Edit and Verify

Radu Grigore[1] and Michał Moskal[2]

[1] UCD CASL, University College Dublin, Belfield, Dublin 4, Ireland
[2] Institute of Computer Science, University of Wrocław, ul. Joliot-Curie 15, 50-383 Wrocław, Poland, mjm@ii.uni.wroc.pl

Abstract. Automated theorem provers are used in extended static checking, where they are the performance bottleneck. Extended static checkers are run typically after incremental changes to the code. We propose to exploit this usage pattern to improve performance. We present two approaches of how to do so and a full solution.

1 Introduction

Extended static checking [1] is a technology that makes automated theorem proving relevant to a wide group of programmers. The architecture of an Extended Static Checker (ESC) is similar to that of a compiler (see Fig. 1). It has a front-end that translates high-level code and specifications into a simpler intermediate representation, and a back-end that formulates first order logic formulas as queries for a theorem prover. The queries are called *verification conditions* (VCs). If the ESC is sound then the VC is UNSAT only if the code meets its specifications; if the ESC is complete then the program meets its specification only if the VC is UNSAT. ESC/Java2 [1] is an ESC that was designed to be unsound and incomplete (as a tradeoff to make it more usable in practice); Spec$^\#$ [2] is an ESC that was designed to be sound.

In this article we shall assume an ideal ESC that is both sound and complete. Automated first order theorem provers used in extended static checking are incomplete: They either find a proof that a formula is UNSAT or they give an assignment that *probably* satisfies the formula. As a result, even if the ESC is sound and complete, spurious warnings are possible.

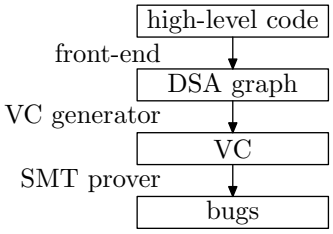

Fig. 1. The architecture of an ESC

The purpose of an ESC is to provide warnings that help programmers to write high-quality code. In practice it is used much like a compiler. Either the programmer runs it periodically or the Integrated Development Environment (IDE) runs it in the background. Because of these usage patterns, performance is quite important. The bottleneck is the prover. Luckily, the fact that the ESC is run often can be exploited since it means that the program does not change much between two runs. Compilers already exploit this by doing incremental compilation [3]. ESCs do checking in a modular way, method by method. Nevertheless, once the contract of a method is altered all its clients must be rechecked. In such a scenario the VCs of the clients do not change much.

```
// blank line                          // (1)
class Day {
  //@ ensures 1 <= \result && \result <= 12;
  public abstract int getMonth();

  //@ ensures 1970 <= \result;
  //@ ensures \result <= 2038;              // (2)
  public abstract int getYear();

  //@ ensures 1 <= \result && \result <= 31;
  public abstract int getDay();

  //@ ensures 1 <= \result;
  //@ ensures \result <= 366;               // (3)
  public int dayOfYear() {
    int offset = 0;
    if (getMonth() > 1) offset += 31;
    if (getMonth() > 2) offset += 28;
    if (getMonth() > 3) offset += 31;
    if (getMonth() > 4) offset += 30;
    if (getMonth() > 5) offset += 31;
    if (getMonth() > 6) offset += 30;
    if (getMonth() > 7) offset += 31;
    if (getMonth() > 8) offset += 31;
    if (getMonth() > 9) offset += 30;
    if (getMonth() > 10) offset += 31;
    if (getMonth() > 11) offset += 30;
    boolean isLeap = getYear() % 4 == 0 &&
                     (getYear() % 100 != 0 || getYear() % 400 == 0);
    //@ assert offset <= 335;               // (4)
    if (isLeap && getMonth() > 2) offset++;
    return offset + getDay();
  }
}
```

Fig. 2. Typical evolution of annotated Java code

This paper (1) argues for the importance of using techniques analogous to incremental compilation in software verification, (2) formalizes the problem and explores possible solutions (Sect. 2), (3) presents a specific solution that works exclusively inside an automated theorem prover (Sect. 3), in the process (4) presents a technique to heuristically determine similarities between formulas, and (5) gives a mechanically verified proof for the correctness of a part of the specific solution presented.

