Disjunctive Image Computation for Software Verification

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

Existing BDD-based symbolic algorithms designed for hardware designs do not perform well on software programs. We propose novel techniques based on unique characteristics of software programs. Our algorithm divides an image computation step into a disjunctive set of easier ones that can be performed in isolation. We use hypergraph partitioning to minimize the number of live variables in each disjunctive component, and variable scopes to simplify transition relations and reachable state subsets. Our experiments on nontrivial C programs show that BDD-based symbolic algorithms can directly handle software models with a much larger number of state variables than for hardware designs.

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

General Terms: Verification, Algorithms

Additional Key Words and Phrases: Model checking, reachability analysis, image computation, binary decision diagram, formal verification

ACM Reference Format:

Wang, C., Yang, Z., Ivančić, F., and Gupta, A. 2007. Disjunctive image computation for software verification. ACM Trans. Des. Autom. Electron. Syst. 12, 2, Article 10 (April 2007), 26 pages. DOI = 10.1145/1230800.1230802 http://10.1145/1230800.1230802

A preliminary version of this article was presented at the *DATE 2006 Conference* [Wang et al. 2006].

Authors' addresses: C. Wang, F. Ivančić, and A. Gupta, NEC Laboratories America, 4 Independence Way, Suite 200, Princeton, NJ 08540; Z. Yang (contact author), Department of Computer Science, Western Michigan University, 1903 W. Michigan Ave., Kalamazoo, MI 49008-5200; email: zijiang.yang@wmich.edu.

1. INTRODUCTION

Model checking [Clarke and Emerson 1981; Quielle and Sifakis 1981] is a formal method for proving that a finite-state model satisfies a user-specified temporal logic property. With the development of symbolic-state space traversal algorithms based on binary decision diagrams (BDDs [Bryant et al. 1986]), symbolic model checking [Burch et al. 1990; McMillan 1994] has become a widely accepted technique in hardware verification, and has shown promise for verifying embedded software programs, as well [Ball and Rajamani 2000]. In this article, we consider verifying C programs that include integer arithmetic, pointers, arrays, function calls, and bounded memory allocation. Although program verification in general is undecidable, the problem becomes decidable under certain conditions. Here we consider the case where the number of recursive function calls and the data size are bounded. With these assumptions, we can always build a finite state model from the software program and apply model checking.

In practice, both recursive functions and dynamic memory allocation are strongly discouraged in embedded software programs due to limited memory capacity. Even for software systems that have unbounded recursions and data (therefore an unbounded number of states), we can build finite-state verification models by abstraction of the original systems. In this case, model checking is still applicable, although the result is often conservative. For instance, false negatives may be introduced during an overapproximate abstraction.

In symbolic model checking, the transition relation of the model and sets of reachable states are represented symbolically as Boolean functions, which in turn can be represented by BDDs. The complexity of symbolic traversal algorithms depends more directly on the BDD sizes, rather than the actual number of states they represent. Therefore, the search for heuristics to reduce the BDD sizes has been one of the major research topics in symbolic model checking. Image computation is the core computation in symbolic model checking. Given a state transition system, image computation is used to find all the successors of a given set of states according to a set of transitions. The performance of this computation depends heavily on the size of the BDDs that represents the set of states, transition relations, and intermediate products created during the computation. Although BDD-based symbolic image computation has been extensively studied in past decades [Coudert et al. 1989; Burch et al. 1991; Ranjan et al. 1995; Moon et al. 2000; Chauhan et al. 2001; Jin et al. 2002; Wang et al. 2003], most of these existing algorithms were developed mainly for hardware systems.

Software models have some characteristics that are significantly different from hardware models. For instance, software models often have larger sequential depths and significantly more state variables. Their state variables also have a higher degree of locality, since most program variables are effective only in parts of the program. At any program location, only a limited number of program variables can change their values. Note that this argument remains valid even for analysis of concurrent software, since in the interleaving execution model [Holzmann and Peled 1994], only one thread is assumed to be active at each point of time. In contrast, state variables (or latches) in hardware are updated simultaneously at every clock cycle and therefore cannot be easily localized. Due to these differences, even the most fine-tuned symbolic image computation algorithms in the area of hardware verification do not work well on software models. To efficiently handle software models, the symbolic model checking engine and image computation algorithm in particular need to be reengineered with these characteristics in mind.

