Mixed Symbolic Representations for Model Checking Software Programs

Zijiang Yang [1] Chao Wang [2] Aarti Gupta [2] Franjo Ivančić [2]

1 Western Michigan University, Kalamazoo, Michigan, USA

2 NEC Laboratories America, Princeton, New Jersey, USA

Abstract

We present an efficient symbolic search algorithm for software model checking. The algorithm combines multiple symbolic representations to efficiently represent the transition relation and reachable states and uses a combination of decision procedures for Boolean and integer representations. Our main contributions include: (1) mixed symbolic representations to model C programs with rich data types and complex expressions; and (2) new symbolic search strategies and optimization techniques specific to sequential programs that can significantly improve the scalability of model checking algorithms. Our controlled experiments on real-world software programs show that the new symbolic search algorithm can achieve several orders-of-magnitude improvements over existing methods. The proposed techniques are extremely competitive in handling sequential models of non-trivial sizes, and also compare favorably to popular Boolean-level model checking algorithms based on BDDs and SAT.

1 Introduction

Model checking as an automatic verification technique has been successfully used in the design of complex circuits and communication protocols [10, 20]. The procedure normally uses an exhaustive search of the state space of the considered system to determine whether a specification is true or false. Various symbolic representation and manipulation techniques [20, 6] have been proposed to improve the scalability of the procedure. 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 [26]. Existing symbolic model checking tools in this category, including [4, 11, 18], 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) [7] and SAT [12] to perform verification. Although modeling all variables as bit-vectors is accurate, such a high precision approach is often not needed and may generate models of very large sizes.

In [9, 28], Bultan *et al.* proposed a composite symbolic representation in an infinite-state model checker by combining the relative strengths of two symbolic representations: they used BDDs to represent Boolean formulas and union of polyhedrons to represent 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 [9, 28], 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 paper, we follow the general framework of [9, 28] 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. The number of program variables is often orders-of-magnitude larger than in previous studies [9]. 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 [15, 3]. 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 disjunctive set of polyhedrons; the union of two state sets is approxi-

mated into their convex union. Since a convex hull is often expensive to compute, this approach is also known to have scalability problems. The Symbolic Analysis Laboratory (SAL) [5] also provides a method for combining different decision procedures. However, it is different from our approach in the sense that the different search engines 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. Word-level model for C programs has also been used in linear programs where program variables can range over a numeric domain [1, 2]. However, our work emphasizes the combination of different modeling techniques such that each domain can be solved by the most efficient verification engine.

We have implemented the proposed techniques in our C model checking tool F-SOFT [18, 17], and compare our new method with the related work [28] in a set of controlled experiments. Our experimental results show that the new algorithm significantly outperforms existing methods in terms of both CPU time and memory usage. We note that the performance gains achieved by our new method do not come from improvement of any elementary symbolic engines, but is a result of combining the individual engines *suitably* for the particular task of verifying sequential programs. Our experimental study also 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.

The remainder of this paper is organized as follows. We introduce our software modeling approach in Section 2, 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 3, we present our software specific optimization techniques in decomposing and minimizing the transition relation representations. In Section 4, 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 5 and then conclude in Section 6.

2 Preliminaries

In this section, we review the software modeling in F-SOFT [18, 17] relevant to the automatic construction of a mixed symbolic model. F-SOFT is a tool for analyzing safety properties in C programs, by checking whether certain labeled statements are reachable from an entry point of the program. A large set of programming bugs, such as array bound violations, use of uninitialized variables, memory leaks, locking rule violations, and division by zero, can be formulated into reachability problems by adding suitable property monitors to the given program.

2.1 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. Below are details relevant to the construction of a mixed symbolic model (for a comprehensive description of the transformations, please refer to [17]).

