Model Checking Sequential Software Programs Via Mixed Symbolic Analysis

ZIJIANG YANG
Western Michigan University
and
CHAO WANG, AARTI GUPTA, and FRANJO IVANČIĆ
NEC Laboratories America

We present an efficient symbolic search algorithm for software model checking. Our algorithms perform word-level reasoning by using a combination of decision procedures in Boolean and integer and real domains, and use novel symbolic search strategies optimized specifically for sequential programs to improve scalability. Experiments on real-world C programs show that the new symbolic search algorithms can achieve several orders-of-magnitude improvements over existing methods based on bit-level (Boolean) reasoning.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aid—*Verification*

General Terms: Verification

Additional Key Words and Phrases: Model checking, reachability analysis, image computation, binary decision diagram, presburger arithmetic, composite symbolic formula

ACM Reference Format:

Yang, Z., Wang, C., Gupta, A., and Ivančić, F. 2009. Model checking sequential software programs via mixed symbolic analysis. ACM Trans. Des. Autom. Electron. Syst. 14, 1, Article 10 (January 2009), 26 pages, DOI = 10.1145/1455229.1455239 http://doi.acm.org/10.1145/1455229.1455239

1. INTRODUCTION

Software has become pervasive in civilian and military infrastructure as increasingly capable and less expensive hardware processors and communication devices are routinely being embedded into an ever-growing list of physical devices. Software is now being used for controlling critical systems such as nuclear power plants, patient monitors, and planes. Consequently, the

A preliminary version of this article was presented at the Memocode'06 conference [Yang et al. 2006].

Author's address: Z. Yang, Department of Computer Science, Western Michigan University, Parkview Campus, Kalamazoo, MI 49008; email: zijiang.yang@wmich.edu.

problem of making software reliable has become one of today's most important challenges.

While specialized domain-specific languages are available for programming embedded hardware, C and its derivatives, such as nesC [Gay et al. 2003] are still the most popular languages in this domain due to their ability to manipulate the runtime memory using low-level constructs. However, C is inherently an unsafe language [Cardelli 1997]. In particular, pointers and explicit memory management lead to erroneous programs with buffer overruns, uninitialized variables, memory leaks, and null pointer dereferences.

Model checking is a method to algorithmically verify formal systems [Clarke and Emerson 1981; Quielle and Sifakis 1981]. This is achieved by verifying if the model, often derived from a hardware or software design, satisfies a formal specification. It has been successfully used in the design of complex circuits and communication protocols [Clarke et al. 2000; McMillan 1994]. The procedure normally uses an exhaustive search of the model under consideration. As a result, model checking tools face a combinatorial blowup of the state-space, commonly known as the state explosion problem. In order to address the problem, various symbolic search techniques have been proposed to represent and manipulate set of states instead of each individual state, therefore significantly improving the search capacity [McMillan 1994; Biere et al. 1999].

Although automated detection of software errors has been tackled previously with static analysis, the problem is far from being solved. Static analysis produces a large number of false alarms due to inherent inaccuracy. While symbolic model checking has been extensively studied for hardware verification in industrial settings, its application to analyzing source code programs written in modern programming languages (as opposed to specialized modeling languages) is relatively new [Visser et al. 2000]. Existing symbolic model checking tools in this category, including [Ball and Rajamani 2000; Clarke et al. 2004a], often restrict their representations in the pure Boolean domain; that is, they extract a Boolean-level model from the given program and then apply symbolic decision procedures such as Binary Decision Diagrams (BDDs) [Bryant 1986] and SAT [Davis et al. 1962] to perform verification. Although modeling program variables of various types as bit-vectors is accurate, such a high precision approach is often not needed and may generate models of very large sizes. In addition, it is difficult to model floating point numbers as bit-vectors.

In this article we present an efficient symbolic search algorithm for model checking C programs. In order to design scalable algorithms to verify C programs against programming bugs such as array bound violations, use of uninitialized variables, memory leaks, locking rule violations, and division by zero, we combine multiple symbolic representations at Boolean, integer and real level to efficiently represent the transition relation and reachable states and use a combination of decision procedures for reachability computation. The novel features of the new algorithms include:

— mixed symbolic representations using composite symbolic formulas to model C programs with rich data types and complex expressions;

— new symbolic search strategies and optimization techniques specific to sequential programs that can significantly improve the scalability of model checking algorithms.

We have implemented the proposed techniques and performed a set of experiments on control intensive C programs. Our experimental results show that the new algorithm significantly outperforms existing methods in terms of both CPU time and memory usage. In particular, the experimental study shows that the new algorithm is significantly more scalable than pure Boolean-level algorithms based on BDDs and SAT, indicating that it is advantageous to raise abstraction levels in symbolic model checking.

A preliminary version of this article has been presented at the Memocode'06 conference [Yang et al. 2006]. While the preliminary version only presented the major techniques used in the reachability analysis, this article presents the complete model checker, including the software modeling techniques that transform C programs to mixed symbolic models. In addition, experiments on industrial proprietary software have been performed since the preliminary version to show the effectiveness of the proposed algorithms. The remainder of this article is organized as follows. After describing the related works in Section 2 and the concept of composite symbolic formulas in Section 3, we present the architecture for our mixed symbolic model checker in Section 4. In Section 5 we introduce the software modeling approach by explaining the transformation from C programs into mixed symbolic models. We also review the basic set-theoretic operations on composite formulas and the corresponding model checking procedure. In Section 6, we present our software specific optimization techniques in decomposing and minimizing the transition relation representations. In Section 7, we present two new strategies for symbolic fixpoint computation in order to exploit the unique characteristic of sequential models. We give the experimental results in Section 8 and then conclude in Section 9.

2. RELATED WORK

A plethora of tools exists for analyzing different aspects of C programs. Most well-known program analysis tools for C perform lightweight data flow analysis, such as Splint and LCLint [D. Evans et al. 1994], PREfix and PREfast [Bush et al. 2000], Uno [Holzmann 2002], ESP [Das et al. 2002], and Codesurfer [Anderson et al. 2003]. On the other end of the spectrum, model checkers such as MAGIC [Chaki et al. 2003], CBMC [Clarke et al. 2004b], and BLAST [Beyer et al. 2007] perform finite state model checking of C programs. These tools first generate a finite Boolean model from the source code using bit-vector encoding of program variables or predicate abstraction techniques. Analysis is then carried out on this finite Boolean model using techniques similar to hardware model checking.

Bultan et al. [1997] proposed to use Presburger formulas in an infinite-state model checker. Then in Bultan et al. [2000] and Yavuz-Kahveci and Bultan [2003] they proposed a composite symbolic representation by combining the relative strengths of two symbolic representations: BDDs for Boolean formulas and finite unions of convex polyhedrons for formulas in Presburger arithmetic. Their approach has the advantage of representing both bit-level and word-level expressions uniformly at the suitable abstraction levels. However, the technique in its original form was not aimed at directly handling large sequential programs written in a general purpose programming language. In Bultan et al. [2000] and Yavuz-Kahveci and Bultan [2003], one needs to specify the model in a domain-specific input format called action language, and the published experimental evaluations of their symbolic algorithms were on relatively small concurrent protocols.

In this article, we follow the general framework of Bultan et al. [2000] and Yavuz-Kahveci and Bultan [2003] in combining multiple symbolic representations. However, our focus is on improving the scalability of the composite model checking algorithms, with the application to verifying source code level sequential programs. In our application domain, the number of program variables is often orders-of-magnitude larger than in previous studies [Yavuz-Kahveci et al. 2005]. We differentiate our work from the prior art primarily in the following aspects: (1) we use mixed symbolic representations to model programs with significantly richer data types and more complex expressions; and (2) we develop new search strategies and optimizations specific to sequential programs to improve the scalability of model checking algorithms. In particular, we derive high-level information of the software model using a static control flow analysis, and use it to decompose and minimize the transition relations and to improve the performance of symbolic fixpoint computation.

Linear constraint representations and polyhedral analysis have also been used in the verification of real-time and hybrid systems [Henzinger et al. 1995; Asarin et al. 2000]. These systems are often specified as timed or hybrid automata with variables of infinite data types and continuous dynamics. A state set is represented symbolically as a polyhedron as opposed to a finite union of polyhedrons; the union of two state sets is approximated into their convex union. Since a convex hull is often expensive to compute, this approach is known to have scalability problems. In addition, convex hull causes a precision loss in the analysis result.

The Symbolic Analysis Laboratory (SAL) [Bensalem et al. 2000] also provides a method for combining different decision procedures. However, it is different from our approach in the sense that the different decision procedures and verification tools of SAL are glued together loosely at a very high level by a specification language that models concurrent systems in a compositional manner. On the other hand, a plethora of SMT (satisfiability modulo theories) tools exist to combine Boolean and integers at solvers' level. Note that SMT and our tool are targeting different applications. SMT is solving satisfiability problems, ideally for bounded model checking, while our tool performs fix-point reachability computation with repeated existential quantifications.

Word-level model for C programs has also been used in linear programs where program variables can range over a numeric domain [Armando et al. 2006a, 2006b]. However, our work emphasizes the combination of different modeling techniques such that each domain can be solved by the most efficient decision procedure.