2 Discussion and Definitions

The problem in a nutshell is how to do incremental extended static checking. We shall explore the solution space and then we will see in detail a particular solution, including some experimental data.

Consider the JML-annotated Java code from Fig. 2 When checking the method dayOfYear the ESC will assume the implicit empty precondition holds and will try to prove the postcondition. It will also try to prove all the explicit and implicit assertions in the body. When the method getMonth is called the ESC inserts (implicit) assertions for its preconditions followed by assumptions for its postconditions. Moreover, the ESC will introduce assertions that ensure the absence of runtime exceptions. For example, the receiver object of a method call is asserted to be nonnull.

Notice the lines marked by (1), (2), (3), and (4). Adding these lines represents typical edits that can be done on annotated source code. For example, line (3) is a newly added postcondition. An incremental VC would only check if this new assertion holds, provided that the last VC was UNSAT. It is somehow cumbersome to formulate the problem precisely at the source code level. We can be more precise by descending at the level of an idealized intermediate representation, a *Dynamic Single Assignment (DSA) graph*.

Definition 1 (DSA graph) *The* DSA graph *of a method is a directed acyclic (control flow) graph. Its vertices are* $1, 2, \ldots$ *and they are labeled respectively by the first order logic formulas* $\phi_1, \phi_2, \ldots$. *A vertex represents either an* assertion *(in which case we say it is* black*) or an* assumption *(in which case we say it is* white*). We denote the set of vertices that are predecessors of* v *by* $\mathrm{in}(v)$ *and the set of successors of* v *by* $\mathrm{out}(v)$. *The* in-degree *of* v *is* $|\mathrm{in}(v)|$ *and the* out-degree *is* $|\mathrm{out}(v)|$. *The nodes with in-degree zero are called* initial nodes*; the nodes with out-degree zero are called* final nodes*.*

The assertions model the postconditions of the verified method and the checks inside its body (such as the check that an index in an array access is in-bounds, a receiver of a method call is nonnull, the preconditions of a called method hold, explicit JML assertions, and so on). The assumptions model postconditions of the called methods and semantics of the Java language (including properties ensured by the type system).

For this presentation we simply assume that the intermediate representation is obtained from the source code by some technique, without committing to any

one in particular. The curious reader can start exploring the subject from other papers [2,4,5,6].

The VC is generated from the intermediate representation. The particular algorithm used has a big impact on performance [7,5]. Here we only present a conceptually simple technique that illustrates well the general form VCs have in practice.

Definition 2 (behaviors) *Vertices have associated* preconditions *denoted by* $\alpha_1, \alpha_2, \ldots$, postconditions *denoted by* $\beta_1, \beta_2, \ldots$, *and* wrong behaviors *denoted by* $\gamma_1, \gamma_2, \ldots$ *For all* i *we have*

$$\alpha_i = \begin{cases} \top & \textit{for initial nodes} \\ \bigvee_{v_j \in \mathrm{in}(v_i)} \beta_j & \textit{for non-initial nodes} \end{cases} \tag{1}$$

$$\beta_i = \alpha_i \wedge \phi_i \tag{2}$$

$$\gamma_i = \begin{cases} \alpha_i \wedge \neg\phi_i & \textit{for assertions} \\ \bot & \textit{for assumptions} \end{cases} \tag{3}$$

Definition 3 (verification condition) *The* verification condition *is*

$$\psi = \bigvee_i \gamma_i \tag{4}$$

The wrong behaviors are something we want to avoid, therefore we ask the prover if all the wrong behaviors are impossible which is the same as asking if the VC is UNSAT. If it is, then the ESC concludes that all the assertions are valid and the method is correct. The basic idea behind the more efficient techniques of generating VCs is to generate factored form.

