Efficiency of Extended Static Checkers

PhD research plan: 1 Oct 2005 – 1 Oct 2009

Radu Grigore

March 25, 2008

1 Background

1.1 Programs have bugs

Most research in computer science, software engineering, formal methods, and perhaps a few other programming-related disciplines contributes techniques and ideas that improve the quality of software. The quality has two faces. A program is good if it does what the programmers intended. A program is really good if it does what its users want. The problem of making programmers' intentions agree with the users' requirements will be neatly covered under the rug in what follows. Rest assured, there are many qualified people working on it. So let us shift the focus towards making programs agree with programmers. For sure, no programmer wants their editor to change the selected region after a search and replace operation, or to suddenly display a blue screen and stop react to user input. These disagreements between the program and the programmer are commonly called *bugs*.

An empirical study [CAHM04] of active open source projects revealed that the average number of bugs per project is 509 and the median is 279. Moreover, projects with many downloads tend to also have many developers, many bug reports, and low bug-fixing time. It is not clear how these numbers should be interpreted, but I believe they do show that programs have bugs. They also show that bugs should be fixed early. Extrapolating, it seems a good idea to fix bugs before they reach the users.

Many programming tools, including compilers, are actively pointing to possible trouble spots. Programmers are actively looking for bugs too. Yet, bugs still reach users in high numbers. It is quite unlikely that one idea or technique that decisively impacts the number of bugs released to users will be found [FPB95]. Instead, the constant stream of research output, together with advocacy and teaching, will steadily raise the bar. Even if this is the most likely future evolution, most people, including researchers, can't stop to have personal favorite techniques. My current favorite is *extended static checking* [BLS04]. And, judg-

ing from the fact that it has been my favorite for almost two years, it is a solid favorite.

1.2 Program annotations

Reading code often feels like solving a puzzle. Indeed, the problem column of the first issue of the *ACM Transactions on Algorithms* was essentially "what does this program do?" Sometimes the puzzle is quite easy and you don't even notice. Sometimes it is entertaining. But many times it stands in the way of getting the job done. Even worse, sometimes your understanding of the code (the 'solution of the puzzle') is wrong and affects your subsequent actions.

Using standard algorithms, following code conventions, writing concise code, having an architectural overview — all help. In my experience, documentation close to the code helps most. Norm Schryer noted that "when the code and the comments disagree, both are probably wrong." Because comments are sometimes out-of-sync with the code programmers prefer not to write them, unless they are coerced into doing so. *Program annotations* are machine (and human) readable comments. Extended static checkers verify that program annotation agree with the code. Java Modeling Language (JML) [LBR06] is designed to allow programmers to express part of what they usually write in English comments in a formal way.

1.3 Warnings

Compiler warnings are *the* tool I use to improve the quality of the code I write. Although many articles state that warnings are an effective tool [MRS07], agreeing with my experience, I was surprised to not find any convincing empirical studies.

Many good developers keep warnings turned on and make sure that their number is zero. If the warnings are not turned on from the beginning it is very hard to use them later on. Usually there is a flood of irrelevant ones and it takes to much time to go through all warnings and figure out which are related to bugs and which not.

One typical usage is to run the compiler from the command line periodically and fix any errors or warnings emitted. An alternative usage is to use a development environment that runs in the background the equivalent of a compiler and underlines problematic pieces of code. If compiling after a small change would take forever (that is, more than 30 seconds) or if running the compiler in the background would make the editor's response jerky, then programmers wouldn't take advantage of the static analyzes built into compilers. Efficiency is important.

So far I only gave a high-level motivation of why I am interested in making extended static checkers efficient. Of course, the real reason is that I think it's fun to design algorithms that work fast in practice. In any case, whether you are interested in the possible outcomes or in the problem solving aspect of the

work, I ask you to have a little faith and believe that extended static checkers have a chance to do good.

1.4 Extended static checkers

Extended static checkers are defined by two characteristics:

- how they are meant to be used, and
- their architecture.

Extended static checkers are meant to be used much like compilers, except that they do not generate code. They are either run from the command line periodically or they are run in the background by a development environment. Efficiency is therefore important. (Note: Some authors prefer to use other terms such as *program verifiers* when they refer to extended static checkers.)

We will now descend into more concrete realms to describe the architecture.

