESC4 is able to verify but that ESC/Java2 cannot. Section 4 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 5. The techniques used to speed up the processing in ESC4 are presented in Section 6, and experimental validation is provided in Section 7. Related Work is described in Section 8. Conclusions are presented in Section 9.

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

- numeric quantifiers,
- quantified expressions anywhere a boolean expression is allowed, and
- non-linear arithmetic.

In addition, we also briefly comment on ESC4's ability to report assertions that are provably false.

2.1 Arithmetic quantifiers

Besides existential and universal quantifiers, JML supports the generalized numeric quantifiers **sum**, **product**, **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

$$(\backslash\textbf{sum int } \mathrm{i};\ 3 < \mathrm{i}\ \&\ \mathrm{i} < 7;\ \mathrm{i})$$

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

ESC/Java2 uses Simplify as its underlying Automated Theorem Prover (ATP). ESC4, however, makes simultaneous use of a range of ATPs, currently Simplify, CVC3, and Isabelle/HOL. 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 expressions in higher-order logic, ESC4 faithfully translates all quantified expressions for it. As a result, many methods that use them can be verified automatically.

Fig. 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, the JML expression

$$(\backslash\textbf{sum int } \mathrm{i};\ a <= \mathrm{i}\ \&\ \mathrm{i} <= b;\ E(\mathrm{i}))$$

[1] Most are also beyond the capabilities of Boogie, as will be explained in Section 8.

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

```
//@ requires n >= 0;
//@ ensures \result == (\product int i; 1 <= i & i <= n; i);
public static int factorial(int n) {
    int result = 1, j = 1;
    //@ maintaining result == (\product int i; 1 <= i & i <= j−1; i);
    //@ maintaining 1 <= j;
    //@ decreases n−j+1;
    while (j != n+1)
        result *= j++;
    return result;
}
```

Fig. 2 Arithmetic quantified expression

```
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"
```

Fig. 3 Definition of "sum" from Isabelle UBP

would be translated into Isabelle as

$$\text{sum } a\ b\ (\lambda \text{i}.E(\text{i}))$$

The Universal Background Predicates (UBPs) [1] 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 Fig. 3. This UBP also contains some lemmas for dealing with -1 and **nat**s, 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 [8, §6.1.2] is used, which allows the translation to capture the full meaning of the original JML expression. Hence, the JML expression

$$(\backslash\textbf{sum int i}; R(\text{i}); E(\text{i}))$$

would be expressed as the Isabelle expression

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

While lemmas containing the set-comprehension form are not easily discharged automatically, Section 5 describes how verification conditions (VCs) can be manually discharged.

```
//@ requires x > 0;
//@ ensures \result == x * x * x;
public int cube(int x) {
    int a = 1, b = 0;
    int c = x, 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--;
    }
    return z;
}
```

Fig. 4 Computing x^3 with shifts and additions

2.2 Restoring First-Class Status of Quantified Expressions

As shown in the third assertion in Fig. 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 [1], 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.

2.3 Non-linear arithmetic

Fig. 4 shows a JML-annotated method that computes the cube of its integer parameter using only shifts (multiplication by 2) and additions [9]. 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 4.2). For our Cube example, Simplify is able to discharge all of the VCs except for the ones corresponding to the last two loop invariants, which Isabelle is able to discharge automatically. Since all of the VCs can be discharged by at least one of the ATPs, ESC4 is able to verify that the method is correct.

3 Background

ESC4 is a from-scratch rewrite that builds on the lessons learned from earlier projects, principally ESC/Java2. It is part of the JML4 project.

First-generation tools such as ESC/Java and ESC/Java2 are stand-alone command-line applications that use their own custom Java-compiler front ends to produce an AST. Since the research interest of the maintainers of these tools is JML, and not the underlying Java front end, these tools have not kept up with the latest developments of the Java language.

After much discussion, both within our own research group and with other members of the JML community, it was decided that basing a next-generation JML tooling framework on the Eclipse JDT was the most promising approach.

The result is JML4 [6], a Integrated Verification Environment (IVE) for JML-annotated Java that is built atop the Eclipse Java Development Tooling (JDT).

JML4's first feature set enhanced Eclipse with scanning and parsing of nullity modifiers (nullable and non-null), enforcement of JML's non-null type system (both statically and at runtime) [10] and the ability to read and make use of the extensive JML API library specifications. These include

- recognizing and processing JML syntax inside specially marked comments, both in `.java` files as well as in `.jml` files,
- storing JML-specific nodes in an extended AST hierarchy,
- statically enforcing a non-null type system, and
- generating runtime assertion checking (RAC) code.

Since then, work has been underway by several research groups to flesh out JML4 so that it can process all of JML language-level 2 [4].

The framework has also been enhanced to support static analysis [2], including both ESC and Full Static Program Verification (FSPV). The main compiler phases can be seen in Fig. 5. Note that there are two forms of parsing: Diet and Full. The former ignores method bodies so that the resulting AST only contains signature information, and the latter produces a full AST.

4 Overview of ESC4

ESC4 [2,11] is the ESC component of JML4 and is a ground-up rewrite of ESC. Its VC generation is based on Barnett and Leino's innovative and improved approach to a weakest-precondition semantics for ESC [12]. One of the most significant results of this approach is that the size of the VCs produced are linear in the size of the method being analyzed, where earlier approaches generate VCs whose size can be quadratic in the worst case [13].

Fig. 6 shows the data flow in ESC4. The fully resolved and analyzed AST produced by the JDT's front end is taken as input. Only those with no front-end-reported errors are processed further by ESC4. The source AST is first converted to a control-flow graph (CFG) as described in [12]. This CFG is similar to Dijkstra's Guarded Command Language [14], except that the guards have been replaced with **assume** statements and the choice operator has been replaced with **goto**s. A VC for each source method is generated from this intermediate form. ESC4's Prover Coordinator is responsible for discharging the VC or reporting why it cannot be discharged. A post-processor reports unprovable VCs to the user through the IVE as failed assertions and attaches the results of the analysis to the original AST. Depending on the compiler options in effect, the code-generation phase may make use of these results to optimize runtime checks.

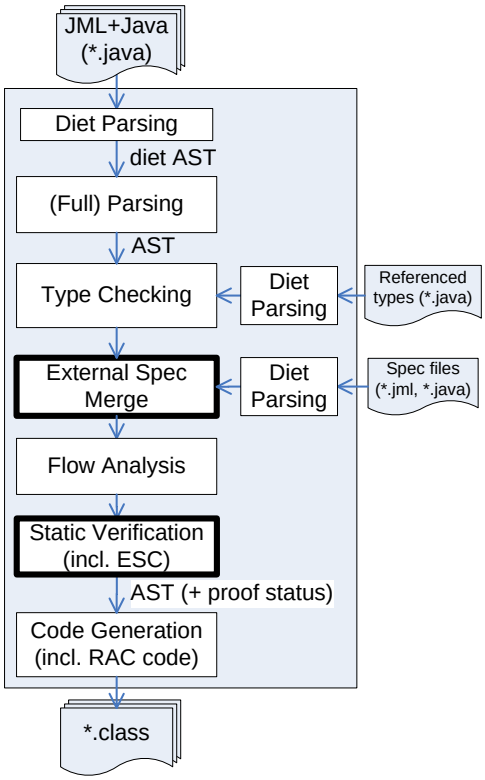

Fig. 5 Compiler phases for JML4

4.1 Generation of VCs

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 [12]. 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 remove side effects.