Old	New	Simplified
		
$\psi_1 = \phi_1 \wedge \neg\phi_2$	$\psi_2 = (\phi_1 \wedge \neg\phi_2) \vee (\phi_1 \wedge \phi_2 \wedge \neg\phi_3)$	$\psi_2' = \phi_1 \wedge \phi_2 \wedge \neg\phi_3$

Table 1. Simplification example

The problem can now be stated as follows: Given two similar formulas ψ_1 and ψ_2, find a formula ψ_2' that is UNSAT if and only if ψ_2 is UNSAT, provided that ψ_1 is UNSAT. An example is given in Table 1. The following equations show step by step how to compute ψ_2 from its corresponding DSA graph.

$$\alpha_1 = \top \qquad \beta_1 = \phi_1 \qquad \gamma_1 = \bot \tag{5}$$
$$\alpha_2 = \phi_1 \qquad \beta_2 = \phi_1 \wedge \phi_2 \qquad \gamma_2 = \phi_1 \wedge \neg\phi_2 \tag{6}$$
$$\alpha_3 = \phi_1 \wedge \phi_2 \qquad \beta_3 = \phi_1 \wedge \phi_2 \wedge \phi_3 \qquad \gamma_3 = \phi_1 \wedge \phi_2 \wedge \neg\phi_3 \tag{7}$$

To make the example concrete the reader might wish to plug in $\phi_1 = x > 2$ and $\phi_2 = x > 1$ and $\phi_3 = x > 0$.

Note that $\psi_2' = \phi_1 \wedge \neg\phi_3$ is sound too, but we do not want to drop parts of the formula that are assumptions because they can make the proof easier. The simplified formula can be obtained in two ways. One is to replace the assertions that appear in both DSA graphs by assumptions and generate the VC for the modified DSA graph; the other is to work directly on the formulas ψ_1 and ψ_2. In this paper we will explore in greater detail the latter.

In both approaches, a solution has to solve two subproblems. First, we must find a correspondence between parts of the two DSA graphs (or formulas). Second, we must simplify one of the DSA graphs (or formulas). The methods we present in the next section for finding a correspondence between parts of the formulas can be partially reused for finding a correspondence between parts of the DSA graphs. Simplifying a formula is harder than changing assertions into assumptions, but on the other hand it is independent of the particular intermediate representation used.

3 Pruning First Order Formulas

One subproblem is to find a correspondence between parts of ψ_1 and parts of ψ_2. We substitute (some) uninterpreted constants in ψ_1 by uninterpreted constants that appear in ψ_2. We also normalize the formulas with respect to commutative operators (Fig. 3). We also use hash-consing [8,9] so later terms are simply compared by reference equality.

Note that if ψ_1 is UNSAT, then any substitution that renames uninterpreted constants leaves it UNSAT. The only assumption we make in solving the second subproblem is that ψ_1 is UNSAT, so there is no 'right' or 'wrong' correspondence between old and new constants. It is true, however, that for different substitutions of constants we will end up with different results ψ_2', some bigger and some smaller. Also we need to remember not to rename interpreted constants (such as 1 and 42).

Assuming that all constants that are 'the same' have the same name in ψ_1 as in ψ_2 would not allow us to prune the VC (to $\bot$) when the programmer only renamed a variable. (Variables in the program appear as uninterpreted constants in the VC.) Even worse, the ESC encodes extra information in identifiers [10] that changes, for example, when a new line is added to the source Java file. Despite these variations, a human that sees both ψ_1 and ψ_2 is generally able to say which sub-term corresponds to which sub-term. So there are good chances to find a heuristic that works well!

```
class Term
  public Name : string
  public Children : list [Term]
def SortTerm(t)
  def CompareTerms(a, b)
    def nc = a.Name.CompareTo(b.Name)
    if (nc != 0) nc
    else LexicographicCompare(a.Children, b.Children, CompareTerms)
  def children = t.Children.Map(SortTerm)
  if (IsCommutative(t)) Term(t.Name, t.Children.Sort(CompareTerms))
  else                  Term(t.Name, children)
def oldVC = SortTerm(oldVC)
def newVC = SortTerm(newVC)
```

Fig. 3. Normalizing queries

We only consider renaming of uninterpreted constants because of the particular algorithm used to build VCs. If some of the function symbols would also need to be renamed, the algorithm can be easily extended by the standard technique of introducing a special function symbol *apply*, and replacing $f(t_1, \ldots, t_n)$ with $apply(f, t_1, \ldots, t_n)$.