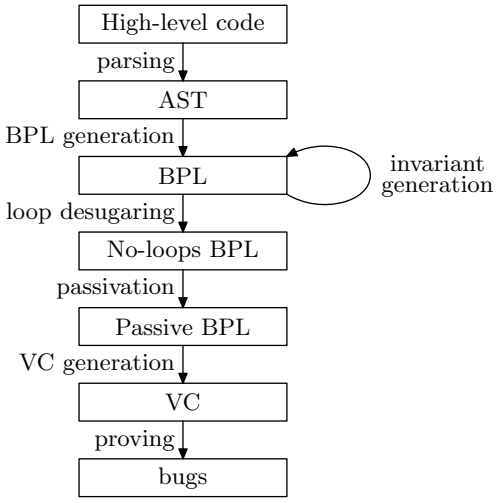

Figure 1: The architecture of an ESC

The input is written in a high-level language, like Modula-3, Java, or C#. The frontend translates the Abstract Syntax Tree (AST) into a simpler language, Boogie Programming Language (BPL). This first phase is responsible for *desugaring* the more complicated features of programming languages such as inheritance into a language that has procedures and only a few types of statements. The frontend is therefore quite similar to that of a compiler (for example, see [gcc]). For languages like Java and C#, which compile to an intermediate representation, it is convenient to use the bytecode as input for the checker. Here I will focus on the backend.

The BPL program then undergoes a few transformations. First, techniques such as abstract interpretation are used to infer properties that are not explicitly

stated in annotations. For example, abstract interpretation can infer that $0 \leq i \leq N$ is an invariant for the loop **for**($i \leftarrow 0$;$i<N$;$++i$) /*...*/. Then loops are cut according to Hoare logic and the flow graph of the remaining program is acyclic. The relevant rule from Hoare logic is the following:

$$\frac{\{I \wedge C\}\, B\, \{I\}}{\{I\}\, \mathbf{while}(C) B\, \{I \wedge \neg C\}} \tag{1}$$

In the *passivation* stage, the code is transformed such that for each execution trace there is at most one assignment to each variable. Such a program could run on a computer with write-once memory. This form is called Dynamic Single Assignment (DSA). At this point all assignments can be replaced by assumes, which are similar in this context to variable definitions in functional languages. The next step is verification condition (VC) generation. This can be done using either a weakest precondition (wp) or a strongest postcondition (sp) calculus. The result is a first-order logic (FOL) formula. A theorem prover sets on to find counterexamples for this formula and, if found, these are translated into warnings that refer to the original code.

The input or output of a few stages is expressed in BPL. Other representations are possible, but having one that is also easily readable by humans helps the development of tools and loosens the coupling between frontends and backends. If we want to support l languages and a analyzes we would typically need la checkers, while by having loosely coupled frontends and backends we need only $l + a$ components.

1.5 The research problem

Extended static checkers are noticeably slower than compilers. The research challenge addressed by this plan is *to improve the efficiency of extended static checkers* so that the chances of seeing them used in practice increase.

2 What was achieved

2.1 Reachability analysis

Reachability analysis is performed by most compilers as an optimization. If code is unreachable because of a preceding return statement then the Java compiler stops with an error. For other types of unreachable code, such as a branch guarded by a condition that always evaluates to **false**, not even a warning is produced. Both situations are suspect and likely to hide bugs, so the user should have an easy way to discover them. Both situations can be compiled, so a warning would suffice.

In the presence of annotations it is possible to do a more precise analysis.

```
public static int indexOf(int x, /*@non_null*/ int [] a) {
  int r;
  //@ loop_invariant 0 ≤ r ∧ r ≤ a.length;
  for (r ← 0; r < a.length ∧ a[r] ≠ x; ++r);
```

```
  if (r > a.length) System.out.println("ouch¬");
  if (r = a.length) r ← −1;
  return r;
}
```

In the preceding code the Java compiler can't determine that the print statement will never be executed, while ESC/Java verifies that the given expression is a loop invariant and uses it to provide the following warning.

```
RA.java:6: Warning: Code not checked. (Pre)
if (r > a.length) System.out.println("ouch!");
                                     ^
```