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

4.2 Discharging VCs

ESC4 uses a Prover Coordinator to discharge VCs whose strategies are selectable through compiler options. This modularization makes it easier to support new provers and implement new prover strategies (see Fig. 7). As a default, 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

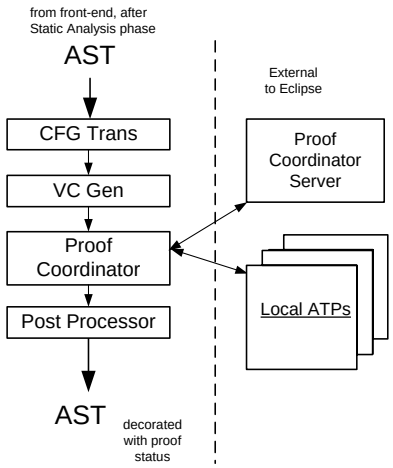

Fig. 6 Data flow in ESC4

the GC program and implications are assumptions. The implications are distributed over the conjunctions to form a set of implications. That is, each sub-VCs represents a single acyclic path from the method's precondition, through its implementation to an assertion. An Isabelle proof that this decomposition is sound and complete can be found in the Appendix.

Instead of reporting an entire assertion as failing, we want to identify the responsible subexpression. This helps users more quickly locate and correct the problem. To do this, we split not only conjunctions generated during translation and VC generation, but also the conjunctions in the source code. Initially, we naïvely split conditional conjunctions (`&&`) the same way as logical (i.e., non-short circuiting) conjunctions (`&`), but this led to incorrect error reporting. For example, consider the assertion **assert** x == 3 `&&` y == x + 3. The second conjunct should never be reported as unprovable if the first conjunct also is. Since the second conjunct should only be reported as unprovable if the first conjunct is true, the second conjunct should be prefixed with this implication. Thus, the conditional conjunction should be translated as

$$a \texttt{ \&\& } b \equiv a \texttt{ \& } (a \rightarrow b).$$

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[3]through an interactive session like that used by the Proof General. 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 5 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 stores the text of the VC as a HashSet. The cache is stored to the file system on a per

[3] For now, "`by (simp add: nat_number | auto | algebra)+`" is used.

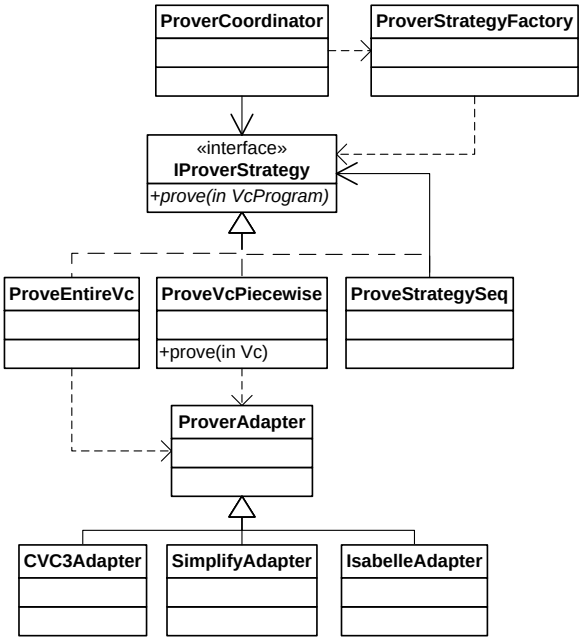

Fig. 7 ESC4's prover back-end

compilation-unit basis. Since there are relatively few VCs in each instance of the cache, the lookup time is insignificant. This cache is consulted before calling any of the ATPs. Also, the Prover Coordinator 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 to fail.

4.3 Prover back-end

A class diagram for the Prover back-end is shown in Fig. 7. A Prover Coordinator is used to discharge VCs. It obtains a proof strategy from a factory whose behavior is governed by compiler options. The default strategy is a sequence of two strategies: The first tries to prove the entire VC using a single ATP. If it fails, the second, ProveVcPiecewise, is used. Both use adapters to access the theorem provers. These adapters hide the mechanism used to communicate with the provers. They use visitors to pretty print the VC to produce input for each ATP's native language. To eliminate wasting time re-discharging a previously discharged VC (or sub-VC), the strategies can make use of a VC cache, which is persisted.

ProveVcPiecewise implements 2D VC Cascading: VCs are broken down into sub-VCs, giving one axis of this 2D technique, and proofs are attempted for each sub-VC using each of the supported ATPs, giving the second axis.

The conjunction of the set of sub-VCs is equivalent to the original VC. Discharging all of the sub-VCs shows that the method is correct with respect to its specification. Any sub-VCs that cannot be discharged reflect either limitations of the provers or faults in the source.

Currently, three ATPs are used: Simplify, CVC3, and Isabelle/HOL. The first two of these are much faster than Isabelle, but Isabelle is able to discharge VCs containing many constructs that the others are not. After trying both Simplify and CVC3 on a sub-VC, we try to prove the negation of the original assertion using Simplify. If this succeeds, it shows that the original assertion never holds. This is usually a faster operation than invoking Isabelle, and since it is easy for programmers to write the negation of the expression wanted, it is useful to know when it occurs (see Section 6.1). Only after all other attempts fail is Isabelle invoked.

5 Offline User-Assisted Extended Static Checking

5.1 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.

To do so, skeleton files are created for any VCs that Isabelle is unable to discharge. If the user provides a valid proof for the lemmas in those files then the next compilation cycle's invocation of ESC4 will use the provided proof instead of the default proof strategy. If Isabelle is able to prove the lemma then the VC is taken as discharged. OUA-ESC opens up the possibility of verifying many more methods than would be possible using only ATPs, and without forcing users to provide proofs if they are not so inclined.

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. Learning just a little bit about

```
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 && 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;
}   }
```

Fig. 8 Calculating the integer square root

```
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
```

Fig. 9 A proof for a VC from the code in Fig. 8

Isabelle and Proof General 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 [15] includes the **sledgehammer** command [16], which can automatically search for some slightly more sophisticated proofs. When the sledgehammer finds a proof, a proof script is provided that can be used as a user-supplied proof.

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

5.2 Example of OUA-ESC

As an example, consider the method in Fig. 8, which computes the integer square root using Newton's method[4]. 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 is output. If a proof is given then ESC4 would be able to discharge the corresponding sub-VC during the next compilation cycle. The first of the 5 unproven sub-VCs corresponds to the loop invariant $x < (y + 1) * (y + 1)$ not holding on entry to the loop and is encoded as the main lemma shown in Fig. 9. We next examine the contents of this file and how we created a proof of the sub-VC.