The heuristic we use to find a good substitution assigns a *similarity* value to each pair of (old, new) constants and then finds a maximum bipartite matching (using the Hungarian method [11]) between the old and the new constants. A complete bipartite graph is constructed from the set V_1 of uninterpreted constants that appear in ψ_1 and the set V_2 of uninterpreted constants that appear in ψ_2. Each pair $(i, j) \in V_1 \times V_2$ has an associated weight, which in this case is the similarity of the two constants. A matching is a subset $M \subset V_1 \times V_2$ such that for all pairs $(i, j) \in M$ and $(i', j') \in M$ we have $i = i'$ if and only if $j = j'$. The weight of the matching is the sum of the weights of all its elements. The similarity has two components: One is the length of the longest common subsequence [12] of the two identifiers; the other, more important, is how many times the constants appear in similar positions in the two VCs.

To measure similarity of position we use path strings [13]. A *path string* is a sequence of function symbols interleaved with the positions, on a path from the root of the term to a particular occurrence of a sub-term. For example $f.2.g.1$ is a path string for the occurrence of b in $f(a, g(b, c))$, and $f.2.g.2$ is a path string for c. We construct a *stripped path string* by treating logical connectives as function symbols, the entire formula as a term, and skipping positions for commutative symbols. For example $\wedge.\vee.f.2.g.1$ is the stripped path string for b in $(f(a, g(b)) \vee g(c)) \wedge g(d)$. The *environment* of a constant c in a formula ψ is the multiset of the stripped path strings for all occurrences of c in ψ. Let E_1 be the environment of x in ψ_1 and E_2 be the environment of y in ψ_2. The similarity of x and y is $2|E_1 \sqcap E_2| - |(|E_1| - |E_2|)|$, where $\sqcap$ is multiset intersection. Other measures, that take environments into account, are also possible.

```
def Prune(p1 : list [ list [Term]], p2 : Term)
  def p1 = Flatten(p1)
  // |p1| is a DNF form, assumed to be UNSAT
  match (p2.Name)
    | "and" =>
        mutable common = []
        foreach (x in p1) foreach (y in x) common = y :: common
        def p1 = p1.Map(x => x.Filter(y => !common.Contains(y)))
        def p2 = p2.Children. Filter (y => !common.Contains(y))
        if (p1.Contains ([])) Term("false", [])
        else                  Term("and", common + p2.Map(x => Prune(p1, x)))
    | "or" =>
        Term("or", p2.Children.Map(x => Prune(p1, x)))
    | _ =>
        if (p1. Exists (x => Implies(p2, Term("and", x)))) Term ("false", [])
        else                                               p2
def prunedVC = Prune([[oldVC]], newVC)
```

Fig. 4. Pruning the VC

The algorithms are presented as Nemerle-like pseudocode [14]. Some obvious optimizations are omitted[3] to improve readability. We also omit textbook algorithms. The algorithm for normalizing queries with respect to commutative operators is given in Fig. 3. It recursively sorts arguments of commutative operators using lexicographic ordering.

The second subproblem, simplification of formulas, is solved by the pruning algorithm in Fig. 4. The function Prune returns a formula equisatisfiable to p2 under the assumption that all elements of p1 are UNSAT. Elements of p1 are conjunctions represented as lists.

