Reachability Modulo Theories

Akash Lal and Shaz Qadeer

Microsoft Research
{akashl,qadeer}@microsoft.com

Abstract. Program verifiers that attempt to verify programs automatically pose the verification problem as the decision problem: *Does there exist a proof that establishes the absence of errors*? In this paper, we argue that program verification should instead be posed as the following decision problem: *Does there exist an execution that establishes the presence of an error*? We formalize the latter problem as Reachability Modulo Theories (RMT) using an imperative programming language parameterized by a multi-sorted first-order signature. We present complexity results, algorithms, and the Corral solver for the RMT problem. We present our experience using Corral on problems from a variety of application domains.

1 Introduction

Practical program verifiers are difficult to design and implement. To be useful, the verification must be automated as much as possible. At the same time, the verifier must be able to model precisely the complex features and abstractions used in real-world programming languages. First-order provers based on satisfiability-modulo-theories (SMT) satisfy these conflicting requirements simultaneously by providing both a rich modeling framework for encoding language semantics and high-degree of automation for deciding satisfiability of expressions. Consequently, many program verifiers use SMT solvers in their core.

Efficiently decidable satisfiability checking of expressions is necessary but insufficient for building practical program verifiers. The reason is that while verification of finite executions can be encoded precisely as satisfiability checking, verification of unbounded executions cannot be similarly encoded. The latter problem is undecidable for the standard theories, e.g., linear arithmetic, uninterpreted functions, arrays, etc. used in modeling program behaviors.

Program verifiers that attempt to verify programs automatically pose the verification problem as the decision problem: *Does there exist a proof that establishes the absence of errors*? As mentioned above, this problem is undecidable; in fact, it is not even recursively enumerable. Intuitively, enumerating program proofs is so difficult because it requires a complete proof system whose assertions come from a statically known and decidable language. Practical verifiers use the Floyd-Hoare proof system and the language of expressions in the programming langauge as the assertion language. Our experience building verifiers for sequential programs indicates that while the Floyd-Hoare proof system is often complete enough, expressions in the programming language are inadequate

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 23–44, 2013.

for capturing proofs of even simple properties. As an example, the theory of arrays which naturally encodes program expressions for looking up and updating the heap is inadequate for expressing a Floyd-Hoare proof of correctness of a heap-manipulating program. Such a proof would typically require a richer theory capable of expressing both quantified facts as well as data abstractions such as objects, lists, and trees. We have encountered this difficulty in practice while deploying the HAVOC verifier [22] to verify simple type-state properties on device drivers [23].

In this paper, we argue that program verification should instead be posed as the following decision problem: *Does there exist an execution that establishes the presence of an error*? This problem is undecidable but recursively enumerable for a programming language with decidable expression language. A prototypical semi-decision procedure for this problem would enumerate program executions in a fair manner to provide complete search in the limit. There are several advantages of this problem formulation. First, it directly matches the most important and common uses of automatic program verification—bug-finding and debugging. When program verification is posed as a proof discovery problem, a counterexample is a by-product of the failure of proof discovery. A direct search for counterexamples could potentially be more efficient at uncovering bugs. Second, it naturally allows the formulation of bounded and decidable versions of the problem. It is possible that as we develop better techniques for solving the bounded problem, we will get incrementally better at solving the harder unbounded problem. As anecdotal evidence from the literature on hardware verification, success in solving the bounded problem (NP-complete) via *Boolean* satisfiability solvers has led to increasing success in solving the harder (PSPACE-complete) unbounded problem. Finally, we note that stating the problem as a search for counterexamples does not preclude the use of proof techniques for pruning search; proofs simply become an opportunity for optimization rather than a goal by themselves.

We present *reachability modulo theories* (RMT), a parameterized framework for modeling program executions and stating verification problems. RMT emphasizes reachability of an error state as opposed to unreachability of all error states. A RMT problem is specified by picking points along two orthogonal axes defining a programming language—control and data. Control is specified using a control-flow graph with an appropriately restricted set of features. For example, in this paper we specifically address sequential and potentially-recursive control flow but it is just as easy to restrict recursion or generalize to allow concurrency by allowing asynchronous and parallel calls. Data is specified by using a multi-sorted first-order signature, much like in the definition of satisfiability modulo theories. Given such a signature, we allow each program variable to be associated with a sort and assignments from well-sorted expression to a variable of a matching sort. In other words, RMT exposes the full power of first-order modeling provided by the satisfiability modulo theories framework and strengthens it with a control flow graph, allowing us to define bounded and unbounded operational semantics over rich data domains.

In addition to presenting the basic definition of RMT in Section 2, this paper also contains the following contributions:

- To improve our understanding of the RMT problem, we studied its complexity for loop-free and recursion-free programs. We call the problem for such programs (with acyclic call graphs) the hierarchical RMT problem. In Section 3, we present complexity results for various expression languages that are relevant for modeling practical problems. Restricting attention to quantifier-free expressions, we show that if the expression language is decidable in NP, then hierarchical RMT is decidable in NEXPTIME; if the only sort is *Boolean*, then hierarchical RMT is PSPACE-complete; if, in addition, uninterpreted functions are available, then hierarchical RMT is NEXPTIME-complete.
- We use the Boogie [9] language as the concrete representation for an RMT problem. We have developed translators from C and .NET bytecode into Boogie. In Section 4, we give examples of how the operational semantics and verification problems for different source languages are encoded into Boogie. Our work enables both sequential and concurrent Boogie programs, regardless of the source language from which they are derived, to be verified without requiring contracts such as preconditions, postconditions, and loop invariants.
- We have implemented Corral [25], a solver for the RMT problem. Corral will replace Slam [6] as the solver inside Static Driver Verifier in the next release of the Microsoft Windows operating system. In Sections 5 and 6, we describe the core techniques and architecture of Corral and our experience applying it to solve reachability problems on device drivers. We also describe other applications, such as analyzing sequentializations of concurrent programs, detecting security vulnerabilities in web applications, and solving debugging queries for .NET bytecode.

Related Work. Software model checkers [7, 21, 29] based on predicate abstraction [19] are the best known examples of program verification as proof search, as opposed to our work which is founded in counterexample search. In addition to this foundational difference, another difference is the programming language on which the problem is typically stated. Software model checking has traditionally been defined for the C programming language, whose expression language makes implicit reference to the heap and and is consequently not directly amenable to logical reasoning. On the other hand, we define our programming language abstractly using a multi-sorted first-order signature; consequently, the well-understood techniques of weakest preconditions and verification-condition generation are immediately available to us.