The theory file shown in Fig. 9 starts by stating its name importing the Universal Background Predicate (UBP), a collection of theorem and function definitions that can be used in proofs. Two lemmas follow: the second (main) is the one created by ESC4 that states the VC that could not be discharged. The first (helper) was added later, as described below. The proof originally left by ESC4 for main was the Isabelle keyword **oops**, which causes Isabelle to stop processing the lemma and ignore it.

To prove this lemma, we opened its file in Proof General, removed the **oops**, and started our proof by applying auto. The sub-VC's original lemma is not very user friendly, but auto is usually able to simplify it greatly. In this case, the single subgoal left is simply

$$\neg x \leq 3 \Longrightarrow x < (x+1) \star (x+1).$$

We copied and pasted this subgoal as a separate helper lemma above the main lemma. If we could prove this helper then Isabelle would be able to prove the main lemma using its simplification procedures.

As a general rule, we first try to find a proof for helper lemmas using Isabelle's **sledgehammer** command [16] to find a Metis proof. Metis [18] is an ATP that tries to prove lemmas using a given list of of theorems. Isabelle's **sledgehammer** tries to find such a list automatically. When **sledgehammer** is able to find a proof, it can be copied and pasted directly into the proof script.

Unfortunately in our case, **sledgehammer** failed to find a proof. We noticed that the helper lemma could be generalized by changing the implication's antecedent from

$$\neg x \leq 3$$

to

$$x > 0,$$

which was enough to allow **sledgehammer** to find a Metis proof. Once the helper lemma was proved, we were able to prove the main lemma by adding the helper to the facts used by the simplificaiton procedure.

If **sledgehammer** is unable to find a proof for a useful helper lemma, it may still be provable manually by crafting an explicit proof. The Isar language [19] 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] This is an adaptation of an example given in the Why distribution [17].

5.3 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, and a slegehammer-provided Metis proof is sufficient since there is no 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 frequently. 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. As the system becomes more stable, it may be of interest to develop an explicit Isar-style proof as this may give insights into the problem domain.

5.4 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 Isabelle 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 is added to identifiers 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 when 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).

5.5 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 can classical ESC, which relies solely on ATPs. 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).

6 Faster ESC

Applying ESC to industrial-scale applications has been difficult because of the time existing tools require. In this section we highlight the enhancements that have been added to ESC4 that reduce the time needed to verify JML-annotated Java code.

6.1 Reducing Prover Invocations

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[5], 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.

6.2 Multi-threading

The biggest gains in speed in ESC4 come from making use of all available CPU resources.

Using the arguments in Leino's thesis, *Toward Reliable Modular Programs* [7], it can be shown that each JML-annotated method in a system can be verified independently of the others. Where there are no dependencies, it is possible to introduce concurrency.

First-generation tools such as ESC/Java [1] and ESC/Java2 [3] were written before multi-threaded and multi-core computers were commonplace. Multi-threading operating systems were already available then, but writing the code to use them would have only increased its complexity without making the processing any faster. This encouraged a serialized approach to the problem, even though the modular nature of ESC is inherently parallelizable. Today, however, multiple-core machines are becoming the norm. Each thread could, in theory, run on its own core and thus reduce the time needed to verify a system to the most time needed to verify a single method. While the number of cores needed to achieve this level of speedup will not be available in the foreseeable future, having such

[5] A true negated VC may also be caused by a contradiction, which would mean either the specification introduced a contradiction or the assertion corresponding to the VC is unreachable.

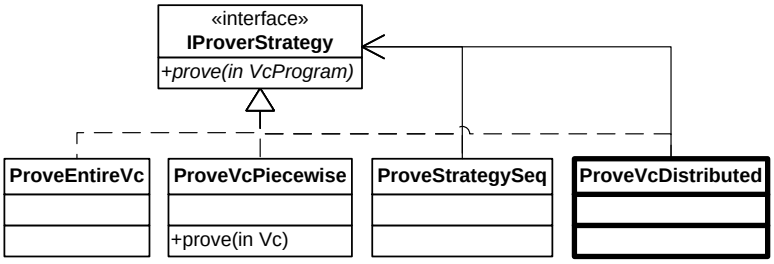

Fig. 10 ESC4's distributed prover back-end

small-grained units of work should make efficient scheduling easier for the operating system and/or virtual machine.

Modifying ESC4 to take advantage of ESC's inherent concurrency simply required adding a thread pool: Instead of processing each method sequentially, we packaged the processing (the body of an inner loop) as a work item and added it to the thread pool's task list. Finally, we added a join point to wait until all of the work for a compilation unit's methods finished before ending the ESC phase for it. This last step is necessary because the results of ESC may be used during code generation.

Version 3.4 of the Eclipse Java compiler added the ability to use separate threads to compile individual source files concurrently [20]. Since ESC4 and JML4 are built on top of this compiler, all we had to do to gain this benefit was to ensure that JML4 is thread safe.

The vast majority of the time doing ESC is spent discharging VCs. Specifically, it is the underlying theorem provers that use the most time. For this reason, most ESC tools only make use of a single ATP per verification session. As mentioned above, ESC4 uses three by default, and 2D VC Cascading can cause those three to be invoked multiple times for each method. Just as the methods in a class can be verified in parallel, the sub-VCs for a method can be discharged in parallel. We just need to put a join point so that we know when the processing of a method's VC has finished.

This gives ESC4 3 layers of parallelism: source files, methods within those files, and sub-VCs for those methods.

6.3 Distributed VC Processing

Once we were able to take advantage of all of the CPU resources on a local machine, it became interesting to ask if we could make use of resources on remote machines. The design of ESC4's Prover Coordinator led to quick discovery of a few deployment scenarios for the distributed discharging of VCs. It was easy to support distributed provers by adding new strategy communication infrastructure.

1. **Prove whole VC remotely**. The first deployment scenario offloads the work of the Prover Coordinator for an entire method. This was done by developing a new subclass of IProverStrategy that sends the VC generated for a method to a remote server for processing. (see Fig. 6.3). A Prover Coordinator is instantiated on the remote server along with its strategies. We initially had it behave like a local Prover Coordinator and discharge the VC itself with its own local provers.

