Analyzing Recursive Programs using a Fixed-point Calculus *

Salvatore La Torre
Dipartimento Informatica e Applicazioni,
Università degli Studi di Salerno, Italia.
latorre@dia.unisa.it

P. Madhusudan
Department of Computer Science,
Univ. of Illinois at Urbana-Champaign,
Illinois, USA.
madhu@illinois.edu

Gennaro Parlato
Department of Computer Science,
Univ. of Illinois at Urbana-Champaign,
Illinois, USA.
Università degli Studi di Salerno, Italia.
parlato@illinois.edu

Abstract

We show that recursive programs where variables range over finite domains can be effectively and efficiently analyzed by describing the analysis algorithm using a formula in a fixed-point calculus. In contrast with programming in traditional languages, a fixed-point calculus serves as a high-level programming language to easily, correctly, and succinctly describe model-checking algorithms. While there have been declarative high-level formalisms that have been proposed earlier for analysis problems (e.g., Datalog), the fixed-point calculus we propose has the salient feature that it also allows *algorithmic* aspects to be specified.

We exhibit two classes of algorithms of symbolic (BDD-based) algorithms written using this framework— one for checking for errors in sequential recursive Boolean programs, and the other to check for errors reachable within a bounded number of context-switches in a concurrent recursive Boolean program. Our formalization of these otherwise complex algorithms is extremely simple, and spans just a page of fixed-point formulae. Moreover, we implement these algorithms in a tool called GETAFIX which expresses algorithms as fixed-point formulae and evaluates them efficiently using a symbolic fixed-point solver called MUCKE. The resulting model-checking tools are surprisingly efficient and are competitive in performance with mature existing tools that have been fine-tuned for these problems.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program VerificationModel checking; F.3.1 [*Theory of Computation*]: Specifying and Verifying and Reasoning about Programs; F.4.1 [*Theory of Computation*]: Mathematical LogicTemporal logic.

General Terms Algorithms, Experimentation, Verification.

Keywords Software verification, abstraction, logic, μ-calculus, model-checking, recursive systems.

* This work was partially funded by NSF CAREER Award CCF 0747041, by the Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign (a center sponsored by Intel Corporation and Microsoft Corporation), and the MIUR grants ex-60% 2007-2008 and FARB 2009 Università degli Studi di Salerno (Italy).

PLDI'09, June 15–20, 2009, Dublin, Ireland.

1. Introduction

Abstraction is a key concept in verification. Analyzing complex programs against relatively simple properties has become practical in recent years primarily due to abstraction based techniques, which include type-checking, static-analysis, and abstract-interpretation. Abstraction-based verification is often composed of two distinct methods— the abstraction engine that simplifies a complex program into a tractable model and an analysis engine that, using fixed-point computations, deduces properties of the model.

This paper is devoted to the key analysis component in the second phase of verification— model-checking the state-spaces of recursive *Boolean* models of imperative programs.

A Boolean program is a recursive imperative program where all variables range over the Boolean domain only. Boolean programs are a standard model used in abstraction based verification. The fairly successful paradigm of verifying control-intensive properties using *predicate abstraction*, for instance, generates Boolean program models. Microsoft Research's SLAM engine [5], its commercial developer-kit cousin SDV (Static Driver Verifier [3]), the checker BLAST [6], and the tool Terminator [10] that checks termination for programs, are all prominent artifacts that construct Boolean models whose analysis leads to finding errors or proving programs correct.

Several model-checkers for Boolean programs exist, and many of the efficient checkers use symbolic techniques that represent sets of states using Boolean Decision Diagrams (BDDs). The SLAM tool utilizes the reachability solver BEBOP [4] that computes procedure summaries using BDDs (see [21, 18] for the traditional algorithms for reachability). The MOPED tool is a very efficient and mature checker: one of its versions computes the reachable set by computing the finite automaton that represents the set of all reachable configurations (including the stack), while the other two versions use a weighted pushdown system library [11].

Implementing symbolic model-checking algorithms is, in general, a complex task. A researcher who comes up with a new algorithm to model-check a system often implements in a traditional programming languages like C or Java, utilizing a BDD-library to perform the symbolic operations. Myriad heuristics needs to be put in place, including decisions on ordering of variables, managing memory and caching, deciding order of evaluations, when to reset temporary terms, and a host of other tricks, to obtain a reasonably efficient implementation. Moreover, small changes to the algorithm may require considerable change in the design of the program, discouraging the testing of new ideas, and severely affecting the productivity of the programmer.

In this paper, we propose that most symbolic model-checking algorithms are essentially *fixed-point computations* that can be compactly described in a high-level fixed-point calculus. Our main

thesis is that model-checking algorithms for Boolean programs can be described easily using a fixed-point calculus, and by utilizing a symbolic fixed-point solver, we can obtain efficient model-checkers, without resorting to low-level programming.

Our main contribution is to substantiate the above thesis by building efficient symbolic model-checking algorithms for solving the reachability problem for sequential and concurrent recursive Boolean programs, writing the algorithms in a fixed-point calculus. We show that by using a fixed-point solver, we can automatically obtain an efficient implementation of these algorithms. We have implemented our ideas in a new tool, GETAFIX ("get a fix using fixed-points"), which take as input sequential and concurrent Boolean programs, and model-checks them against reachability specifications. A detailed experimental evaluation shows that our solvers are competitive with sophisticated mature checkers that have been engineered for these problems.

For sequential Boolean programs, we show that we can encode the traditional algorithm for reachability using simple fixed-point formulae. We adapt the traditional algorithm into two algorithms, which ensure better performance by restricting themselves to searching using only the reachable parts of the search-space. The formulae depicting the two algorithms are very readable small formulae, one of 40 lines and the other of 50 lines!

Concurrent recursive Boolean programs with shared memory do not have a decidable reachability problem. Since concurrency bugs are notoriously hard to find, and since abstracting a program naturally yields a recursive model, the problem of checking reachability in concurrent programs is an important one. A recent proposal [16] is to search the space reached by a concurrent program using at most k context switches, where k is a fixed bound. Intuitively, exploring a few number of context-switches exhaustively searches a very interesting space for concurrency errors, and studies have shown that a few context-switches give very high coverability of the entire search space, and that many concurrency errors manifest within a small context-switch bound [15].

Bounded context-switching reachability for concurrent recursive programs (with a fixed number of threads) was shown to be decidable by Qadeer and Rehof in 2005 [16]. The first algorithms proposed to solve this problem involved, unfortunately, keeping the entire description of stack contents reached by each individual process using finite automata (similar to the way BEBOP computes reachability). A model-checker for bounded context-switching was not implemented until recently as it involved a complex algorithm involving automata represented as BDDs. Recently, Lal and Reps [12] have shown how to reduce the context-bounded reachability problem to the reachability problem for sequential recursive programs. However, this conversion and the implementation are algorithms that search using even unreachable parts of the search space (called the *eager approach* in their paper). They also propose a set of rules for computing the reachable states lazily, exploring only the reachable parts of the search space, but this has not been implemented. Also, very recently, there has been an implementation of context-bounded reachability in an extension of Moped (called JMoped) that solves the problem by iteratively computing the automata representing reachable configurations.

In this paper, we show a new (and simple) fixed-point formulation of the algorithm to solve bounded context-switching reachability of concurrent recursive programs. Our algorithm is described using a formula that adds a couple of clauses to the sequential reachability algorithm, and computes only using the reachable states of the concurrent program. While our fixed-point formulation is similar to the set of rules for lazy evaluation described in [12], it is new and more efficient. In particular, the rules presented in [12] involve keeping tuples, where the tuple contains as many as $3k$ global variables, where k is the context-switching bound. Ours, in contrast, involves a fixed-point computation of tuples that keep track of at most $k+1$ copies of global variables only. As k increases, this tuple's length depicts the precise space increase in model-checking, and hence we believe that our formulation is significantly more efficient. It is worth mentioning that in [12] the authors assume that along any computation the control switches happen according to a round-robin scheduling. Thus, for computations allowing up to r round-switches (which, for 2 threads, means $2r$ context switches), they can implement their rules by using just $3r$ global variables. We observe that this improvement is orthogonal to what we propose in this paper, and our ideas are applicable in their setting too, thus reducing the number of global variables required.

Turning to the implementation, we encode our algorithm for bounded-context switching reachability as a fixed-point formula, and obtain an efficient symbolic model-checking algorithm. The fixed-point formulation is quite simple, and in fact it took us only a couple of days to write the formula (algorithm), and rearrange terms to get efficiency! This algorithm is also incorporated in our implementation, and we show that it works well in practice on a class of Bluetooth examples.

Our proposal of using a high-level fixed-point calculus as a programming language to easily write symbolic model-checking algorithms has several advantages. First, it eases the effort in building model-checking tools considerably. We have implemented scores of variants of the model-checking algorithm, using a simple rewriting of the formula, and evaluated the results. If we had implemented our checker using a language such as C, we would have certainly not have tried these many variants, nor would we have implemented the concurrent program checker (let alone implementing it two days). We believe that our approach greatly eases programming, letting novice programmers and theoreticians implement their algorithmic ideas with little effort, while not sacrificing efficiency.

Secondly, we believe that our framework aids building of *correct* correctness tools. Most model-checking tools have bugs, primarily due to the hard task of managing computations and algorithms natively in a traditional programming language. During our experiments, we found errors in MOPED, which were duly corrected by its authors. Seeing our entire algorithm on one page aided us greatly in reviewing the "code" (formula) and arguing its correctness. We found the transformation from the mathematical formulation of the algorithm on paper to the fixed-point formula easy and intuitive.

The fixed-point calculus that we propose to use is a quantified Boolean logic with least-fixed point operators (more precisely, we use first-order logic on the domain $\{true, false\}$ with an additional least fixed-point operator). While this calculus has a natural semantics based on least-fixed points, we also endow it with an *operational* semantics that precisely describes how the fixed point will be computed. This operational semantics is the natural Tarskian iterative computation, and is intuitive and natural. Furthermore, even if the formulae we write are not monotonic (and hence least-fixed points may not exist), the operational semantics gives precise algorithmic meaning to our formulae. In fact, in one of our algorithms (the optimized entry-forward algorithm), it becomes essential to use a non-monotonic operator. In this case, the operational semantics is what gives us algorithmic control over the way the fixed-points are computed.