The function Implies explores the structure of two formulas and returns **true** only if the first is stronger than the second. The last branch is clearly correct: If p2 is stronger than a conjunct known to be UNSAT then it is also UNSAT. In the case that p2 is a disjunction we can treat its children independently. The case when p2 is a conjunction is more interesting. To understand why it works consider a small example.

$$\psi_1 = (\phi_1 \wedge \phi_2) \vee (\phi_3 \wedge \phi_4) \tag{8}$$

$$\psi_2 = \phi_2 \wedge \phi_4 \wedge (\phi_1 \vee \phi_3) \tag{9}$$

$$\psi_2' = \phi_2 \wedge \phi_4 \wedge \bot = \bot \tag{10}$$

We write $\mathrm{P}(\psi_1, \psi_2) = \psi_2'$ for the result of pruning ψ_2 under the assumption that ψ_1 in UNSAT. The common part of ψ_1 and ψ_2, as computed in the variable common in Fig. 4, is $\phi_2 \wedge \phi_4$. Pruning $\phi_1 \vee \phi_3$ knowing that $\phi_1 \vee \phi_3$ is UNSAT results in $\bot$. The formulas that appear in both ψ_1 and ψ_2 can always be factored.

[3] See `http://nemerle.org/svn.fx7/branches/fx8/Pruner.n` for all details.

$$(\phi_1 \wedge \phi_2) \vee (\phi_3 \wedge \phi_4) \tag{11}$$
$$\Leftarrow (\phi_1 \wedge \phi_2 \wedge \phi_4) \vee (\phi_3 \wedge \phi_2 \wedge \phi_4) \tag{12}$$
$$\Leftrightarrow \phi_2 \wedge \phi_4 \wedge (\phi_1 \vee \phi_3) \tag{13}$$

Hence, we can always reduce the problem to the form

$$\psi_1 = \phi_1' \wedge \phi_2' \tag{14}$$
$$\psi_2 = \phi_1' \wedge \phi_3' \tag{15}$$
$$\psi_2' = \phi_1' \wedge \mathrm{P}(\phi_2', \phi_3') \tag{16}$$

where ϕ_1' is the common part and ϕ_2' is what we assume to be UNSAT while pruning ϕ_3' (see also Fig. 4). In this example $\phi_1' = \phi_2 \wedge \phi_4$ and $\phi_2' = \phi_3' = \phi_1 \vee \phi_3$. It is easy to see that the above is correct, by doing a case analysis on whether $\phi_1'(\mathbf{x})$ holds for some vector $\mathbf{x}$. The formalization[4] in Coq [15] of a simplified version of the pruning function emphasizes the main points of the proof. The formulas abstract theories by arbitrary predicates over the domain of uninterpreted constants.

```
Inductive Formula : Type :=
  | FPred : (Dom -> Prop) -> Formula
  | FAnd : Formula -> Formula -> Formula
  | FOr: Formula -> Formula -> Formula.
Fixpoint Eval (f : Formula) (x : Dom) {struct f} : Prop :=
  match f with
    | FPred p => p x
    | FAnd fa fb => Eval fa x /\ Eval fb x
    | FOr fa fb => Eval fa x \/ Eval fb x
  end.
```

The simplified version of the algorithm whose proof we check mechanically is

```
Fixpoint Prune (p1 p2 : Formula) {struct p2} : Formula :=
  match p1, p2 with
    | FAnd a b, FAnd aa c => if eq a aa then FAnd a (Prune b c) else p2
    | _, FOr a b => FOr (Prune p1 a) (Prune p1 b)
    | _, _ => if eq p1 p2 then FPred PFalse else p2
  end.
```

This function has two important invariants.

```
Lemma PruneInvA : forall p1 p2 : Formula, forall x : Dom,
  (~ Eval p1 x -> Eval p2 x -> Eval (Prune p1 p2) x).
Lemma PruneInvB : forall p1 p2 : Formula, forall x : Dom,
  (~ Eval p1 x -> Eval (Prune p1 p2) x -> Eval p2 x).
```