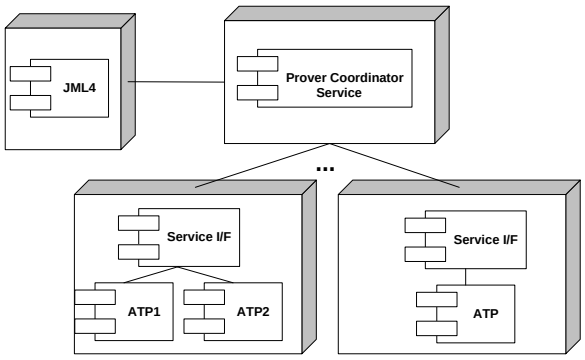

Fig. 11 Deployment

2. **Prove sub-VCs remotely**. A second deployment scenario was to split the VC into sub-VCs and send each of them off for remote discharging. This was done by extending the ProveVcPiecewise strategy discussed in Section 4.3 and having it use remote services to discharge the sub-VCs in parallel.
3. **Doubly Remote Prover Coordinator**. Combining the two previous approaches, so that the remote Prover Coordinator itself delegates the responsibility for discharging the sub-VCs to remote services by using the ProveVcPiecewiseDistributed strategy, provides yet another alternative. A deployment view can be seen in Fig. 11.

Scenario 1 uses the least bandwidth, since only the original VC is transmitted. Scenario 2 uses the next least, although it can be exponentially more than 1. Scenario 3 uses the most, the sum of 1 and 2, but it is split into two groups: the same is used between the local machine and the remote Prover Coordinator as in 1, and between the remote Prover Coordinator and its servers as in 2.

Splitting a VC into sub-VCs can cause exponential growth in size, since these sub-VCs each represent a single acyclic path from the method's precondition, through its implementation to an assertion.

As a result, scenarios 1 and 3 would be preferred over 2 when the remote machines are not on the same local area network. Scenario 3 can be thought of as providing the best parts of the other two: low bandwidth requirements to reach the prover service, and 2D VC Cascading.

In addition, scenario 3 is the most likely to be used when a large farm of servers is available or when the Prover Coordinator service provides a façade that hides load balancing and other details from ESC4.

6.4 Prover service

Independent of the strategy used, the proving resources may be local or remote. The initial prover adapters communicated with local resources using Java's Process mechanism. After facing some difficulties installing some provers on all of our development platforms, we hit on the idea of Prover Services.

The adapters that use the provers locally can be taken as base classes to subclasses that access them remotely. Part of the purpose of the adapter classes is to hide the interface with

Table 1 VCs discharged with provers

Prover	No. VCs	No. Proved	(%)
Simplify	235	193	82
CVC3	42	0	0
Negation[a]	42	23	55[b]
Isabelle	19	13	68
failed	6		

[a] Simplify used to prove the negation of the VC
[b] 80% of all false

the provers. Applying the same concept lets us hide whether the prover is hosted locally or on a remote machine.

This has the advantage of making the provers OS independent. If a prover is needed on an OS for which there is no executable, it can be hosted on another machine with the appropriate OS and an adapter can hide the extra communication needed to access it.

7 Validation

To confirm that our approach produces speedups, we performed some preliminary timing tests. The source tested was a single Java class with 51 methods. For this code, ESC4 produced 235 VCs. Table 1 shows the number of times each of provers was invoked. Simplify was able to discharge over 80% of the VCs. It was also able to show as false almost 80% of those that were indeed false $(23+6)$. In this sample, CVC3 was not able to prove any of the VCs that Simplify was also unable to prove, and Isabelle was needed for just over 5% of the original VCs.

We ran the test with two deployment scenarios, both based on the Doubly Remote Prover Coordinator described in Section 6.3. In the first, the Prover Coordinator was hosted on the same PC as ESC4. In the second, it was hosted on a faster remote machine.

ESC4 was run on a 2.4 GHz Pentium 4. The Prover Coordinator was hosted either locally to the ESC4 machine or on a 3.0 GHz Pentium 4. Neither of these machines' CPUs is hyperthreaded. The provers were hosted on servers, each with a 2.4 GHz Quad-core Xeon processor. The timing results are shown in Table 2. Each entry in the last two columns is the average of three test runs, which were made after an initial run with the configuration being tested to remove initialization costs. Even so, the timings varied from 0.5 s to 1.6 s. Network usage may account for some of this variation.

For comparison, running the test with the Prover Coordinator and provers were all on the same PC as ESC4 took 72 s. It should be noted that when using remote provers, the CPU of the local machine stayed at 100% during the first few seconds and then dropped to below 20% while gathering the results. When the Prover Coordinator was on a separate machine, that machine's CPU was never went above 50%.

The data gathered indicate that there is little difference between hosting the Prover Coordinator locally or remotely. We had thought that hosting it remotely would allow the VCs to reach the provers faster, thus giving a greater speedup. Surprisingly, as more processing cores were made available, it was actually faster to send the VCs directly. Further testing will have to be done to confirm this. For the sample shown, the timing difference between the two scenarios is within the range of error.

Table 2 Timing results

No. servers	No. cores	Time (s) with Prover Coordinator local	remote
1	4	26.6	26.4
2	8	16.9	16.2
3	12	12.8	13.3

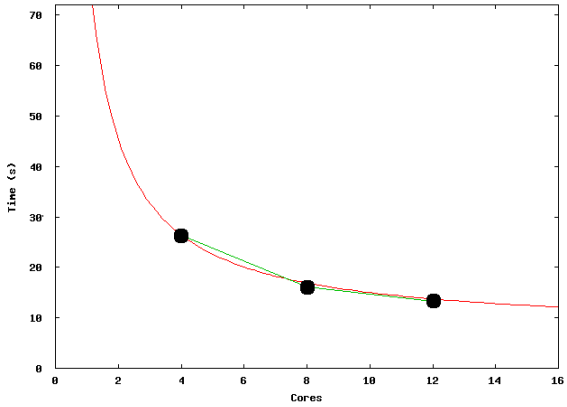

Fig. 12 Time (s) vs. Cores

A function from the number of processors used to the time taken to analyze a given piece of code can be derived by applying simple algebra to Amdahl's law [21,22]. It should have the form

$$t = C_1 + \frac{C_2}{n},$$

where C_1 is the time taken to complete the portion that cannot be serialized and C_2 is the time for the portion that can. Replacing n with 4 and 8 cores and t with the times for the remote Prover Coordinators gives a system of 2 linear equation with 2 unknowns. Solving this system gives

$$t = 7.4 + \frac{76.0}{n}$$

The experimental result of 13.3 s for 12 cores is within the error range of the predicted time of 13.7 s.

These initial results with up to 12 cores suggest that over 90% of the ESC analysis is amenable to parallelization. One question that future study will have to address is, "Can the 7.4 s that was not parallelized by using distributed provers be made parallelizable by hosting ESC4 on a multi-core machine?" Contained in the serial part is the JDT's front-end generation of the AST and ESC4's generation of VCs from it.

After adding 12 cores, the serial portion takes longer than the portion that is parallelized. We did not test the generation of VCs on a multi-core system. Doing so may show that at least part, and maybe even most, of this segment is parallizable.

8 Related Work

8.1 ESC/Java and ESC/Java2

ESC/Java2 [3] is the successor to the earlier ESC/Java project [1], 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 an 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" [1]. The ESC/Java2 project first unified the original program's input language with JML before becoming the platform developed by many research groups [3]. There has been mention of experimental support for multiple prover back-ends, but it is unclear when this feature will be released. In contrast to ESC4's use of a cache and removing position information from VCs (Sec. 5.4), Grigore and Moskal [23] describe an approach to reducing the time to re-verifiy a method after minor edits are made by comparing the original and modified VCs to prune the VC that will be sent to the ATP.