3. BACKGROUND: COMPOSITE SYMBOLIC FORMULAS

In this section, we review the definition of composite symbolic formulas and the corresponding set theoretic operations.

3.1 Composite Symbolic Formulas

Let $\mathbb{Z}$ be the set of integer numbers and $\mathbb{R}$ be the set of real numbers. An integer linear constraint is denoted by $\mathbf{a_i x} \leq b$, where $\mathbf{x}, \mathbf{a_i} \in \mathbb{Z}^n$ (x is the variable) and $b \in \mathbb{Z}$ is a scalar constant. Similarly, a real linear constraint is denoted by $\mathbf{c_i y} \leq d$, where $\mathbf{y}, \mathbf{c_i} \in \mathbb{R}^n$ (y is the variable) and $d \in \mathbb{R}$ is a scalar constant.

Definition 3.1 (*c.f. [Yavuz-Kahveci et al. 2001]*). The composite symbolic formula F is defined as follows,

$$F := F \wedge F | \neg F | F^B | F^I | F^R,$$

where F^B, F^I, and F^R are formulas in Boolean logic, Presburger arithmetic, and Boolean combination of real linear constraints, respectively.

This definition extends the one in Yavuz-Kahveci et al. [2001] by introducing one more elementary formula type: Boolean combination of linear constraints on reals. A formulation of composite symbolic representation for arbitrary number of types is given in Bultan et al. [2000]. A composite symbolic formula can be put into the *Disjunctive Normal Form (DNF)* as follows:

$$F = \bigvee_i F_i^B \wedge F_i^I \wedge F_i^R,$$

where each disjunct is called an *atom*, and the number of atoms is finite in a DNF. Assume that all expressions in a composite formula are type-consistent, then subformulas of different types share no common variables.

3.2 Basic Set-Theoretic Operations

The general approach of carrying out set-theoretic operations on composite symbolic formulas is to rewrite the operands into DNF, process the corresponding subformulas with suitable decision procedures, and assemble the result back into DNF. Note that the DNF representation is not canonical, and there are heuristic algorithms [Yavuz-Kahveci et al. 2001] to make the result more compact. One can use the CUDD package [Somenzi 1995] to represent Boolean formulas, the Omega library [Pugh 1991] to represent Presburger formulas, and the Parma Polyhedral Library [Bagnara et al. 2002] to represent linear constraints on reals. These underlying manipulation packages all support set-theoretic operations such as union ($\vee$), conjoin ($\wedge$), negation ($\neg$), and quantification ($\exists$).

Since there is no common variable shared by F^B, F^I, and F^R, subformulas in different domains do not interfere with each other during these set-theoretic operations. Given two composite symbolic formulas $f = \bigvee_{i=1}^{n_f} f_i^B \wedge f_i^I \wedge f_i^R$ and $g = \bigvee_{j=1}^{n_g} g_j^B \wedge g_j^I \wedge g_j^R$, we briefly illustrate how to perform set-theoretic operations at composite level. For more details, please refer to Yavuz-Kahveci et al. [2001].

The union of two composite formulas is defined as:

$$f \vee g = \bigvee_{i=1}^{n_f+n_g} \left(k_i^B \wedge k_i^I \wedge k_i^R\right), \tag{1}$$

where $k_i = f_i$ for $1 \leq i \leq n_f$, and $k_i = g_{i-n_f}$ for $n_f + 1 \leq i \leq n_f + n_g$. The number of disjuncts in the results is $O(n_f + n_g)$.

The conjunction of two composite formulas is the union of pair-wise conjunctions of their subformulas.

$$f \wedge g = \bigvee_{i=1}^{n_f} \bigvee_{j=1}^{n_g} \left(f_i^B \wedge g_j^B\right) \wedge \left(f_i^I \wedge g_j^I\right) \wedge \left(f_i^R \wedge g_j^R\right). \tag{2}$$

The number of disjuncts in the results is $O(n_f \times n_g)$.

The negation of a composite formula is defined as

$$\neg f = \bigwedge_{i=1}^{n_f} \left(\neg f_i^B\right) \vee \bigwedge_{i=1}^{n_f} \left(\neg f_i^I\right) \vee \bigwedge_{i=1}^{n_f} \left(\neg f_i^R\right). \tag{3}$$

The number of disjuncts in the results is $O(3^{n_f})$.

Due to the fact that the sets of Boolean variables v^B, integer variables v^I, and real variables v^R are disjoint, existential quantification distributes not only over unions (which is true in the pure Boolean domain) but also over conjunctions of subformulas of different types; that is,

$$\exists_{v^B,v^I,v^R} f = \bigvee_{i=1}^{n_f} \left(\exists_{v^B} f_i^B\right) \wedge \left(\exists_{v^I} f_i^I\right) \wedge \left(\exists_{v^R} f_i^R\right). \tag{4}$$

Although the number of DNF terms can be as large as $(n_F \times n_G)$ for conjunction and 3^{n_F} for negation, such a worst-case blowup does not happen in any of our experiments performed in Section 8. There may be two explanations: (1) Our benchmarks are all control intensive embedded software. We may observe a worst-case blowup in other types of applications such as numerical computation programs. (2) There exist efficient heuristics in Omega and Parma Polyhedral Library to reduce the number of DNF terms.

4. TOOL ARCHITECTURE

Figure 1 shows the overall architecture of the mixed symbolic model checker, which is part of the F-SOFT [Ivančić et al. 2005a, 2005b, 2009] tool suite for analyzing safety properties of C programs. In reachability analysis, we check whether certain statements are reachable from an entry point of the program. A large set of programming bugs, such as assertion failure, buffer overruns, use of uninitialized variables, memory leaks, locking rule violations, and division by zero, can be formulated as reachability problems by adding suitable property monitors automatically to the given program.

The F-SOFT tool tries to combine the complimentary strengths of static program analysis based on techniques such as abstraction interpretation and model checking to ensure high efficiency with low false alarm rate. Specifically, we use static program analysis techniques such as program slicing [Jayaraman

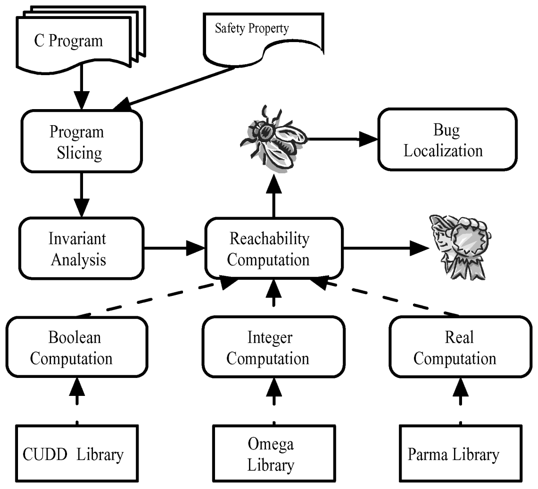

Fig. 1. Mixed symbolic model checker architecture.

et al. 2005; Tip 1995], constant folding, range or interval analysis [Cousot and Cousot 1976; Rugina and Rinard 2000; Zaks et al. 2008], and other techniques to reduce the model size. We also use several sound numerical analysis techniques to prove the correctness of as many properties of interest as possible [Sankaranarayanan et al. 2006, 2007]. These techniques extend the analysis capabilities of the interval domain, which allows us to compute invariants over multiple variables such as the octagon domain [Miné 2001].

Since these static analysis techniques are quite efficient and thus have a relatively low overhead compared to model checking, we generally try to resolve as many properties as possible in this phase of the analysis. Though several such properties get resolved during this phase, most of them still remain unresolved. Model checking technology as described in this article is then employed to reduce the burden on the user of the tool by finding precise witness traces that can guide the debugging effort. The details of the static analysis techniques are out of scope of this article. Interested readers are referred to the cited articles for more information. The experimental results presented in this article, both with and without the mixed symbolic analysis, are obtained after using the static analysis techniques. In addition to the reduction in the model by proving certain properties, we use the interval analysis [Zaks et al. 2008] also to decide how to represent certain variables during the model checking stage. For integer variables that are found to have relatively small ranges, we choose to model them as finite-length bit-vectors using BDDs as opposed to integer variables using polyhedrons.

As mentioned before, counterexamples are an important advantage that model checkers can provide to the user. Although the counterexample returned by a model checker can help in reproducing the symptom related to a defect, a significant amount of effort is still often required for the programmer to interpret it in order to locate the cause of the problem. Note that the model checker finds a counterexample in the model of the software, which means that it need

```
int foo(int s){              void bar(){
  int t=s+2;                   int x=3;
  if (t>6)                     int y=x-3;
    t -= 3 ;                   while (x<=4){
  else                           y++ ;
    t--;                         x = foo(x);
  return t;                    }
}                              y = foo(y);
                             }
```

Fig. 2. Sample C code.

not correspond to an actual bug in the original code but rather in the modeling assumptions employed to handle the unknown environment. In the bug localization component, we provide an automated procedure to focus on potential model defects (and relate them back to the original software) by analyzing a single concrete counterexample. For a detailed explanation of the algorithms, please refer to Wang et al. [2006].