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 appropriate 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 (e.g., elements of an array) are given consecutive memory addresses.
- We perform a points-to analysis [16] 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 reads via pointers (pointer-deref), we adopt an approach from hardware synthesis [24] and for each pointer variable `p` create a new variable `STAR_p` representing the current value of `*p`. Each read of `*p` is then rewritten as simply a read of `STAR_p`. (Reads of the form `*(p+i)` continue to be handled as described earlier.) To keep `STAR_p` up-to-date, after each assignment `p=q` we add an *inferred assignment* `STAR_p = STAR_q`. Furthermore, we need to add *aliasing assignments* to the model that keep `STAR_p` up-to-date, when the value may have been changed by an assignment through `*q` or some other variable in `p`'s points-to set.

Unbounded Data, Recursion and Function. The C language specification does not bound heap or stack size, but our focus is on generating a bounded model only[1]. Therefore, we model the heap as a finite array, adding a simple implementation of `malloc()` that returns pointers into

[1] Our bounded 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.

```
int foo(int s){
  int t=s+2;
  if (t>6)
    t -= 3 ;
  else
    t--;
  return t;
}

void bar(){
  int x=3;
  int y=x-3;
  while (x<=4){
    y++ ;
    x = foo(x);
  }
  y = foo(y);
}
```

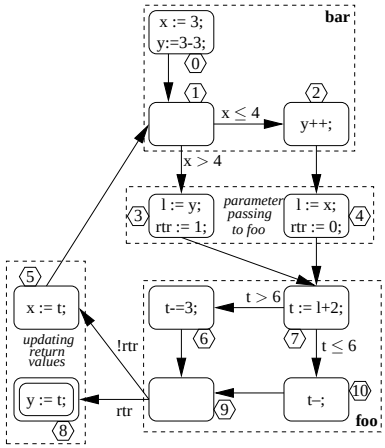

Figure 1. Sample code and its graph representation

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.

As a running example, Figure 1 shows a simplified control flow graph structure obtained from the C program on the left-hand side. The example pictorially shows how non-recursive function calls are included in the control flow of the calling function. 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 of the right-hand side graph is a basic block consisting of a set of parallel assignments. The edges are labeled by conditional expressions, e.g., the transition from block 1 to block 2 is guarded by $x \leq 4$. In case an edge is not labeled by any condition, the default condition is true. Finally, block 0 is the entry block and block 8 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 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 non-empty 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.*
- *$\delta : B \rightarrow 2^{\Sigma}$ is a labeling function that labels each basic block with a set of parallel assignments, where Σ represents the set of all possible C expressions.*
- *$\theta : E \rightarrow \Sigma$ is a labeling function that labels each edge with a conditional C expression. These conditionals are based on 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 $Q = B \times \mathcal{X}$. we define a state to be a tuple $q = (\vec{b}, \vec{x}) \in Q$. 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 $x_1, \ldots, x_n \leftarrow e_1, \ldots, e_n$, where $\{x_1, \ldots, x_n\} \subseteq X$ and $\{e_1, \ldots, e_n\} \subseteq \Sigma$.

For checking reachability properties, we define a subset $B_{Err} \subseteq B$ of blocks to be unsafe; model checking is then used to prove or disprove that these basic blocks 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 $(\vec{b_0}, \vec{x_0}), \ldots, (\vec{b_k}, \vec{x_k})$ such that $(\vec{b_0}, \vec{x_0}) \in Q_0$ and for all $0 \leq i < k-1$, $(\vec{b_i}, \vec{x_i}) \rightarrow (\vec{b_{i+1}}, \vec{x_{i+1}})$. A counterexample is a path that ends in an unsafe basic block $\vec{b_k} \in B_{Err}$.

2.2 Composite Symbolic Formulas

We now review the definition of composite symbolic formulas and the corresponding set theoretic operations. 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 $\mathbf{x}, \mathbf{a_i} \in \mathbb{Z}^n$ are vectors and $b \in \mathbb{Z}$ is a scalar. Similarly, a real linear constraint is denoted by $\mathbf{c_i y} \leq d$, where $\mathbf{y}, \mathbf{c_i} \in \mathbb{R}^n$ are vectors and $d \in \mathbb{R}$ is a scalar. A formula in Presburger arithmetic is an arbitrary Boolean combination of integer linear constraints, which can be represented as a union of polyhedrons.