8.2 Spec#, VCC, and HOL-Boogie

Spec# is Microsoft's extension to C# for supporting verified software [5]. 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, only the first was correctly handled, while the third caused the IDE to throw an exception. 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 [24] report a way to handle arithmetic quantifiers using ATPs. This addition was enough to allow the simple example in the first assertion in Fig. 1 to be verified, but not the example in Fig. 2.

Böhme, Leino, and Wolff [25] 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.

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.

8.3 Caduceus

Caduceus [26] is a tool for the static verification of C programs annotated with contracts similar to those of JML. Verification in Caduceus is essentially a three-step, manual process: C programs are first translated into the language of the Why system [27], then Why is used to translate VCs into the language of a user-selected prover. The supported provers include Simplify, CVC3, and Isabelle/HOL that are used by ESC4 as well as PVS, Coq, Z3, and others. Finally, the user runs or interacts with the selected prover in order to discharge the VC proof obligations. The user is left to interpret any prover output, including tracking undischarged VCs back to the source. Such an *offline* approach to verification is like that adopted by the JML4 FSPV Theory Generator [2,28] and contrasts with ESC4's fully automated mode of extended static checking.

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 prover would allow them to be proved manually.

8.4 Omnibus

The Omnibus language was developed to be "similar to Java but easier to formally reason about" [29]. It compiles to Java bytecode and is aimed at component developers, who can use the code's specification as part of its documentation. One of the main differences between Omnibus and Java is that the former uses value semantics (as in many functional languages) while the latter uses reference semantics (as in most imperative languages). Other simplifications include the lack of static data, exceptions, interface inheritance, arithmetic overflow, and concurrency.

Wilson et al. provide support for RAC, ESC, and FSPV through a custom IDE. Among the results of their research is the discovery of problems that arise when reasoning about a class that uses another class that was verified with another method. Several guidelines are provided to avoid or minimize these problems [30,31].

Each of the source files in an Omnibus project has an associated verification policy that gives the level of verification required for it, which can be RAC, ESC, or FSPV. After being parsed and type checked, each file is analyzed by the static verifier. The source is translated into the logics of the theorem provers, as required by the verification policy. Simplify and PVS are the two theorem provers used for ESC and FSPV, respectively [31,32].

8.5 JACK

Jack is a set of tools integrated into Eclipse that was developed by Gemplus for FSPV of a subset of Java and JML. The Jack converter translates JML-annotated Java to a form usable by tools that support the B Method, specifically the automated and interactive theorem provers that are included with the Atelier B. Each Java class is translated into a single B machine in which the class's inheritance hierarchy has been flattened and other classes' members are in-lined. B lemmas are generated whose proof indicates that the class's invariants always hold and its methods' postconditions are guaranteed by their preconditions and bodies [33]. The Atelier B automated prover is able to discharge, on average, 80% of proof obligations. While it was a goal for the tool to be sound and complete, shortcomings in the weakest-precondition calculus used make it impossible to prove that the lemmas corresponding to method specifications are "necessary and sufficient to ensure the correctness of the [code]" [34].

Jack differs from other FSPV tools for JML in that it both makes use of an automated theorem prover and provides an easy-to-use interface to an interactive theorem prover. Instead of requiring users to learn the B notation, lemmas and their proof status are converted back to Java for presentation. A goal for the project is to eventually allow fuller access to the interactive theorem prover, but as of [34], only two forms of "indicating a false hypothesis" are supported. Until this goal is realized, users must either manually prove the leftover lemmas in the Atelier B interactive prover or accept that the tool provides only a more rigorous form of ESC.

8.6 Other Tools

As noted in the introduction, we have not been able to find other existing tools that make use of distributed or parallel processing to enhance fully automatic program verification. Two related aspects of the work presented here have been previously examined: multi-threaded, distributed compilation and interactive, distributed theorem proving for program verification. These are discussed in the following subsections.

8.6.1 Compilation

As mentioned in Section 6.2, Eclipse 3.4 supports mutithreaded compilation of Java programs. The Gnu make command `gmake` has a `--jobs[==n]` option that executes up to *n* build tasks concurrently. If an integer *n* is not supplied then as many tasks are started as possible [35]. Microsoft's Visual C++ compiler has the "Build with Multiple Processes" option (`/MP`) that launches multiple compiler processes. If no argument is given, the number of effective processors is used. The number of effective processors is the number of threads that can be executed simultaneously and considers the number of processors, cores per processor and any hyperthreading capabilities. Apple's Xcode IDE also supports distributed builds, with options for using either shared-workgroup machines or dedicated build servers [36].

Several open-source projects and commercial products are available that can distribute the tasks in a build process to networked machines. These only launch a process on a remote machine and do not make use of a service-based approach. Open-source projects include distcc [37] and Icecream [38]. Xoreax sells a product called IncrediBuild incredibuild that coordinates distributed builds from within with Microsoft's VisualStudio.

8.6.2 Interactive, distributed theorem proving for program verification

The MetaPRL system [39,40] is, in part, an interactive theorem prover written in OCaml. It is a tactic-based prover that uses a higher-order term-rewriting engine to automatically explore the proof space. To speed up this search, tactics have been written that distribute the work to local and remote processors using Ensemble [41], whose last release was in 2004 and no longer appears to be maintained.

Vandevoorde and Kapur describe the Distributed Larch Prover (DLP), "a distributed and parallel version of LP, an interactive prover" [42]. Like LP, DLP is not an ATP, as users must guide the proof-discovery process. It achieves parallelism by allowing users to simultaneously try several techniques to prove a subgoal. This is done by distributing the attempts among computers on a network. Some automation is provided by heuristics that chose the inference methods to be launched in parallel.

Hunter et al. attempt to use distributed provers to increase the adoption of formal techniques in industry [43]. Like the DLP, their approach requires interaction, but their goal is to reduce that interaction. Reducing the amount of user interaction would reduce the cost of using formal tools to prove software correct and thus remove one of the impediments to its more widespread use. A user interacts with software agents that try to automatically prove a goal. User interaction is needed when one of these agents is unable to automatically prove subgoals.

9 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 non-linear arithmetic. It is able to do 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 never hold 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 Proof General plugin for Eclipse. Performance gains can easily be made by making the interface to the theorem provers more efficient.

Applying ESC to industrial-scale applications has been difficult because of the time required. Invoking a theorem prover for every method in a system is computationally expensive. We attacked this by applying the divide-and-conquer strategy to allow processing by multiple computing resources, both local and remote. Generating and discharging the VC for Java methods is a problem that can be easily decomposed into many independent tasks. This makes it very amenable to multi-threading and distributed processing. Given the power of today's desktop PCs, most of an organization's desktop computers' CPUs are under-utilized. Installing a distributed proving service on these machines would allow the

organization's developers to tap into existing resources without requiring the acquisition of additional hardware.