5. SOFTWARE MODELING

F-Soft begins with a program in full-fledged C and applies a series of source-to-source transformations into smaller subsets of C, until the program state is represented as a collection of simple scalar variables and each program step is represented as a set of parallel assignments to these variables. For a comprehensive description of the transformations, please refer to Ivančić et al. [2005b]. In this section, we present the details relevant to the construction of a mixed symbolic model.

5.1 Labeled Transition Graph

Figure 3 shows the graphical representation of the sample code in Figure 2, by separating the graph into two subgraphs. The control logic subgraph at the left-hand is used to define the transition relation in terms of basic block changes. The data logic subgraph at the right-hand side concentrates on how variables are updated in each individual block, and therefore is used to define the transition relation for program variables. A preprocessing analysis determines that function `foo` is not called in a recursive manner. The two return points are recorded by an encoding that passes a unique return location as a special parameter using the variable `rtr`.

Each rectangle in the data logic subgraph is a basic block consisting of a set of parallel assignments that can be executed at the same time. Compared with sequential assignments that need one transition relation expression for each assignment, the set of parallel assignments in a basic block requires a single transition relation expression; thus reduces the number of image computations in reachability analysis. The assignments in block 1 of Figure 3 show the parallel assignments converted from the original sequential assignments $x = 3; y = x - 3; x = y + 4;$ in the function `bar()` of Figure 2. For any variable that is written by right-hand expression e in the basic block, we substitute all the subsequent reads of the variable with e. After the substitution, the three

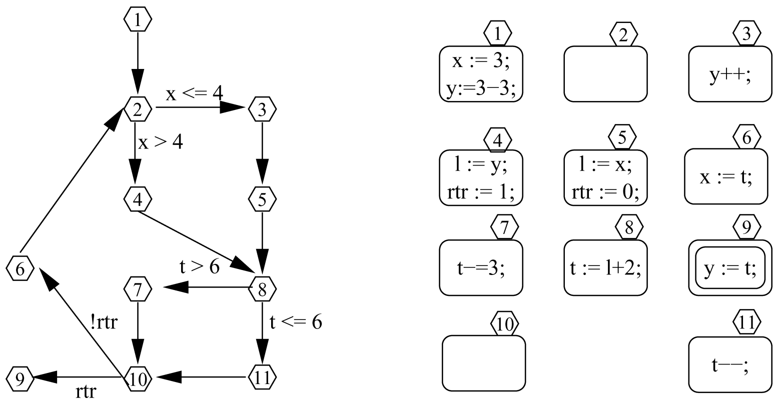

Fig. 3. Labeled transition graph (divided into control logic and data path).

assignments become $x = 3; y = 3-3; x = (3-3)+4;$. Then the dead assignments are removed, which leads to two remaining assignments $y = 3-3; x = 3-3+4$ in this example.

The edges in Figure 3 are labeled by conditional expressions, for example, the transition from block 2 to block 3 is guarded by $x \leq 4$. In case an edge is not labeled by any condition, the default condition is true. Finally, block 1 is the entry block and block 9 is the one that leaves the analysis scope.

Formally, the transformations produce a simplified program that can be represented as a labeled transition graph.

Definition 5.1. A labeled transition graph G is a 5-tuple $\langle B, E, X, \delta, \theta \rangle$, wherein

—$B = \{b_1, \ldots, b_n\}$ is a finite nonempty set of basic blocks. $b_s \in B$ is an initial basic block.
—$E \subseteq B \times B$ is a set of edges representing transitions between basic blocks.
—X is a finite set of variables that consists of actual source variables and auxiliary variables added for modeling and property monitoring.
—$\delta : B \rightarrow 2^{\Sigma_A}$ is a labeling function that labels each basic block with a set of parallel assignments, where Σ_A represents the set of all possible C assignment expressions.
—$\theta : E \rightarrow \Sigma_C$ is a labeling function that labels each edge with a condition, where Σ_C represents the set of all possible C conditional expressions, such as the conditions in the C code as part of `if-then-else` or `while` expressions.

We denote a valuation of all variables in X by $\vec{x}$, and the set of all valuations by $\mathcal{X}$. The state space of the entire program is $Q \subseteq B \times \mathcal{X}$. we define a state to be a tuple $q = (b, \vec{x}) \in Q$. The initial states of the program are in the initial basic block b_s with an arbitrary data valuation, denoted by $Q_0 = \{(b_s, \vec{x}) | \vec{x} \in \mathcal{X}\} \subseteq Q$. The set of parallel assignments in each $b_i \in B$, denoted by $\delta(b_i)$, can be written as $\{x_1 := e_1, \ldots, x_n := e_n\}$, where $\{x_1, \ldots, x_n\} \subseteq X$ and $\{e_1, \ldots, e_n\} \subseteq \Sigma_A$.

For checking reachability properties, we define a subset $Q_{Err} \subseteq Q$ to be error states; model checking is then used to prove or disprove that these error states can be reached. Let $q_1 \rightarrow q_2$ denote a valid transition between the two states $q_1, q_2 \in Q$. We define a path in the state space Q to be a sequence of states $(b_0, \vec{x}_0), \ldots, (b_k, \vec{x}_k)$ such that $(b_0, \vec{x}_0) \in Q_0$ and for all $0 \leq i < k-1$, $(b_i, \vec{x}_i) \rightarrow (b_{i+1}, \vec{x}_{i+1})$. A counterexample is a path that ends in an error state $(b_k, \vec{x}_k) \in Q_{Err}$.

5.2 Pointer and Memory Modeling

One difficulty in modeling C programs lies in modeling indirect memory accesses via pointers, such as `x=*(p+i)` and `q[j]=y`. We replace all indirect accesses with equivalent expressions involving only direct variable accesses, by introducing additional variables and conditional expressions as described below.

— To facilitate the modeling of pointer arithmetic, we build an internal memory representation of the program by assigning to each variable a unique natural number representing its memory address. Adjacent variables in C program memory (for example, elements of an array) are given consecutive memory addresses.

— We perform a points-to analysis [Hind and Pioli 2001] to determine, for each indirect memory access, the set of variables that may be accessed (called the *points-to set*). If a pointer can point to a set of variables at a given program location, we rewrite a pointer read as a conditional assignment expression using the numeric memory addresses assigned to the variables.

— For accesses via pointers, we adopt an approach from hardware synthesis [Séméria and Micheli 1998] and create additional variables for each pointer variable. The declaration `int **p` creates three variables: v_p for pointer p, $\dot{v}_p$ for pointer dereference $*p$, and $\ddot{v}_p$ for pointer dereference $**p$. A pointer reference in the C code, such as $\&q$, also leads to an additional variable–in this case the variable $\hat{v}_q$.

As an example, consider the assignment `p=&q`, where p is declared as `int **p` and q as `int *q`. This assignment directly corresponds to the assignment $v_p = \dot{v}_q$. However, since $p = \&q$ implies $*p = q$ and $**p = *q$, two new assignments $\dot{v}_p = v_q$ and $\ddot{v}_p = \dot{v}_q$ are also inferred from the original assignment in the source code. We call such assignments *implied assignments*. Consider another assignment $q = \&m$ in the same program where m is declared as `int m`. This assignment gives rise not only to the assignments $v_q = \dot{v}_m$ and $\dot{v}_q = v_m$, but also to conditional assignments due to aliasing. Since p may equal $\&q$, it is possible that $*p$ and $**p$ are assigned new values when q is assigned. This results in the conditional assignments $\dot{v}_p = (v_p == \hat{v}_q)?\hat{v}_m : \dot{v}_p$ (which stands for $*p = (p == \&q)?\&m : *p$) and $\ddot{v}_p = (v_p == \hat{v}_q)?v_m : \ddot{v}_p$ (which stands for $**p = (p == \&m)?m : **p$). These kind of assignments are called *aliasing assignments*.

5.3 Unbounded Data and Recursion

The C language specification does not bound heap or stack size, but our focus is on generating a model with bounded dynamic arrays and recursions. Such modeling approach works well on control intensive programs such as device drivers and embedded software in portable devices, although it may not be suitable for programs in some application domains such as scientific computing and memory management. We model the heap as a finite array, adding a simple implementation of `malloc()` that returns pointers into this array. We also add a bounded depth stack as another global array in order to handle bounded recursion, along with code to save and restore local state for recursive functions only. Therefore, we only handle bounded recursions with the bound provided by users. However, we argue that there is always a tradeoff between precision and termination, given the fact that program verification in general is undecidable.

5.4 Symbolic Representation of the Model

Let P denote the set of program counter (PC) variables for encoding the set B of basic blocks (or program locations);[1] then P and X form the complete set of state variables of the model. Their next-state values are represented by the *primed* version P' and X'. The verification model is represented by $\langle T, I\rangle$, wherein $T(P, X, P', X')$ is the transition relation and $I(P, X)$ is the initial state predicate. An evaluation of the characteristic function $T(b, \vec{x}, b', \vec{x}')$ is $true$ if and only if there is a transition from the state $(b, \vec{x})$ to the state $(b', \vec{x}')$. Similarly, the evaluation of function $I(b, \vec{x})$ is $true$ if and only if $(b, \vec{x})$ is an initial state.