Declarative approaches to building analysis tools have been proposed earlier. Notably, Lam et al [13] propose to reduce context-sensitive static analysis questions to answering queries in Datalog. Using an efficient BDD-based solver for Datalog (bddbddb), the authors obtain automatic implementation for program analysis problems. They too argue that using an intermediate high-level lan-

guage to state analysis problems greatly eases the engineering of an efficient static analysis tool.

It is in fact well known that reachability of recursive programs reduces to querying Datalog (see [1] for a precise reduction). However, though Datalog is a convenient way to declaratively state the reachability problem, Datalog is too weak to express the *algorithmic control* that we desire in guiding the search. One may of course wonder whether a clever Datalog solver like bddbddb may outperform the algorithms that we wish to encode. We actually did try this idea, and in fact built an automatic transformation of Boolean programs to Datalog. However, the initial experimental results were poor, and we abandoned this approach. The programmer has very little control on how the Datalog queries are computed; heuristics like magic-set transformations and other optimizations are often implemented and discourage control by the programmer. We believe that some control on how exactly the algorithm searches the state-space (like the control required to describe the optimized entry-forward algorithm we formulate in this paper) is required to build an efficient reachability solver.

In summary, this paper makes the following contributions:

- The thesis that symbolic model-checking algorithms can be easily, correctly, and efficiently engineered by expressing the model-checking algorithm as a formula in a fixed-point calculus over the Boolean domain.
- The formulation of three algorithms for checking reachability in recursive Boolean programs as fixed-point formulae. The algorithms get increasingly complex to describe but are increasingly efficient, and the final one is competitive with existing mature solvers.
- A new fixed-point formulation of the reachability problem for concurrent recursive Boolean programs under a context-switching bound. Our formulation involves a fixed-point over a tuple that is considerably shorter than previous formulations.
- The implementation of a model-checker, called GETAFIX, that automatically translates Boolean programs to Boolean formulae that capture its behavior, and by incorporating the above mentioned algorithms written using fixed-point formulae, implements them using the symbolic fixed-point solver MUCKE.
- An extensive evaluation of GETAFIX on sequential programs, derived from SLAM device-driver benchmarks and TERMINATOR benchmarks that show that GETAFIX is extremely competitive with existing tools.
- An evaluation of the GETAFIX for reachability in concurrent programs on a class of BlueTooth benchmarks, where it performs well, competitive with existing prototype tools that have been developed to solve this problem.

Related work

The BEBOP tool implements reachability in recursive Boolean programs using summaries, similar to the algorithms proposed in this paper. The tool MOPED [11, 20] implements both forward and backward symbolic reachability algorithms for Boolean programs. The tool is built for pushdown systems with local and global variables ranging over finite domains, and the implemented algorithms construct a finite automaton accepting the set of reachable configurations (correctness relies on the fact that this set is regular [8]). The algorithms proceed by growing the automaton until saturation. Each edge added to the automaton in this process corresponds to adding a pair to the summary of a procedure, and in these regards, the forward algorithm builds the summary by taking the entry forward (ϵ-edges summarize the entry-to-exit reachability relation) and the backward algorithm by taking the exit backward.

Weighted pushdown systems, where edges of the pushdown graph are endowed with weights, have been used to analyze interprocedural dataflow analysis problems [19]. A new version of MOPED has been built using the weighted pushdown system library to compute reachability of Boolean programs as well.

In [1], the model-checking of recursive state machines (a formalism equivalent to Boolean programs) is studied. There, the authors propose a reachability algorithm that computes the summary for each procedure going either forward or backward depending on whether the number of entries and the number of exits is smaller. This yields an $O(n\,\theta^2)$ bound on the time complexity for solving reachability, where n is the number of vertices and θ is the maximum over all procedures of the minimum between the number of entries and the number of exits. Though the algorithm can be implemented symbolically via Datalog rules, the backward-computation utilized in this can discover unreachable states. Recently, a subcubic algorithm for recursive state machines was given in [9], though it is not clear how to implement it symbolically. Symbolic reachability has been implemented for hierarchic reactive modules (no recursion) in the tool HERMES [2].

The idea of studying context-bounded reachability stems from a paper by Qadeer and Wu [17] where the authors study the problem for two context-switches. In [16], it was shown that the bounded context-switching problem for recursive concurrent programs is decidable for any bound. In a recent paper, Lal and Reps propose a constructive reduction of the bounded context-switching reachability problem to reachability on sequential recursive programs. They also provide an eager implementation of their algorithm. More recently, the paper [22] describes an implementation of the original algorithm by Qadeer and Rehof [16] that checks bounded context-switching reachability using a tuple of automata that represent the configurations of the individual threads.

As mentioned earlier, Datalog has also been successfully used as a high-level language to express context-sensitive analysis of recursive programs, by using a BDD-based solver (bddbddb) for Datalog [24, 13].

A convenient intermediate level between using high-level specifications and programming directly with BDDs in order to implement program analysis tools is given by JEDD [14], where BDDs are abstracted as database-style relations and operations on relations. Jedd, however, does not provide fixed-point operators that we need to express algorithms.

2. Recursive Boolean programs

Let us fix the syntax of a simple *sequential* programming language with variables ranging only over the Boolean domain, and with explicit syntax for nondeterminism, (recursive) function calls, and tuples of return values.

Let us fix a set of variable names Var. Programs are described by the following grammar:

$$
\begin{array}{lll}
\langle pgm\rangle & ::= & \langle gvar\text{-}decl\rangle\ \langle proc\text{-}list\rangle \\
\langle gvar\text{-}decl\rangle & ::= & \textbf{decl}\ x\textbf{;} \mid \langle gvar\text{-}decl\rangle\ \langle gvar\text{-}decl\rangle \\
\langle proc\text{-}list\rangle & ::= & \langle proc\rangle\ \langle proc\text{-}list\rangle \mid \langle proc\rangle \\
\langle proc\rangle & ::= & f^{h,k}(x_1,\ldots,x_h)\ \textbf{begin}\ \langle lvar\text{-}decl\rangle\ \langle stmt\rangle\ \textbf{end} \\
\langle lvar\text{-}decl\rangle & ::= & \textbf{decl}\ x\ \textbf{;} \mid \langle lvar\text{-}decl\rangle\ \langle lvar\text{-}decl\rangle \\
\langle stmt\rangle & ::= & \langle stmt\rangle\ \textbf{;}\ \langle stmt\rangle \mid \textbf{skip} \mid \langle assign\rangle \mid \\
 & & \quad \textbf{call}\ f^{h,0}(\ \langle expr_1\rangle,\ldots,\langle expr_h\rangle) \mid \\
 & & \quad \textbf{return}\ \langle expr_1\rangle,\ldots,\langle expr_k\rangle \mid \\
 & & \quad \textbf{if}\ (\langle expr\rangle)\ \textbf{then}\ \langle stmt\rangle\ \textbf{else}\ \langle stmt\rangle\ \textbf{fi} \mid \\
 & & \quad \textbf{while}\ (\langle expr\rangle)\ \textbf{do}\ \langle stmt\rangle\ \textbf{od} \\
\langle assign\rangle & ::= & x_1,\ldots,x_m\ \textbf{:=}\ \langle expr_1\rangle,\ldots,\langle expr_m\rangle \mid \\
 & & \quad x_1,\ldots,x_k\ \textbf{:=}\ f^{h,k}(\langle expr_1\rangle,\ldots,\langle expr_h\rangle) \\
\langle expr\rangle & ::= & T \mid F \mid * \mid x \mid \neg\ \langle expr\rangle \mid \langle expr\rangle \vee \langle expr\rangle \mid \\
 & & \quad \langle expr\rangle \wedge \langle expr\rangle
\end{array}
$$

In the above, x_i are variables in Var, and $f^{h,k}$ denotes a procedure with h formal parameters and k return values.

A program has a global variable declaration followed by a list of procedures. Each procedure is a declaration of local variables followed by a sequence of statements, where statements can be simultaneous assignments, calls to functions (call-by-value) that take in multiple parameters and return multiple values, conditional statements, while-loops, or return statements. Boolean expressions can be true, false, or non-deterministically true or false ($*$), and can be combined using standard Boolean operations. Functions that do not return any values are called using the **call** statement.

We will assume several obvious restrictions on the above syntax: global variables and local variables are assumed to be disjoint; formal parameters are local variables; the body of a function $f^{h,k}$ has only variables that are either globally declared, locally declared, or a formal parameter; a return statement in the body of $f^{h,k}$ returns exactly k values.

Let us also assume that there is a procedure `main`, which is the procedure where the program starts, and that there are no calls to this procedure in the code of P.

The semantics is the obvious one: a configuration of a program consists of a *stack* which stores the history of positions at which calls were made, along with valuations for local variables and formal parameters, and the top of the stack contains the local and global valuations, and a pointer to the current statement being executed.

Let us fix a program P, and let the set of formal parameters, local variables and global variables in P be respectively *Par*, L and G. As we observed above *Par* $\subseteq L$. A *global valuation* is a function $v_G : G \to \{T, F\}$ that assigns true or false to every variable in G. Analogously, a *local valuation* is a function $v_L : L \to \{T, F\}$ that assigns true or false to every variable in L. Let V_G and V_L denote the set of all global and local valuations, respectively.

Let us assume that each individual statement in each procedure in P is *labeled* uniquely by an element from a set *PC* (called *program counter*).

A *state* of the program P is given by a program counter, a local valuation and a global valuation. We will denote with $\overline{u}$ the state defined by the tuple $(\overline{u}.pc, \overline{u}.Local, \overline{u}.Global)$. The configuration of the program P is a *call-stack*— a stack of elements, each element being a pair containing a program counter and a local valuation, except the top of the stack which is the current state of P (i.e. the stack is a word in $(PC \times V_L \times V_G).(PC \times V_L)^*$).