A case study that presents what kind of problems are discovered by such warnings is available [JGM07]. The Java code above is first transformed into a corresponding BPL program.

```
procedure indexOf(x : int , a : ref ) returns (r : int );
  requires a ≠ null;
{
start :
  r ← 0;
  assert 0 ≤ r ∧ r ≤ length(a);
  goto loop_body, after_loop
loop_body:
  assume 0 ≤ r ∧ r ≤ length(a);
  assume r < length(a) ∧ elem(a, r) ≠ x;
  r ← r + 1;
  assert 0 ≤ r ∧ r ≤ length(a);
  assume false;
after_loop :
  assume 0 ≤ r ∧ r ≤ length(a);
  assume ¬(r < length(a) ∧ elem(a, r) = x);
  goto first_if_true , first_if_false ;
 first_if_true :
  assume r > length(a);
  call System.out. println (/* ... */)
  goto second_if_true , second_if_false ;
  first_if_false :
  assume ¬(r > length(a));
  goto second_if_true , second_if_false ;
second_if_true :
  assume r = length(a);
  r ← −1;
  goto exit ;
second_if_false :
  assume ¬(r = length(a));
  goto exit ;
exit :
}
```

The code above already has an acyclic control-flow thanks to one application of the Hoare rule (1). It is not passive because there are execution traces that assign to r more than once. In such situations r is replaced by a set of variables, essentially an array. To my knowledge, there is no formal presentation of how passivation is done and what algorithms are used.

Variables can be analyzed one-by-one. The reader can think of the variable r in the previous example. We replace the variable by an array and specify for each node in the control-flow graph a *read index* and *write index*. If the statement originally read variable r then in the modified program it will read from the array the element in the position given by the read index; and analogously for writing. (Note that we can assume at most one read and one write per statement.) We say that the modified program is equivalent to the original if the read (and written) values are the same for each statement for all executions. The extra requirement we pose is that we never write more than once to any element of the array.

A passive version of the BPL example follows.

```
procedure indexOf(x : int , a : ref ) returns ( r_1 : int );
   requires a ≠ null;
{
start :
   assume r_0 = 0;
   assert 0 ≤ r_0 ∧ r_0 ≤ length(a);
   goto loop_body, after_loop
loop_body:
   assume 0 ≤ r_0 ∧ r_0 ≤ length(a);
   assume r < length(a) ∧ elem(a, r_0) ≠ x;
   assume r_1 = r_0 + 1;
   assert 0 ≤ r_1 ∧ r_1 ≤ length(a);
   assume false;
after_loop :
   assume 0 ≤ r_0 ∧ r_0 ≤ length(a);
   assume ¬(r_0 < length(a) ∧ elem(a, r_0) = x);
   goto first_if_true , first_if_false ;
  first_if_true :
   assume r_0 > length(a);
    call System.out. println (/* ... */)
   goto second_if_true , second_if_false ;
   first_if_false :
   assume ¬(r_0 > length(a));
   goto second_if_true , second_if_false ;
second_if_true :
   assume r_0 = length(a);
   assume r_1 = −1;
   goto exit ;
 second_if_false :
   assume ¬(r_0 = length(a));
   assume r_1 = r_0;    // copy operation
   goto exit ;
exit :
}
```

Since all variables are written to once, the assignments were replaced by assumes. Note also that it was necessary to introduce a *copy* operation.

At this point there are two obvious criteria that can be optimized by passivation: the size of the array and the number of copy operations. It is relatively easy to see that some examples of 'optimal' passive versions of programs in the literature [BL05] are not optimal for either of the two criteria mentioned.

Optimizing for the size of the array is trivial. One merely has to see the

control-flow graph of write operations as a *comparison graph* and then use a minimal coloring for the later. This is done in linear time. The minimal size of the array is always the maximal number of write operations on some execution path. In contrast, I believe it is not known whether an efficient algorithm for minimizing the number of copy operations exists. I (and the authors of [BL05]) suspect it does not.

At this point the program is represented by a directed acyclic graph whose nodes are tagged with either an **assume** or an **assert** followed by a FOL formula. The *precondition* of these nodes can be computed according to the definition.

$$\psi_a = \begin{cases} \top & \text{when } a \text{ is an initial node} \\ \bigvee_{b \to a} \psi_b \wedge \phi_b & \text{otherwise} \end{cases} \tag{2}$$

where ϕ_b is the formula attached to node b. It is explained in the paper [JGM07] that (semantically) unreachable nodes are exactly those with an unsatisfiable precondition. Thus, a naive solution would be to ask the theorem prover if the precondition of each node is satisfiable. This is not usable in practice.

A better alternative is to design an algorithm than tries to minimize the number of calls to the prover, which is by far the most expensive operation. Such an algorithm can take advantage of the particular way preconditions are computed. For example, suppose that the program is a chain of n statements $1 \to 2 \to \cdots \to n$ and the theorem prover says that n is reachable. We can immediately conclude that $1, \cdots, n-1$ are also reachable. In general, the information about reachability can be propagated in the flow-graph using the following two rules.