We choose to represent expressions related to PC variables as Boolean formulas. That is, we allocate a finite set of Boolean variables $P = \{p_1, p_2, \ldots, p_k\}$ with p_1 the least significant bit. This is based on the observation that formulas involving the PC variable are often control-intensive, for which the representation of linear constraints is ill-suited. For instance, a PC value of 5 ($p_1 = p_3 = 1, p_2 = 0$) is encoded as $(p_3 \wedge \neg p_2 \wedge p_1)$. For convenience, in the rest of the article we use $P = val$ to denote the value of the PC variables is val, although it should be understood that P is a set and it presents multiple Boolean variables.

On the other hand, we use integer and real linear constraints to model the data-path. Individual expressions in $\delta(b_i)$ such as $(x'_k = e_{ik})$ are represented either by a Boolean formula, Presburger formula, or polyhedrons on real, depending on the type of the variable x'_k. If there are expressions that are not type-consistent, we perform over-approximation similar to that for non-linear operators to be discussed next. In this case, the correlation of variables from two different types are ignored.

Reachable states are also represented disjunctively as the union of subformula. For instance, given a set of initial values $\{x_1 = e_{01}; \ldots; x_m = e_{0m}\}$ and the entry block b_s, we have the initial predicate $I \equiv (P = b_s) \wedge \bigwedge_{k=1}^{m}(x_k = e_{0k})$. Given a composite formula representing an arbitrary state set, we can easily partition the conjuncts and convert it to DNF.

[1] P consists of $\lceil \log |B| \rceil$ Boolean variables in a pure bit-level representation, or a single integer variable in a word-level representation.

5.5 Handling Non-Linear Operators

Since non-linear operators on integer and real variables cannot be modeled by polyhedrons, they need special treatment. If all operands are of integer type and of bounded size, we can model a non-linear operation as Boolean-level operations through the instantiation of predefined logic components such as multipliers. However, not all non-linear operations can be handled this way: if a bounded integer variable x is treated as a fixed-length bit-vector, then (1) any operation on x must be treated as a bit-vector operation; and (2) any other operand of the same bit-vector operation must be treated as a bit-vector. Therefore, the definition of bit-vector variable is transitive. If a non-linear operation involves both fixed-length bit-vectors and unbounded integers, it cannot be modeled in pure Boolean logic. The requirement of disallowing common variables shared among different symbolic decision procedures clearly differentiates this modeling approach from the Nelson-Oppen framework for cooperating decision procedures [Nelson 1984].

If this requirement is not satisfied, we resort to approximate modeling. A straightforward way is to assume that the result of a non-linear operation takes an arbitrary value. For instance, the assignment $x_k \leftarrow x_i * x_j$ becomes $x_k \leftarrow w$, where w is a nondeterministic *pseudo input* variable of the suitable type. During post-condition computation, w will be existentially quantified out, therefore modeling the fact that x_k can take an arbitrary value. If an upper and/or lower bound on the values of its operands is known, we can improve the approximation by estimating the output value range of the non-linear operation. For instance, given $1 \leq x_i \leq 4$ and $2 \leq x_j \leq 5$, we can impose the additional constraint $2 \leq w \leq 20$. The bound information of variables x_i and x_j may come from an interval analysis [Zaks et al. 2008], which determines a conservative value interval of each variable in the given program.

6. MIXED SYMBOLIC TRANSITION RELATIONS

Now we present our software specific optimizations that decompose and simplify mixed symbolic representations of the transition relation and the reachable state set.

6.1 Disjunctive Transition Relations

From the labeled transition graph of a given program, we construct the symbolic representation of its verification model as follows. We define transition relation of the entire model as:

$$T \equiv \bigvee_{(b_i, b_j) \in E} t_i^d \wedge t_{ij}^c,$$

where t_{ij}^c denotes the transition of control flow from block b_i to block b_j, and t_i^d denotes the data assignments inside block b_i. Given a transition from b_i to b_j under the condition $\theta(b_i, b_j)$, the transition relation t_{ij}^c is defined as follows,

$$t_{ij}^c \equiv (P = i) \wedge (P' = j) \wedge \theta(b_i, b_j).$$

Given a block $b_i \in B$, t_i^d describes the conjunction of all assignments in $\delta(b_i)$, and therefore is defined as follows,

$$t_i^d \equiv (P = i) \wedge \bigwedge_{k=1}^{|X|} (x'_k = e_{ik}).$$

Inside a block b_i, for each variable $x_k \in X$, the elementary transition relation is $x'_k = e_{ik}$ such that

$$e_{ik} = \begin{cases} e & \text{, if } (x_k := e) \in \delta(b_i), \\ x_k & \text{, otherwise} \end{cases}$$

A disjunctively partitioned T is naturally suited for sequential software programs. Let $T = \bigvee T_{ij}$ and $T_{ij} = t_i^d \wedge t_{ij}^c$; then T_{ij} corresponds to a transition in the labeled transition graph.

$$T_{ij} \equiv (P = i) \wedge (P' = j) \wedge \theta(b_i, b_j) \wedge \bigwedge_{k=1}^{|X|} (x'_k = e_{ik}).$$

Note that the partitioning of T into T_{ij} is independent of any symbolic representation. When we use composite formulas to represent each T_{ij}, there will be another level of decomposition which further partitions each component T_{ij} into individual conjuncts based on their formula types. It is worth pointing out that these two levels of decomposition are different, and indeed complementary.

6.2 Simplifying Transition Relations

The main reason for state explosion inside symbolic model checking is the exponential dependency of the state space on the number of state variables of the model. For many realistic C programs, the number of variables of the verification model can easily be in the hundreds (including those added for modeling indirect memory accesses, function calls, and encoding properties), which is well above the capacity of state-of-the-art BDD and polyhedral analysis algorithms. Although all elementary decision procedures can dynamically simplify representations—variable sifting in CUDD and `simplify` in the Omega library—they are time-consuming in the presence of many variables.

The symbolic model checking algorithm as outlined up to this point still suffers from performance problems. In a normal reachability fixpoint computation, it is often the case that both the number and the size of polyhedrons in the reachable state set quickly become too large for the underlying polyhedral libraries.

Our observation is that most variables in sequential programs are inherently local, and therefore should be considered as state-holding only when they affect the control flow or the data-path. In our previous work [Wang et al. 2007], we have successfully exploited this characteristic of sequential program to simplify BDD-based image computation, and have obtained significant performance improvement. Here, we extend the technique to simplify the transition relation as well as reachable state sets for model checking using mixed representations.

code fragment		live variables
L1:	x = y = 0;	{ }
L2:	x = 7;	{ }
L3:	s = x;	{ x }
L4:	y = 8;	{ s }
L5:	s = s + y ;	{ s, y }
L6:	if (s) goto L2;	{ s }
L7:	ERROR:	{ }

Fig. 4. An example of live variables.

Definition 6.1. Variable $x \in X$ is *live* in block $b_m \in B$ if and only if b_m is on an execution path π from b_i to b_j (including b_i and b_j) such that,

—x is defined in $\delta(b_i)$ and not redefined in any other blocks in π, and

—x is read in an assignment in $\delta(b_j)$ or in the condition $\theta(b_j, b_k)$ where $(b_j, b_k) \in E$.

In our reachability procedure, we associate a reachable state subset with each basic block (i.e., a disjunctive partition of the reachable state set). From this definition, it is clear that if x is not live in block b_i, there is no need to record its value in the associated reachable state subset. Note that according to the definition a variable x may be live in a block even if x is not accessed in the block. Locally defined variables are live only inside the program scopes in which they are defined; these variables can be identified syntactically. However, we note that even globally defined variables may not be live (according to our definition) at all basic blocks. We use the code fragment in Figure 4 to show that global variables are often live at a limited number of locations. Assume that x, y, and s are global variables but do not appear elsewhere in the program. Then none of them are live at program locations 1 and 2, since their values will not affect the control flow and data-path. Variable x is considered live at L3 because its value will be assigned to s, and similarly for y at L5. We consider s as live at L6 because its value may affect the control flow. (Figure 4 is for illustration purposes only.)

Finding the set of blocks in which variable x is live is a standard program analysis problem. We use the live variable analysis for the following optimization. During the construction of the transition relation T_{ij}, if a certain variable x_k is not alive in the destination block b_j, we remove $x'_k = e_{ik}$ from the transition relation component, since the value of x_k would be immaterial in the destination block. The next-state variable x'_k in this case can assume an arbitrary value, thereby providing an abstraction of the search state space. Note that the live variable analysis can achieve significantly more reduction of the transition relation size than a simple program slicing. In Figure 4, for instance, we can remove the implicit assignments $x' = x$ from the transition relations at Lines 4-6 where x is not live; however, a property dependent program slicing along cannot remove them. Our experience shows that in practice, live variables with respect to any individual block comprise typically less than 30% of the entire program variables in X.