The *reachability* problem asks whether a particular statement in the program marked using a special label *Goal* is reachable. More precisely, given a program counter *pc*, the problem is to determine if there is a reachable configuration where *pc* is the program counter of the current state.

3. Fixed-point calculus over Boolean domains

In this paper, we will use a basic first-order logic along with a fixed-point formula to represent model-checking algorithms for both sequential and concurrent recursive Boolean programs. Intuitively, a symbolic model-checking algorithm takes a set of *input* relations, typically describing the model that is being checked (for example, the transition relation and the set of initial states are input relations). Using these relations, the algorithm proceeds to compute several sets in order to check the problem at hand— for instance, a model-checker for *non-recursive* Boolean programs will start from a set of initial states, and compute the set of all reachable states.

The central theme of this paper is that a high-level fixed-point calculus serves as an adequate programming language to describe typical model-checking algorithms, and moreover, using an efficient underlying platform to execute these formulae, we can obtain fast and efficient verification tools. Most importantly, the formulae are very easy to write, follows the mathematical algorithm a programmer intends to write, and leads often to bug-free implementations. We believe that this programming paradigm will make novice programmers quickly implement their ideas, without the need to learn the (often undocumented) heuristics and yet obtain efficient implementations.

The fixed-point calculus we use in this paper is a first-order variant of the μ-calculus (more precisely, first-order logic with a least-fixed point operator suffices) that has operators Boolean combinations of sets, existential quantification over the Boolean domain, and least fixed-point operators.

Let us fix some notation.

A Boolean relation $R^k(x_1, \ldots, x_k)$ is any k-ary relation over the Boolean domain $\mathbb{B} = \{true, false\}$, for some $k \in \mathbb{N}$. I.e. $R^k \subseteq \mathbb{B}^k$.

Fix a set of variables *Var*. A Boolean expression over *Var* is given by the following syntax:

$$\begin{aligned} BoolExp ::= {} & T \mid F \mid R^k(x_1, \ldots, x_k) \mid \neg BoolExp \mid \\ & BoolExp \wedge BoolExp \mid BoolExp \vee BoolExp \mid \\ & \exists x.(BoolExp) \mid \forall x.(BoolExp) \end{aligned}$$

In the above, $x_1, \ldots, x_k$ range over variables in *Var* and R^k denotes any k-ary Boolean relation. The semantics of Boolean expressions is the standard one, and an expression defines some m-ary relation (where m is the number of free variables in the expression).

We say that a boolean expression is *positive* if the relations R^k used in the expression do not occur within an odd number of negations. (Intuitively, the formula is allowed to "test membership" of tuples in R^k, but not test non-membership.) In this case, the relation defined by the expression is *monotonic*, in the sense that when the relations R^k are interpreted as larger relations, the expression also relates a (not-strictly) larger relation as well.

An equation over R is an equation of the form $R = BoolExp$. Note that R may appear also in $BoolExp$ and thus this depicts a recursive definition of R.

By Tarski's fixed-point theorem, it follows that any *positive* equation system (set of equations) has a *unique* least fixed-point (and unique greatest fixed-point). That is, there is a unique least interpretation for the relation that satisfy the equations.

Algorithmic semantics of a fixed-point formula

Consider a system of equations (not necessarily positive): $Eq = \{R_1 = B_1, R_2 = B_2, \ldots R_k = B_k\}$.

Then we define a precise algorithmic computation of a particular relation, say R_1, as follows:

```
Evaluate(R,Eq) :-
  Let (R = B) occur in Eq;
  Set S := Emptyset;
  while (S does not stabilize) {
    -Eq' := Eq \ R = B;
    -Replace R in each expression of Eq' with S
      to get Eq'';
    -for every relation R_i that occurs in B {
      S_i := Evaluate(R_i,Eq');
     }
    -Substitute each R_i in B with S_i, replace
     R in B with S, and evaluate B to obtain
     the new value of S;
  }
  return S;
```

While the above may look daunting, it is actually quite intuitive. The algorithm above, in order to compute a relation R in an equa-

tion system with $R = B$ in it, essentially tries to find a fixed-point by starting with interpreting R as the empty set, and in each round replacing R with its current interpretation, evaluating the remaining equation system, and substituting those relations in B to obtain the next interpretation of R.

The Tarski-Knaster theorem says that the above iterative algorithm will always converge to the *least fixed-point* of the relations, when all the expressions are positive. In other words, there is a smallest interpretation for each of the R_i's that satisfies all the equations in the system, and the above algorithm computes it.

In this paper, we will confine ourselves to algorithms that can be described as a least fixed-point of a set of equations. While most of our algorithms will use only positive expressions, there are certain situations when we may want to use expressions negatively (for instance, to compute on a set smaller than the current set of reachable states, in order to speed up computation). When non-positive expressions are used, the above algorithm need not terminate (the interpretation for a relation may never stabilize), but we will ensure that the particular algorithms we write do indeed terminate.

Let's give an example of an algorithm written as a formula. Consider a transition system, where states are valuations over the Boolean variables $\bar{v} = v_1, \ldots v_k$, let the transition relation be $Trans(\bar{v}, \bar{v}')$ and let $Init(\bar{v})$ the set of initial states, both expresses as Boolean formulae.

Then the least fixed-point of the equation system with a single equation

$$Reach(\bar{u}) = Init(\bar{u}) \vee \exists \bar{x}.(Reach(\bar{x}) \wedge Trans(\bar{x}, \bar{u}))$$

is the set of *all* states reachable in the transition system. Also, the above formula gives a concrete way of computing it: we start with the $Reach$ set empty, and evaluate the expression, to get the initial states. We then substitute it back in the formula, and obtain the set of states reachable from the initial states in zero or one steps. We continue this till we reach a fixed-point, and stop.

The above formula is in fact all that we need to write an algorithm that symbolically model-checks a (non-recursive) transition system. We will have an underlying fixed-point calculus solver that will perform the iterations efficiently using BDDs, employing heuristics to order variables, evaluate sub-expressions, project away variables that are existentially/universally quantified, etc. The aim of this paper is to show that efficient model-checking algorithms for recursive sequential and concurrent programs can be obtained by simply writing such formulae.

4. The entry-forward summary-based algorithm

In this section, we show how to implement a symbolic reachability algorithm for recursive Boolean programs. Note that a recursive Boolean program has, in general, an infinite number of states it can reach (due to the unbounded stack). Traditional algorithms for reachability (which are sound and complete) compute the reachability sets using *summaries*.

Our aim in this section is to gently introduce the primary ideas in model-checking recursive programs, and encode the algorithms using fixed-point formulae. We will first recall the most simple of the sound and complete summary-based algorithms, and show how to express it as a fixed-point formula. This algorithm, however, is not efficient, as it can compute the reachable states using intermediate sets that are not reachable. As we progress, we will introduce better algorithms that are competitive with existing tools.

In order to describe the algorithms for solving reachability of Boolean programs, let us assume the following template formulae that describe the semantics of a program. Let us also assume that the *Goal* program counter that we test reachability for is in the main module (we can easily transform a general reachability problem to such a one).

Internal Transitions: Transitions that happen within a module (that is, those that are neither a call nor a return) are described by a formula $ProgramInt(\overline{u}, \overline{v})$. For instance, an assignment will compute the variables of the next state $\overline{v}$ depending on those of the current state $\overline{u}$, and update the program counter.

Transitions into a Call: Transitions that take the program from a call in a procedure to the beginning of the procedure being called are described by a formula $IntoCall(\overline{u}, \overline{v})$. This formula ensures that $\overline{v}.Global = \overline{u}.Global$ (global variables are preserved) and that $\overline{v}.Local$ initializes the local variables of the called module (in particular, it sets the value for the formal parameters).

Entry, Exit: Let us assume formulae $Entry(\overline{u}.pc)$ and $Exit(\overline{u}.pc)$ that capture the entry and exit labels of all procedures in the program. Entries correspond to labels of the first statement in all procedures. Exits are labels of return statements or after the last line of a procedure.

Initial states: The formula $Init(\overline{u}.pc)$ captures the constraints on the initial program counter, i.e. states that $\overline{u}.pc$ is the label that precedes the first statement of `main`.

Across a call: Let us assume a formula $Across(\overline{u}.pc, \overline{v}.pc)$ that captures the pairs of program counters denoting from where a call occurs to where the call returns to (both program counters in the same procedure).

Return from a call: Let us assume a formula $Return(\overline{u}, \overline{v}, \overline{w})$ that captures the update of variables according to the values computed in a procedure call. In this formula, $\overline{u}$ refers to the state at the procedure call, $\overline{v}$ to the state on exiting from the called procedure, and $\overline{w}$ to the state of the calling procedure on returning from the call. Formula *Return* ensures that $Across(\overline{u}.pc, \overline{w}.pc)$ holds, and also that $\overline{w}.Global$ matches $\overline{v}.Global$ and $\overline{w}.Local$ matches $\overline{u}.Local$ except for those variables that are assigned with return values (in particular, if the called procedure does not return values, we have $\overline{w}.Global = \overline{v}.Global$ and $\overline{w}.Local = \overline{u}.Local$).

4.1 A simple summary-based algorithm

We are now ready to write a summary-based least-fixed point definition of the reachable states according to a classical summary-based algorithm. A *summary* captures the set of all pair of states $(\overline{u}, \overline{v})$, where $\overline{u}$ is an entry to a procedure and $\overline{v}$ is a state of the same procedure that is reachable from $\overline{u}$.

We can easily write a recursive definition of the summary relation, using the following three clauses:

- If $\overline{u}$ is an entry, then $Summary(\overline{u}, \overline{u})$ holds.
- If $Summary(\overline{u}, \overline{x})$ holds and there is an internal move from $\overline{x}$ to $\overline{v}$, then $Summary(\overline{u}, \overline{v})$ holds.
- If $Summary(\overline{u}, \overline{x})$ holds, $\overline{x}$ is a call, $\overline{y}$ is the entry $\overline{x}$ leads to, $\overline{v}$ is a return matching the call $\overline{x}$ and further $Summary(\overline{y}, \overline{z})$ holds where $\overline{z}$ is an exit, then $Summary(\overline{u}, \overline{v})$ holds provided $\overline{v}$ matches $\overline{x}$ on the local variables and matches $\overline{z}$ on the global variables modulo the assignment of the returned values.