- If u dominates a reachable node v then u is reachable, and
- if each path from the initial node to u contains an unreachable statement then u is unreachable.

(Note that we can assume there is only one initial node, that is, without predecessors, without any loss of generality.)

We say that a statement u *dominates* a statement v if all paths from the initial statement to v contain u. We say that u is an *immediate dominator* of v if it does not dominate any other dominator of v. With these definitions the algorithm I designed can be presented as follows.

PROPAGATE-UNREACHABLE(u)
  label u as unreachable
  for each child v of u
    such that v has only unreachable parents
      do PROPAGATE-UNREACHABLE(v)

PROPAGATE-REACHABLE(u)
  label u as reachable
  if u has an immediate dominator d
    then PROPAGATE-REACHABLE(d)

```
REACHABILITY-ANALYSIS()
  while there are unlabeled nodes
      do choose an unlabeled node u that has
           a maximal number of unlabeled dominators
         if the prover says that
           the precondition of u is reachable
           then PROPAGATE-REACHABLE(u)
           else use binary search with prover queries
                  to identify the farthest
                  unreachable dominator d of u
                PROPAGATE-UNREACHABLE(d)
                RECOMPUTE-DOMINATORS
                if d has an immediate dominator d'
                  then PROPAGATE-REACHABLE(d')
```

This algorithm is greedy and it queries the prover a minimal number of times when all the statements are reachable, which is common. For the case when there are unreachable statements I do not have a formal analysis of the performance of the algorithm. A case study that presents running times and further details are found in the paper [JGM07].

2.2 Edit and verify

What is even more important than making new analyzes run in reasonable time is to make faster the core analysis, the search for possible assertion failures. One possible approach is to exploit the incremental nature of verification. In other words, we optimize for the typical case. Programmers are envisioned to use extended static checkers just like good programmers use compilers today. The tool is run often and the code (plus annotations) is adjusted so that dubious idioms are avoided. The second and subsequent runs of the extended static checker should be faster because something similar was already 'seen'.

To exploit the incremental nature of verification the extended static checker must identify differences and use results of previous runs. The code plus annotations is first transformed into BPL and then into FOL. A program that identifies differences can work on any of these three representations. I and Michał Moskal investigated in detail how to identify differences at the FOL level and how to exploit the last previous known good state of the program [GM07].

The formulas are typically a few hundred kilobytes in size. Small changes to the code may result in the formulas being shuffled and constants being renamed. Still, a human can typically tell what are the important differences. The task was to design an algorithm that does the same. The result was a heuristic that works in most cases and whose running time is small. In some cases the algorithm reports a set of difference much bigger than a human would. In such cases we say that it failed; the overall runtime increases, but the results of the extended static checker are still correct. The runtime of the algorithm that finds differences is quadratic in the worst case, but in practice it is very fast.

The problem of doing a 'smart diff' on two FOL formulas seems general enough to warrant some hope that the developed algorithm will be useful in other circumstances. Let us see how it works. The key ingredients are: (1) find a mapping between old constants and new constants, (2) sort formulas in tree lexicographic order, and (3) use hash-consing. These steps are not executed sequentially. Hash-consing is a standard technique to implement data structures such that structural equality implies reference equality. The data structures used to represent the formulas must be implemented in this way. The time ordering of steps (1) and (2) is not important.

(Note: Lexicographic ordering is usually defined for sequences, but there is a fairly obvious way to extend the definition for trees.)

To mapping find a, we begin by assigning a similarity score to each pair of (old, new) constants. The similarity is based on the edit distance and, more importantly, on the *environment*. The environment of a constant that appears in a formula is the multiset of modified path strings that point to all appearances of the constant. The environment of x in $f(g(x), h(a, x), x)$, assuming that h is the only commutative function, is $\{f.1.g.1, f.2.h, f.3\}$; the argument index is missing for commutative functions.

The similarity score is used as cost in a bipartite graph whose left nodes are the old constants and whose right nodes are the new constants. The mapping is given by a maximum bipartite matching in this graph. New constants substitute their old counterparts.

At this point, reference equality can be used to quickly check if a sub-term of the old formula corresponds to a sub-term of the new formula. This operation suffice for the later pruning stage.