Recently, another attempt to state the proof search problem using first-order signatures has been made using a formulation based on Horn clauses [12, 13]. The main difference from our work is the focus on proof discovery as opposed to counterexample discovery; in this regard, their approach is similar to software model checking. However, similar to our formulation, their approach is independent of the syntax and semantics of source-level programming languages. While

```
// Identifiers
Id

// Sorts
Sort

// Expressions
Expr

// Variable declarations
VarDecl ::=  var Id: Sort

// Commands
Cmd ::=  assume Expr  |  Id := Expr  |  call Id* := Id(Id*)  |  havoc Id

// Basic Blocks
Block ::=  Id: Cmd goto Id*  |  Id: return

// Procedures
Proc ::=  procedure Id (VarDecl*) returns VarDecl* { VarDecl* Block+ }

// Program
Program ::=  Proc*
```

Fig. 1. The grammar of programs

RMT uses control flow and variables whose values can be updated, the Horn clause formulation uses side-effect free logical expressions. The Horn clause formulation has the advantage that the representation can encode not just program semantics but various proof systems such as Floyd-Hoare for sequential programs and Owicki-Gries for concurrent programs. Our formulation has the advantage that we can reason directly about sequential or concurrent program executions.

2 Reachability Modulo Theories

We define the RMT problem over a simple imperative programming language. The syntax of the language is shown in Fig. 1. A program (Program) is a list of procedure declarations. A procedure (Proc) can have any number of input and output parameters. The procedure body is a list of local variable declarations followed by a list of basic blocks; the first block is the one where execution of the procedure starts. The output variables of a procedure (if any) act like any other local variable, except that their value at a **return** command is the tuple of values returned on a call to the procedure. A basic block (Block) is a label followed by a list of commands. A command (Cmd) is either an assume command, or an assignment, or a **havoc** command, or a procedure call. The command **havoc** `x` non-deterministically assigns an arbitrary value (of the right type) to

x. The rest of the commands have the standard meaning. We disallow loops in our programs (they can be encoded using tail-recursion). Thus, the control-flow graph of a procedure is always acyclic. We refer to a recursion-free program as a *hierarchical* program.

We leave the syntax of expressions and sorts unspecified in the syntax and require only the following two properties:

- Expr is generated from a multi-sorted first-order signature containing the *Boolean* sort and the equality relation $=$.
- It is decidable to check satisfiability of *Boolean*-valued expressions in Expr.

The smallest expression language that satisfies the above properties is quantifier-free and contains only *Boolean* sort. For this expression language, the satisfiability problem is NP-complete.

$$\frac{P \vdash (l : \textbf{assume } e;\ \textbf{goto } ls') \qquad l' \in ls' \qquad e(M,\sigma) = \mathit{true}}{P, M \vdash (\sigma, l) \cdot ss \rightarrow_b (\sigma, l') \cdot ss}$$

$$\frac{P \vdash (l : x := e;\ \textbf{goto } ls') \qquad l' \in ls' \qquad e(M,\sigma) = v}{P, M \vdash (\sigma, l) \cdot ss \rightarrow_b (\sigma[x := v], l') \cdot ss}$$

$$\frac{P \vdash (l : \textbf{havoc } x;\ \textbf{goto } ls') \qquad l' \in ls' \qquad v \in M(\mathit{Sort}(x))}{P, M \vdash (\sigma, l) \cdot ss \rightarrow_b (\sigma[x := v], l') \cdot ss}$$

$$\frac{\begin{array}{c} P \vdash (l : \textbf{call } \boldsymbol{y} := p(\boldsymbol{x});\ \textbf{goto } ls') \\ \boldsymbol{a} = \mathit{Ins}(p) \qquad \boldsymbol{b} = \mathit{Outs}(p) \qquad \boldsymbol{c} = \mathit{Locals}(p) \qquad \forall i.\ \sigma'(\boldsymbol{a}_i) = \sigma(\boldsymbol{x}_i) \\ \forall j.\ \sigma'(\boldsymbol{b}_j) = M(\mathit{Sort}(\boldsymbol{b}_j)) \qquad \forall k.\ \sigma'(\boldsymbol{c}_k) = M(\mathit{Sort}(\boldsymbol{c}_k)) \qquad \mathit{Count}((\sigma, l) \cdot ss, p) < b \end{array}}{P, M \vdash (\sigma, l) \cdot ss \rightarrow_b (\sigma', \mathit{Start}(p)) \cdot (\sigma, l) \cdot ss}$$

$$\frac{P \vdash (l : \textbf{return}) \qquad P \vdash (l' : \textbf{call } \boldsymbol{y} := p(\boldsymbol{x});\ \textbf{goto } ls') \qquad l'' \in ls' \qquad \boldsymbol{b} = \mathit{Outs}(p)}{P, M \vdash (\sigma, l) \cdot (\sigma', l') \cdot ss \rightarrow_b (\sigma'[\boldsymbol{y} := \sigma(\boldsymbol{b})], l'') \cdot ss}$$

Fig. 2. Operational semantics

Figure 2 presents the operational semantics of our programming language. The semantics is given using the derivation $P, M \vdash ss \rightarrow_b ss'$ that refers to the following elements:

- P is a program.
- M is a model for the first-order signature of program P.
- Each of ss and ss' is a stack of activation records, essentially a list of pairs, with each pair comprising a label and a valuation to program variables.
- $\rightarrow_b$ is the transition relation for an integer bound $b > 0$.

The semantics also use an auxiliary derivation of the form $P \vdash l : c; \textbf{goto } ls$ or $P \vdash l : \textbf{return}$. This derivation indicates that the program P contains an appropriately labeled basic block. We assume that all labels in the program are distinct.

The first rule in Figure 2 is for the assume statement; it allows execution to proceed only if the expression e evaluates to true. The next rule is for the assignment statement. The rule for havoc updates the variable x to an arbitrary value belonging to the interpretation of $Sort(x)$ in the model M. Next are the rules for procedure call and return. The activation record of the called procedure gets arbitrary initial values for the output and local variables. Upon return, actual output parameters in the caller are updated by looking up the appropriate variables in the callee. For any stack ss and procedure p, $Count(ss, p)$ returns the total number of activation records of procedure p on the stack. The call rule ensures that the number of activation records for any procedure does not go beyond b. We further define

$$\rightarrow = \bigcup_{b>0} \rightarrow_b$$

as the full transition relation of the program.

Let p be a procedure with no input or output parameters. We say that p has a *terminating execution* if and only if there is a model M, label l, and variable evaluations σ, σ' such that $P \vdash (l : \textbf{return})$ and $P, M \vdash (Start(p), \sigma) \rightarrow^* (l, \sigma')$. We say that p has a *b-terminating execution* for some $b > 0$ if and only if there is a model M, label l, and variable evaluations σ, σ' such that $P \vdash (l : \textbf{return})$ and $P, M \vdash (Start(p), \sigma) \rightarrow_b^* (l, \sigma')$. Using these definitions, we can define the following two decision problems.

Reachability Modulo Theories. Given a program P and a procedure p in the program with no input or output parameters, return YES if p has a terminating execution and return NO otherwise. We use $\text{RMT}(P, p)$ to denote an instance of this problem. If P is hierarchical, the problem is referred to as the hierarchical reachability modulo theories problem.