The Eclipse JDT compiler is able to process multiple source files in parallel. We showed how we modified ESC4 to support verifying multiple methods in parallel. Similarly, a method's sub-VCs are discharged in parallel. Because of the potential reduction in time to verify a system, it became useful to explore distributed prover resources. This in turn led to exposing individual provers as distributed resources. All of these combined make the verification of Java programs scalable: The time ESC4 needs to verify a system should be inversely proportional to the CPU resources made available to it.

There are several possible next steps. We modified ESC4 to take advantage of many local and non-local computing resources. The implementation was done to quickly get a usable and stable framework in place, without much regard for optimization. While we are pleased with the initial results, there are ample opportunities for improvement. These include using more efficient communication mechanisms to interact with remote resources. Load balancing and other techniques from service-oriented architectures are obvious candidates for consideration. Proof-status caching, as described in [2], would also improve performance during iterative development since only methods that were changed would need to be re-verified.

After making the obvious enhancements, we plan to conduct timing studies to evaluate the deployment scenarios mentioned in this paper, varying the number and kinds of local and remote resources as well as the characteristics (speed and reliability) of the network.

Acknowledgements This papers is an extended version of [11]. We would like to thank the anonymous reviewers for their helpful comments and suggestions. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT).

Appendix: Soundess and Completeness Proof for VC-Splitting Algorithm

The following Isabelle/HOL 2009 proof is available online at `http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/jml4/trunk/org.eclipse.jdt.core/notes/esc4/Vc2Vcs.thy`

theory *Vc2Vcs* **imports** *Main* **begin**

A Introduction

ESC4 produces a single VC for each method to be verified. We would like to split unprovable VCs into a collection of sub-VCs, the conjunction of which is equivalent to the original. This Appendix describes the decomposition algorithm as well as provides a proof that it is sound and complete.

B VC Language and Splitting Algorithm

typedecl *BExp*

datatype *VC* = *VcIf VC VC*

| *VcAnd VC VC*
| *VcAndAnd VC VC*
| *VcOther BExp*

The VC language that we work with in this Appendix contains only conjunction, implication, and undefined Boolean expressions. Two forms of conjunction are supported: logical and conditional. The former always evaluates both operands, while the latter only evaluates its second operand if the first evaluates to *True*.

fun *distribImp* :: *VC* ⇒ *VC list* **where**
distribImp (*VcIf a b*) = *map* (*VcIf a*) (*distribImp b*)
| *distribImp* (*VcAnd a b*) = *distribImp a* @ *distribImp b*
| *distribImp* (*VcAndAnd a b*) = *distribImp a* @ *map* (*VcIf a*) (*distribImp b*)
| *distribImp* (*VcOther b*) = [*VcOther b*]

The VC splitting is accomplished by distributing implications over the conjunctions, while keeping the conditional definition of the conditional conjunction. This special treatment was needed to provide proper error reporting.

C Semantics

consts *M′* :: *BExp* ⇒ *bool*

To be able to show that our approach is sound and complete, we must first give meaning to VCs. Since *VcOther* is meant to be an uninterpreted boolean expression, its meaning is given by the uninterpreted function *M′*.

fun *M* :: *VC* ⇒ *bool* **where**
M (*VcIf x y*) = ((*M x*) ⟶ (*M y*))
| *M* (*VcAnd x y*) = ((*M x*) ∧ (*M y*))
| *M* (*VcAndAnd x y*) = ((*M x*) ∧ (*M y*))
| *M* (*VcOther x*) = *M′ x*

The definition of *M* for *VcAndAnd* could have been written as

$$((\mathrm{M\ x}) \wedge (\mathrm{M\ x} \longrightarrow \mathrm{M\ y})),$$

but in a two-valued logic this is equivalent to the definition given. This is shown by the following lemma.

lemma ((*M x*) ∧ (*M y*)) = ((*M x*) ∧ (*M x* ⟶ *M y*))
by *blast*

D Auxilliary Lemmas About foldr and map

To make the proof of the main theorem easier, we first prove some properties HOL's *foldr* and *map* functions.

It is useful to be able to move one of the conjuncts from the base expression of the *foldr* expression to the outside. This is similar to HOL's *List.foldr_add_assoc*.

lemma *foldr-conj-assoc*:
shows (*foldr op* ∧ *zs* (*x* ∧ *y*)) = (*x* ∧ (*foldr op* ∧ *zs y*))
by (*induct zs*) (*simp-all*, *blast*)

For the whole *foldr* expression to be *True*, the base expression must be *True*.

lemma *foldr-conj-base*:
shows *foldr* (*op* ∧ ∘ *M*) *xs base* ⟹ *base*
by (*induct xs*) *simp-all*

If the *VcIf*'s antecedant evaluates to *False* then the whole *foldr* expression is *True*.

lemma *foldr-negM*:
shows (¬ *M vc1*) ⟹ *foldr* (*op* ∧ ∘ *M* ∘ *VcIf vc1*) *vc2 True*
by (*induct vc2*) *simp-all*

If the *VcIf*'s antecedant evaluates to *True* then the whole *foldr* expression depends on the value of the consequent.

lemma *M-VcIf-cong*:
shows *M vc1* ⟹ *foldr* (*op* ∧ ∘ *M* ∘ *VcIf vc1*) *vc2 True* = *foldr* (*op* ∧ ∘ *M*) *vc2 True*
by (*induct vc2*) *simp-all*

The simplification procedure can introduce λ expressions, but it is sometimes easier to remove them before proceeding.

lemma *foldr-lambda*:
shows *foldr* (λ*vc*. *op* ∧ (*M vc*)) *vcs base* = *foldr* (*op* ∧ ∘ *M*) *vcs base*
by (*induct vcs*) *simp-all*

foldr-append is defined in HOL's List.thy, but sometimes it is easier to work with an appended list than a nested *foldr*. Care must be taken that the result of applying this lemma not be undone, since *foldr-append* is defined as a *simp* rule.

lemma *foldr-unappend*:
shows *foldr f xs* (*foldr f ys base*) = *foldr f* (*xs* @ *ys*) *base*
by (*induct xs*) *simp-all*

If a *foldr* expresion with the given second parameter evaluates to *True* for a list, it must also evaluate to *True* for sublists. Two particular sublists are of interest.

lemma *foldr-append-left*:
shows *foldr* (*op* ∧ ∘ *M*) (*xs* @ *ys*) *True* ⟹ *foldr* (*op* ∧ ∘ *M*) (*xs*) *True*
by (*induct xs*) *simp-all*

lemma *foldr-append-right*:
shows *foldr* (*op* ∧ ∘ *M*) (*xs* @ *ys*) *True* ⟹ *foldr* (*op* ∧ ∘ *M*) (*ys*) *True*
by (*induct ys*) (*simp-all*, *rule foldr-conj-base*, *simp*)