A much-simplified version of the pruning algorithm uses the following data structures:

```
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 algorithm 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.
```

The correctness of the algorithm is captured by the following theorem:

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

Further details are given in the paper [GM07].

2.3 Lightweight abstract grammars

Language processing tools written in object-oriented programming languages are traditionally organized in a way that makes it easy to modify the abstract grammar of the input language or in a way that makes it easy modify the processing done to the input, but not both. Simple object-oriented design leads to the first situation; the use of the Visitor pattern leads to the second situation. A number of solutions were designed to address this issue [Par, GH, Sun, HM03]. All seem to be heavyweight, to have a steep learning curve, and to not allow aggressive customization. As a result I developed yet another solution that targets a sweet spot between complexity and functionality.

The idea is to have a precise and simple language in which an explicit abstract grammar of the input language can be described. The abstract grammar is then used to populate arbitrary textual code and documentation templates. These are typically used to generate AST data structures, base classes for visitors, and documentation for these.

The accompanying tool is used in the development of the FreeBoogie frontend. Further details are given in the technical report [Gri07].

2.4 FreeBoogie

The best extended static checker is Boogie, the static verifier included in the Spec$^\#$ programming system. Unfortunately, it is developed by Microsoft Research (MSR) and hence it is closed source. An open implementation can serve as a platform for research experimentation. At least one group has already expressed interest in using FreeBoogie in such a way. A focused and intense development effort may result in a big service to the research community.

The input of FreeBoogie is a BPL program. The output is a list of warnings (that preferably refer to the original high-level source code). A first implementation of FreeBoogie will be considered successful when it can be plugged in the Spec$^\#$ system instead of the Boogie static verifier and the whole pipeline will still function.

(For the architecture of FreeBoogie we refer the reader back to Figure 1.)

To date, the parser, the typechecker, and the flow graph builder are implemented. The main component, which is still missing, is the VC generator. This component will construct prover queries by calling into the prover backend. At the moment, the prover backend implemented by Michał Moskal for the Mobius project is being refactored and integrated in FreeBoogie.

The theorem provers that will be supported include Simplify, Z3, and Fx7.

There is no work planned for invariant inference engines but hopefully Mikoláš Janota will be interested enough to contribute an implementation of his work on invariant generation [Jan07] or some evolution of it.

Since it is in its early stages, no comprehensive description of FreeBoogie exists.

3 Next steps

3.1 FreeBoogie

The task is to make FreeBoogie a drop-in replacement for MSR's Boogie and is important because most other tasks depend on it. There is little formal methods research needed, and most of the effort will go in engineering.

FreeBoogie must

- make flow graphs reducible,
- passivate the program,
- generate VCs,
- interface with various automated theorem provers,
- provide readable warning messages, and allow the frontend of the extended static checker to do the same.

Previous papers [JC97, FS01, BL05, LMS05] describe these phases, the challenge being to integrate all in a high quality implementation.

The interface with various automated theorem provers is implemented and the design borrows heavily from that of the package `escjava.sortedProver`, implemented by Michał Moskal. Although implemented, it is not well tested and supports only Simplify and Z3.

Reachability analysis builds prover queries in a way very similar to how VCs should be generated in FreeBoogie. In particular, the code that smartly unfolds the expression trees will be reused.

It is not at all clear what guarantees FreeBoogie will offer the frontend about the format of errors and location tracking, nor how those guarantees will be realized.

3.2 Reachability analysis

The task is to move this module from ESC/Java to FreeBoogie and, in the process, enhance it. That module was designed to work on unstructured flow graphs but was only tested on structured ones, because such is the nature of the ESC/Java intermediate representation. Therefore, the performance impact of the transplant is not completely clear, and might dictate further work. Since FreeBoogie supports the better theorem prover Z3, all performance measures need to be taken again anyway.

Here are some possible improvements of the analysis itself.

Unreachable code has an unsatisfiable precondition. At the moment only unreachable node that has at least one reachable parent is reported. The reason is that we want to report an error close to where the bug is and at the same time we want to avoid flooding the user with irrelevant messages. A common scenario, though, is that some piece of code is reachable but has an unsatisfiable postcondition, which makes all the nodes it dominates unreachable. It would be nice to report separately such situations, again with minimum help from the prover.

In general, it is desirable to heuristically guess the type of error (such as "caused by the unsound treatment of loops").