Bounded Reachability Modulo Theories. Given a program P, a procedure p in the program with no input or output parameters, and a bound $b > 0$, return YES if p has a b-terminating execution and return NO otherwise. We use $\text{RMT}(P, p, b)$ to denote an instance of this problem.

Fig. 3 shows a simple program P for which $\text{RMT}(P, main)$ and $\text{RMT}(P, main, 100)$ holds, but $\text{RMT}(P, main, b)$ does not hold for $b < 100$.

3 Complexity of the RMT Problem

The RMT problem is undecidable in general. The presence of recursion along with say, linear arithemetic in expressions, is enough to encode Turing-powerful computations. However, the bounded and hierarchical RMT problems are decidable. This section first gives an algorithm for deciding bounded RMT and then

```
procedure main() {
   call bar(0);
}

procedure bar(i: int) {
  if (i < 100) {
    i := i + 1;
    call bar(i);
  }
}
```

Fig. 3. An example program over *Boolean* and *Integer* sorts. Structured command `if` is used for convenience and can be easily compiled to labeled blocks.

refines the complexity analysis depending on the choice of the expression language. We first consider the special case of call-free single-procedure programs. Deciding RMT in this case can be reduced to the satisfiability of a single expression (which is decidable) through a process called Verification Condition (VC) generation.

3.1 Verification-Condition Generation

Let Expr be the expression language and P be a program consisting of a single call-free procedure f. The VC generation algorithm converts the body of f to a *Boolean* expression $VC(f)(\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{t})$ such that $\boldsymbol{i}$ is the list of all input parameters to the procedure, $\boldsymbol{o}$ is the list of all output parameters, and $\boldsymbol{t}$ are some temporary variables. We call the tuple $(\boldsymbol{i}, \boldsymbol{o})$ the interface variables of f. The expression $VC(f)$ satisfies two important properties. First, $VC(f)$ is an expression in Expr. Second, $VC(f)$ is satisfiable if and only if some execution of f starting in state $\boldsymbol{i}$ can return with state $\boldsymbol{o}$. Therefore, $\mathrm{RMT}(P, f)$ can be decided by checking the satisfiability of $VC(f)$.

We now describe a standard VC generation algorithm [10]. The first step is to *passify* f by converting all commands to **assume** statements. Consider the single procedure shown on the left of Fig. 4. Its passified version is shown on the right of the figure. Passification can be done in two steps: first, do *single-static assignment* (SSA renaming) [28] by introducing fresh variable incarnations so that each variable has at most one assignment. Next, push the ϕ functions to their definitions. For instance, the SSA renaming of Fig. 4 would create the statement `x4 := ϕ(x2,x3)` at label `l3` where control-flow merges. This statement can be realized by instead having the statements **assume** `x4 = x2` and **assume** `x4 = x3` right after the definitions of `x2` and `x3`, respectively. Next, as is standard, an assignment `x := e` after SSA renaming can be replaced by **assume** `x = e`, and **havoc** statements can be dropped (assuming that uninitialized variables are unconstrained). Finally, we add an **assume** statement at each **return** stating that the output parameters are constrained to the appropriate incarnation. This results in the passified version shown in Fig. 4 (right). Note that passification requires Expr to be closed under $=$.

Once the body of procedure f is passified, it can be converted to $VC(f)$ as follows. Let l be an arbitrary block label in f. We define $C(l)$ to be the

```
procedure f(w: int)
   returns (x: int, y: int, z: int)
{
start:
   havoc x;
   y := x + w;
   goto l1, l2;

l1:
   x := x + 1;
   goto l3;

l2:
   x := x + 2;
   goto l3;

l3:
   assume !(x > y);
   return;
}
```

```
procedure f(w: int)
   returns (x: int, y: int, z: int)
{
   var x0, x1, x2, x3, x4: int;
   var y0, y1: int;
   var z0: int;

start:
   assume y1 = x1 + w;
   goto l1, l2;

l1:
   assume x2 = x1 + 1;
   assume x4 = x2;
   goto l3;

l2:
   assume x3 = x1 + 2;
   assume x4 = x3;
   goto l3;

l3:
   assume !(x4 > y1);
   assume (x = x4 && y = y1 && z = z0);
   return;
}
```

Fig. 4. A program and its passified version

conjunction of the expressions in the assume statements in the block. We also define $\texttt{Succ}(l)$ to be the set of successor labels of block l if l ends with a **goto** and the empty set otherwise. We create a set of fresh *Boolean* variables $\{\ldots, b_l, \ldots\}$, one for each block l in f. We define the equation $E(l)$ of a block l as $b_l = C(l)$ if the block ends with a **return** and $b_l = c(l) \wedge \bigvee_{n \in \texttt{Succ}(l)} b_n$ otherwise. Then we get the following expression for $VC(f)$:

$$VC(f) = b_e \wedge \bigwedge_l E(l)$$

The expression $VC(f)$ refers to input and output variables of f; the temporary variables are the incarnation variables created for the SSA renaming. As an example, for the program in Figure 4, the block equations are as follows:

$$\begin{aligned} b_{start} &\equiv y1 = x1 + w \wedge (b_{l1} \vee b_{l2}) \\ b_{l1} &\equiv x2 = x1 + 1 \wedge x4 = x2 \wedge b_{l3} \\ b_{l2} &\equiv x3 = x1 + 2 \wedge x4 = x3 \wedge b_{l3} \\ b_{l3} &\equiv \neg(x4 > y1) \wedge x = x4 \wedge y = y1 \wedge z = z0 \end{aligned}$$

We capture the correctness of VC generation in the following lemma.

Lemma 1. *Let P be a call-free program and p be a procedure in P. Then the answer to* $\mathrm{RMT}(P, p)$ *is Yes iff $VC(p)$ is satisfiable.*

The above VC generation algorithm shows that RMT for call-free programs is decidable. This result can be extended to the hierarchical RMT problem for

arbitrary (non-recursive) programs: one can simply inline all procedures to obtain a single call-free procedure (because there is no recursion) and then decide reachability using VC generation. We call this algorithm the *static inlining* algorithm. This algorithm is easily extended to to the bounded RMT problem as well. Given a program P with a procedure p and bound b, it is possible to create a recursion-free program P' with a procedure p' such that $\mathrm{RMT}(P, p, b)$ is equivalent to $\mathrm{RMT}(P', p')$. Therefore, bounded RMT is decidable as well. The decision procedure for bounded RMT also implies that RMT is recursively enumerable: one can start with $b = 1$ and increment b until a witness for RMT is found.