More succinctly, *Summary* is the *least fixed-point* of the following equation:

$$\begin{aligned} &Summary(\overline{u}, \overline{v}) = \\ &\quad (Entry(\overline{u}.pc) \wedge \overline{u} = \overline{v}) \\ &\vee (\exists \overline{x}.(Summary(\overline{u}, \overline{x}) \wedge ProgramInt(\overline{x}, \overline{v}))) \\ &\vee (\exists \overline{x}, \overline{y}, \overline{z}.(Summary(\overline{u}, \overline{x}) \wedge IntoCall(\overline{x}, \overline{y}) \wedge Summary(\overline{y}, \overline{z}) \\ &\qquad \wedge Exit(\overline{z}.pc) \wedge Return(\overline{x}, \overline{z}, \overline{v}))) \end{aligned}$$

THEOREM 1. *Let P be a program. Let $\overline{u}$ be an entry of procedure* `main`*. The least fixed-point of the equation for Summary given above relates $\overline{u}$ and a state $\overline{v}$ if and only if $\overline{v}$ is a state of* `main` *that is reachable in P.*

The above formula, surprisingly, can be immediately written using a mu-calculus formula, and using a tool like MUCKE, we would obtain a symbolic model-checker for recursive Boolean programs! The standard approach to evaluate a fixed-point formula, when applied to the above, will result in an algorithm that starts with summaries capturing all entries of all modules (whether they are reachable or not) and compute the set of all reachable states in each process, utilizing summaries of called procedures. Symbolic model-checking algorithms that search spaces that are not reachable often do much worse than those explore forward from the initial state, ensuring that only reachable states are discovered. We will modify the above scheme into one that computes only the reachable states.

4.2 The entry-forward summary-based algorithm

Let us now refine the scheme above so that, at any point of the computation of the set of reachable states, we keep only states that are reachable. In other words, if $\mathit{Summary}(\overline{u}, \overline{v})$ holds, we insist that $\overline{u}$ and $\overline{v}$ are reachable (with some stack content) from the entry of the `main`-procedure.

Intuitively, we start with the set of only the entry summary of the `main` procedure. At a call to a procedure, we allow the entry-summary of the called procedure to be discovered. Computation proceeds otherwise in a similar manner as above. Let us call this relation $\mathit{Summary}_{EF}$ (for "Entry-forward", as we are intuitively taking the entry forward to compute the summary).

$$
\begin{array}{l}
\mathit{Summary}_{EF}(\overline{u}, \overline{v}) = \\
\quad (\mathit{Entry}(\overline{u}.pc) \wedge \overline{u}{=}\overline{v} \wedge \mathit{Init}(\overline{u}.pc)) \\
\vee\ (\exists \overline{x}.(\mathit{Summary}_{EF}(\overline{u}, \overline{x}) \wedge \mathit{ProgramInt}(\overline{x}, \overline{v}))) \\
\vee\ (\exists \overline{x}, \overline{y}.(\mathit{Summary}_{EF}(\overline{x}, \overline{y}) \wedge \mathit{IntoCall}(\overline{y}, \overline{u}) \wedge \overline{u}{=}\overline{v})) \\
\vee (\exists \overline{x}, \overline{y}, \overline{z}.(\mathit{Summary}_{EF}(\overline{u}, \overline{x}) \wedge \mathit{IntoCall}(\overline{x}, \overline{y}) \wedge \mathit{Summary}_{EF}(\overline{y}, \overline{z}) \\
\qquad \wedge \mathit{Exit}(\overline{z}.pc) \wedge \mathit{Return}(\overline{x}, \overline{z}, \overline{v}))\ )
\end{array}
$$

The third clause above, which has been added new, generates the pair of identical entries of a module that is reachable. It is easy to see that $\mathit{Summary}_{EF}$ computes reachability, and any two states related by summary must be individually reachable from the initial state of the program.

The above formula (algorithm) can be optimized further. Notice that the fixed-point computation will, in each round, evaluate the formula inside out, representing each set using a BDD. Now let us look closely at the last disjunctive clause which computes the discovery of a return ($\mathit{Summary}_{EF}(\bar{u}, \bar{v})$) to a module, from the caller ($\mathit{Summary}_{EF}(\bar{u}, \bar{x})$) and a summary of a called procedure ($\mathit{Summary}_{EF}(\bar{y}, \bar{z})$). This clause has two $\mathit{Summary}_{EF}$ relations that are combined conjunctively, and given that this set gets very large, and the conjunction of two BDDs takes time product of their sizes, this is a serious bottle-neck in the computation.

We aim to rewrite this clause such that the two summary sets are first combined with other sets with a presumably small BDD representation and then their conjunction is computed. We start by arranging the constraints into two groups: (A) the constraints involving the caller ($\bar{x}$) and the return ($\bar{v}$), and (B) the constraints involving the exit ($\bar{z}$), the entry ($\bar{y}$) and the return ($\bar{v}$). For this purpose, we split the predicate *Return* into two parts, called respectively $\mathit{ReturnA}$ and $\mathit{ReturnB}$, each restricted to state constraints only on a subset of the variables.

Recall that formula $\mathit{IntoCall}(\overline{x}, \overline{y})$ captures the transitions from a call $\overline{x}$ to an entry $\overline{y}$. In particular, it ensures $\overline{x}.\mathit{Global} = \overline{y}.\mathit{Global}$ and that $\overline{y}.\mathit{Local}$ is consistent with the parameters passed to the module of $\overline{y}$ when called from $\overline{x}$. Thus, to express such constraints, $\overline{y}.pc$ is not really needed provided that we can refer to the module of $\overline{y}$. Therefore, instead of $\mathit{IntoCall}(\overline{x}, \overline{y})$ we can use an equivalent formula $\mathit{IntoCall1}(\overline{x}, pc, \overline{y}.\mathit{Local}, \overline{y}.\mathit{Global})$ where pc is a program counter of the same module as $\overline{y}.pc$.

Thus, we quantify over $\bar{x}$ the constraints (A), and extract the variables in $\bar{x}$ that are required in (B). Then, we quantify over $\bar{y}$ and $\bar{z}$ the constraints (B) and extract the variables of $\bar{y}$ and $\bar{z}$ which are in use in (A).

Rewriting the algorithm as a formula, we replace the last disjunctive clause of the above formula with:

$$
\begin{array}{l}
\vee\ (\exists\ \mathit{xpc},\ \ \mathit{zpc},\ \ \mathit{yLocal},\ \ \mathit{yGlobal}. \\
\quad (\exists \overline{x}.(\mathit{Summary}_{EF}(\overline{u}, \overline{x}) \wedge \mathit{ReturnA}(\overline{x}, \mathit{zpc}, \overline{v}) \\
\qquad \wedge\ \mathit{IntoCall1}(\overline{x}, \mathit{zpc}, \mathit{yLocal}, \mathit{yGlobal}) \wedge \mathit{xpc} = \overline{x}.pc)\ ) \\
\quad \wedge\ (\exists \overline{y}, \overline{z}.(\mathit{Summary}_{EF}(\overline{y}, \overline{z}) \\
\qquad \wedge\ (\mathit{yGlobal} = \overline{y}.\mathit{Global} \wedge \mathit{yLocal} = \overline{y}.\mathit{Local}) \\
\qquad \wedge\ (\mathit{zpc} = \overline{z}.pc \wedge \mathit{Exit}(\overline{z}.pc)) \\
\qquad \wedge\ \mathit{ReturnB}(\overline{y}, \overline{z}, \mathit{xpc}, \overline{v}))) \\
\ )
\end{array}
$$

The above formula, except for some implementation details, is exactly the formula for one of the algorithms that we report in the implementation (see the Appendix for the exact implemented formula).

The following theorem holds.

THEOREM 2. *Let P be a program. The least fixed-point of the equation for $\mathit{Summary}_{EF}$ given above relates a state $\overline{u}$ and a state $\overline{v}$ if and only if $\overline{u}$ is an entry of a procedure that is reachable in P and $\overline{v}$ is a state in the same procedure reachable from the entry $\overline{u}$ using a run that returns from all calls. In particular, $\mathit{Summary}_{EF}(\overline{u}, \overline{v})$ holds when $\overline{u}$ is an entry of* `main` *iff $\overline{v}$ is a state of* `main` *that is reachable in P.*

4.3 An optimized entry-forward algorithm

We now present a high-level description of an optimization of the entry-forward algorithm. This algorithm involves several issues that we will highlight.

We compute the set $\mathit{Summary}_{EFopt}$ as described by the following iterative algorithm. $\mathit{Summary}_{EFopt}$ is initialized with the initial configurations. Now, let X be the configurations discovered in the last iteration, and *Relevant* be set of all $\mathit{Summary}_{EFopt}$ configurations that have as a program counter the program counter of a configuration of X. We define *New*1 to be the the image-closure of *Relevant* on internal transitions, and define *New*2 as the image of *Relevant* on transitions that call a module or skip a called module using a summary. Thus, $\mathit{Summary}_{EFopt}$ is the union of $\mathit{Summary}_{EFopt}$ in the previous round, *New*1, and *New*2. The algorithm terminates when $\mathit{Summary}_{EFopt}$ stabilizes.

Two aspects are relevant in the algorithm described above. Typically, programs exhibit many more internal transitions than calls and/or returns, and the computation of calls and returns are quite expensive compared to internal transitions— a call or a return involves at least two configurations of different modules and the parameter/return values are typically set in a non-trivial way. Thus, the first idea is to compute, in one iteration, only one set of new calls and returns, but compute the internal moves *to completion* (ignoring calls and the returns that may become active). Another important intuition is that we would compute only starting from configurations that are newly discovered in the previous round. Unfortunately, the BDD representing such a set can be quite complex and completely different from the one for $\mathit{Summary}_{EFopt}$. A better solution is to take the configurations to process by considering only the pc. Thus, at a given iteration, a configuration is processed if its program counter is the same as that of a state newly discovered in

the previous iteration. Usually the BDD configuration for it is less complex of the one for $Summary_{EFopt}$, resulting in faster computation.