Definition 2 (c.f. [9])**.** *The composite symbolic formula F is defined as follows,*

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

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

The above definition extends the one in [9] 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 [8]. A composite symbolic formula can be put into

the *Disjunctive Normal Form (DNF)* as follows

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

Assume that all expressions in a composite formula are type-consistent, then subformulas of different types share no common variables.

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 engines, and assemble the result back into DNF. One can use CUDD [25] to represent Boolean formulas, the Omega library [22] to represent Presburger formulas, and the Parma Polyhedral Library [13] 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 ($\exists$).

The union of two composite formulas is simply the union of their subformulas. The conjunction of two composite formulas is the union of pair-wise conjunctions of their subformulas. Let $F = \bigvee_{i=1}^{n_F} F_i^B \wedge F_i^I \wedge F_i^R$ and $G = \bigvee_{j=1}^{n_G} G_j^B \wedge G_j^I \wedge G_j^R$; then

$$F \wedge G = \bigvee_{i=1,j=1}^{n_F,n_G} (F_i^B \wedge G_j^B) \wedge (F_i^I \wedge G_j^I) \wedge (F_i^R \wedge G_j^R) \ .$$

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. The negation of a composite formula can be implemented in a way similar to conjunction. Note that the DNF representation is not canonical, and there are heuristic algorithms [9] to make the result more compact. Although the number of mixed terms can be as large as $(n_F \times n_G)$ for conjunction (3^{n_F} for negation), such a worst-case blowup rarely happens in our application domain.

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,

$$\exists v^B, v^I, v^R \,.\, F = \bigvee_{i=1}^{n_F} (\exists v^B \,.\, F_i^B) \wedge (\exists v^I \,.\, F_i^I) \wedge (\exists v^R \,.\, F_i^R) \ ,$$

due to the fact that v^B, v^I, and v^R are disjoint sets.

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);[2] 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 $T(\vec{b}, \vec{x}, \vec{b}', \vec{x}')$ is true if and only if there is a transition from the state $(\vec{b}, \vec{x})$ to the state $(\vec{b}', \vec{x}')$. Similarly, the evaluation of function $I(\vec{b}, \vec{x})$ is true if and only if $(\vec{b}, \vec{x})$ 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 $P = \{p_1, p_2, ..., p_k\}$ so that, for instance, $(P = 5)$ is encoded as $(p_3 \wedge \neg p_2 \wedge p_1)$. 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. 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.

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

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 engines clearly differentiates this modeling approach from the Nelson-Oppen framework for cooperating decision procedures [21].

If the above 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 nondeterminstic *pseudo input* variable of the suitable type. During post-condition computa-

[2] 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.

tion 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 a range analysis [17], which determines a conservative value range of each variable in the given program. Also, the user can "sharpen" the over-approximation with the help of pre- and post-conditions (or asserts and assumes) in such cases.

3 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.

3.1 Disjunctive Transition Relations

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

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

where t_{ij}^c denotes the transition of control flow from b_i to 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,

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

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,

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

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 LTG.

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

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.

Given a transition relation T and a set Z of states, the post-condition or *image* of Z with respect to T consists of all the successors of Z in the state transition graph. Let $f_{(X/X')}$ denote the substitution of X' variables in f by the corresponding X. Then

$$post(T, Z) = (\exists X, P \,.\, T \wedge Z)_{(X/X', P/P')} \ .$$

The post-condition computation can be decomposed into a set of easier steps as follows,

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

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.

Reachability analysis is a least fixpoint computation,

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

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.

3.2 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

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:	{ }

Figure 2. An example of live variables.

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 [27], 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.