Theorem 1. *Let P be a program and p be a procedure in P. Then* $\mathrm{RMT}(P, p, b)$ *is decidable and* $\mathrm{RMT}(P, p)$ *is recursively enumerable.*

It is worth contrasting the $\mathrm{RMT}(P, p)$ problem with a different $UMT(P, p)$ problem which asks the question where there is no terminating execution of procedure p in program P. Let us consider Expr such that $\mathrm{RMT}(P, p)$ is undecidable (easy as soon as either arithmetic or uninterpreted sorts are introduced). If $UMT(P, p)$ is recursively enumerable, then we can use Theorem 1 to conclude that $\mathrm{RMT}(P, p)$ is decidable which would be a contradiction. Therefore, $UMT(P, p)$ is neither decidable nor recursively enumerable. The $UMT(P, p)$ problem captures the problem definition being solved by software model checkers whose goal is to discover program proofs automatically. Intuitively, it appears that searching for a proof is more difficult than searching for a feasible path and an RMT solver is solving an "easier" problem than the one being solved by a software model checker.

3.2 Complexity of Hierarchical RMT

In this section, we demonstrate certain complexity results for the hierarchical RMT problem. As discussed earlier, the hierarchical RMT problem is decidable. We can refine the complexity analysis further by restricting the sorts in Expr.

Theorem 2. *If checking satisfiability of Boolean expressions in* Expr *is decidable in* NP*, then hierarchical RMT is decidable in* NEXPTIME*.*

Proof. Let D be a non-deterministic machine that does satisfiability of expressions in polynomial time. Let P be a non-recursive program. Then the length of any execution σ of P will be at most exponential in the size of P. Construct a non-deterministic machine M that guesses an execution σ_w of P, then rewrites it as straightline program P_w. (The size of P_w is at most exponential in the size of P.) Next, M does VC generation on P_w to obtain a single expression e_w and feeds it to D. M says that $\mathrm{RMT}(P)$ holds if and only if D says "satisfiable". It is easy to see that M solves $\mathrm{RMT}(P)$ in time at most exponential in the size of P.

The upper bound of NEXPTIME can be tightened if we restrict programs to use only the *Boolean* sort. When we only allow the *Boolean* sort, then we end up with the class of programs commonly known as *Boolean programs* [8, 18] that have been extensively studied in the literature.

Theorem 3. *If* Expr *is quantifier-free and contains only Boolean sort, hierachical RMT is* PSPACE*-complete.*

Proof. It is known that reachability for recursion-free Boolean programs is PSPACE-complete [1]; this result is enough to show membership in PSPACE. We only need to show that the problem is PSPACE-hard. We do that by reduction from the PSPACE-complete problem of checking whether there is a path from an initial state to a bad state in a transition system over a vector $\boldsymbol{x}$ of n *Boolean* variables. Let *Init* be the predicate representing the set of initial states, *Good* the predicate representing the set of good states, and *Trans* the transition relation. We construct a program P with procedure p

```
procedure p() {
  var y;
  assume Init(y);
  call y := p_0(y);
}
```

procedure p_i for $i \in [0, n)$

```
procedure p_i(x) returns (y) {
  y := x;
  call y := p_{i+1}(y);
  ...
  call y := p_{i+1}(y);
}
```

and procedure p_n

```
procedure p_n(x) returns (y) {
  assert Good(x);
  assume Trans(x, y);
}
```

The desired problem is $RMT(P, p)$. A transition system over n *Boolean* variables can have non-repeating paths of length at most 2^n; thus, the executions of p encode all non-repeating paths of the input program. The procedure p_n uses the assert statement; we show in Section 6 that assert statements can be compiled away with at most a linear cost.

Theorem 4. *If* Expr *is quantifier-free and contains only uninterpreted sorts (in addition to Boolean sort), hierarchical RMT is* NEXPTIME*-complete.*

Proof. Checking satisfiability of quantifier-free first-order logic is decidable in NP. Therefore, Theorem 2 gives membership in NEXPTIME. To show hardness, we demonstrate a polynomial-time reduction from the satisfiability problem for the EPR fragment of first-order logic. This fragments is given as $\exists \boldsymbol{x}.\ \forall \boldsymbol{y}.\ \varphi(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x} = x_1, \ldots, x_m$ and $\boldsymbol{y} = y_1, \ldots, y_n$ and φ refers only to uninterpreted relation symbols. The problem of checking satisfiability of EPR formulas is known to be NEXPTIME-complete [27]. The decision procedure is straightforward. Skolemize $\boldsymbol{x}$ and then create a ground formula by taking the conjunction of φ

for all possible instantiations of $\boldsymbol{y}$ using only the skolem constants. The resulting ground formula is exponentially larger and equisatisfiable to the original. We show how to encode this decision procedure using a polynomial-size hiearchical RMT problem over a single uninterpreted sort S.

Let P be a hierachical program constructed as follows. First, declare m constants named $x_1, \ldots, x_m$ each of sort S. Next, declare procedures $p_0, p_1, \ldots, p_n$ such that procedure p_i has i input parameters each of sort S and no output parameters. Finally, define procedures p_i for $i \in [0, n)$

```
procedure p_i(v) {
  call p_{i+1}(v, x_1);
  ...
  call p_{i+1}(v, x_m);
}
```

and procedure p_n

```
procedure p_n(v) {
  assume φ(x, v);
}
```

The desired RMT problem is $\mathrm{RMT}(P, p_0)$.

4 Encoding Verification Problems

The Boogie language [9] is a concrete instance of an RMT language. In particular, the expression language of Boogie consists of the usual SMT theories (uninterpreted functions, theory of arrays, etc.) whose quantifier-free subset can be decided in NP using SMT solvers. The presence of such an expressive language offers a convenient way to encode many software verification tasks.

In this section, we briefly survey past effort on compiling programs in languages such as C and C# down to Boogie. Instead of rigorously describing the compilers, we illustrate using examples how source-level features are modeled in Boogie using decidable theories. More details can be obtained from the original papers on HAVOC, for C to Boogie [22] and the ByteCode Translator (BCT), for C# to Boogie [11].

Fig. 5 shows the encoding of a simple C program in Boogie, focusing on the treatment given to pointers and the heap in C. The HAVOC tool treats a pointer as simply an `int`. Memory allocation happens through a special variable called `alloc` that monotonically increases, as captured by the procedure `malloc`. It is easy to verify that successive calls to `malloc` will return distinct pointers.

The heap is modeled using arrays. Conceptually, the entire heap can be encoded using a single map `Mem` of type `int` $\rightarrow$ `int`, and each dereference `*x` can be translated to `Mem[x]`. However, for efficiency reasons, HAVOC splits the `Mem` map to multiple maps, one for each type and field, assuming certain type safety conditions on the program [14]. In Fig. 5, the use of two maps, one for field `f` and the other for `g` statically encodes the non-aliasing constraint that `x->f` cannot alias `y->g`, irrespective of the values of `x` and `y`. Such a constraint enables local reasoning.