In the following equations, we use a new bit fr; $fr=1$ is used to mark each pair of states $(\overline{u},\overline{v})$ which is added to $Summary_{EFopt}$, and $fr=0$ to mark all such pairs that have been added in any of the previous iterations but the last one. According to this marking, we can determine the set of pairs $(\overline{u},\overline{v})$ which have been added to $Summary_{EFopt}$ for the first time in the last iteration— the tuple $(1,\overline{u},\overline{v})$ must be in $Summary_{EFopt}$ while the tuple $(0,\overline{u},\overline{v})$ should not. This is used to compute the relevant set of program counters *Relevant* which the next iteration works on. Formally, $Summary_{EFopt}$ is defined as the *least fixed-point* of the relation $Summary_{EFopt}(fr,\overline{u},\overline{v})$ in the system of equations below:

$$\begin{aligned}
&Summary_{EFopt}(fr,\overline{u},\overline{v}) = \\
&\quad (fr{=}1 \wedge Entry(\overline{u}.pc) \wedge \overline{u}{=}\overline{v} \wedge Init(\overline{u}.pc)) && [1]\\
&\quad \vee Summary_{EFopt}(1,\overline{u},\overline{v}) && [2]\\
&\quad \vee (fr{=}1 \wedge (New1(\overline{u},\overline{v}) \vee New2(\overline{u},\overline{v}))\ ) && [3]
\end{aligned}$$

$$\begin{aligned}
Relevant(pc) = \exists \overline{u},\overline{v}.(&Summary_{EFopt}(1,\overline{u},\overline{v}) \\
&\wedge \neg Summary_{EFopt}(0,\overline{u},\overline{v}) \\
&\wedge\ \overline{v}.pc = pc) && [4]
\end{aligned}$$

$$\begin{aligned}
&New1(\overline{u},\overline{v}) = \\
&\quad (Summary_{EFopt}(1,\overline{u},\overline{v}) \wedge Relevant(\overline{v}.pc)) && [5]\\
&\qquad \vee (\exists \overline{x}.(New1(\overline{u},\overline{x}) \wedge ProgramInt(\overline{x},\overline{v}))) && [6]
\end{aligned}$$

$$\begin{aligned}
&New2(\overline{u},\overline{v}) = \\
&\quad (\exists \overline{x}.(Relevant(\overline{x}.pc) \wedge Summary_{EFopt}(1,\overline{u},\overline{x}) \\
&\qquad \wedge IntoCall(\overline{x},\overline{v}))) && [7]\\
&\qquad \vee (\exists \overline{x},\overline{y},\overline{z}.(Summary_{EFopt}(\overline{u},\overline{x}) && [8]\\
&\qquad\quad \wedge IntoCall(\overline{x},\overline{y}) \wedge Summary_{EFopt}(\overline{y},\overline{z}) && [9]\\
&\qquad\quad \wedge Exit(\overline{z}.pc) \wedge Return(\overline{x},\overline{z},\overline{v}) && [10]\\
&\qquad\quad \wedge (Relevant(\overline{x}.pc) \vee Relevant(\overline{z}.pc)))) && [11]
\end{aligned}$$

All the above equations are interpreted as least-fixed-point equations, with the main computation being that of $Summary_{EFopt}$ (note that this is important as some of the equations, such as *Relevant* employs *negation*— see Section 3 for the precise algorithmic semantics of evaluation).

By clause 1, the initial states are added to the current set in each iteration. Note that they will contribute to determining the set of "relevant" program counters only once (when they are first inserted). By clause 2, for each tuple $(1,\overline{u},\overline{v})$, the tuple $(0,\overline{u},\overline{v})$ is added to $Summary_{EFopt}$, i.e., all the pairs $(\overline{u},\overline{v})$ already discovered so far are marked as not newly discovered. Clause 3 adds to $Summary_{EFopt}$ the tuples belonging to either one of the sets $New1$ and $New2$, and mark them with $fr=1$. Equation 4 computes the set *Relevant* of all program counters pc such that there are pairs of states $(\overline{u},\overline{v})$ in $Summary_{EFopt}$ which are marked with $fr=1$ but not $fr=0$, and pc is the program counter of the newly discovered states $\overline{v}$. In the equation defining $New1$, clause 5 adds to $New1$ each pair $(\overline{u},\overline{v})$ which has been already discovered and such that $\overline{v}.pc$ is in the set *Relevant*, and clause 6 allows us to discover new states by taking internal transitions. The least fixed-point of $New1$ computes the set of all pairs $(\overline{u},\overline{v})$ such that v can be reached within any number of internal transitions from a state w such that $\overline{w}.pc$ is in *Relevant* and $(1,\overline{u},\overline{w})$ is in $Summary_{EFopt}$. The above equation for $New2$ defines $New2$ as the set of all pairs $(\overline{u},\overline{v})$ such that $\overline{v}$ is either reachable by a call from a reachable relevant state (clause 7) or by a transition across a call to a module (clauses 8–11). Notice that for the correctness of the algorithm in this second case it is crucial to require just that for either one among the caller state and the exit state involved in the transition to be in the relevant set, and not both (clause 11). In fact, such caller and exit states could be discovered in different times and thus it might be the case that they never happen to be in the relevant set at the same time.

The following that captures the correctness of the above fixed-point algorithm:

THEOREM 3. *Let P be a program. Consider the least fixed-point of the equation for $Summary_{EFopt}$ given above. Then $Summary_{EFopt}(1,\overline{u},\overline{v})$ holds if and only if $\overline{u}$ is an entry of a procedure that is reachable in P and $\overline{v}$ is a state in the same procedure reachable from the entry $\overline{u}$ using a run that returns from all calls. In particular, $Summary_{EFopt}(1,\overline{u},\overline{v})$ holds when $\overline{u}$ is an entry of* `main` *iff $\overline{v}$ is a state of* `main` *that is reachable in P.*

The above algorithm is similar to *frontier-set simplification* or *incrementalization* techniques often implemented in solvers. However, we emphasize that they are not the same. In frontier-set simplification, the precise set of states discovered in the previous round are used to compute the image. However, in many situations, the BDD for the frontier-set can be much larger than the BDD for all reachable states. In our algorithm, we wish to take the set of all reachable nodes whose pc-value was part of some state on the frontier-set. This is a restriction of the reachable set to a particular set of program-counter values, and hence does not blow up in practice.

5. Bounded-context reachability in concurrent programs

A *concurrent Boolean program* is a set of boolean recursive programs running in parallel and sharing some (global) variables. Formally, the syntax of concurrent boolean programs is defined by extending the syntax of boolean recursive programs (Section 2) with the following rules:

$$\begin{aligned}
\langle conc\text{-}pgm\rangle &::= \langle gvar\text{-}decl\rangle\ \langle pgm\text{-}list\rangle\\
\langle pgm\text{-}list\rangle &::= \langle pgm\rangle\ \langle pgm\text{-}list\rangle \mid \langle pgm\rangle
\end{aligned}$$

Let $\mathcal{P}$ be a concurrent program formed by the sequential programs $P_1,\ldots,P_n$ (where $n>0$). Each program P_i has its own global and local variables, and can access also to other variables which are shared with the other component programs. We denote with S the set of the shared variables and with V_S the set of their valuations.

We give the semantics of $\mathcal{P}$ using interleaved the behaviors of $P_1,\ldots,P_n$. At any point of a computation, only one of the programs is *active*. Therefore, a *configuration* of $\mathcal{P}$ is denoted by a tuple $(i,u_S,\overline{u}_1,\ldots,\overline{u}_n)$ where P_i is the currently active program, $u_S \in V_S$ and $\overline{u}_j$ is a configuration of P_j (valuation of global and local variables and stack) for $j=1,\ldots,n$. From the configuration $(i,u_S,\overline{u}_1,\ldots,\overline{u}_n)$ the computation of $\mathcal{P}$ can evolve either according to the local behavior of P_i or by switching to another program P_j, which then becomes the new active program. The consecutive portion of a run that executing only program P_i is called a *context*.

Given a program counter pc (of any component program), the reachability problem for concurrent programs asks whether a state containing pc can be reached in a computation of $\mathcal{P}$. It is well-known that the reachability problem for concurrent recursive programs is undecidable. For model-checking such programs several authors have considered a simpler reachability problem, the *bounded context-switch reachability*. Given a program counter pc and an integer k, the k-bounded context-switch reachability problem asks whether a state containing pc can be reached in a computation where the active program has changed at most k times (i.e.,

at most k context-switches happened). In the following, we refer to a component program of a concurrent program also as a thread.

5.1 An algorithm for bounded context-switching reachability in concurrent programs

Our fixed-point algorithm for computing the set of states reachable by a concurrent Boolean program in k context-switches is a bit involved— we will give here the main intuition of the construction of the fixed-point equations, and skip formal proofs of correctness.

Fix a bound k on the number of context-switches and fix n the number of threads in the program. Our fixed-point formula is parameterized over k and n. Assume, for simplicity, that each thread of the concurrent program is over the same set of local and global variables, and further, that all global variables are shared by the threads. Let L denote the local variables and G the global (and shared) variables.

We compute a predicate $\mathit{Reach}(\overline{u}, \overline{v}, ecs, cs, \{g_i\}_{i=1}^{k}, \{t_i\}_{i=0}^{k})$, where both $\overline{u}$ and $\overline{v}$ represent valuations of $L \cup G$, cs and ecs are variables ranging over $\{0, 1, \ldots, k\}$, each g_i is a valuation of the global shared variables G, and each t_i is a variable over $\{1, \ldots, n\}$. Intuitively, the pair $(\overline{u}, \overline{v})$ captures a summary relation of *one* particular thread, where $\overline{u}$ represents the point of entry into the current procedure of the thread and $\overline{v}$ represents the current state of the thread. The variable cs denotes the current context-switch number (i.e. the number of context-switches that have taken place) while ecs represents the number of context-switches that had happened at the point of entry into the current procedure (and hence $ecs \leq cs$). The variable t_i ($i = 0, \ldots cs$) represents the thread that is active at the i'th context. The variable g_i represents the valuation of the shared global variables at the point at which the i'th context-switch happened.