In our previous work [Wang et al. 2007], the live variable information was used to existentially quantify dead variables out of image results at each iteration. In this article, however, we use live variables to directly simplify the mixed symbolic representations of individual transition relation components. This prevents transition relations of dead variables from being involved in the often costly post-condition computation. Existential quantification of dead variables from the post-condition results, as was done in Wang et al. [2007], is avoided since dead variables never appear in the result in the first place.

Removing dead variables not only reduces the sizes of the symbolic representations, but also leads to a potentially faster convergence of reachability analysis. Take the code fragment in Figure 4 as an example. With the live variable based simplification, one can declare the termination of reachability fixpoint computation after going from L1 through L6 only once. This is because the post-condition of L6 is ($P = 2 \wedge s = 15$), which has already been covered by ($P = 2$), the post-condition of L1 (wherein s can take any value). However, if x and y are assumed to be live everywhere, we will have much larger polyhedrons to represent in the reachable states at each location. In addition, we can no longer declare convergence after L6, since the post-condition ($P = 2 \wedge s = 15 \wedge x = 7 \wedge y = 8$) is not covered by ($P = 2 \wedge x = 0 \wedge y = 0$), the post-condition of L1. As a result, we need a few more iterations in order to declare convergence.

7. SYMBOLIC SEARCH STRATEGIES

In this section, we present symbolic search algorithms that are suitable for software verification using mixed symbolic representations.

7.1 Disjunctive Symbolic Reachability Computation

In symbolic model checking, the state transition graph of the model is represented symbolically by $\langle T, I \rangle$, where $T(X, P, X', P')$ is the characteristic function of the transition relation, and $I(X, P)$ is the initial state predicate. Reachability analysis is a least fixpoint computation,

$$R = \mu Z.I \vee post(T, Z) \ .$$

Here, μ denotes the least fixpoint and Z is an auxiliary variable for iteration. Reachability fixpoint computation starts from the initial state set and repeatedly adds the post-condition of already reached states until convergence. Given a transition relation T and a set $D(X, P)$ of states, the post-condition or *image* of D with respect to T consists of all the successors of D in the state transition graph. Let $f_{(X/X')}$ denote the substitution of X' variables in f by the corresponding X. Then

$$post(T, D) = \left(\exists_{X,P} T \wedge D\right)_{(X/X', P/P')} \ .$$

Partitioned transition relations for symbolic image computation were proposed in conjunctive forms. In Narayan et al. [1997], multiple variable orders were used together with partitioned BDDs [Bryant 1986] to reduce the peak memory usage in reachability analysis. However, these works were not targeted for handling software models. In general, image computation based on a disjunctively partitioned transition relation is effective only if a good partition can be efficiently computed. For hardware verification, previous applications of disjunctively partitioned transition relation were not successful, since creating a good disjunctive partition itself is a non-trivial task. Our work demonstrates that disjunctive partitioning is naturally suited for software models due to their sequential nature. Our new method differs from the prior work in the criteria we use for decomposition and in our software-specific simplifications.

Since we have disjunctive transition relation, the post-condition computation can be decomposed into a set of easier steps as follows,

$$\begin{aligned} post(T, D) &= \left(\exists_{X,P} \bigvee_{(b_i,b_j)\in E} (T_{ij} \wedge D)\right)_{(X/X',P/P')} \\ &= \bigvee_{(b_i,b_j)\in E} \left(\exists_{X,P} (T_{ij} \wedge D)\right)_{(X/X',P/P')}. \end{aligned}$$

Based on Equation 2, we can convert $T_{ij} \wedge D$ into DNF format. Let $T_{ij} \wedge D = \bigvee (f_{ij}^B \wedge f_{ij}^I \wedge f_{ij}^R)$, where f_{ij}^B, f_{ij}^I, and f_{ij}^R are sub-formulas in Boolean logic, Presburger arithmetic, and Boolean combination of real linear constraints, respectively. Let $X = X^B \cup X^I \cup X^R$, where X^B, X^I and X^R represent the set of Boolean, integer and real variables. The post-condition computation can be decomposed further as follows,

$$\begin{aligned} post(T, D) &= \bigvee_{(b_i,b_j)\in E} (\exists_{X^B,X^I,X^R,P} \bigvee \left(f_{ij}^B \wedge f_{ij}^I \wedge f_{ij}^R)\right)_{(X/X',P/P')} \\ &= \bigvee_{(b_i,b_j)\in E} \left(\bigvee \exists_{X^B,X^I,X^R,P} \left(f_{ij}^B \wedge f_{ij}^I \wedge f_{ij}^R\right)\right)_{(X/X',P/P')} \\ &= \bigvee_{(b_i,b_j)\in E} \left(\bigvee \left(\exists_{X^B,P} f_{ij}^B\right) \wedge \left(\exists_{X^I} f_{ij}^I\right) \wedge \left(\exists_{X^R} f_{ij}^R\right)\right)_{(X/X',P/P')}. \end{aligned}$$

Computing post-condition subsets individually is often more efficient than computing the entire set on a monolithic transition relation, since it reduces the peak size of symbolic representations for intermediate products. For BDD-based operations, the disjunctive decomposition can reduce the peak live BDD nodes [Wang et al. 2007]; for polyhedron-based operations, it can reduce the peak number of polyhedrons and the number of linear constraints.

7.2 The Reach_Frontier Strategy

Let R^{i-1} and R^i represent the sets of reachable states at two consecutive steps; in computing R^{i+1}, one can use $post(T, R^i \setminus R^{i-1})$ instead of $post(T, R^i)$ if the symbolic representation of $(R^i \setminus R^{i-1}) = R^i \wedge \not R^{i-1}$ is smaller than that of R^i. In BDD based symbolic model checking, the set $R^i \setminus R^{i-1}$ is called the *frontier set* [Ranjan et al. 1995]. However, in order to detect convergence, one still needs to store the entire reachable state set R^i (in order to stop as soon as $R^{i+1} = R^i$).

We have observed that maintaining the entire reachable state set R^i at every iteration is costly. In symbolic model checking, it is a known fact that the size of symbolic representation of R^i often increases in the middle stages of fixpoint computation and then decreases when it is close to convergence. The case becomes even more severe with polyhedrons in our mixed representations, which is largely due to the fact that composite formula representation is not canonical —after being propagated through various branching and re-converging points, polyhedrons are fragmented more easily into smaller pieces.

We propose a specialized symbolic search strategy called REACH_FRONTIER to improve reachability fixpoint computation. The idea is to avoid storing the

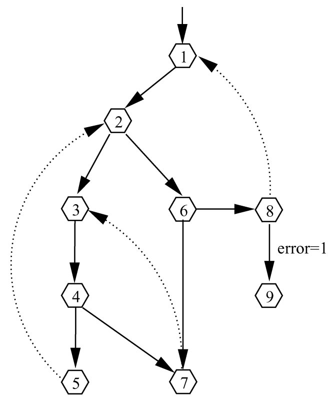

Fig. 5. Control flow graph of a C code and its dominator tree.

entire reachable state set at each iteration, but use an augmented frontier set to detect convergence. Given an LTG G, we say block b_i *dominates* block b_j if every path from the initial block to b_j goes through b_i. A *back edge* is an edge (b_j, b_i) whose head b_i dominates its tail b_j. Figure 5 shows the control flow graph of a C code on its left, where the dotted line indicate a back edge. To its right is the corresponding *dominator tree*, in which each block dominates only its descendants in the tree. There exist classic algorithms [Aho et al. 2006] to compute the set of dominators and back edges.

Let $E_{back} \subseteq E$ be the set of back edges in the LTG, whose removal will make the graph acyclic[2]. Let $Q_{bt} \equiv \vee_{(b_j,b_i)\in E_{back}}(P = b_j)$ be the subspace associated with the tails of the back edges. In Figure 5, for instance, the subspace is represented by $Q_{bt} \equiv (P = 3)$. If we record all the reached states falling inside Q_{bt}, which is $S = R \wedge Q_{bt}$, then the emptiness of the set $(F \setminus R \wedge Q_{bt})$ can be used to detect convergence.

Algorithm 1. Reachability computation with the REACH_FRONTIER approach

```
bool Reach_Frontier($T$,$I$,$Q_{Err}$,$Q_{bt}$)
{
    $F = I$;
    $S = I \wedge Q_{bt}$;
    while ($F \neq false$)
        if (($F \wedge Q_{Err}$) $\neq false$)
            return $false$;
        $F = (post(T, F) \setminus F) \setminus S$;
        $S = S \vee (F \wedge Q_{bt})$;
    return $true$;
}
```

[2]Removing back edges will make LTG acyclic if the graph is derived from programs with exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue and break statements. However, programs written using goto statements may cause the resulted LTG to be not *reducible*, in which the LTG may contain cycle even after remove back edges. If this case, we remove *retreating edges*, a super set of back edges. For more information on retreating edges and reducibility, please refer to [Aho et al. 2006].