HAVOC supports the option of encoding pointers using bitvectors as well. In that case, arithmetic operations are compiled to bitvector operations that

```
struct S {
  int f;
  int g;
};

void main() {
   S *x = malloc(sizeof(S));
   S *y = malloc(sizeof(S));
   x->f = 1;
   y->g = 2;
   assert(x->f == 1);
}
```

```
var Mem.f_S: [int]int;
var Mem.g_S: [int]int;

var alloc: int;

procedure malloc(size: int)
       returns (ptr: int) {
  var old_alloc: int;

   assume size >= 0;
   old_alloc := alloc;
   havoc alloc;
   assume alloc >
    old_alloc + size;
   ptr := alloc;
}

procedure main() {
   var x: int;

   assume alloc > 0;

   call x := malloc(8);
   call y := malloc(8);

   Mem.f_S[x] := 1;
   Mem.g_S[y] := 2;

   assert Mem.f_S[x] == 1;
}
```

Fig. 5. A C program (left) and the corresponding compiled Boogie program (right)

still fall under the QFBV theory (quantifier-free bit-vectors) supported by SMT sovlers.

The SMACK compiler [32] is another tool for compiling C programs to Boogie. It has the option of using a memory model where pointers are not `int`s but rather a type that encapsulates the actual pointer value and meta-data such as the block of memory the pointer belongs to and the size of that block. Such a memory model allows asserting of memory safety (i.e., every pointer dereference is inside allocated memory).

The compilation of a C# program to Boogie, using BCT, also encodes the heap using a series of maps, one for each field declared in the program. However, C# being a higher-level language than C, BCT has to model several other features of C#. For instance, the sub-typing relation can be encoded using a series of axioms (written in Boogie syntax):

```
// a type for C# types
type Type;

// an uninterpreted function
function SubType(Type,Type): bool;

// whenever A inherits from B
axiom SubType(A, B);
```

```
// reflexive, transitive, anti-symmetric
axiom forall t: Type :: SubType(t, t);
axiom forall t1, t2, t3: Type :: SubType(t1, t2) && SubType(t2, t3
    ) ==> SubType(t1, t3);
axiom forall t1, t2: Type :: t1 != t2 && SubType(t1, t2) ==> !
    SubType(t2, t1)
```

Axioms are structural constraints that are assumed to hold in any valid program state. Such an encoding of subtyping, even though it uses quantifiers, falls under the *effectively propositional* class of formulas [30] that turns out to be decidable.

The use of SMT theories allows Boogie to capture many verification tasks, but there are still certain important aspects of software that are hard to model. For example, there is no easy way to encode floating-point computation or string operations in a decidable theory. One can use quantifiers or recursive procedures to model string operations, but it remains to be seen if this leads to an effective end-to-end solution for string-manipulating programs.

5 Stratified Inlining

Section 4 shows that even the most common software verification tasks requires the use of linear arithmetic, uninterpreted functions and maps. For these problems, the bounded RMT problem is NEXPTIME-hard (Section 3) and one cannot hope for a better algorithm than static inlining, in the worst case. On one hand, it is unlikely that a polynomially-sized formula captures a bounded RMT problem; on the other hand, static inlining is inefficient on practical problems. In an experiment on safety verification of device drivers [25], we found that static inlining ran out of memory during VC generation. Even when the VC did fit in memory, the SMT solver (Z3) was overwhelmed by its size and timed out in many instances.

This section presents the *stratified inlining* (SI) algorithm for solving the bounded RMT problem with respect to a bound $b > 0$. SI tries to delay the construction of an exponentially-sized formula as much as possible, in hope of efficiently solving RMT for most programs. Instead of inlining all procedures upfront, SI inlines procedures on-demand, in a goal-directed manner. Experiments validated stratified inlining to be much more efficient than static inlining in practice [25].

Overview. SI works as follows: at any point in time, SI maintains a partially-inlined program P, along with the set of call-sites C of P that have not been inlined so far. Initially, P is `main` and C is the set of all call-sites in `main`. Next, it queries the theorem prover to see if P has a valid execution of `main` that does not go through any call-site in C. If so, it returns this execution. If not, then it queries the theorem prover again, this time allowing executions to go through C and simulating the effect that open call-sites can modify state arbitrarily. This query represents an over-approximation of the input program. If no valid

```
procedure main() {
   var i: int;
   i := 0;
   if( * )
     call i := foo(i);
   else
     call i := bar(i);
   assume i >= 5;
}
```

```
procedure foo(i: int)
     returns (i: int) {
   if( * ) {
     i := i + 1;
     call i := foo(i);
     call i := bar(i);
   }
}
```

```
procedure bar(i: int)
     returns (i: int) {
   if( * ) {
    i := 2*i;
    call i := bar(i);
   }
}
```

Fig. 6. An example program

```
procedure main() {
   var i1,i2,i3,i4: int;
   assume i1 = 0;
   if( * )
     call i2 := foo(i1);
     assume i4 = i2;
   else
     call i3 := bar(i1);
     assume i4 = i3;
   assume i4 >= 5;
}
```

```
procedure main() {
   var i1,i2,i3,i4: int;
   var c1,c2: bool;

   assume i1 = 0;
   if( * )
     assume c1;
     assume i4 = i2;
   else
     assume c2;
     assume i4 = i3;
   assume i4 >= 5;
}
```

```
c1 ↦ (foo, (i1, i2))
c2 ↦ (bar, (i1, i3))
```

Fig. 7. The passified version of `main` of Fig. 6; replacing procedure calls with fresh Boolean constants; and the mapping between such constants and the input-output variables of the corresponding procedure call

execution of P is found, then the original program is safe, i.e., RMT does not hold. If there is a valid execution of P, then it must go through some call-sites in C. These call-sites are inlined, provided they are under the recursion bound, and the process continues. We now describe this process in more detail.

The VC generation algorithm used by SI is similar to the one described in Section 3, with slight modifications to handle procedure calls. Given a passified procedure `f`, we replace each procedure call with **assume** c for a fresh Boolean constant c, and then do the VC generation as usual. An example is shown in Fig. 7. In this case: (1) constraining c to **false** blocks executions that go through the call, underapproximating the behaviors of the call, and (2) constraining c to **true** allows executions in which the return values of the call can be arbitrary. For example, in Fig. 7, if `c1` is **true** then there is no constraint between `i1` and `i2`, i.e., the call to `foo` could return any output. This represents an overapproximation to the call. When these Boolean constants are introduced, we also record the mapping between them and the input-output variables of the calls that they replace, as shown on the right of Fig. 7.