Definition 3. *Variable $x \in X$ is* live *in block $b_i \in B$ if and only if there exists an execution path from b_i to b_j such that,*

- *x appears either in $\theta(b_j, b_k)$ or in the right-hand side of an assignment in $\delta(b_j)$;*
- *x does not appear in the left-hand side of an assignment in any block between b_i and b_j along the path.*

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 the above 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.

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 Fig. 2 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. (Fig. 2 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 Fig. 2, 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 [27], the live variable information was used to existentially quantify dead variables out of image results at each iteration. In this paper, 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 [27], 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 Fig. 2 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.

4 Specialized Symbolic Search Strategies

Let R^{i-1} and R^i be two reachable state sets 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 $(R^i \setminus R^{i-1})$ 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* [23]. However, in order to detect convergence, one still needs to store the entire reachable state set R^i (in

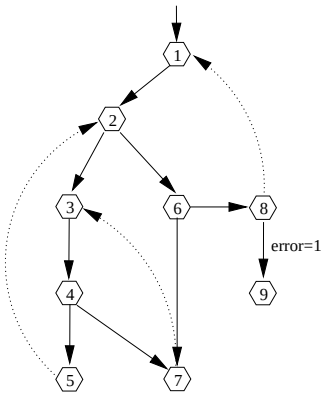

Figure 3. Removing back edges to break cycles

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.

4.1 The Frontier Strategy

We propose a specialized symbolic search strategy called REACH_FRONTIER to improve reachability fixpoint computation. The idea is to avoid storing the entire reachable state set at each iteration, but use an augmented frontier set to detect convergence. In reachability computation, a frontier set consists of all the new states reached at the previous iteration; that is, $F^0 = I, F^i = post(T, F^{i-1}) \setminus F^{i-1}$. For straight-line code (without loops and backward gotos in the LTG), we can declare convergence when F^i becomes empty (and the set is guaranteed to become empty after $|B|$ iterations). However, in the presence of loops (cycles), the frontier set may never become empty—an example would be any program with an infinite loop.

In the presence of cycles, we need to identify a set of back edges $E_{back} \subseteq E$ in the LTG, whose removal will make the graph acyclic (an example is given in Fig. 3). Let $Spa \subseteq Q$ denote the state subspace associated with tail blocks of those back edges. In Fig. 3, for instance, the subspace is represented by $Spa = (P = 5 \vee P = 7 \vee P = 8)$. If we record all the reached states falling inside Spa, which is $S = R \cap Spa$, then the emptiness of the set $(F \setminus R \cap Spa)$ can be used to detect convergence.

Algorithm 1 REACH_FRONTIER(T,I,Err,Spa)

```
1:  F = I;
2:  S = I ∩ Spa;
3:  while F ≠ ∅ do
4:    if (F ∩ Err) ≠ ∅ then
5:      return false;
6:    end if
7:    F = (post(T, F) \ F) \ S;
8:    S = S ∪ (F ∩ Spa);
9:  end while
10: return true;
```

Our new reachability procedure in Algorithm 1 takes as parameters the symbolic model $\langle T, I \rangle$, the state subspace $Err = B_{Err} \times X$ associated with a set of error blocks B_{Err}, as well as the state subspace Spa associated with tail blocks of back edges E_{back}. We use set S to represent the subset of already reached states that falls inside Spa. When we define $Spa = \mathsf{true}$, the algorithm becomes the same as the ordinary reachability analysis procedure.

Finally, we 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 Frontier procedure is able to terminate as long as the ordinary procedure terminates.

Theorem 1. *Let D be the longest path starting from the entry block in the LTG after the removal of back edges. Then* REACH_FRONTIER *terminates with at most D more iterations after the conventional reachability analysis procedure terminates.*

Note that by definition, we have $S = R \cap Spa$ and therefore $S^i = R^i \cap Spa$. It follows that if $R^i \setminus R^{i-1}$ is empty, then $S^i \setminus S^{i-1}$ is also empty. The set F^i may not become empty immediately after $R^i \setminus R^{i-1}$, 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. Also note that if we remove all the back edges in E_{back}, the LTG becomes a directed acyclic graph with a maximal depth D.