[4] Available at `http://radu.ucd.ie/hp/papers/ev.html`

These are proved by double induction on the structure of p1 and p2. We use one extra fact.

```
Lemma UnsatImp : forall a b : Formula,
  ( forall x : Dom, Eval a x -> Eval b x) -> Unsat b -> Unsat a.
```

At this point we can prove that the algorithm is sound and complete.

```
Lemma PruneSound : forall p1 p2 : Formula,
  Unsat p1 -> Unsat (Prune p1 p2) -> Unsat p2.
Lemma PruneComplete : forall p1 p2 : Formula,
  Unsat p1 -> Unsat p2 -> Unsat (Prune p1 p2).
Theorem PruneCorrect : forall p1 p2 : Formula,
  Unsat p1 -> (Unsat p2 <-> Unsat (Prune p1 p2)).
```

The algorithm in Fig. 4 is more efficient since it exploits the associativity and commutativity of the $\wedge$ and $\vee$ operators. The worst case time complexity is $O(mn)$, and arises when the formula known to be UNSAT and the formula to be simplified have, respectively, the form

$$\psi_1 = \bigvee(\phi_1, \ldots, \phi_{m-1}) \tag{17}$$

$$\psi_2 = \underbrace{\wedge \ldots \wedge}_{n \text{ times}} \phi_m \tag{18}$$

where $\wedge$ and $\vee$ are written as $n$ary operators. Unfortunately, the average case that appears in practice is hard to describe. Experimental data from 20 cases suggests that the running time grows linearly with the size of the formulas. But we need more data before we can make a definite statement (see Sect. 5 for details).

4 Case Study

In this section, we explain how the common way of editing programs affects the DSA and therefore also the VC and how pruning exploits the changes.

Let us again consider the program from Fig. 2. We used ESC/Java2 to generate VCs for a version without any of the lines marked (1), (2), (3), and (4). This was the base case. Next we ran it on a method with only line (1) added, only line (2) added and so forth. Finally we ran the pruning algorithm with the old formula being the base case and the new formula being being VC for a method with an added line. Table 2 lists three times for each such formula. The first is the time it takes to prove the formula using Simplify [16]; the second is the time it takes to prune the formula; the third is the time it takes to prove the pruned formula. The reader can note that the running times of Simplify on the original formulas vary rather nondeterministically. In particular, one would expect the base case and the one with an added empty line to have the same running time, but they do not. The reason for this is a "butterfly effect" in the prover, where for example a slight change in the selection of a literal for a case split can cause large changes in the final shape of the proof search tree.

Marker	Description	Original	Pruning	Pruned
	base case	20.91s		
(1)	empty line	17.59s	2.23s	0.01s
(2)	irrelevant postcondition	16.91s	2.31s	0.06s
(3)	additional postcondition	21.65s	2.19s	19.34s
(4)	assertion in the middle	22.81s	2.16s	7.67s

Table 2. Case study results

The first edit operation (marked by (1)) is adding an empty line somewhere, or in general changing the locations of symbols. As ESCs often use location information for encoding symbol names, the uninterpreted constants in the second VC are different than in the first one. Our algorithm generates a query that is just $\bot$.

The second edit strengthens the postcondition of a method getYear used in the verified dayOfYear method. Here, we are able to prune almost everything, i.e. the resulting query is propositionally UNSAT.

The third edit adds a postcondition to the verified method. We can imagine that the DSA graph gets one more black node at the end, so this is the only thing that should be verified now. In this case we do prune parts of the formula, it however fails to speed up checking.

Finally the last edit adds an assertion near the end of the method. Here the heuristics work well and the time is reduced considerably.

The dayOfYear method (Fig. 2) is an example of a case where the VC is relatively small (around 60 kilobytes), but hard to prove. This is due to the large number of possible paths in the method. There are other reasons methods can be hard to prove: methods can be more complicated, the specifications can be complicated, the modelling of the language can be more accurate (for example in multi-threading programs). All those scenarios are good for our pruning algorithm as it runs in polynomial time and can potentially save a lot of proving time. The bad case is when the formula is large, but not that hard to prove. In particular it sometimes happen that most of the time is spent just reading/writing the formula and doing basic preprocessing, like skolemization.