For a procedure `f`, let VcGen(`f`) be a tuple $(\phi(\boldsymbol{io}, \boldsymbol{t}), d)$ such that $\phi(\boldsymbol{io}, \boldsymbol{t})$ is the VC of `f`, $\boldsymbol{io}$ are the interface (input, output) variables of `f`, $\boldsymbol{t}$ are some internal variables and d is a map from Boolean variables to information about

Procedure INIT(`main`)

1: Let $(\phi(\boldsymbol{io}, \boldsymbol{t}), d)$ = VCGEN(`main`)
2: CallTree := (`main`)
3: Assert $\phi(\boldsymbol{io}, \boldsymbol{t})$
4: **for all** $c \mapsto (\mathtt{f}, \boldsymbol{io'})$ **in** d **do**
5: Create edge $(\mathtt{main}, (\mathtt{f}, \boldsymbol{io'}, c))$ in CallTree
6: **end for**

Procedure INSTANTIATE(Node n)

1: Let $(\mathtt{f}, \boldsymbol{io'}, c) = n$
2: Let $(\phi(\boldsymbol{io}, \boldsymbol{t}), d)$ = VCGEN(`f`)
3: Let $\boldsymbol{t'}$ be fresh variables
4: Assert $c \Rightarrow \phi(\boldsymbol{io'}, \boldsymbol{t'})$
5: Let $d' = d[\boldsymbol{t'}/\boldsymbol{t}][\boldsymbol{io'}/\boldsymbol{io}]$
6: **for all** $c \mapsto (\mathtt{b}, \boldsymbol{v})$ **in** d' **do**
7: Create edge $(n, (\mathtt{b}, \boldsymbol{v}, c))$ in CallTree
8: **end for**

Fig. 8. Procedures for initializing and growing the CallTree

Procedure QUERYUNDER()

1: Push
2: **for all** leaves $l = (\_, \_, c)$ **do**
3: Assert $\neg c$
4: **end for**
5: Check
6: **if** Satisfiable **then**
7: **return** "RMT$_b$ holds"
8: **end if**
9: Pop

Procedure QUERYOVER()

1: Push
2: **for all** leaves $l = (\_, \_, c)$ **do**
3: **if** $\mathtt{RB}(l) > b$ **then**
4: Assert $\neg c$
5: **end if**
6: **end for**
7: Check
8: **if** Unsatisfiable **then**
9: **return** "RMT_b does not hold"
10: **end if**
11: Let τ be the error trace
12: Pop
13: **for all** leaves l on τ **do**
14: INSTANTIATE(l)
15: **end for**

Fig. 9. Querying the theorem prover with under- and over-approximations

the procedure calls that they replaced. For Fig. 7, d(`c1`) = `(foo, (i1, i2))` and d(`c2`) = `(bar, (i1, i3))`.

The SI algorithm maintains a partially-inlined program in the form of a tree, called the *CallTree*. Nodes of the tree represent a dynamic instance of a procedure and children of a node are the procedures called by that node. Thus, the *CallTree* is a partial unrolling of the call graph of the program. Internal nodes of the tree are all the procedure that have been inlined so far by SI, and leaves represent non-inlined procedure calls. We also use the term "open call-sites" to refer to leaves or the non-inlined calls. SI maintains the invariant that at any time, the VCs of all internal nodes are asserted in the theorem prover stack. All nodes in the *CallTree*, except the root node, are a triple $(\mathtt{f}, \boldsymbol{io}, c)$ where `f` is the name of the procedure, $\boldsymbol{io}$ are the input-output variables of this particular dynamic instance of `f`, and c is the unique Boolean variable that substituted the call to

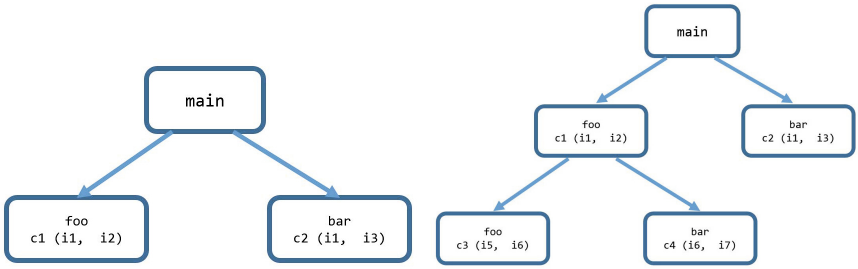

Fig. 10. The shape of the *CallTree* after initialization (left) and after instantiating `foo` (right)

`f` from its parent node during VC generation. SI uses the standard SMT solver API:

- Push: Creates a backtracking point.
- Pop: Undo all asserted formulae until the most recent backtracking point.
- Assert: Add (conjoin) a formula to already asserted formulae.
- Check: Check for satisfiability of asserted formulae.

The procedures Init and Instantiate, shown in Fig. 8, initialize and grow the *CallTree*, respectively. Init takes the name of the starting procedure (`main`) and creates a tree with root labeled `main` and one leaf for each procedure called by `main`. Fig. 10 shows the initial tree for our running example of Fig. 6. Instantiate takes a leaf node and inlines the procedure represented by that node. It does so by generating the VC (line 2), renaming the interface variables and asserting the VC (line 4). The asserted formula says that if c is **true**, then the constraint imposed by `f` must be satisfied. Next, we then create new leaves for all callees of `f` (line 7).

Given a *CallTree*, SI makes two kinds of queries, shown in Fig. 9. QueryUnder tries to see if RMT_b holds: it first blocks all open call-sites (line 3) and then checks if the currently asserted formula is satisfiable (line 5). If so, then we have found a valid program execution (because it only goes through inlined calls) and RMT_b holds. QueryOver tries to see if RMT_b does not hold. First, it blocks all open call-sites whose recursion bound exceeds b (line 4). The sub-routine `RB` takes a leaf node, say $l = (\texttt{f}, \_, \_)$ and simply counts the number of instances of `f` along the path from l to the root. (`RB` simulates the *Count* function of Figure 2.) It is easy to see that this count is the number of times `f` must appear on the call-stack when execution reaches l. For example, `RB` of the leaf node `foo` in Fig. 10 is 1. If the check on line 7 is satisfiable, then we use the model to construct the corresponding program execution. Next, we inline all open calls on this path using Instantiate (line 13). Note: (1) QueryOver will never inline a leaf that has crossed the recursion bound b because such open calls are blocked (line 4), and (2) the blocking of open calls in both QueryUnder and QueryOver is nested inside Push-Pop operations, hence this blocking does not persist beyond a single query.

We can now write down the full SI algorithm. It simply calls the two queries in alternation, until one of them returns "RMT_b holds" or "RMT_b does not hold".

```
1: Init(main)
2: while true do
3:    QueryUnder();
4:    QueryOver();
5: end while
```

The SI algorithm is guaranteed to terminate because each iteration of the while loop, if it is not the last iteration, must grow the *CallTree*. This is because when QueryUnder is not able to find a path within the inlined part, it must be that the trace τ on line 11 of QueryOver passes through some open call. Moreover, the size of *CallTree* is bounded because of the recursion bound. Hence, SI will terminate in bounded time. SI makes at most exponential number of queries on formulas that are atmost exponential in the size of the program. Asymptotically, SI has the same complexity as static inliining.