4.2 The Lock-Step Strategy

Our 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 re-convergence 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 our frontier strategy gradually evaporates.

To address this problem, we propose another search strategy called REACH_LOCKSTEP, which is an improvement of the 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 bi-partition 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 in lock-step, by first propagating the frontier set through T_f until convergence, and then feeding back the set $R \cap Spa$ through T_b. Note that this may introduce some stuttering steps, where propagation from some cycles is delayed.

Algorithm 2 REACH_LOCKSTEP(T_f,T_b,I,Err,Spa)

1: $F = I$;
2: $S = S_{new} = (I \cap Spa)$;
3: **while** $F \neq \emptyset$ **do**
4: **if** $(F \cap Err) \neq \emptyset$ **then**
5: return false;
6: **end if**
7: $F = (post(T_f, F) \setminus F) \setminus S$;
8: $S = S \cup (F \cap Spa)$;
9: $S_{new} = S_{new} \cup (F \cap Spa)$;
10: **if** $F = \emptyset$ **then**
11: $F = post(T_b, S_{new}) \setminus S$;
12: $S_{new} = \emptyset$;
13: **end if**
14: **end while**
15: return true;

The new procedure in Algorithm 2 takes as inputs the symbolic model $\langle T_f, T_b, I \rangle$, the state subspace Err associated with error blocks, as well as the state subspace Spa 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. By synchronizing the propagation through back edges, we can significantly reduce the size of F. Note that with the lock-step 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 frontier strategy takes much longer runtime or may not even finish in the allocated time; in these cases, the lockstep strategy becomes a viable option.

5 Experiments

We have implemented the new techniques on the F-SOFT verification platform [18, 17]. Our implementation builds upon CUDD [25], the Omega library [22], and the Parma Polyhedral library [13]. At this time the integration with CUDD and Omega has been completed, whereas the interface to Parma is still work in progress. We are able to evaluate the proposed techniques by comparing to the best known composite model checking algorithm in [28], 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 (e.g., device drivers, embedded software of portable devices). For all test examples, we check reachability properties expressing the absence of out-of-bound array and pointer accesses. Among the eleven test cases, *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.

5.1 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). We note that without all these new features, our implementation of the underlying composite model checking algorithm becomes comparable to the action language verifier of [28].

The results are given in Table 1, 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 frontier search strategy, and *lstep* denotes the lockstep strategy. 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.

Of the 11 examples, the baseline reachability algorithm can complete only 2, while the one with our optimizations and the new lock-step 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 reduc-

Table 1. Comparing search strategies in reachability fixpoint computation

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

ing the size of the mixed symbolic representation.

5.2 Comparison with Boolean Engines

We also give the comparison of *mix-lockstep* against pure Boolean-level symbolic engines, including BDD-based model checking and SAT-based bounded model checking. Both of these two Boolean level engines are based on matured techniques and have been fine-tuned for handling sequential programs [17, 27]. In particular, the BDD-based algorithm also uses decomposition and simplification based on live variables.

The results are given in Table 2. 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. Sometimes the comparison may not be entirely fair, since BDD/SAT models non-linear operations as bit-vector operations (maximum 32 bits), while the new method may approximate them. When approximation happens, we put a star in the last column.

Table 2 shows that our new algorithm *mix-lockstep* is the only method that can complete traversal in all examples. This, we believe, is due to the fact that mix-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 [14]. 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 counter-example driven predicate abstraction algorithm [19]. 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 1 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 [19] 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.

6 Conclusions

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 significantly richer data types and more complex expressions, and develop optimizations and new symbolic search strategies to improve the scalability of model checking algorithms. 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 search engines using BDDs and SAT. For future work, we want to explore various approximate state space traversal algorithms and extend our method to handle concurrent software programs.