One serious lack of the work done so far is the superficial analysis of the algorithm. It is necessary to have a precise idea of how the runtime is influenced by things such as the number of nodes in the flow graph and the (maximum) branching factor of the flow graph. This can be done by studying analytically the algorithm for a given model of how reachability errors appear in annotated code and for specific types of flow graphs. The model needs to be validated with experimental data. Also, general flow graphs may be to hard to analyze analytically, in which case experimental analysis is necessary.

3.3 Edit and verify

The current implementation of edit-and-verify is in the Satisfiability Modulo Theories (SMT) solver Fx7 and acts on FOL formulas. The task is to implement it in FreeBoogie and act on BPL.

One of the contributions of [GM07] is the heuristic that finds differences between two huge trees representing FOL formulas. If we act on BPL flow graphs then we need a way to quickly find differences between flow graphs that are very similar and whose nodes contain FOL formulas. It is not clear how to achieve this task.

Silvio Ranise suggested that some techniques used by the existent edit-and-verify implementation might be useful to speed up bounded model checkers. Therefore I intend to see if this can be done, using Bogor [RDH06] as a framework.

The current implementation of edit-and-verify has the same problem as the one for reachability analysis: It is not clear what the performance is and a better algorithm analysis is needed. Such analysis might suggest even better solutions.

Michał Moskal recently pointed out an article by Babić and Hu [BH07] that seems closely related and needs to be well digested.

Another improvement of the edit-and-verify approach is to exploit not only the last known good state but a longer history. How to do so is not clear.

From an engineering point of view, the FreeBoogie implementation will be much more user friendly — it will require the user to do nothing to use these performance improvements.

3.4 Separation logic

Separation logic is good for reasoning about programs that manipulate heap [MAY06]. This task is to add heap commands and separating operators to BPL. The focus is on how to do static verification efficiently for the enhanced BPL and, at the same time, still support all the other static checks that work on BPL, such as reachability analysis.

Most previous work on using separation logic to do automated verification constrained either the logic [BCO04] or the programming language [Web04, BCO05b] for the sake of automation, or used interactive verification [Pre06, MAY06]. Here I take the view that, even if the problem is undecidable, an implementation that gives the correct answer automatically in 90% of the cases is desirable. Also, previous implementations (such as [BCO05a]) process small languages and are not integrated with any other static check.

This task is the most fuzzy to me at the moment, and will likely require the highest amount of research. The implementation work is divided in these steps:

- translate separating operators into FOL and translate heap commands into standard BPL commands
- carry over separating operators in the VC and process heap commands during VC generation
- implement a decision procedure for separating operators that integrates well in an SMT solver (Fx7)

Desugaring. In the initial phase heap commands and separating operators will be desugared into standard BPL. I will take care to have a desugaring that is compatible with the heap model used by the Spec$^\#$ frontend. This means that I should be able to produce BPL from a C$^\#$ program of a reasonable size using Spec$^\#$, add one procedure that uses heap commands to manipulate fields that existed in C$^\#$ and have the desugaring produce something consistent. This compatibility requirement is, of course, somewhat arbitrary but it should help testing a lot. Calcagno et al. [CGH05] show how to translate a fragment of separation logic into a decidable fragment of FOL. I want to translate everything in separation logic in full FOL, making sure that the formulas do not become too hard to prove *in practice*.

After this phase is completed it will be possible to verify BPL programs that contain heap commands and separating operators.

Enhance VC generation. The second phase involves handling heap commands during VC generation. It is not clear yet what this involves, but Christian Haack and Clement Huerlin reported progress on this point that they will make available before publication. Carrying over separating operators in the VC should not be difficult.

Decision procedure. An SMT solver consists of a SAT solver and a group of cooperating decision procedures. The decision procedures handle arithmetic, arrays, and so on. Equality is a special symbol that is understood by all decision

procedures and is the key to cooperation. The third phase involves developing a decision procedure that handles separating operators and integrate it in an existing SMT solver.

4 Dissemination

The results obtained so far appear in two published papers [JGM07, GM07] and in one technical report [Gri07].