More formally, $\mathit{Reach}(\overline{u}, \overline{v}, ecs, cs, \{g_i\}_{i=1}^{k}, \{t_i\}_{i=0}^{k})$ holds iff there is a global execution ρ that switches contexts cs times, the i'th context-switch happening at global valuation g_i (for each $i \in \{1, \ldots, cs\}$), and reaches a global state where thread $T = t_{cs}$ is in the state $\overline{v}$, and furthermore, the valuation of the global variables at the entry to the procedure in thread T was $\overline{u}.\mathit{Global}$, and the number of context-switches done before the last entry to the procedure was ecs. This is the precise semantics of *Reach*, and is essential to understand the fixed-point formulation that we provide below. Notice that the values of g_j and t_j, where $j > cs$, are not relevant at all, and do not contribute to the semantics of *Reach*.

Note that a program counter pc is reachable by the concurrent program within k context-switches iff $\mathit{Reach}(\overline{u}, \overline{v}, cs, \{g_i\}_{i=1}^{k}, \{t_i\}_{i=0}^{k})$ holds for some tuple with $\overline{v}.pc = pc$. Hence computation of *Reach* suffices to solve the reachability problem.

One property of our formulation of the fixed-point greatly simplifies understanding it and hence is worth noting. When inferring that $\mathit{Reach}(\overline{u}, \overline{v}, ecs, cs, \{g_i\}_{i=1}^{k}, \{t_i\}_{i=0}^{k})$ holds, we will use the fact that other tuples of the form $\mathit{Reach}(\overline{u'}, \overline{v'}, ecs', cs', \{g'_i\}_{i=1}^{k}, \{t'_i\}_{i=0}^{k})$ hold. However, these tuples will *always* be such that $g'_i = g_i$ and $t'_i = t_i$. In other words, when we infer tuples in *Reach* using other tuples, we will ensure that the g_i and t_i tuples are precisely the same, for all values of i. Furthermore, $cs' \leq cs$ also will always hold.

We are now ready to define the fixed-point. In the following, let $\overline{g}$ denote $\{g_i\}_{i=1}^{k}$ and let $\overline{t}$ denote $\{t_i\}_{i=0}^{k}$.

$$\mathit{Reach}(\overline{u}, \overline{v}, ecs, cs, \overline{g}, \overline{t}) = \varphi_{init} \vee \varphi_{int} \vee \varphi_{call} \vee \varphi_{ret} \vee \varphi_{1st\text{–}switch} \vee \varphi_{switch}$$

where the sub-formulae are defined as follows:

[Initial tuples:]

$$\varphi_{init} = (cs{=}\, ecs{=}0 \wedge \mathit{Entry}(\overline{u}.pc) \wedge \overline{u}{=}\overline{v} \wedge \mathit{Init}(t_0, \overline{u}.pc)\ )$$

This is similar to the initial clause we had for sequential summaries, except that we set the number of context-switches that have happened to 0 and make sure $\overline{u}.pc$ is the initial program-counter for thread t_0, the thread active in the first context.

[Internal transitions:]

$$\varphi_{int} = \exists \overline{x}.\ \big(\mathit{Reach}(\overline{u}, \overline{x}, ecs, cs, \overline{g}, \overline{t}) \wedge \mathit{ProgramInt}(\overline{x}, \overline{v})\big)$$

[Call transitions:]

$$\varphi_{call} = \exists \overline{x}, \overline{y}, ecs'.(\ \mathit{Reach}(\overline{x}, \overline{y}, ecs', cs, \overline{g}, \overline{t}) \wedge \mathit{IntoCall}(\overline{y}, \overline{u}) \wedge ecs = cs \wedge \overline{u} = \overline{v}\ )$$

[Return transitions:]

$$\varphi_{ret} = \exists \overline{x}, \overline{y}, \overline{z}, cs'.(\ \mathit{Reach}(\overline{u}, \overline{x}, ecs, cs', \overline{g}, \overline{t}) \wedge \mathit{IntoCall}(\overline{x}, \overline{y}) \wedge \mathit{Reach}(\overline{y}, \overline{z}, cs', cs, \overline{g}, \overline{t}) \wedge \mathit{Exit}(\overline{z}.pc) \wedge \mathit{Return}(\overline{x}, \overline{z}, \overline{v}) \wedge cs' \leq cs)$$

The above three clauses explore states reached using internal transitions, call transitions, and return transitions of the current thread, and are very similar to the corresponding clauses we had for sequential summaries. Note that taking these transitions preserves the value of cs.

However, there are some subtleties to note regarding the formula for return transitions. The state of the caller of the current module, $\overline{x}$, could have been reached using a different number of context-switches cs'; but we must then insist that cs' is not larger than cs.

The soundness argument for this last rule is subtle. Since $\mathit{Reach}(\overline{u}, \overline{x}, ecs, cs', \overline{g}, \overline{t})$, there is a run ρ that reaches $\overline{x}$ using cs' context-switches. From the fact that $\mathit{Reach}(\overline{y}, \overline{z}, cs', cs, \overline{g}, \overline{t})$ holds, it follows that there is a run ρ' that reaches $\overline{z}$ using cs context-switches. The crucial argument is that we can construct from ρ and ρ' a run that reaches $\overline{v}$ using cs context-switches. This run ρ'' is obtained as follows: from ρ, we take the portions of the run in the current thread that reaches the caller $\overline{x}$ in cs' context-switches, and stitch it with the local run of the current thread in ρ' within the current procedure from entry $\overline{y}$ to exit $\overline{z}$, and then proceed to $\overline{v}$. Moreover, the runs in the *other threads* are obtained by using appropriate portions in the run ρ'. The fact that this results in a valid global run ρ'' depends crucially on the fact that the two *Reach*-tuples are over the same vectors $\overline{g}$ and $\overline{t}$. In particular, the local run which we pull out from ρ assumes context-switches to other threads that result in changes to global variables, and cannot be invalidated when we substitute the runs in other threads using different runs (from ρ') that result in the same changes to global variables.

[Context-switching to a thread for the first time:]

$$\varphi_{1st\text{–}switch} = \exists \overline{x}, \overline{y}, cs', ecs'.\ (\ \mathit{Reach}(\overline{x}, \overline{y}, ecs', cs', \overline{g}, \overline{t}) \wedge (cs = cs' + 1) \wedge \mathit{First}(t_{cs}, cs, \overline{t}) \wedge (\overline{v}.\mathit{Global} = g_{cs} = \overline{y}.\mathit{Global}) \wedge (\overline{u} = \overline{v}) \wedge (ecs = cs) \wedge \mathit{Init}(t_{cs}, \overline{v}.pc)\ )$$

In the above, $\mathit{First}(t, s, \overline{t})$ is a predicate that is true iff $t_s = t$ and there is no $r < s$ such that $t_r = t$. In other words, $\mathit{First}(t_{cs}, cs, \overline{t})$ checks whether cs is the first context the thread we are switching to (t_{cs}) is active in.

This clause deals with the first time we switch to a thread, and allows a tuple to be added to *Reach* provided there was some state reachable in a different thread with the same global variables and one less context-switch; we record the globals in g_{cs} (i.e., the component of $\overline{g}$ at index cs).

[Context-switching back to a thread:]

$$\varphi_{switch} =$$

$(\exists \overline{x}, \overline{y}, cs', ecs'.\ (\mathit{Reach}(\overline{x}, \overline{y}, ecs', cs', \overline{g}, \overline{t}) \wedge (cs = cs' + 1) \wedge$

$\neg \mathit{First}(t_{cs}, cs, \overline{t}) \wedge (\overline{v}.\mathit{Global} = g_{cs} = \overline{y}.\mathit{Global})\ )\ )$

$\wedge$

$(\exists \overline{v'}, cs''.(\mathit{Reach}(\overline{u}, \overline{v'}, ecs, cs'', \overline{g}, \overline{t}) \wedge (cs'' < cs) \wedge$

$\mathit{Consecutive}(cs'', cs, \overline{t}) \wedge \overline{v}.\mathit{Local} = \overline{v'}.\mathit{Local})\ )$

This case handles the scenario when we switch to a thread we had executed previously at some point.

The first conjunct above checks whether a state is reachable in another thread, and imbibes the global valuation from there (similar to the previous case). The second conjunct is more involved as it retrieves the local valuation of the thread we are context-switching to.

The predicate $\mathit{Consecutive}(r, s, \overline{t})$ checks whether $t_s = t_r$ and further that there is no $r < i < s$ such that $t_s = t_i$. Hence $\mathit{Consecutive}(cs'', cs, \overline{t})$ checks whether cs'' was the last context the thread we are switching to was last active in. We recover the local state of the thread by checking for a compatible tuple in *Reach* with cs'' context-switches.

The soundness of this rule also involves stitching runs. Given two runs that witness the two tuples in *Reach* in the above clause, we can stitch them into a single run that witnesses the new tuple we are adding to *Reach*. The crucial condition that helps in showing this is again that the runs in other threads can be freely substituted using runs that effect the same change in global variables.