E Auxilliary Lemmas for Induction Steps

A few final lemmas are useful to simplify the induction steps in the proofs of the soundness and completeness lemmas.

lemma *mVcIf*:
assumes *foldr* (*op* ∧ ∘ *M*) (*distribImp vc1*) *True* ⟹ *M vc1*
foldr (*op* ∧ ∘ *M*) (*distribImp vc2*) *True* ⟹ *M vc2*
foldr (*op* ∧ ∘ *M*) (*distribImp* (*VcIf vc1 vc2*)) *True*
shows *M* (*VcIf vc1 vc2*)
proof (*cases M vc1*)
case *True*
thus *M* (*VcIf vc1 vc2*)
using *assms* **by** (*simp add*: *foldr-map M-VcIf-cong*)
next case *False*
thus *M* (*VcIf vc1 vc2*)

```
    using assms by (simp add: foldr-map)
qed

lemma mVcAnd:
  assumes foldr (op ∧ ∘ M) (distribImp vc1) True ⟹ M vc1
      foldr (op ∧ ∘ M) (distribImp vc2) True ⟹ M vc2
      foldr (op ∧ ∘ M) (distribImp (VcAnd vc1 vc2)) True
  shows M (VcAnd vc1 vc2)
proof −
  have foldr (op ∧ ∘ M) (distribImp vc1) (foldr (op ∧ ∘ M) (distribImp vc2) True)
    using assms by simp
  hence append: foldr (op ∧ ∘ M) (distribImp vc1 @ distribImp vc2) True
    by (simp only: foldr-unappend)
  hence foldr (op ∧ ∘ M) (distribImp vc1) True
    by (rule foldr-append-left)
  hence M vc1
    using assms by simp
  moreover have foldr (op ∧ ∘ M) (distribImp vc2) True
    using append by (rule foldr-append-right)
  hence M vc2
    using assms by simp
  ultimately show M (VcAnd vc1 vc2)
    by simp
qed

lemma mVcAndAnd:
  assumes foldr (op ∧ ∘ M) (distribImp vc1) True ⟹ M vc1
      foldr (op ∧ ∘ M) (distribImp vc2) True ⟹ M vc2
      foldr (op ∧ ∘ M) (distribImp (VcAndAnd vc1 vc2)) True
  shows M (VcAndAnd vc1 vc2)
proof (cases M vc1)
  case True
    hence foldr (op ∧ ∘ M) (distribImp vc1) (foldr (op ∧ ∘ M) (distribImp vc2) True)
      using assms by (simp add: foldr-map M-VcIf-cong)
    hence foldr (op ∧ ∘ M) (distribImp vc2) True
      by (simp only: foldr-unappend foldr-append-right)
    hence M vc2
      using assms by simp
    thus M (VcAndAnd vc1 vc2)
      using prems by simp
  next case False
    thus M (VcAndAnd vc1 vc2)
      using assms by (simp add: foldr-map foldr-negM)
qed

lemma distribVcIf:
  assumes M vc1 ⟹ foldr (op ∧ ∘ M) (distribImp vc1) True
      M vc2 ⟹ foldr (op ∧ ∘ M) (distribImp vc2) True
      M (VcIf vc1 vc2)
  shows foldr (op ∧ ∘ M) (distribImp (VcIf vc1 vc2)) True
proof (simp add: foldr-map, cases M vc1)
  case True
    thus foldr (op ∧ ∘ M ∘ VcIf vc1) (distribImp vc2) True
      using assms by (simp add: foldr-map M-VcIf-cong)
  next case False
    thus foldr (op ∧ ∘ M ∘ VcIf vc1) (distribImp vc2) True
      using assms by (simp add: foldr-map foldr-negM)
qed
```

F Soundness and Completeness

Showing the soundness of our splitting algorithm amounts to showing that if the conjunction of the evaluations of the sub-VCs is true then the original VC evaluates to true. Showing the algorithm's completeness is showing the converse. Thus, showing that the decomposition is both sound and complete is simply showing the implication in both directions.

lemma *soundness*:
  shows *foldr op* $\wedge$ (*map M* (*distribImp vc*)) *True* $\Longrightarrow$ *M vc*
proof (*simp only*: *foldr-map*, *induct vc*)
  case (*VcIf vc1 vc2*)
    thus *M* (*VcIf vc1 vc2*) **by** (*rule mVcIf*)
  next case (*VcAnd vc1 vc2*)
    thus *M* (*VcAnd vc1 vc2*) **by** (*rule mVcAnd*)
  next case (*VcAndAnd vc1 vc2*)
    thus *M* (*VcAndAnd vc1 vc2*) **by** (*rule mVcAndAnd*)
  next case (*VcOther b*)
    thus *M* (*VcOther b*) **by** *simp*
qed

lemma *completeness*:
  shows *M vc* $\Longrightarrow$ *foldr op* $\wedge$ (*map M* (*distribImp vc*)) *True*
proof (*simp only*: *foldr-map*, *induct vc*)
  case (*VcIf vc1 vc2*)
    thus *foldr* (*op* $\wedge$ $\circ$ *M*) (*distribImp* (*VcIf vc1 vc2*)) *True*
      by (*rule distribVcIf*)
  next case (*VcAnd vc1 vc2*)
    thus *foldr* (*op* $\wedge$ $\circ$ *M*) (*distribImp* (*VcAnd vc1 vc2*)) *True*
      by (*simp add*: *foldr-lambda*)
  next case (*VcAndAnd vc1 vc2*)
    thus *foldr* (*op* $\wedge$ $\circ$ *M*) (*distribImp* (*VcAndAnd vc1 vc2*)) *True*
      by (*simp add*: *foldr-lambda foldr-map M-VcIf-cong*)
  next case (*VcOther b*)
    thus *foldr* (*op* $\wedge$ $\circ$ *M*) (*distribImp* (*VcOther b*)) *True*
      by *simp*
qed

theorem *sound-and-complete*:
  shows *foldr op* $\wedge$ (*map M* (*distribImp vc*)) *True* = *M vc*
  using *soundness completeness* **..**
end

References

1. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference, pp. 234–245. ACM Press, New York, NY (2002). DOI http://doi.acm.org/10.1145/512529.512558
2. Chalin, P., James, P.R., 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 Verified Systems: Theories, Tools, and Experiments (2008)
3. Cok, D.R., Kiniry, J.R.: ESC/Java2: Uniting ESC/Java and JML. In: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, *LNCS*, vol. 3362/2005, pp. 108–128. Springer Berlin (2005)
4. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D.R., Müller, P., Kiniry, J.R., Chalin, P.: JML reference manual (2008). Available at `http://www.jmlspecs.org`
5. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: G. Barthe, L. Burdy, M. Huisman, J.L. Lanet, T. Muntean (eds.) CASSIS 2004: Construction and Analysis of