Related Work. Stratified Inlining draws inspiration from multiple sources. Previous work on *structural abstraction* [4] and *inertial refinement* [33] has similarities with SI. However, work on structural abstraction does not use an underapproximation-based query (by blocking open call sites). Inertial refinement does use both over and under approximations to iteratively build a view of the program that is then analyzed. A distinguishing factor is our use of recursion bounding as well as using lazy inlining to construct a single VC for the entire program view.

It has been illustrated in the SMT community that dealing with eager instantiation of either theory lemmas or quantifiers (e.g. as done in UCLID tool [24]) does not provide the most scalable way to reason about SMT. Instead lazy instantiation tends to scale much better. Similarly, we believe that lazy approaches like SI have much better chance of being successful than full static inlining.

6 The Corral Solver

The Corral tool is a practical realization of a solver for bounded RMT. It is designed for the Boogie programming language. It takes a Boogie program (with assertions) as input, the name of the starting procedure (`main`) and a recursion bound. The assertions in the input program are removed using a source-to-source transformation to obtain a usual bounded RMT problem as follows.

- Introduce a Boolean variable `error` and initialize it to **false** at the entry to `main`.
- Replace **assert** e with `error := `e`; if(error) return`.
- After each procedure call, insert **if** `(error) return`.
- At the exit of `main`, assume that `error` is **true**.

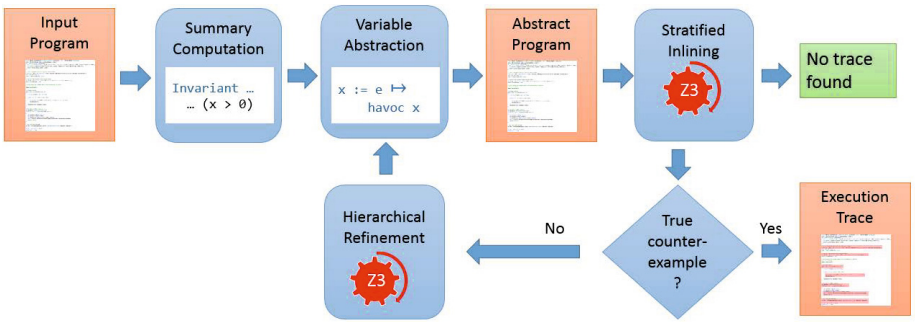

Fig. 11. Corral's architecture

After such a transformation, there is an execution that exits `main` if and only if the original program had a failing assertion.

As output, Corral can either return a feasible path that ends in an assertion violation (i.e., RMT holds), or a message saying that no such path exists (i.e., RMT does not hold), or a message saying that no such path was found within the recursion bound (i.e., bounded RMT does not hold, but no conclusion can be made for unbounded RMT). Corral was the subject of a previous publication [25]. This paper only briefly covers the ideas and algorithms behind Corral.

The design of Corral is shown in Fig. 11. The main component of Corral is the stratified inlining (SI) algorithm that was described in Section 5. Instead of directly giving the input RMT problem to SI, Corral uses two optimizations to reduce the computational burden on SI.

The first optimization is to compute program invariants. In principle, any technique for invariant generation may be used. Corral uses the Houdini algorithm [17] to compute invariants in the form of procedure summaries. The user provides, as additional input to Corral, *candidate* expressions for procedure summaries. Houdini uses theorem prover queries, each on the VC of at most one procedure, to compute the strongest inductive summaries within the given candidates. The number of single-procedure queries is quadratic in the worst case but linear in the common case. The invariants, once computed, are injected back into the program as assume statements. These invariants can help SI prune search because they (soundly) constrain the over-approximate query used by SI, which can rule out many abstract counterexamples.

The second optimization is an abstraction-refinement loop. Corral uses a very simple abstraction in this loop, called *variable abstraction*. Let G be the set of global variables of the input program. Note that G is always a finite set. Variable abstraction is parameterized by a set T of *tracked* variables, where $T \subseteq G$. Variable abstraction works by abstracting away all variables in $G - T$, using a simple source rewriting. For instance, the assignment `x := e`, where `x` $\in T$ and the expression e has some variable in $G - T$, is re-written to **havoc** `x`. The abstracted program is fed to SI. This abstraction can lead to SI returning spurious counterexamples. These counterexamples are used to refine the abstraction by

increasing T. The variable-abstraction based refinement loop substantially differs from one based on predicate abstraction that is used by most software model checking tools: (1) variable abstraction is easy to compute, unlike predicate abstraction that may need an exponential number of SMT queries, and (2) the refinement loop of variable abstraction is bounded because T is bounded above by G, unlike predicate abstraction where the number of predicates is unbounded.

The abstraction-refinement loop of Corral is useful when only a few variables are relevant in solving the RMT problem. Abstracting away variables considerably reduces the size of the program and allows the theorem prover to focus only on the relevant part of the program's data. Moreover, VC generation is quadratic in the number of variables, and a reduction in the number of variables also significantly decreases the VC size.

6.1 Experience Using Corral

Corral is ideally suited for applications where one is more interested in finding bugs than in finding proofs, or when finding proofs is simply too difficult. We now describe our experience with such applications.

The first application is the Static Driver Verifier (SDV), a product supported by the Driver Quality team of Microsoft Windows. SDV supports any driver written using one of four diferent driver models: WDM, KMDF, NDIS, STORPORT. It also comes with a list of rules (or properties) that drivers must satisfy. A driver and rule pair forms a *verification instance* that is fed to the verification engine. SDV has traditionally used Slam as the verification engine. After a comprehensive evaluation, a dual-engine system of Corral and Yogi [20, 29] will replace Slam inside SDV in the next release of the Microsoft Windows operating system. Going into the details of the evaluation is outside the scope of this paper. We briefly present our experience with Corral in comparison to Slam.

Corral is executed with a modest recursion bound of 3 to 6, depending on the driver model. In an initial study [25], this bound was sufficient to find all but 9 defects in a test suite containing around 400 defects in total. The missed defects were due to loops with a constant upper bound, for example, the loop **for**`(int i = 0; i < 27; i++)` requires a bound of at least 27 before code after the loop is reachable. We designed custom techniques to deal with such loops, after which Corral was able to find almost all defects reported by Slam. Furthermore, Corral has 2.5X reduction in timeouts and 40% improvement in running time.

While Corral was able to overtake Slam in the number of defects found, it was also important to compare the number of instances that were proved correct. This will indicate a measure of confidence that Corral will not miss defects in yet unseen drivers. Along with each verification instance, we also supplied summary candidates for Houdini. The candidates are derived heuristically looking solely at the property (not the driver) being verified. On an initial test suite, Corral was able to prove correctness in 91% of the cases (regardless of the recursion bound), with summaries inferred by Houdini playing an important role

in establishing proofs. This means that for most part our heuristically-generated invariant candidates were sufficient.