In this article, we propose a new symbolic image computation algorithm that exploits the unique characteristics of software models. It disjunctively decomposes the computation into a set of steps that can be performed in isolation on submodules. Breaking the expensive computation into a set of less expensive ones can significantly reduce the peak memory size during image computation. In addition, we derive the variable live scope information of the software model using a static analysis. Variable live scope information can be used to minimize the transition relation and to improve the performance of symbolic fixpoint computation. Our algorithm for creating the submodules is also geared towards exploitation of variable locality. Using a hypergraph partitioning heuristic, we are able to produce a small set of submodules, and at the same time minimize the number of live variables. We further improve the performance by preventing irrelevant variables from appearing in the transition relations, and by existentially quantifying dead variables from the reachable state subsets.

We have implemented and evaluated our new algorithm using the public domain symbolic model checker VIS [Brayton et al. 1996] as well as our program verification tool F-Soft [Ivančić et al. 2005, 2004]. However, for the purpose of conducting controlled experiments, we have used VIS as the performance comparison platform for evaluating our new algorithm. Our benchmarks are typical embedded C programs from the public domain as well as industry, including Linux device drivers, network application software, and programs embedded in portable devices. We will demonstrate on this set of benchmark programs that our new algorithm outperforms the best known conjunctive image computation algorithms in terms of both CPU time and memory usage. The improvement is both consistent and significant, namely, typically orders of magnitude. Although previous experience with hardware verification shows that BDD-based methods often lose robustness when the number of state variables of the model exceeds 100–200, our work demonstrates that with enough domain-specific optimizations, BDD-based symbolic model checking can *directly handle* software models with thousands of state variables.

The rest of this article is organized as follows. After reviewing the related work, we start by providing a brief background on symbolic model checking in Section 3. In Section 4 we describe our approach on modeling and verifying C programs. We present our disjunctive image computation algorithm in Section 5, followed by the application of relevant and live variables to simplify transition relations and reachable state subsets in Section 6. In Section 7, we present the use of the hypergraph-based partitioning algorithm to decompose the model in order to minimize the number of live variables shared by different submodules. We give our experimental results in Section 8 and then conclude in Section 9.

2. RELATED WORK

Partitioned transition relations for symbolic image computation were proposed in Burch et al. [1991] and Cabodi et al. [1997] in both disjunctive and conjunctive forms. In Narayan et al. [1997], multiple variable orders were used together with partitioned BDDs [Bryant et al. 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 nontrivial task. Our work demonstrates that disjunctive partitioning is naturally suited for software models due to their sequential nature. Our new method differs from prior work in terms of the criteria we use for decomposition and our software-specific simplifications.

Edwards, Ma, and Damiano [Edwards et al. 2001] applied a commercial model checker to software by synthesizing C programs into circuits. However, only very small programs can be directly verified. Although they pointed out that model checking algorithms must be reengineered, they did not provide any solution. Ball and Rajamani [2000] presented a tool for verifying Boolean programs abstracted from C code. Since predicate abstraction [Graf and Saïdi 1997; Das et al. 1999; Ball et al. 2001] has been applied, all the variables in their Boolean program are binary-state variables. Their underlying algorithm was a generalization of an interprocedural data flow analysis algorithm [Reps et al. 1996]. Their core idea is to efficiently compute *summaries* that record the input/output behavior of a procedure. Once a summary is computed, it is reused whenever the same input context arises at another call to the same procedure. Our work is different, since it builds upon symbolic model checking and therefore takes full advantage of the decade-long research in BDD-based algorithms and matured implementations (e.g., SMV [McMillan 1994] and VIS [Brayton et al. 1996]).

The algorithm by Barner and Rabinovitz [2003] was also based on symbolic model checking and used disjunctively partitioned transition relations. They start with a conjunctively partitioned transition relation and then disjunctively partition it so that each cluster corresponds to a subset of program variables. However, their partitioning method is significantly different from ours, since it requires a conjunctive transition relation and expensive *and-quantify* operations in order to build disjunctive transition relations. In contrast, we do not need the entire transition relation or quantification operations in order to build disjunctive transition relations. Furthermore, they did not exploit the variable locality, which we use for both transition relation decomposition and subsequent optimizations of symbolic fixpoint computation.

To summarize, the main contributions of our article are a new method for deriving and using disjunctively partitioned transition relations for software model checking, and further optimizations using variable locality information derived from static analysis of the given program.