The high quality conferences in the field are:

- Foundations of Software Engineering
- Formal methods
- Tools and Algorithms for Construction and Analysis of Systems
- European Symposium on Programming
- Static Analysis
- International Conference on Software Engineering

Other good venues include:

- Fundamental Approaches to Software Engineering
- Foundations of Software Science and Computation Structure
- Compiler Construction
- European Software Engineering Conference
- Automated Software Engineering
- Asia-Pacific Software Engineering Conference
- International Conference on Engineering of Complex Computer Systems
- TOOLS

5 Related work

Reference. The Java language and bytecode are defined in [GJSB00, LY99]. The BPL is described in [DL05]. The JML is described in [JML]. Design patterns [GHJV95] are tried solutions for organizing code in object-oriented languages. The "Dragon Book" [ASU86] (still) is the Bible of compiler developers. CLRS [CLRS01] beautifully presents a wide range of algorithms. Knuth [Knu97a, Knu97b, Knu97c] provides in-depth analysis of frequently used algorithms (that are most of the time buried in standard libraries nowadays)

and a good repertoire of algorithm analysis techniques. The "Coq Book" [BC04] is a good guide to interactive and mechanical theorem proving. Fitting [Fit96] presents the theory behind tableaux-based and resolution-based first-order automated theorem provers. A comprehensive book on model checking is [CGP99].

Extended static checkers and similar tools. The theoretical underpinnings where explored by Floyd [Flo67], Hoare [Hoa69], and Dijkstra [Dij76]. Old implementations include AFFIRM, the Stanford Pascal Verifier, the Ford Aerospace Pascal verification system, Penelope [GMP90], and ESC/Modula [Det96]. The implementations that are still maintained or that are in active development are ESC/Java [FLL+02, CK05], the Key tool [ABB+05], Jack [BRL03], Krakatoa [MPMU04], Caduceus [FM07], and Spec# [BLS04]. Krakatoa and Caduceus share the backend Why. The input language for Why, like BPL, is an intermediate language for extended static checkers.

The backend of extended static checkers. FreeBoogie must (1) make flow graphs reducible, (2) passivate the program, (3) generate a VC, and (4) interface with the prover. I will implement the method presented in [JC97]; it is unlikely that this part will influence considerably the overall performance. Flanagan and Saxe [FS01] explain why it is a good idea to passivate the program before generating VCs and how to carry on both steps. The approach is refined for unstructured code by Barnett and Leino [BL05]. The interface with multiple provers is based on the `escjava.sortedProver` package. The approach taken by the Why tool is described in [Fil03]. A technical but important issue is to produce high-quality error messages, and ESC/Java has a good solution [LMS05].

The frontend of extended static checkers. The frontend is out of the scope of this PhD. Still, a lot of interesting work is happening in this closely related area [BDF+04, DM05, JSPS06, LM05].

Separation logic. Making an extended static checker use separation logic involves changes in almost all the pipeline — it is a cross-cutting concern, like tracking location information for building good warning messages. O'Hearn [ORY01] and Reynolds [Rey02] present separation logic per se. The translation from BPL with separating operators to (standard) BPL may be done along the lines of [CGH05]. The papers [BCO05b, BCO05a, Web04, Pre06] show progress in automatic verification and will be useful in designing the VC generator in FreeBoogie. Finally, there is some preliminary work [BCO04] that will be quite useful in implementing a decision procedure for separation logic in an SMT solver.

SMT solvers. The best theorem provers used by extended static checkers are actually SMT solvers, which put together, using equality as a common point [NO79], decision procedures (that know about theories such as arithmetic with integers), and a SAT solver with instantiation techniques for universal quantifiers [DP60, DLL62, GHN+04].

References

[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter Schmitt. The KeY tool. *Software and Systems Modeling*, 4(1):32–54, 2005.

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: principles, techniques, and tools*. Addison–Wesley, 1986.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art*. Springer, 2004.

[BCO04] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS*, volume 3328 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 2004.

[BCO05a] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[BCO05b] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer–Verlag, 2005.

[BDF+04] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[BH07] Domagoj Babić and Alan J. Hu. Exploiting shared structure in software verification conditions. In *Haifa Verification Conference, HVC 2007, October 23–25, 2007, Proceedings*, Lecture Notes in Computer Science. Springer, 2007. To appear.

[BL05] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. *Workshop on Program Analysis for Software Tools and Engineering*, pages 82–87, 2005.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. *Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 3362, 2004.

[BRL03] Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: a developer-oriented approach. *FME*, pages 422–439, 2003.

[CAHM04] Kevin Crowston, Hala Annabi, James Howison, and Chengetai Masango. Towards a portfolio of FLOSS project success measures. In *ICSE Open Source Workshop*, 2004.

[CGH05] Cristiano Calcagno, Philippa Gardner, and Matthew Hague. From separation logic to first-order logic. In Vladimiro Sassone, editor, *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 395–409. Springer, 2005.