Table I. Reachability computations with or without the REACH_FRONTIER approach

iteration	F	S	R
1	$P = 1 \wedge x = 0$	$false$	$P = 1 \wedge x = 0$
2	$P = 2 \wedge x = 0$	$false$	$P = 1 \wedge x = 0 \vee$ $P = 2 \wedge x = 0$
2	$P = 3 \wedge x = 1$	$P = 3 \wedge x = 1$	$P = 1 \wedge x = 0 \vee$ $P = 2 \wedge x = 0 \vee$ $P = 3 \wedge x = 1$
4	$P = 2 \wedge x = 1$	$P = 3 \wedge x = 1$	$P = 1 \wedge x = 0 \vee$ $P = 2 \wedge (x = 0 \vee x = 1) \vee$ $P = 3 \wedge x = 1$
5	$P = 4 \wedge x = 1$	$P = 3 \wedge x = 1$	$P = 1 \wedge x = 0 \vee$ $P = 2 \wedge (x = 0 \vee x = 1) \vee$ $P = 3 \wedge x = 1 \vee$ $P = 4 \wedge x = 1$
6	$false$	-	-

Our new reachability procedure in Algorithm 1 takes as parameters the symbolic model $\langle T, I \rangle$, the set of error states Q_{Err}, as well as the state subspace Q_{bt} associated with tail blocks of back edges E_{back}. We use set S to represent the subset of already reached states that falls inside Q_{bt}. When we define $Q_{bt} \equiv \vee_{b_i \in B}(P = b_i)$, the algorithm becomes the same as the ordinary reachability analysis procedure.

Table I illustrates the effectiveness of the reachability computation with the REACH_FRONTIER approach. The reachability computation is performed on the example shown in Figure 5. In the table Column 1 shows the index of the iterations, Column 2 gives the formula that represents the frontier set, Columns 3 and 4 show the formulas for the set of reachable states S at Q_{bt} using the REACH_FRONTIER approach, and the set of reachable states R in conventional reachability computation, respectively. It can be observed that the size of S is much smaller than that of R. Note that even the ordinary reachability analysis procedure may not converge since program verification in general is undecidable in the polyhedral abstract domain. However, what we can guarantee is that, our REACH_FRONTIER approach is able to terminate as long as the ordinary procedure terminates.

THEOREM 7.1. *Let D be the longest path starting from the entry block in the LTG after the removal of back edges. Then the* REACH_FRONTIER *procedure terminates with at most D more iterations after the conventional reachability analysis procedure terminates.*

PROOF. By definition, $S = R \wedge Q_{bt}$ where R is the set of reached states. Therefore at the i-th iteration we have $S^i = R^i \wedge Q_{bt}$. It follows that if $R^i \setminus R^{i-1}$ becomes empty, then $S^i \setminus S^{i-1}$ is also empty. The set F^i may not become empty immediately after $R^i \setminus R^{i-1}$ becomes empty , but it will never add any new state inside S^i. Therefore, the frontier set F is guaranteed to become empty after going through all the forward edges one more time. Since the LTG becomes a directed acyclic graph with a maximal depth D if we remove all the back edges in E_{back}, the REACH_FRONTIER procedure terminates with at most D more iterations after the conventional reachability analysis. □

7.3 The Reach_Lockstep Strategy

Our REACH_FRONTIER search strategy can significantly reduce the peak memory usage in the middle stages of fixpoint computation. However, there are still cases for which even the mixed representation of F^i becomes too large. When an LTG has multiple cycles of different lengths and the cycles are not well synchronized at the reconvergence points, new states (in frontier set) may easily scatter in a large number of basic blocks. Since this often means a larger number of polyhedrons (and more linear constraints), the gain by the REACH_FRONTIER strategy gradually evaporates.

To address this problem, we propose another search strategy called REACH_LOCKSTEP, which is an improvement of the REACH_FRONTIER procedure in Algorithm 1. The idea is to synchronize multiple cycles by controlling the time when new states are propagated through back edges. For this we bipartition the transition relation T into T_f and T_b, such that T_f consists of forward edges only and T_b consists of back edges only. We conduct reachability analysis using the REACH_LOCKSTEP approach, by first propagating the frontier set through T_f until convergence, and then feeding back the set $R \wedge Q_{bt}$ through T_b. Note that this may introduce some stuttering steps, where propagation from some cycles is delayed.

The new procedure in Algorithm 2 takes as inputs the symbolic model $\langle T_f, T_b, I\rangle$, the set of error states Q_{Err}, as well as the state subspace Q_{bt} associated with tail blocks of back edges. It terminates only when no new state is reached by post-condition computations on both T_f and T_b. Note that REACH_LOCKSTEP has the same effect with REACH_FRONTIER on straight-line code where there are no back edges. However, by synchronizing the propagation through back edges, we can significantly reduce the size of F. Note that with

Algorithm 2. Reachability computation with the REACH_LOCKSTEP approach.

```
bool Reach_Lockstep($T_f$,$T_b$,$I$,$Q_{Err}$,$Q_{bt}$)
{
    $F = I$;
    $S = S_{new} = I \wedge Q_{bt}$;
    while($F \neq false$)
        if (($F \wedge Q_{Err}) \neq false$)
          return $false$;
        $F = (post(T_f, F) \setminus F) \setminus S$;
        $S = S \vee (F \wedge Q_{bt})$;
        $S_{new} = S_{new} \vee (F \wedge Q_{bt})$;
        if ($F = \emptyset$)
          $F = post(T_b, S_{new}) \setminus S$;
          $S_{new} = \emptyset$;
    return $true$;
}
```

the Reach_Lockstep strategy, we may get longer counterexamples due to the addition of stuttering steps. This may be a disadvantage considering the fact that counterexamples may take more iterations to generate. However, we shall show that there are some examples on which the Reach_Frontier strategy takes much longer runtime or may not even finish in the allocated time; in these cases, the Reach_Lockstep strategy becomes a viable option.

8. EXPERIMENTS

We have implemented the new techniques on the F-Soft verification platform [Ivančić et al. 2005a; Ivančić et al. 2009]. We are able to evaluate the proposed techniques by comparing to the best known composite model checking algorithm in Yavuz-Kahveci et al. [2005], as well as pure Boolean level algorithm using BDDs and SAT. Our experiments were conducted on a workstation with 2.8 GHz Xeon processors and 4GB of RAM running Red Hat Linux 7.2. We set the CPU time limit to one hour for all runs.

Our benchmarks are control intensive C programs from public domain as well as industry (for example, device drivers, embedded software of portable devices). For all test examples, we check reachability properties. Program counter variables are modeled as Boolean type and pointer variables are modeled as integers. Other program variables are modeled according their types in the program.

Among the eleven test cases in Sections 8.1 and 8.2, *bakery* is a C model of Leslie Lamport's bakery protocol; *tcas* is an air traffic control and avionic system; *ppp* is C public domain implementation of the Point-to-Point protocol. The examples starting with *mcf* are from an industry embedded software of a portable device, for which we only have the verification models but no source code information (such as the lines of C code). The *ftpd* examples are from the FTP daemon code in Linux. The benchmarks in Section 8.3 are all industrial proprietary software programs.

8.1 Comparing Search Strategies

First, we evaluate the proposed techniques by comparing the performance of composite model checking with and without the new features (i.e., program-specific optimizations and search strategies).

The results are given in Table II, wherein for each test example, we list in Columns 1-4 the name, the lines of C code, the number of variables, and the number of blocks. Columns 5-8 compare the runtime performance of the four implementations, where *old* denotes the baseline algorithm, *live* denotes the live variable based simplification, *front* denotes the one augmented with the Reach_Frontier algorithm, and *lstep* denotes the Reach_Lockstep algorithm. Live variable analysis is applied in both new search strategies. On average about 70% of the program variables are eliminated in each basic block by the live variable analysis. We use T/O to indicate time out and M/O memory out. Columns 9-12 compare the peak number of linear equalities and inequalities used in Omega library. We omit the peak BDD sizes, since for these examples the BDD sizes are all very small.

Table II. Comparing Search Strategies in Reachability Fixpoint Computation Where T/O Stands for Time Out, M/O for Memory Out, and - for no Data

Test Program				Total CPU Time (s)				Peak GEQ Formulas			
name	loc	vars	blks	old	live	front	lstep	old	live	front	lstep
bakery	94	10	26	T/O	755	35	13	-	1518	264	128
tcas-1a	1652	59	133	T/O	T/O	T/O	374	-	-	-	17656
tcas-any	1652	65	215	T/O	T/O	T/O	415	-	-	-	14920
ppp	2623	91	720	T/O	T/O	T/O	51	-	-	-	3782
mcf1_as	-	92	92	2475	57	3	2	3394	355	45	45
mcf2_afr	-	126	155	T/O	91	7	5	-	344	110	165
mcf3_mrr	-	80	299	T/O	79	4	4	-	407	55	55
bftpd_useringrp	1115	242	13	12	1	1	1	829	6	4	4
bftpd_chkuser	2584	591	175	M/O	59	20	20	-	187	57	57
bftpd_chkshell	2931	674	364	M/O	576	47	48	-	995	358	358
bftpd_chkpasspwd	1166	547	463	M/O	681	760	760	-	579	2362	2362

Of the 11 examples, the baseline reachability algorithm can complete only 2, while the one with our optimizations and the new REACH_LOCKSTEP strategy completes all. For the cases where all methods can do a complete traversal, the performance gained by our optimizations can be several orders-of-magnitude. The results clearly show that exploiting sequentiality and variable locality is a key to making symbolic software model checking scalable. The comparison of the number of linear constraints at each iteration shows that our proposed techniques are also extremely effective in reducing the size of the mixed symbolic representation.