5 Related and Future Work

The work presented here parallels the work done in the compiler community under the name *incremental compilation*. In the context of software verification by theorem proving the term *incremental verification* is taken—it refers to the process of proving stronger assertions using weaker ones as lemmas [17]. Hence, we use the distinct term *edit and verify* for the related idea of proving only what has not been proven before, and doing so automatically. In the context of interactive theorem proving the term *proof reuse* is used for a similar technique [18].

A Program Verification Environment (PVE) is the same for an ESC, as an Integrated Development Environment (IDE) is for a compiler. It provides an easy to use interface to the tool. As incremental compilation is very useful in IDEs, we expect Edit and Verify to be even more useful in PVEs. This is because static verification consumes much more resources than compilation. There is much research on software verification using PVEs, there is also vast amount of interest from the industry in PVEs.

One of the goals of the MOBIUS research project [19] is to produce a PVE for Java. Penelope [20] is an early PVE that processes a subset of Ada. Its designers chose to rely on interactive theorem proving. The KeY Tool [21] is a modern PVE for Java that uses the same approach but differs in the mechanisms and theory of verification condition generation. Spec$^\#$ [2] is a modern PVE for C$^\#$ that uses automated theorem proving. ESC/Java2 [1,22] is an ESC for JML-annotated [23] Java code. It produces VCs in the Simplify [16] format and in the SMT format [24] for other automated theorem provers. It also generates VCs for the Coq interactive theorem prover [15].

Whether an ESC is considered a PVE or not depends chiefly on how well integrated it is with the editor. ESC/Java2 is integrated into Eclipse using a plugin. Spec$^\#$ is more tightly integrated into Visual Studio using a plugin. Work on incremental compilation [3] suggests that an even tighter integration leads to important performance benefits.

There are two improvements that we will try in the near future. One is to prune the DSA graph. The other is to modify Fx7 [25] to produce a formula weaker than the query but still UNSAT, and use that to prune subsequent queries. Another idea that is worth exploring is to integrate pruning more tightly not with the ESC but instead with the proving process. For example, we could save the relevance of specific axioms in the old proof, so they can be prioritized while searching for a proof of the new query.

To assess the effectiveness of these improvements we need a better benchmark. The amount of JML-annotated Java is still modest. Moreover, code from the version control history is not appropriate because the commit cycle is typically much longer that the duration between two invocations of ESC/Java2. Therefore we need to collect such data ourselves and this is a time consuming effort. Such a benchmark would hopefully nicely complement the existing (very useful) Boogie benchmarks and SMT-COMP benchmarks [24]. A theoretical analysis seems to require a good model for the type of queries that are produced as verification conditions.

An idea very similar to the one explored in this paper did lead to interesting results in model checking [26], the so called *extreme model checking*. Model checking is sometimes used together with unit testing and therefore it is run often on code with minor modifications. Therefore, it is natural to take advantage of the results of previous runs.

6 Conclusion

We described the typical usage pattern of automated theorem proving in extended static checking and two approaches that exploit it to improve performance. We gave a detailed solution that processes first order formulas. The implementation is a part of the Fx7 theorem prover [25]. It was tested on queries generated by ESC/Java2, without requiring any modifications to the latter. The other approach, working on the intermediate representation of the extended static checker, promises to be more efficient but requires a tighter integration of the prover with the checker.

The first part of the solution is a heuristic that, given two formulas, finds which sub-terms of one formula correspond to which sub-terms of the other. This heuristic may prove to be a useful technique in solving related problems since it performs well and there is ample room for tuning. The second part of the solution is a formula pruning algorithm. This algorithm is proven correct, and part of the proof is mechanically verified. Its efficiency is reasonable because of the use of hash-consing and because formulas are normalized with respect to commutative operators. The pruned formulas are clearly easier to prove.