We can now check whether *pc* is reachable using the following predicate:

$\mathit{Reachable}(pc) = \exists \overline{u}, \overline{v}, ecs, cs, \overline{g}, \overline{t}.$

$(\ \mathit{Reach}(\overline{u}, \overline{v}, ecs, cs, \overline{g}, \overline{t}) \wedge \overline{v}.pc = pc\ )$

The following theorem states the correctness of the above fixed-point formulation:

THEOREM 4. *Fix an integer $k > 0$. Let $\mathcal{P}$ be a concurrent Boolean program with n threads, and let Goal be a program counter of $\mathcal{P}$. Let Reach and Reachable be the predicates as defined above. Then a state with program counter Goal is reachable in $\mathcal{P}$ within k context-switches iff $\mathit{Reachable}(\mathit{Goal})$ holds.*

One of the salient features of the above formulation is the economic use of copies of global variables. A natural formulation would keep track of the $\overline{g}$ and $\overline{t}$ vectors at the entry to a called procedure; however, since these vectors never really change, we can do away with keeping these vectors at the entry to a procedure. In other words, summaries do not involve *two* copies of the k-tuple of global variables. Hence the number of global variables we use in the fixed-point formulation in $k \cdot |G| + 2|G|$ only (we can in fact reduce this even to $k \cdot |G| + |G|$ by getting rid of the $\overline{v}.\mathit{Global}$ variables, and instead use g_{cs}). Previous formulations of capturing the reachable states using fixed points required more copies of global variables [12].

A nice feature of the formulation presented in [12] is that the authors show how to handle k *round-robin schedules* (not context-switches) using $O(k)$ copies of global variables only. We have extended our formulation above to rounds as well, and can capture a k round-robin schedule using $2k$ copies of global variables. However, this is considerably more complex, and is out of the scope of this paper. We do emphasize that this formulation too can be captured using our fixed-point calculus.

6. Implementation and experiments

In this section, we describe the implementation of the ideas presented in this paper in a new tool GETAFIX ("get-a-fix using fixed-points"), and is available at the website below:

`http://www.cs.uiuc.edu/~madhu/getafix` .

6.1 GETAFIX on sequential recursive programs

GETAFIX is a BDD-based symbolic verifier for Boolean programs which answers reachability queries. The tool takes as an input a Boolean program and a statement label, and gives as an output a YES/NO answer to the question "is the statement at the input label reachable in the input program?". The answer can be computed using one of the algorithms presented in the previous sections, which have all been implemented in the tool. The model-checking engine of the tool is the mu-calculus symbolic model checker MUCKE [7]: the input program and the reachability algorithm are all translated into a mu-calculus formula which is fed to MUCKE for evaluation.

The high level architecture of the tool is shown in Figure 1. GETAFIX takes the input Boolean program and a target program label, and generates Boolean formulae (without fixed-points) that describe the various predicates in the Boolean program. These are the template formulae described in Section 4, and includes predicates that capture the internal transition relation, the transition relation on calls, etc. These templates are succinctly described as Boolean formulae. GETAFIX also computes a fairly simple BDD-ordering suggestion to MUCKE, encoding it as allocation constraints. This BDD ordering is based on a simple algorithm which looks at the assignments in the program, and tries to allocate the variables involved in the assignment together. This algorithm is essentially the same algorithm followed by both MOPED VERSION 1 and BEBOP (the Boolean program checker used in SLAM).

The formulae resulting from the translation step has a first part that concerns the encoding of the input program and a second part that concerns the encoding of the chosen reachability algorithm. The two parts have a clear interface represented by the formulae *ProgramInt*, *IntoCall*, *Entry*, *Exit*, *Init*, *Across* and *Return* that have been described in Section 4. Therefore, each of the parts can be implemented independently of the other as long as the program translation will define those formulae and the algorithm will use only those abstractions of the program.

The table in Figure 2 summarizes our experimental results. For each analyzed program, we report the number of non-blank lines of code (LOC), the maximal number of return values and input parameters in any procedure in the program, the number of global variables, the total number of local variables, the maximal number of local variables in any procedure, and the number of procedures in the program. For the GETAFIX tool, we report the final BDD size of the summary set, and the time taken to compute the answer, in seconds, for both the entry-forward algorithm and the optimized entry-forward algorithm. Note that the final BDD size is the same for both algorithms. We also report the time MOPED VERSION 1, MOPED VERSION 2 and BEBOP took to answer the same queries on the same machine. As regards to BEBOP, on each suite we have tried different options which enable/disable counter-example generation and switch on/off optimizations; we report in the table the *best time* (minimum time) taken by BEBOP for each experiment. The programs were all in compatible syntax so that they could be input to any of the tools. For aggregated suites, the time and space reported are the average of the times and spaces for the suite. We took care not to combine examples with very different complexities into a sub-suite.

The first suite is the set of Regression examples; the positive ones are those where the target label is reachable, and the negative ones are where it is unreachable. The Regression suite consists of 177 small programs meant to test SLAM on correctness of abstraction of language features.

The next suite is a set of drivers from SLAM benchmarks, which we obtained from Stefan Schwoon. Each sub-suite contains several Boolean programs. For all these suites, we report average statistics.

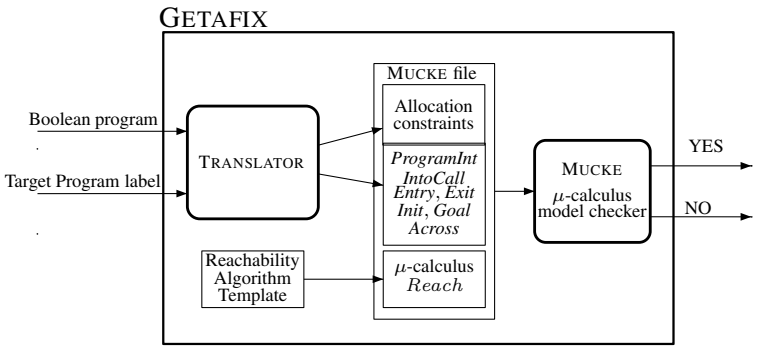

Figure 1. Architecture of the GETAFIX tool

	# LOC	#ret	#param.	#globals (g)	#locals	max locals per proc. (l)	# procedures	Reach?	GETAFIX #Nodes in BDD	GETAFIX Time (s) EF	GETAFIX Time (s) EF opt	MOPED 1 Time (s)	MOPED 2 Time (s)	BEBOP Time (s)
REGRESSION														
Positive (99 programs)	368	1.8	2.3	0.81	4.8	3.2	6.9	Yes	240	1	1	1	1	1
Negative (79 programs)	224	1.9	2.5	2	8.2	3.6	6.5	No	625	1	1	1	1	1
SLAM														
Driver iscsiprt (positive) (15 programs)	10K	10	9	3.5	211	11.8	148	Yes	8K	2	2	1	1	1
Driver floppy (positive) (12 programs)	17K	15	12	5.3	172.5	16	103	Yes	9K	2	2	1	1	1
Driver (negative) (4 programs)	10K	11	8.25	10.7	154.2	12	116.2	No	17K	2	2	1	1	1
iscsi (positive) (16 programs)	12K	18	18	17.1	358.8	18	158	Yes	81K	5	5	2	4	6
TERMINATOR														
Terminator-A (iterative)	284	18	12	9	97	68	4	Yes	141K	4	3	-	1	4
(schoose)	284	18	12	9	97	68	4	Yes	162K	3	3	14	1	3
Terminator-B (iterative)	471	9	7	14	77	42	4	No	997K	72	12	-	-	-
(schoose)	471	9	7	14	77	42	4	No	660K	32	9	24	490	953
Terminator-C (iterative)	908	1	0	23	19	19	4	No	48K	2	3	1	1	711
(schoose)	908	1	0	23	19	19	4	No	48K	2	3	1	1	709

Figure 2. Experimental results: "–" denotes time-out in 30 minutes

Finally, the last suite contains three programs generated by the TERMINATOR tool [10] and provided to us by Byron Cook.

First, note that the entry-forward optimized algorithm outperforms the entry-forward algorithm, essentially, overall.

In SLAM driver suites, notice that GETAFIX takes a little longer than MOPED, and except for the ISCSI suite, also than BEBOP. This is primarily due to an overhead in using MUCKE— MUCKE first computes allocations of BDD-variables using an algorithm which actually takes a few seconds. These examples are extremely large in code-length (not particularly complex in the set of reachable states). Profiling the tool, we found that the actual image computation time is less than a second on these examples in GETAFIX.

Let us turn to the TERMINATOR benchmarks, which are actually quite complex and with large BDDs representing reachable states. The benchmarks we obtained had a *dead* statement that currently the tools MOPED and GETAFIX do not support. A declaration of a set of variables as dead essentially means that the variables will no longer be used in the computation of a module. We modeled the *dead* statement in two ways— one way was to iteratively assign the variable to a nondeterministic value using conditional *if-then-else* statements (marked 'iterative' in the table), and the other was to replace them using an assignment called *schoose* which can assign non-deterministic values (marked 'schoose' in the table). Notice that Moped does not terminate on two of the three examples, in the iterative version. We cannot believe that it is actually experiencing difficulty in solving the problem; we suspect this may just be an error in Moped. Also, these examples are quite challenging for BEBOP. On TERMINATOR-B and TERMINATOR-C it takes several minutes and on the iterative version of TERMINATOR-B it did not terminate in 30 minutes. In all the TERMINATOR examples GETAFIX terminates in less than 30 seconds.

In summary, the experiments clearly show that GETAFIX is competitive against well-tuned mature tools. Note that this is despite the fact that GETAFIX lacks several other low-level heuristic tricks built into other tools, such as static analysis to eliminate dead code, etc.

Context switches	Reachable	Max reach set size	Time sec.
Two processes: one adder and one stopper (12 local variables and 8 global variables)			
1	No	0.6k	5.4
2	No	1.5k	5.5
3	No	5.5k	5.7
4	No	7.0k	5.8
5	No	13.3k	6.3
6	No	23.2k	7.3
Three processes: one adder and two stoppers (18 local variables and 8 global variables)			
1	No	0.7k	5.5
2	No	2.9k	5.5
3	Yes	9.9k	5.7
4	Yes	34.6k	6.8
5	Yes	115.0k	9.2
6	Yes	370.5k	17.4
Three processes: two adders and one stopper (18 local variables and 8 global variables)			
1	No	0.7k	5.4
2	No	2.6k	5.6
3	No	10.4k	6.0
4	Yes	41.1k	7.6
5	Yes	90.9k	9.1
6	Yes	250.1k	9.1
Four processes: two adders and two stoppers (24 local variables and 8 global variables)			
1	No	0.8k	5.5
2	No	3.6k	5.6
3	Yes	15.8k	5.9
4	Yes	67.8k	7.6
5	Yes	296.9k	13.0
6	Yes	1227.4k	57.1

Figure 3. Experimental results on the Bluetooth driver example

6.2 GETAFIX on concurrent programs

The table in Figure 3 resumes the experiments we have run on our tool to test a Boolean model of Windows NT Bluetooth driver [17]. Briefly, this driver has two types of threads, *stoppers* and *adders*. A stopper calls a stopping procedure to halt the driver, while an adder calls a procedure to perform I/O in the driver. We have considered four different configurations: an adder and a stopper, two adders and a stopper, an adder and two stoppers, and two adders and two stoppers. For each of this configurations we report the results allowing up to six context switches. Except for the configuration with only two programs, GETAFIX discovered a bug using at least three context switches when using two stoppers and at least four context switches when using only one stopper. The execution times shown in Figure 3 are competitive with those obtained by other tools on which this driver was tested [22, 12].