8.2 Comparison with Boolean Decision Procedures

We also give the comparison of the REACH_LOCKSTEP approach against pure Boolean-level symbolic decision procedures, including BDD-based model checking and SAT-based bounded model checking. Both of these two Boolean level decision procedures are based on matured techniques and have been fine-tuned for handling sequential programs [Ivančić et al. 2005b; Wang et al. 2007]. In particular, the BDD-based algorithm also uses decomposition and simplification based on live variables.

The results are given in Table III. Columns 1-3 give the name of the program, the number of bit variables in the Boolean model, and the sequential depth at which point all given properties can be decided. Columns 4-6 show, for each of the three methods, whether verification can be completed, and the maximum reached depth for the incomplete cases. Note that the BDD-based methods may time out before the transition relation is built, in which cases the maximum reached depth is 0. Finally, Columns 7-9 list the run time of each method in seconds. Note that the comparison may not be entirely fair for `mcf1_as` because the program contains non-linear operations. While BDD/SAT models nonlinear operations as bit-vector operations (maximum 32 bits), our approach approximates them. We put a star besides the name of the program.

Table III shows that the REACH_LOCKSTEP algorithm is the only method that can complete traversal in all examples. This, we believe, is due to the fact that

Table III. Comparing the REACH_LOCKSTEP Algorithm with Pure Boolean-level Algorithms, Where T/O Stands for Time Out

Test Program			Completed			CPU Time (s)		
name	bvars	depth	bdd-mc	sat-bmc	mix-ls	bdd-mc	sat-bmc	mix-ls
bakery	84	172	Y	(68)	Y	2	T/O	13
tcas-1a	307	119	Y	(103)	Y	433	T/O	374
tcas-any	362	181	(103)	(100)	Y	T/O	T/O	415
ppp	1435	132	Y	(84)	Y	687	T/O	51
mcf1_as⋆	500	192	Y	(98)	Y	150	T/O	2
mcf2_afr	508	211	Y	(60)	Y	110	T/O	5
mcf3_mrr	1212	148	Y	(43)	Y	190	T/O	4
bftpd_useringrp	1163	11	Y	Y	Y	1	1	1
bftpd_chkuser	5000	75	(0)	(70)	Y	T/O	T/O	20
bftpd_chkshell	7849	94	(0)	(44)	Y	T/O	T/O	48
bftpd_chkpasspwd	2826	147	(10)	(13)	Y	T/O	T/O	760

REACH_LOCKSTEP models the different behaviors of the system at the right levels of abstractions. Note that our method is significantly different from static analysis based on the polyhedral abstract domain [Halbwachs et al. 1997]. Although both methods use polyhedral representations, we are conducting an exact state space exploration—none of our results relies on convex hull based approximation or widening; when a property fails, we can generate a concrete counterexample trace.

We also checked the same test examples with a counterexample driven predicate abstraction algorithm [Jain et al. 2005]. Since the predicate abstraction procedure was designed for checking one property at a time, whereas all the other methods used in our experimental study can check multiple properties simultaneously in one run, a fair comparison was possible only on the first four examples (each of which has a single property). The results are as follows: (1) predicate abstraction completed *bakery*, *tcas-1a*, and *tcas-any* in one second, 137 seconds, and 836 seconds, respectively; (2) on *ppp* it timed out after one hour. This indicates that our exact composite reachability computation algorithm has already better performance than an advanced predicate abstraction procedure. Note that the procedure in Jain et al. [2005] builds upon a pure Boolean-level model. We believe it is possible to combine predication abstraction with our mixed symbolic algorithm, which we leave as a future work.

8.3 Experiments on Industrial Proprietary Software Programs

Finally, we applied our approach to verify real-world examples. All the examples in Table IV are industrial proprietary embedded software programs for portable devices. We substitute the real names by numbers in the first column. Column 2 shows the number of lines of code in the three examples. Columns 3 and 4 list the types and the number of properties under verification. There are three types of properties: `ary` checks array bound violations, `ptr` checks null pointer dereference, and `str` checks string related properties. All property monitors are automatically generated. Each property ID given one hour time limit.

Columns 5-8 compare the performance of F-SOFT with or without mixed symbolic analysis. After preprocessing and static analysis as outlined in Section 4,

Table IV. Experiments on Industrial Proprietary Software Programs

Test Program				F-Soft w/o Mix		F-Soft w Mix		Improvement	
Name	LOC	Type	#Prop	Proof	Witness	Proof	Witness	More Proof	More Witness
1	7669	ary	350	246	57	260	65	14	8
		ptr	173	128	37	132	27	4	0
		str	362	95	11	94	11	−1	0
2	1521	ary	2552	2220	7	2270	8	50	1
		ptr	1005	910	3	910	3	0	0
		str	39	35	4	35	4	0	0
3	5797	ary	938	758	79	825	85	67	6
		ptr	484	387	54	429	55	42	1
		str	215	200	0	207	7	7	7
Total				run time 16.2 hrs		run time 8.1 hrs		183	23

F-Soft without Mix uses SAT solver to perform bounded model checking (Columns 5-6), while F-Soft with Mix performs fixpoint reachability computation using the approach illustrated in this article (Columns 7-8). If a property is proved to be correct within the time limit, we find a `proof`. Note that although in general a proof cannot be obtained by analyzing under-approximated models, we can prove properties in this group of experiments because no pre-set bound is needed for the embedded software benchmarks that do not have recursive function calls. On the other hand, if a counter-example is reported within the time limit, we find a `witness` to show that the property does not hold in the program. If it cannot prove or find a witness within the one-hour time limit, F-Soft cannot solve the current property in time and moves on to verify the others. Columns 5 and 7 reports the number of properties that can be proved, and Columns 6 and 8 the number of properties with witnesses found. Columns 9-10 show the improvement (additional number of properties that can be solved within the time limit) gained by mixed symbolic analysis. Finally, the last row in Table IV summarizes the comparison.

F-Soft with Mix outperforms SAT-based F-Soft in most benchmarks. The one case where F-Soft with Mix is worse (benchmark 1 with string related properties) is due to the large number of branches in the benchmark, which causes large symbolic representations. In total, F-Soft with Mix solves 206 more properties. In addition, the total run time of F-Soft with Mix is 8.1 hours, compared with 16.2 hours used by F-Soft without Mix.

9. CONCLUSION

We have presented a symbolic model checking algorithm that combines multiple decision procedures for verifying sequential programs. We apply mixed symbolic representations to programs with rich data types and complex expressions, and develop optimizations and new symbolic search strategies to improve the scalability of model checking algorithms. Based on the algorithm, we are able to statically detect software bugs such as assertion failure, buffer overruns, use of uninitialized variables, memory leaks, locking rule violations, and division by zero, thus making software more secure against malicious applications. Our experimental results show that these proposed techniques can significantly reduce the run time and peak memory usage required in fixpoint computation. It also compares favorably to pure Boolean level decision procedures using BDDs and SAT.

For future work, we want to explore various approximate state space traversal algorithms and guided search algorithms to further improve the performance. So far the mixed symbolic approach is not suitable for numerical computation software. For example, our treatment of floating point arithmetic cannot handle underflow or overflow. In the future work we will develop modeling and analysis heuristics for such applications. We also plan to extend our method to handle concurrent software programs.

REFERENCES

AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. 2006. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

ANDERSON, P., REPS, T., TEITELBAUM, T., AND ZARINS, M. 2003. Tool support for fine-grained software inspection. *IEEE Softw. 20,* 4, 42–50.

ARMANDO, A., BENERECETTI, M., AND MANTOVANI, J. 2006a. Model checking linear programs with arrays. *Electr. Notes Theor. Comput. Sci. 144,* 3, 79–94.

ARMANDO, A., MANTOVANI, J., AND PLATANIA, L. 2006b. Bounded model checking of software using smt solvers instead of sat solvers. In *Proceedings of SPIN*. 146–162.

ASARIN, E., BOURNEZ, O., DANG, T., AND MALER, O. 2000. Approximate reachability analysis of piecewise-linear dynamical systems. In *Proceedings of the International Workshop on Hybrid Systems: Computation and Control*. Lecture Notes in Computer Science, vol. 1790, Springer-Verlag, 21–31.

BAGNARA, R., RICCI, E., ZAFFANELLA, E., AND HILL, P. M. 2002. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *Proceedings of the Static Analysis Symposium*.

BALL, T. AND RAJAMANI, S. K. 2000. Bebop: a symbolic model checker for Boolean programs. In *Proceedings of the SPIN Workshop*. Springer-Verlag, 113–130. Lacture Notes in Computer Science 1885.

BENSALEM, S., GANESH, V., LAKHNECH, Y., MUNOZ, C., OWRE, S., RUEB, H., RUSHBY, J., RUSU, V., SAIDI, H., SHANKAR, N., SINGERMAN, E., AND TIWARI, A. 2000. An overview of SAL. In *Proceedings of the 5th Langley Formal Methods Workshop*.