Safe, Secure, and Interoperable Smart Devices, International Workshop, Marseille, France, March 10-14, 2004, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 3362, pp. 49–69. Springer (2004)
6. Chalin, P., James, P.R., Karabotsos, G.: An integrated verification environment for JML: Architecture and early results. In: SAVCBS '07: Proceedings of the 2007 Workshop on Specification and Verification of Component-Based Systems, pp. 47–53 (2007)
7. Leino, K.R.M.: Toward reliable modular programs. Ph.D. thesis, California Institute of Technology, Pasadena, CA (1995)
8. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)
9. Kolman, B., Busby, R.C.: Discrete mathematical structures for computer science (2nd ed.). Prentice-Hall, Inc., Upper Saddle River, NJ (1986)
10. Chalin, P., James, P.R.: Non-null references by default in Java: Alleviating the nullity annotation burden. In: Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07). Berlin, Germany (2007)
11. James, P.R., Chalin, P.: Enhanced extended static checking in JML4: Benefits of multiple-prover support. In: ACM SAC 2009 (24th Annual ACM Symposium on Applied Computing) (2009)
12. Barnett, M., 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, pp. 82–87. ACM Press, New York, NY (2005)
13. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: Generating compact verification conditions. In: POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 193–205. ACM Press, New York, NY (2001). DOI http://doi.acm.org/10.1145/360204.360220
14. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Inc., Englewood Cliffs, NJ (1976)
15. Isabelle (2008). Homepage at http://isabelle.in.tum.de
16. Paulson, L.C., Susanto, K.W.: Source-level proof reconstruction for interactive theorem proving. In: K. Schneider, J. Brandt (eds.) Theorem Proving in Higher Order Logics: TPHOLs 2007, LNCS 4732, pp. 232–245. Springer (2007). DOI 10.1007/978-3-540-74591-4. URL http://www.cl.cam.ac.uk/~lp15/papers/Automation/reconstruction.pdf
17. Why: software verification platform (2008). Homepage at http://why.lri.fr.
18. Metis theorem prover (2008). Homepage at http://www.gilith.com/software/metis/.
19. 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, pp. 167–184. Springer-Verlag, London, UK (1999)
20. Bug 142126 - utilizing multiple CPUs for Java compiler (2008). Homepage at https://bugs.eclipse.org/bugs/show_bug.cgi?id=142126
21. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of AFIPS Conference, pp. 79–81. San Francisco, CA (1967)
22. Krishnaprasad, S.: Uses and abuses of Amdahl's law. The Journal of Computing in Small Colleges **17**(2), 288–293 (2001)
23. Grigore, R., Moskal, M.: Edit and verify. In: Proceedings of the 6th International Workshop on First-Order Theorem Proving (FTP 2007). Liverpool, UK (2007)
24. Leino, K.R.M., Monahan, R.: Reasoning about comprehensions with first-order smt solvers. In: ACM SAC 2009 SVT (Software Verification and Testing Track of the 24th Annual ACM Symposium on Applied Computing) (2009)
25. Böhme, S., Leino, R., 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), LNCS 5170. Springer (2008). URL http://www-wjp.cs.uni-sb.de/publikationen/boehme_tphols_2008.pdf
26. Filliâtre, J.C., Hubert, T., Marché, C.: The Caduceus verification tool for C programs: Tutorial and reference manual (2008). Available at http://caduceus.lri.fr.
27. Filliâtre, J.C.: The WHY verification tool: Tutorial and reference manual (2008). Available at http://why.lri.fr.
28. Karabotsos, G., Chalin, P., James, P.R., Giannas, L.: Total correctness of recursive functions using JML4 FSPV. In: SAVCBS '08: Proceedings of the 2008 Workshop on Specification and Verification of Component-Based Systems (2008)
29. Wilson, T., Maharaj, S., Clark, R.G.: Omnibus: A clean language and supporting tool for integrating different assertion-based verification techniques. In: Proceedings of REFT 2005. Newcastle, UK (2005). URL http://www.cs.stir.ac.uk/$\sim$twi/omni/papers/reft2005.pdf

30. Wilson, T., Maharaj, S., Clark, R.G.: Omnibus verification policies: A flexible, configurable approach to assertion-based software verification. In: SEFM'05, The 3rd IEEE International Conference on Software Engineering and Formal Methods (September 2005). URL http://www.cs.stir.ac.uk/$\sim$twi/omni/papers/sefm2005.pdf
31. Wilson, T.: The omnibus language and integrated verification approach. Ph.D. thesis, University of Stirling, Stirling, UK (2008)
32. Wilson, T., Maharaj, S., Clark, R.G.: Push-button tools for application developers, full formal verification for component vendors. Tech. rep., Department of Computing Science and Mathematics, University of Stirling, Stirling, UK (2006)
33. Burdy, L., Requet, A.: JACK: Java applet correctness kit. In: 4th Gemplus Developer Conference (2002)
34. Burdy, L., Requet, A., Lanet, J.L.: Java applet correctness: a developer-oriented approach. In: Formal Methods (FME'03), LNCS 2805, pp. 422–439 (2003)
35. Parallel - GNU 'make' (2006). Homepage at http://www.gnu.org/software/automake/manual/make/Parallel.html
36. Inc., A.: Xcode build system guide (2009). Available at http://developer.apple.com/documentation/DeveloperTools/Conceptual/XcodeBuildSystem/Xcode_Build_System.pdf.
37. distcc: a fast, free distributed C/C++ compiler (2008). Homepage at distcc.org
38. Icecream - openSUSE (2006). Homepage at http://en.opensuse.org/Icecream
39. Hickey, J., Nogin, A., Constable, R.L., Aydemir, B.E., Barzilay, E., Bryukhov, Y., Eaton, R., Granicz, A., Kopylov, A., Kreitz, C., Krupski, V.N., Lorigo, L., Schmitt, S., Witty, C., Yu, X.: MetaPRL — A modular logical environment. In: D. Basin, B. Wolff (eds.) Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003), pp. 287–303. Springer-Verlag, London, UK (2003)
40. Hickey, J.: Fault-tolerant distributed theorem proving. In: CADE-16: Proceedings of the 16th International Conference on Automated Deduction, pp. 227–231. Springer-Verlag, London, UK (1999)
41. Rodeh, O., Birman, K., Dolev, D.: The architecture and performance of security protocols in the ensemble group communication system: Using diamonds to guard the castle. Journal of ACM Transactions on Information Systems and Security (TISSEC) **4**(3), 289–319 (2001). DOI http://doi.acm.org/10.1145/501978.501982
42. Vandevoorde, M.T., Kapur, D.: Distributed Larch Prover (DLP): An experiment in parallelizing a rewrite-rule based prover. In: RTA '96: Proceedings of the 7th International Conference on Rewriting Techniques and Applications, pp. 420–423. Springer-Verlag, London, UK (1996)
43. Hunter, C., Robinson, P., Strooper, P.: Agent-based distributed software verification. In: ACSC '05: Proceedings of the Twenty-eighth Australasian Conference on Computer Science, pp. 159–164. Darlinghurst, Australia (2005)