3. BACKGROUND: MODEL CHECKING

This section provides a brief overview and terminology for model checking [Clarke et al. 2000], which is an algorithmic procedure for checking the correctness of a property in a model. In model checking, the design to be verified is represented as a finite state transition system, and the property is specified as a temporal logic formula. Temporal logics are useful for specifying dynamic behavior over time. In this article, we focus on a class of temporal logic properties called *safety properties*. Informally, a safety property states that something bad should never happen. According to Kupferman and Vardi [2001], a safety property can always be reduced to an invariant in the form of $\mathbf{AG}p$ by a compilation process. Here $\mathbf{AG}p$ means that on *all* ($\mathbf{A}$) paths of a system, the property p holds *globally* ($\mathbf{G}$) in each state. Such properties can be verified by an exhaustive traversal of the state space to check that p holds in every reachable state. When a safety property fails, the model checker can also produce a counterexample trace which shows the reachability of an error state (where p is false).

Model checking can be applied directly for verification of finite-state systems, such as sequential circuits and protocol controllers [Clarke et al. 1999]. In addition, by use of suitable abstractions, finite state models can also be extracted from infinite state systems for subsequent verification using model checking. This can be used in, for instance, real-time and hybrid system verification [Alur et al. 1993], parameterized system verification [Browne et al. 1999], and software program verification [Visser et al. 2000; Corbett et al. 2000]. Furthermore, model checking techniques have also been extended to a pushdown system [Boujjani et al. 1997; Ball and Rajamani 2000] which has a finite control but an unbounded stack, and therefore allows a direct modeling of recursion.

Most model checkers can be classified into two categories based on their underlying state space traversal algorithms. Explicit state model checkers such as SPIN [Holzmann 1997] use an explicit representation of states and transitions in the system, and enumerate the reachable states explicitly. They also utilize additional techniques such as state hashing [Wolper and Leroy 1993] and partial order reduction [Holzmann and Peled 1994]. The scalability issue in explicit state enumeration makes these checkers unsuitable for models with extremely large state space, although they have found practical success in verification of controllers.

In contrast, symbolic model checkers such as SMV [McMillan 1994] and VIS [Brayton et al. 1996] avoid an explicit enumeration of the state space by using symbolic representations of sets of states and transitions. They typically use BDDs to provide a canonical symbolic representation of Boolean formulas and efficient graph-based algorithms for symbolic manipulation. For many finite-state systems, especially digital circuits, symbolic representations can effectively capture the regularity in the state space, and therefore can significantly extend the ability of model checker to handle large systems. Several more recent works have applied SAT-based methods to software verification [Clarke et al. 2004; Ivančić et al. 2004; Cook et al. 2005].

4. SOFTWARE MODEL CHECKING

We first explain how our verification model is constructed from a source-code-level C program, and then review symbolic model checking in this context.

4.1 Software Model

We begin with a program in full-fledged C and apply 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. To follow are details relevant to the construction of a symbolic model (for a comprehensive description of the transformations, please refer to Ivančić et al. [2004]).

—*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 next.

—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 [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 expression using the numeric memory addresses assigned to the variables.