BEYER, D., HENZINGER, T. A., JHALA, R., AND MAJUMDAR, R. 2007. The software model checker Blast: Applications to software engineering. *International J. Softw. Tools for Tech. Transfer*.

BIERE, A., CIMATTI, A., CLARKE, E., AND ZHU, Y. 1999. Symbolic model checking without BDDs. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 1579, 193–207.

BRYANT, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput. 35,* 8, 677–691.

BULTAN, T., GERBER, R., AND LEAGUE, C. 2000. Composite model checking: verification with type-specific symbolic representations. *ACM Trans. Softw. Eng. Method. 9*, 1, 3–50.

BULTAN, T., GERBER, R., AND PUGH, W. 1997. Symbolic model checking of infinite state systems using presburger arithmetic. In *Proceedings of the 9th International Conference on Computer Aided Verification*. Springer-Verlag, London, UK, 400–411.

BURCH, J. R., CLARKE, E. M., AND LONG, D. E. 1991. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the Design Automation Conference*. San Francisco, CA, 403–407.

BUSH, W., PINCUS, J. D., AND SIELAFF, D. J. 2000. A static analyzer for finding dynamic programming errors. *Softw. Prac. Exper. 30,* 7, 775–802.

CABODI, G., CAMURATI, P., LAVAGNO, L., AND QUER, S. 1997. Disjunctive partitionining and partial iterative squaring: an effective approach for symbolic traversal of large circuits. In *Proceedings of the Design Automation Conference*. Anaheim, CA, 728–733.

CARDELLI, L. 1997. Type systems. In *Handbook of Computer Science and Engineering*.

CHAKI, S., CLARKE, E., GROCE, A., JHA, S., AND VEITH, H. 2003. Modular verification of software components in c. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

CLARKE, E., GRUMBERG, O., AND PELED, D. 2000. *Model Checking*. MIT Press.

CLARKE, E., KROENING, D., AND LERDA, F. 2004a. A tool for checking ANSI-C programs. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 2988, Springer, 168–176.

CLARKE, E., KROENING, D., AND LERDA, F. 2004b. A tool for checking ANSI-C programs. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. K. Jensen and A. Podelski, Eds. Lecture Notes in Computer Science, vol. 2988. Springer, 168–176.

CLARKE, E. M. AND EMERSON, E. A. 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the Workshop on Logics of Programs*. Lecture Notes in Computer Science, vol. 131, Springer-Verlag, Berlin. 52–71.

COUSOT, P. AND COUSOT, R. 1976. Static determination of dynamic properties of programs. In *Proceedings of the the 2$^{\text{nd}}$ International Symposium on Programming*.

DAS, M., LERNER, S., AND SEIGLE, M. 2002. Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

DAVIS, M., LOGEMANN, G., AND LOVELAND, D. 1962. A machine program for theorem proving. *Comm. ACM 5*, 394–397.

D. EVANS, J. GUTTAG, HORNING, J., AND TAN, Y. M. 1994. Lclint: A tool for using specifications to check codd. In *Proceedings of the SIGSOFT Symposium on the Foundations of Software Engineering*.

GAY, D., LEVIS, P., BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. 2003. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

HALBWACHS, N., PROY, Y. E., AND ROUMANOFF, P. 1997. Verification of real-time systems using linear relation analysis. *Formal Methods Syst. Des. 11*, 2, 157–185.

HENZINGER, T. A., HO, P.-H., AND WONG-TOI, H. 1995. HYTECH: the next generation. In *Proceedings of the IEEE Real-Time Systems Symposium*. 56–65.

HIND, M. AND PIOLI, A. 2001. Evaluating the effectiveness of pointer alias analyses. *Sci. Comput. Program. 39,* 1, 31–55.

HOLZMANN, G. 2002. Software analysis and model checking. In *Proceedings of the International Conference on Computer-Aided Verification*.

IVANČIĆ, F., SHLYAKHTER, I., GUPTA, A., GANAI, M., KAHLON, V., WANG, C., AND YANG, Z. 2005. Model checking C programs using F-Soft. In *Proceedings of the IEEE International Conference on Computer Design*. San Jose, CA, 297–308.

IVANČIĆ, F., YANG, Z., GANAI, M., GUPTA, A., AND ASHAR, P. 2009. Efficient sat-based bounded model checking for software verification. *Theor. Comput. Sci.* To Appear.

IVANČIĆ, F., YANG, Z., SHLYAKHTER, I., GANAI, M., GUPTA, A., AND ASHAR, P. 2005a. F-SOFT: Software verification platform. In *Proceedings of the Computer-Aided Verification*. Lecture Notes on Computer Science, vol. 3576, Springer-Verlag, 301–306.

JAIN, H., IVANČIĆ, F., GUPTA, A., AND GANAI, M. 2005. Localization and register sharing for predicate abstraction. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes on Computer Science, vol. 3440. Springer-Verlag, 394–409.

JAYARAMAN, G., RANGANATH, V. P., AND HATCLIFF, J. 2005. Kaveri: Delivering the indus java program slicer to eclipse. In *Proceedings of the Fundamental Approaches to Software Engineering (FASE)*. 269–272.

MCMILLAN, K. L. 1994. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA.

MINÉ, A. 2001. The octagon abstract domain. In *Proceedings of the Workshop on Analysis, Slicing, and Transformation (AST) in the Working Conference on Reverse Engineering (WCRE)*. IEEE, 310–319.

NARAYAN, A., ISLES, A. J., JAIN, J., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. L. 1997. Reachability analysis using partitioned ROBDDs. In *Proceedings of the International Conference on Computer-Aided Design*. 388–393.

NELSON, G. 1984. Combining satisfiability procedures by equality-sharing. *Contemp. Math. 29*, 201–211.

PUGH, W. 1991. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *ACM/IEEE Conference on Supercomputing*. ACM, 4–13.

QUIELLE, J. P. AND SIFAKIS, J. 1981. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Annual Symposium on Programming*.

RANJAN, R. K., AZIZ, A., BRAYTON, R. K., PLESSIER, B. F., AND PIXLEY, C. 1995. Efficient BDD algorithms for FSM synthesis and verification. In *Proceedings of the International Workshop on Logic Synthesis (IWLS'95)*.

RUGINA, R. AND RINARD, M. 2000. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 182–195.

SANKARANARAYANAN, S., IVANČIĆ, F., AND GUPTA, A. 2007. Program analysis using symbolic ranges. In *Proceedings of the Static Analysis Symposium*.

SANKARANARAYANAN, S., IVANČIĆ, F., SHLYAKHTER, I., AND GUPTA, A. 2006. Static analysis in disjunctive numerical domains. In *Proceedings of the Static Analysis Symposium*.

SÉMÉRIA, L. AND MICHELI, G. D. 1998. Spc: synthesis of pointers in c: application of pointer analysis to the behavioral synthesis from c. In *Proceedings of the International Conference on Computer-Aided Design*. 340–346.

SOMENZI, F. 1995. CUDD: CU Decision Diagram Package. University of Colorado at Boulder, ftp://vlsi.colorado.edu/pub/.

TIP, F. 1995. A survey of program slicing techniques. *J. Program. Lang. 3*, 121–189.

VISSER, W., HAVELUND, K., BRAT, G., AND PARK, S. 2000. Model checking programs. In *Proceedings of the International Conference on Automated Software Engineering*. 3–12.

WANG, C., YANG, Z., GUPTA, A., AND IVANČIĆ, F. 2006. Whodunit? causal analysis for counterexamples. In *Proceedings of the 4th International Symposium on Automated Technology for Verification and Analysis*.

WANG, C., YANG, Z., IVANCIC, F., AND GUPTA, A. 2007. Disjunctive image computation for software verfication. *ACM Trans. Des. Autom. Electr. Syst. 12,* 2.

YANG, Z., WANG, C., IVANČIĆ, F., AND GUPTA, A. 2006. Mixed symbolic representations for model checking software programs. In *ACM/IEEE International Conference on Formal Methods and Models for Codesign (Memocode)*.

YAVUZ-KAHVECI, T., BARTZIS, C., AND BULTAN, T. 2005. Action language verifier, extended. In *Proceedings of the Conference on Computer Aided Verification*. Lecture Notes in Computer Science, vol. 3576, Springer-Verlag, 413–416.

YAVUZ-KAHVECI, T. AND BULTAN, T. 2003. A symbolic manipulator for automated verification of reactive systems with heterogeneous data types. *Int. J. Softw. Tools Tech. Trans. 5,*1, 15–33.

YAVUZ-KAHVECI, T., TUNCER, M., AND BULTAN, T. 2001. A library for composite symbolic representations. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

ZAKS, A., YANG, Z., SHLYAKHTER, I., IVANČIĆ, F., CADAMBI, S., GANAI, M. K., GUPTA, A., AND ASHAR, P. 2008. Bitwidth reduction via symbolic interval analysis for software model checking. *IEEE Trans. Comput. Aid. Des. Integr. Circ. Syst., 27*, 8.

Received November 2007; revised July 2008; accepted August 2008