Acknowledgements. This work is partly funded by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. The article contains only the authors' views and the Community is not liable for any use that may be made of the information therein. The second author is partially supported by Polish Ministry of Science and Education grant 3 T11C 042 30.

The authors would like to thank Joseph Kiniry, Mikoláš Janota, and Fintan Fairmichael for their detailed feedback on a draft of this article. The authors would also like to thank the anonymous reviewers who pointed out that a formal analysis of the performance gains is needed. We will try to include such an analysis once the work progresses.

References

1. Flanagan, C., Leino, K., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002). (2002) 234–245
2. Barnett, M., Leino, K., Schulte, W.: The Spec$^{\#}$ programming system: An overview. In: Proceeding of CASSIS. Volume 3362 of Lecture Notes in Computer Science., Springer–Verlag (2004)
3. Schwartz, M.D., Delisle, N.M., Begwani, V.S.: Incremental compilation in Magpie. Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction (1984) 122–131
4. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology **3**(6) (2004) 27–56

5. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In Ernst, M.D., Jensen, T.P., eds.: Workshop on Program Analysis For Software Tools and Engineering, ACM Press (September 2005) 82–87
6. Darvas, A., Müller, P.: Reasoning about method calls in JML specifications. Formal Techniques for Java-like Programs (2005)
7. Flanagan, C., Saxe., J.B.: Avoiding exponential explosion: generating compact verification conditions. Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (2001) 193–205
8. Ershov, A.P.: On programming of arithmetic operations. Communications of the ACM **1**(8) (1958) 3–6
9. Filliâtre, J.C., Conchon, S.: Type-safe modular hash-consing. In: Proceedings of the 2006 workshop on ML, New York, NY, USA, ACM Press (2006) 12–19
10. Leino, K.R.M., Millstein, T., Saxe, J.B.: Generating error traces from verification-condition counterexamples. Science of Computer Programming **55**(1–3) (2005) 209–226
11. Knuth, D.E.: The Stanford GraphBase: A platform for combinatorial computing. ACM Press (1993) See the program `assign_lisa`.
12. Hirschberg, D.S.: A linear space algorithm for computing maximal common subsequences. Communications of the ACM **18**(6) (1975) 341–343
13. Ramakrishnan, I.V., Sekar, R.C., Voronkov, A.: Term indexing. In Robinson, J.A., Voronkov, A., eds.: Handbook of Automated Reasoning. Elsevier and MIT Press (2001) 1853–1964
14. : The Nemerle programming language website `http://nemerle.org/`.
15. Casteran, P., Bertot, Y.: Interactive Theorem Proving And Program Development: Coq'Art—the Calculus of Inductive Constructions. Springer (2004)
16. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Journal of the ACM **52**(3) (2005) 365–473
17. Uribe, T.E.: Combinations of model checking and theorem proving. Proceedings of the Third International Workshop on Frontiers of Combining Systems **1794** (2000) 151–170
18. Beckert, B., Klebanov, V.: Proof reuse for deductive program verification. Software Engineering and Formal Methods (2004) 77–86
19. : The Mobius project website `http://mobius.inria.fr/`.
20. Guaspari, D., Marceau, C., Polak, W.: Formal verification of Ada programs. IEEE Transactions on Software Engineering **16**(9) (1990) 1058–1075
21. Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software: The KeY Approach. LNCS 4334. Springer-Verlag (2007)
22. Cok, D., Kiniry, J.: ESC/Java2: Uniting ESC/Java and JML. Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices **3362** (2005) 108–128
23. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. Behavioral Specifications of Businesses and Systems (1999) 175–188
24. : SMT-LIB: The satisfiability modulo theories library `http://www.smt-lib.org/`.
25. Moskal, M.: Fx7 or it is all about quantifiers (SMTCOMP, 2007) Also, `http://nemerle.org/fx7/`.
26. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme model checking. In: Verification: Theory and Practice. Springer (2004)