—For reads via pointers (pointer-deref), we adopt an approach from hardware synthesis [Séméria and Micheli 1998] 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. Therefore, 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. 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);
}
```

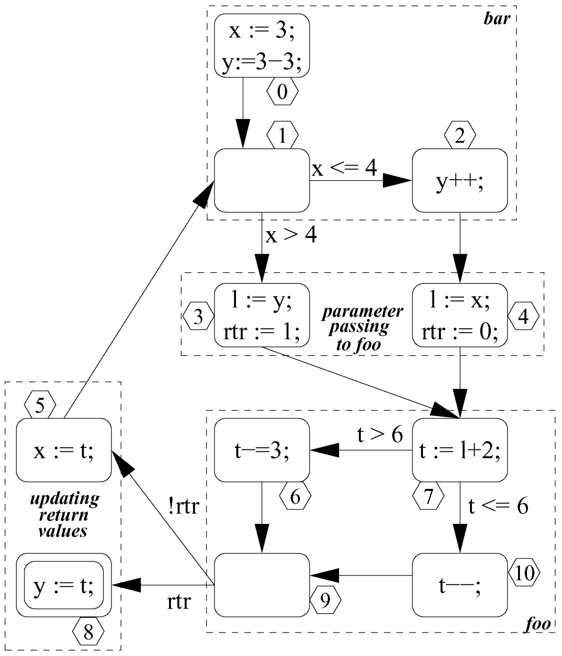

Fig. 1. Sample, code and its graph representation.

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 nonrecursive function calls are included in the control flow of the calling function. A preprocessing analysis determines that function `foo` is not called in any 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, for example, 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 4.1. 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.
—$E \subseteq B \times B$ is a set of edges representing transitions between basic blocks.
—X is a finite set of variables that consists of actual source-code variables and auxiliary variables added for modeling and property monitoring.
—$\delta : B \to 2^{\mathcal{A}}$ is a labeling function that labels each basic block with a set of parallel assignments, where $\mathcal{A}$ represents the set of all possible C assignments.

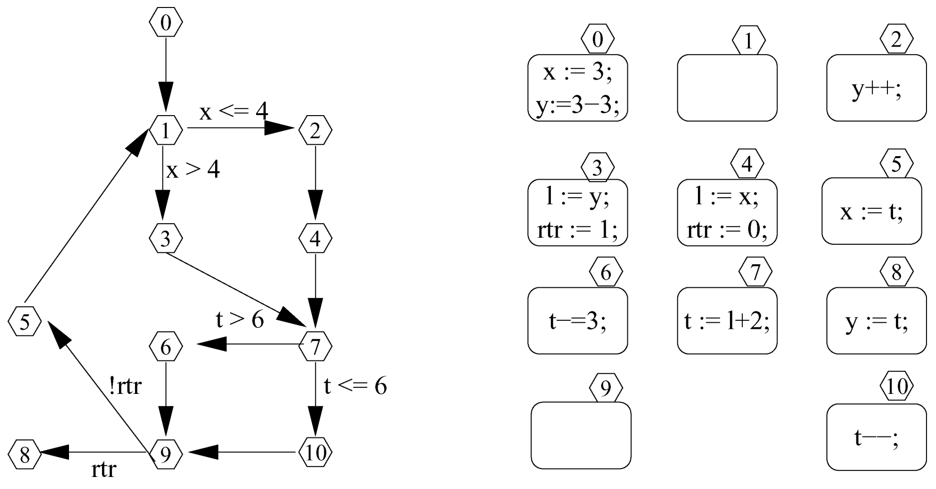

Fig. 2. Control and data logic subgraphs.

—$\theta : E \rightarrow \mathcal{C}$ is a labeling function that labels each edge with a conditional expression, where $\mathcal{C}$ represents the set of all possible C conditional expressions. 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 $b_s \in B$, 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 $e_1, \ldots, e_n$ are valid C expressions.

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 $\vec{b}_k \in B_{Err}$.

In order to represent the transition relation of C programs symbolically, we consider two subgraphs of G. The *control logic subgraph* $G_C = \langle B, E, X, \theta \rangle$ discards the labeling function δ, since it is used to define the transition relation in terms of basic block changes in a control flow graph. On the other hand, the *data logic subgraph* $G_D = \langle B, X, \delta \rangle$ concentrates on how variables are updated in individual basic blocks, and is used to define the transition relation for variables in X. Figure 2 shows the control and data logic subgraphs of the example in Figure 1.

Table I. A Partial State Transition Table

p_4	p_3	p_2	p_1	guard	q_4	q_3	q_2	q_1
0	0	0	0	true	0	0	0	1
0	0	0	1	$x \leq 4$	0	0	1	0
0	0	0	1	$x > 4$	0	0	1	1
...	...	...	...	...	...	...	...	...

4.2 Next-State Function

To build the next-state functions, we encode each simple variable by a set of binary-state variables. The number of bits needed to encode a certain program variable depends on the variable range. Note that for many program variables, although they are declared as integers, the actual ranges used in the program are usually very small. Typical variables falling into this category include function pointer ids, process ids, offsets in pointer arithmetic, array indices, etc. Variables created during the removal of pointers and flattening of structure variables also have very limited ranges. In Zaks et al. [2006], we formulate each range analysis problem [Rugina and Rinard 2000] as a system of inequality constraints between symbolic bound polynomials, then reduce the constraint system to a linear program (LP) that can be analyzed by available LP solvers. The solution to the LP problem provides symbolic lower and upper bounds for the values of all integer variables.