References

[1] A. Armando, M. Benerecetti, and J. Mantovani. Model checking linear programs with arrays. *Electr. Notes Theor.*

Table 2. Comparing Mix-LockStep with Pure Boolean-level algorithms

Test Program			Completed			CPU Time (s)			non
name	bvars	depth	bdd-mc	sat-bmc	mix-ls	bdd-mc	sat-bmc	mix-ls	-lin
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	$\star$
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	

Comput. Sci., 144(3):79–94, 2006.

[2] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using smt solvers instead of sat solvers. In *SPIN*, pages 146–162, 2006.

[3] E. Asarin, O. Bournez, T. Dang, and O. Maler. Approximate reachability analysis of piecewise-linear dynamical systems. In *Hybrid Systems: Computation and Control*, pages 21–31. Springer-Verlag, 2000. LNCS 1790.

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

[5] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueb, J. Rushby, V. Rusu, H. Saidi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In *Proc. of the Fifth Langley Formal Methods Workshop*, Jan. 2000.

[6] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, Mar. 1999. LNCS 1579.

[7] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computer*, C-35(8):677–691, Aug. 1986.

[8] T. Bultan, R. Gerber, and C. League. Composite model checking: Verification with type-specific symbolic representations. *ACM Transactions on Software Engineering and Methodology*, 9(1):3–50, Jan 2000.

[9] T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *International Conference on Automated Software Engineering*, pages 382–386, 2001.

[10] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2000.

[11] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004. LNCS 2988.

[12] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[13] T. C. S. Group. *The Parma Polyhedra Library*. University of Parma, Italy, http://www.cs.unipr.it/ppl/.

[14] N. Halbwachs, Y. E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in Systems Design*, 11(2):157–185, 1997.

[15] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: the next generation. In *IEEE Real-Time Systems Symposium*, pages 56–65, 1995.

[16] M. Hind and A. Pioli. Evaluating the effectiveness of pointer alias analyses. *Sci. Comput. Program.*, 39(1):31–55, 2001.

[17] F. Ivančić, I. Shlyakhter, A. Gupta, M. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking C programs using F-Soft. In *IEEE International Conference on Computer Design*, pages 297–308, San Jose, CA, Oct. 2005.

[18] F. Ivančić, Z. Yang, I. Shlyakhter, M. Ganai, A. Gupta, and P. Ashar. F-SOFT: Software verification platform. In *Computer-Aided Verification*, pages 301–306. Springer-Verlag, 2005. LNCS 3576.

[19] H. Jain, F. Ivančić, A. Gupta, and M. Ganai. Localization and register sharing for predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 394–409. Springer-Verlag, 2005. LNCS 3440.

[20] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.

[21] G. Nelson. Combining satisfiability procedures by equality-sharing. *Contemporary Mathematics*, 29:201–211, 1984.

[22] W. Pugh and et al. *The Omega Project*. University of Maryland, http://www. cs.umd.edu/projects/omega/.

[23] R. K. Ranjan, A. Aziz, R. K. Brayton, B. F. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. Presented at IWLS95, May 1995.

[24] L. Séméria and G. D. Micheli. Spc: synthesis of pointers in c: application of pointer analysis to the behavioral synthesis from c. In *International Conference on Computer-aided design*, pages 340–346, 1998.

[25] F. Somenzi. *CUDD: CU Decision Diagram Package*. University of Colorado at Boulder, ftp://vlsi.colorado.edu/pub/.

[26] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *International Conference on Automated Software Engineering*, pages 3–12, 2000.

[27] C. Wang, Z. Yang, F. Ivancic, and A. Gupta. Disjunctive image computation for emebedded software verification. In *Design, Automation and Test in Europe (DATE'06)*, Munich, Germany, Mar. 2006.

[28] T. Yavuz-Kahveci, C. Bartzis, and T. Bultan. Action language verifier, extended. In *Computer Aided Verification*, pages 413–416. Springer-Verlag, July 2005. LNCS 3576.