[CGP99] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999. To buy.

[CK05] David Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. *Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 3362:108–128, 2005.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition edition, 2001.

[Det96] David L. Detlefs. An overview of the extended static checking system. In *Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, January 1996. To read.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. 1976.

[DL05] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical report, Microsoft Research, 2005.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DM05] Ádám Darvas and Peter Müller. Reasoning about method calls in jml specifications. *Formal Techniques for Java-like Programs*, 2005.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the Association of Computing Machinery*, 7(3):201–215, 1960.

[Fil03] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, 2003.

[Fit96] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Springer–Verlag, 1996. To buy.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James Saxe, and Raymie Stata. Extended static checking for Java. *ACM SIGPLAN Notices*, 37(5):234–245, 2002.

[Flo67] Robert W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19(19–32):1, 1967. Not available online.

[FM07] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

[FPB95] Jr. Frederick P. Brooks. *The mythical man-month*. Addison-Wesley, 1995.

[FS01] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. *Symposium on Principles of Programming Languages*, pages 193–205, 2001.

[gcc] GNU C compiler internals. `http://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/GCC_4.1`.

[GH] Etienne M. Gagnon and Laurie J. Hendrie. The SableCC homepage.
`http://sablecc.org/`.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. AW, 1995.

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, 2000.

[GM07] Radu Grigore and Michał Moskal. Edit and verify. In Silvio Ranise, editor, *Proceedings of the 6th International Workshop on First-Order Theorem Proving*, pages 101–113. University of Liverpool, September 2007.

[GMP90] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, 1990.

[Gri07] Radu Grigore. Generating code and documentation from lightweight abstract grammars. Technical report, University College Dublin, 2007.

[HM03] Görel Hedin and Eva Magnusson. JastAdd—an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[Hoa69] C. Anthony R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Jan07] Mikoláš Janota. Assertion-based loop invariant generation. In *1st International Workshop on Invariant Generation (WING)*, 2007.

[JC97] Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. *ACM Transaction on Programming Languages and Systems*, 19(6):1031–1052, 1997.

[JGM07] Mikoláš Janota, Radu Grigore, and Michał Moskal. Reachability analysis for annotated code. In *Specification and Verification of Component-Based Systems*, June 2007.

[JML] The JML reference manual. `http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman.html`.

[JSPS06] Bart Jacobs, Jan Smans, Frank Pissens, and Wolfram Schulte. A statically verifiable programming model for concurrent object-oriented programs. *International Conference on Formal Engineering Methods*, 2006.

[Knu97a] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison–Wesley, third edition edition, 1997.

[Knu97b] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison–Wesley, third edition edition, 1997.

[Knu97c] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 2. Addison–Wesley, third edition edition, 1997.

[LBR06] Gary Leavens, Albert Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[LM05] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. *Formal Methods (FM)*, pages 26–42, 2005.

[LMS05] K. Rustan M. Leino, Todd Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55(1–3):209–226, 2005.

[LY99] Tim Lindholm and Frank Yellin. *The Java virtual machine specification*. Sun Microsystems, 1999.

[MAY06] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In Zhiming Liu and Jifeng He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 400–419. Springer, 2006.

[MPMU04] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/Javacard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.

[MRS07] Raimund Moser, Barbara Russo, and Giancarlo Succi. Empirical analysis on the correlation between GCC compiler warnings and revision numbers of source files in five industrial software projects. *Empirical Software Engineering*, 12(3):295–310, 2007.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transaction on Programming Languages and Systems*, 1(2):245–257, October 1979. to read carefully.

[ORY01] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. *Computer Science Logic*, 2142:1–19, 2001.

[Par] Terence Parr. The ANTLR system homepage. `http://www.antlr.org/`.

[Pre06] Viorel Preoteasa. Mechanical verification of recursive procedures manipulating pointers using separation logic. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 508–523. Springer, 2006.

[RDH06] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: A flexible framework for creating software model checkers. In Phil McMinn, editor, *TAIC PART*, pages 3–22. IEEE Computer Society, 2006.

[Rey02] John Reynolds. Separation logic: a logic for shared mutable data structures. *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74, 2002.

[Sun] Sun Microsystems. The JavaCC homepage. `https://javacc.dev.java.net/`.

[Web04] Tjark Weber. Towards mechanized program verification with separation logic. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *CSL*, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2004. To read carefully.