We obvserved similar speedups against the YOGI tool as well. We now list some other lessons learned from these comparisons.

- While SLAM and YOGI were designed and trained on drivers for SDV, CORRAL was initially designed for finding concurrency bugs. The fact that CORRAL performed well inside SDV demonstrates a degree of robustness of CORRAL in handling programs from multiple sources.
- SLAM avoided using array theory to model the heap, and instead relied on a *logical* memory model [5]. This decision was justified because implementations of array-theory might have been inefficient ten years ago. YOGI inherited a similar memory model as SLAM. CORRAL, on the other hand, makes heavy use of array theory because the entire heap is modeled using maps. Thus, as theorem-prover technology changes, it is reasonable to expect the design of software verifiers to change as well.
- We found CORRAL to be much more dependent on the performance of Z3 than YOGI. For instance, upgrading Z3 almost always resulted in a significant speedup of CORRAL, whereas, we did not observe such speedups in YOGI. In retrospect, this is not surprising because YOGI was designed to avoid invoking the theorem prover to the greatest extent possible.

The second class of applications for CORRAL are what are now called *sequentializations* of concurrent programs. The original sequentialization was a program transformation that converted a safety property on a multi-threaded concurrent program to a safety property on a sequential program, given a bound on the number of context switches between threads [31, 26]. Subsequently, more sequentializations were proposed: for asynchronous task-buffer programs [16], for liveness properties of concurrent programs [15, 2], and for programs on weak-memory models [3]. In each of these cases, CORRAL was used as the solver for finding defects in the generated sequential program. These applications have two common aspects. First, they require bounding the set of program behaviors (e.g., context switches). Thus, verifying correctness cannot be a goal. Second, the generated sequential programs are complicated and so are their invariants. Consequently, software verifiers like SLAM do not perform well on such programs. On the other hand, CORRAL, which builds off the robustness of SMT solvers, is able to work well uniformly across such programs.

CORRAL has been used to automatically detect security vulnerabilities in models of web applications. In one study, Cashier-as-a-Service web payment systems were modeled in the C language; CORRAL was used to find vulnerabilities that would allow an attacker to shop for free [34]. In another study, authentication and authorization SDKs were modeled in the C# language; CORRAL was used to find improper use of SDK APIs leading to insecure access [35].

Besides these applications, Corral is also used inside a debugging tool for .NET, called GetMeHere, where it is able to successfully operate on Boogie programs compiled from C#. We also evaluated Corral on the Software Verification Competition (SV-COMP) benchmarks and obtained favorable results compared to all the other tools participating in the competition. More details are available in the original Corral paper [25].

References

[1] Alur, R.: Formal analysis of hierarchical state machines. In: Dershowitz, N. (ed.) Verification (Manna Festschrift). LNCS, vol. 2772, pp. 42–66. Springer, Heidelberg (2004)

[2] Atig, M.F., Bouajjani, A., Emmi, M., Lal, A.: Detecting fair non-termination in multithreaded programs. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 210–226. Springer, Heidelberg (2012)

[3] Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 99–115. Springer, Heidelberg (2011)

[4] Babić, D., Hu, A.J.: Structural abstraction of software verification conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 366–378. Springer, Heidelberg (2007)

[5] Ball, T., Bounimova, E., Levin, V., de Moura, L.: Efficient evaluation of pointer predicates with Z3 SMT Solver in SLAM2. Technical Report MSR-TR-2010-24, Microsoft Research (2010)

[6] Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM 54(7), 68–76 (2011)

[7] Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Programming Language Design and Implementation (2001)

[8] Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for boolean programs. In: SPIN, pp. 113–130 (2000)

[9] Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)

[10] Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Program Analysis for Software Tools and Engineering (2005)

[11] Barnett, M., Qadeer, S.: BCT: A translator from MSIL to Boogie. In: Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation (2012)

[12] Bjørner, N., McMillan, K.L., Rybalchenko, A.: Program verification as satisfiability modulo theories. In: SMT (2012)

[13] Bjørner, N., McMillan, K.L., Rybalchenko, A.: On solving universally quantified horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. LNCS, vol. 7935, pp. 105–125. Springer, Heidelberg (2013)

[14] Condit, J., Hackett, B., Lahiri, S., Qadeer, S.: Unifying type checking and property checking for low-level code. In: Principles of Programming Languages (2009)

[15] Emmi, M., Lal, A.: Finding non-terminating executions in distributed asynchronous programs. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 439–455. Springer, Heidelberg (2012)

[16] Emmi, M., Lal, A., Qadeer, S.: Asynchronous programs with prioritized task-buffers. In: Foundations of Software Engineering (2012)
[17] Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)
[18] Godefroid, P., Yannakakis, M.: Analysis of boolean programs. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 214–229. Springer, Heidelberg (2013)
[19] Graf, S., Saïdi, H.: Construction of abstract state graphs with pvs. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
[20] Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: Foundations of Software Engineering (2006)
[21] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Principles of Programming Languages (2002)
[22] Lahiri, S., Qadeer, S.: Back to the future: Revisiting precise program verification using SMT solvers. In: Principles of Programming Languages (2008)
[23] Lahiri, S.K., Qadeer, S., Galeotti, J.P., Voung, J.W., Wies, T.: Intra-module inference. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 493–508. Springer, Heidelberg (2009)
[24] Lahiri, S.K., Seshia, S.A.: The UCLID decision procedure. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 475–478. Springer, Heidelberg (2004)
[25] Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 427–443. Springer, Heidelberg (2012)
[26] Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods in System Design 35(1) (2009)
[27] Lewis, H.R.: Complexity results for classes of quantificational formulas. J. Computer and System Sciences 21, 317–353 (1980)
[28] Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
[29] Nori, A.V., Rajamani, S.K.: An empirical study of optimizations in YOGI. In: International Conference on Software Engineering, pp. 355–364 (2010)
[30] Piskac, R., de Moura, L.M., Bjørner, N.: Deciding effectively propositional logic using DPLL and substitution sets. J. Autom. Reasoning 44(4), 401–424 (2010)
[31] Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: Programming Language Design and Implementation, pp. 14–24 (2004)
[32] Rakamaric, Z., Emmi, M.: SMACK: Static Modular Assertion ChecKer, `http://smackers.github.io/smack`
[33] Sinha, N.: Modular bug detection with inertial refinement. In: FMCAD (2010)
[34] Wang, R., Chen, S., Wang, X., Qadeer, S.: How to shop for free online — security analysis of Cashier-as-a-Service based web stores. In: IEEE Symposium on Security and Privacy, pp. 465–480 (2011)
[35] Wang, R., Zhou, Y., Chen, S., Qadeer, S., Evans, D., Gurevich, Y.: Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization. In: USENIX Security Symposium (2013)