7. Conclusions

We have implemented in the tool GETAFIX two classes of algorithms that solve respectively reachability in Boolean programs and bounded context-switch reachability in concurrent Boolean programs, using formulae in a fixed-point calculus. The experimental results presented in this paper, and the ease of building algorithms using the fixed-point calculus, encourage us also to extend our tool to solve static-analysis problems in program analysis.

GETAFIX uses MUCKE a symbolic fixed-point solver to evaluate formulae. While we have used only alternation-free fixed-point calculus formulae in this work (as we were interested in reachability), our formalism can easily be extended to arbitrary mu-calculus specifications. In fact, it is well-known that any mu-calculus specification on pushdown systems can be reduced to a mu-calculus formula on a finite-state system (modeling a parity game solution on the pushdown graph) [23]. Moreover, MUCKE accepts specifications expressed as a formula of the mu-calculus.

Currently, GETAFIX does not support reporting of counter-examples. Mucke does support counterexamples; we plan to adapt it to report readable counter-examples for reachability to the user.

Acknowledgments

We thank Armin Biere for his technical support on the tool MUCKE, Stefan Schwoon for his technical support on the MOPED tools and for providing us SLAM benchmarks, Byron Cook for providing us with TERMINATOR benchmarks, and Ella Bounimova and Tom Ball for their technical support on the tool BEBOP.

References

[1] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

[2] R. Alur, M. McDougall, and Z. Yang. Exploiting behavioral hierarchy for efficient model checking. In Proc., *CAV*, volume 2404 of *LNCS*, pages 338–342. Springer, 2002.

[3] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In Proc, *IFM*, volume 2999 of *LNCS*, pages 1–20. Springer, 2004.

[4] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In Proc., *SPIN*, volume 1885 of *LNCS*, pages 113–130. Springer, 2000.

[5] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3. ACM, 2002.

[6] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.

[7] A. Biere. μcke - efficient μ-calculus model checking. In *CAV*, volume 1254 of *LNCS*, pages 468–471. Springer, 1997.

[8] J. R. Büchi. Regular canonical systems. *Arch. Math. Logik Grundlagenforschung*, 6:91–111, 1964.

[9] S. Chaudhuri. Subcubic algorithms for recursive state machines. In Proc., *POPL*, pages 159–169. ACM, 2008.

[10] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety. In Proc., *CAV*, volume 4144 of *LNCS*, pages 415–418. Springer, 2006.

[11] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In Proc., *CAV*, volume 2102 of *LNCS*, pages 324–336. Springer, 2001.

[12] A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In A. Gupta and S. Malik, editors, *CAV*, volume 5123 of *LNCS*, pages 37–51. Springer, 2008.

[13] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In Proc., *PODS*, pages 1–12. ACM, 2005.

[14] O. Lhoták and L. Hendren. Jedd: a bdd-based relational extension of java. In Proc., *ACM SIGPLAN PLDI*, pages 158–169, USA, 2004. ACM.

[15] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In Proc., *ACM SIGPLAN PLDI*, pages 446–455. ACM, 2007.

[16] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.

[17] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *Proc. ACM SIGPLAN PLDI, USA*, pages 14–24, ACM, 2004.

[18] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, ACM, 1995.

[19] T. W. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In Proc., *SAS*, volume 2694 of *LNCS*, pages 189–213. Springer, 2003.

[20] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.

[21] M. Sharir and A. Pnueli. Two approaches to inter-procedural data-flow analysis. In *Program Flow Analysis: Theory and Applications*, 1981.

[22] D. Suwimonteerabuth, J. Esparza, and S. Schwoon. Symbolic context-bounded analysis of multithreaded Java programs. In Proc. *SPIN*, volume 5156 of *LNCS*, pages 270–287. Springer, 2008.

[23] I. Walukiewicz. Pushdown processes: Games and model-checking. *Inf. Comput.*, 164(2):234–263, 2001.

[24] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. ACM SIGPLAN PLDI, USA*, pages 131–144, ACM, 2004.

Appendix: Precise fixed-point formula for the entry-forward algorithm

We now describe the implementation of the relation $Summary_{EF}$ from Section 4.2, and is hence the precise algorithm implemented in that section. Notice the brevity of the code ($\sim$40 LOC without comments).

In the following code, we use a structure *Conf* defined as a tuple $(mod, pc, CL, CG, ENTRY\_CL, ENTRY\_CG)$ where: *mod* is a module name, *pc* is a program counter of module *mod*, *CL* and *ENTRY_CL* are vectors of local variables, *CG* and *ENTRY_CG* are vectors of global variables. Note that (mod, pc) uniquely determine a program counter in a program. We assume that *pc* of an entry state is 0 in any module. The intended meaning of a given *Conf* s is as follows. Tuple $((s.mod, s.pc), s.CL, s.CG)$ denotes the current state of the program (in the following, we refer to it as s-current), and $((s.mod, 0), s.CL, s.CG)$ is the state visited when the current module was entered for the last time on a computation leading to the current state (in the following, we refer to it as s-entry).

The following formula *Reachable* defines the relation $Summary_{EF}$ from Section 4.2. In fact, it is possible to show that $Reachable(s)$ holds true if and only if $Summary_{EF}(\overline{u}, \overline{v})$ holds true where $\overline{u}$ is the state s-current and $\overline{v}$ is the state s-entry.

```
mu bool Reachable (Conf s) (
  /* early termination  */
  ( exists Conf t. ( target(t.mod,t.pc) & Reachable(t) ))

  /* add initial configurations */
  |  Init(s)

  /* forward propagation on internal transitions:
     s-current is reachable via a transition internal
     to s-module from  t-current, for a reachable t */
  | (exists Conf t. ( Reachable(t)
      & t.mod=s.mod   /* module does not change */
      & t.ENTRY_CG=s.ENTRY_CG & t.ENTRY_CL=s.ENTRY_CL
     /* entry state does not change */
      & programInt(s.mod,t.pc,s.pc,t.CL,s.CL,t.CG,s.CG)
     /* internal transition from t-current to s-current */
    ))

  /* forward propagation on calling a module:
     there is a reachable t such that t-current is
     a call and s-current is a corresponding entry  */
  | (s.pc=0 & s.ENTRY_CG=s.CG /*s-current is an entry*/
      & CopyLocals(s.mod,s.ENTRY_CL,s.CL)
    /* s.ENTRY_CL=s.CL on local variables in s.mod */
      & (exists Conf t.
         (Reachable(t) & t.CG=s.CG
     /* s-current and t-current match on global vars*/
          & programCall(t.mod,s.mod,t.pc,t.CL,s.CL,s.CG)
     /* t-current is a call to s.mod and parameters
       are passed from t-current to s-current */
    )))

  /* forward propagation from call to matching return:
       s-current is a return,
     (1) there is a reachable t such that
      (1.1) s-entry and t-entry match
      (1.2) t-current is a call matching s-current
      (1.3) s-current and t-current match on local
            variables not assigned with return values
      (1.4) t-current is a call corresponding to u-entry
      (1.5) call parameters are computed from t-current
            and assigned to local variables of u-entry
      and
      (2) there is a reachable
      u=(u_mod,u_pc,u_CL,u_CG,u_ECL,u_ECG) s. t.
      (2.1) u-entry is an entry corresponding to
            t-current (u_ECG = t.CG)
      (2.2) u-current is an exit
      (2.3) u-current corresponds to return s-current
      (2.4) s-current and u-current match on global
            variables not assigned with return values
      (2.5) remaining global variables and assigned
            local variables of s-current are assigned
            with return values computed from u-current
  */
  | (exists PrCount t_pc, Global t_CG, Module u_mod,
            PrCount u_pc, Local u_ECL.
       /* clause (1) */
     (exists  Conf t.
       (Reachable(t)
         & (t.pc=t_pc & t.CG=t_CG)
      /* extract t components needed in clause 2 */
         & (t.mod=s.mod & t.ENTRY_CL=s.ENTRY_CL
           & t.ENTRY_CG=s.ENTRY_CG) /*clause 1.1*/
         & SkipCall(s.mod,t.pc,s.pc) /*clause 1.2*/
         & SetReturn1(s.mod,u_mod,t.pc,u_pc,t.CL,s.CL)
      /* clause 1.3 */
         & programCall(t.mod,u_mod,t.pc,t.CL,u_ECL,t.CG)
      /* clauses 1.4 and 1.5 */
    ))
    &
    /* clause 2 */
    (exists Conf u.(
      (u.mod=u_mod & u.pc=u_pc & u.ENTRY_CL=u_ECL
        & u.ENTRY_CG=t_CG)
      /* u conforms to extracted values & clause 2.1*/
      & Reachable(u)
      /*  u is reachable */
      & Exit(u_mod,u_pc)
      /* clause 2.2 */
      & SetReturn2(s_mod,u_mod,t_pc,u_pc,
                           u_CL,s_CL,u_CG,s_CG)
      /* clauses 2.3, 2.4 and 2.5 */
  )))
);

/* Reachability query: is a target state reachable?*/
(exists Conf s. (target(s.mod,s.pc) &  Reachable(s)));
```