Arithmetic expressions over program variables are modeled by instantiating predefined Boolean logic components (e.g., adders and multipliers).

We assign a program location for each basic block. This approach is very effective in reducing the sequential depth during symbolic search. A set of binary variables, called program counter (PC) variables, are created to encode the program locations. The set of all program variables and PC variables, together with their next-state functions, define the finite-state verification model.

We represent the control flow graph as a state transition table. Table I shows a partial table for the example given in Figure 1, where $P = \{p_1, p_2, p_3, p_4\}$ represents the current value of the program counter (PC) variable, while $Q = \{q_1, q_2, q_3, q_4\}$ represents the next-state value. The column "guard" consists of conditions under which transitions among basic blocks are made. Based on this table, the next-state functions of PC variables, denoted by δ_{q_i}, can be constructed. Next δ_{q_2} and δ_{q_1} are given as follows.

$$\delta_{q_2} = (P \equiv 1 \wedge x \leq 4) \vee (P \equiv 1 \wedge x > 4) \vee \ldots,$$

$$\delta_{q_1} = (P \equiv 0) \vee (P \equiv 1 \wedge x > 4) \vee \ldots,$$

where $(P \equiv 1)$ means that P evaluates to 1, whose equivalent Boolean formula is $\neg p_3 \wedge \neg p_2 \wedge \neg p_2 \wedge p_1$.

Formally, for each row j in a given state transition table, let k_j be the condition, $v_{i,j}$ be the valuation for p_i, and $v'_{i,j}$ be the valuation for q_i. The next-state function for PC variable q_i is given as follows.

$$\delta_{q_i} = \bigvee_{j:v'_{i,j}=1} \left(k_j \wedge \bigwedge_{v_{i,j}=1} p_i \wedge \bigwedge_{v_{i,j}=0} \neg p_i\right)$$

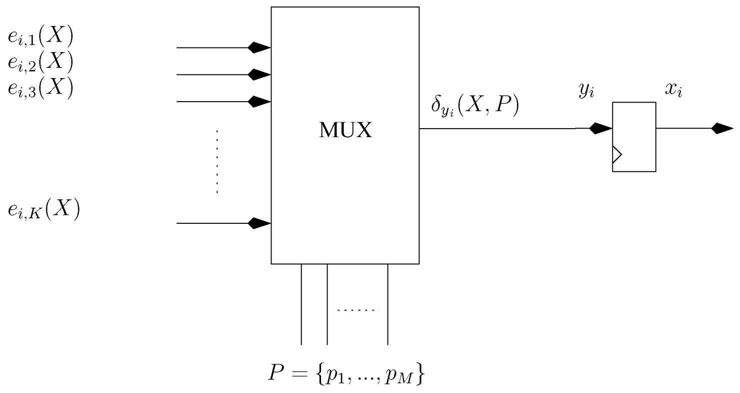

Fig. 3. Next-state logic of a binary state variable.

The next-state functions of program variables are constructed in a similar way. For example, the variable t in Figure 1 is assigned inside blocks 6, 7, and 10 by $t - 3$, $l + 2$, and $t - 1$, respectively. The next-state of t, denoted by $\delta_{t'}$ where t' is the corresponding next-state variable, is given as follows.

$$(P \equiv 6 \; ? \; t - 3 : (P \equiv 7 \; ? \; l + 2 : (P \equiv 10 \; ? \; t - 1 : t)))$$

In other words, the value of t should remain unchanged in blocks where it is not explicitly assigned. Note that the integer arithmetic operations will be translated into Boolean operations by instantiating predefined logic blocks, such as adders and multipliers.

If we use $X = \{x_1, x_2, \ldots, x_N\}$ to denote the entire set of program variables—whose next-state variables are $\{y_1, y_2, \ldots, y_N\}$—the construction of next-state function for x_i is illustrated by Figure 3. Here δ_{y_i} denotes the next-state function of x_i; MUX is a multiplexer; when the control signal $(P \equiv j)$, the output is equal to $e_{i,j}(X)$.

To formalize this, let the model be represented in terms of: (1) a set of present-state program variables $X = \{x_1, \ldots, x_N\}$ and PC variables $P = \{p_1, \ldots, p_M\}$, and (2) a set of next-state program variables $Y = \{y_1, \ldots, y_N\}$ and PC variables $Q = \{q_1, \ldots, q_M\}$. Let δ_{y_i} and δ_{q_i} denote the next-state functions of y_i and q_i, respectively. We have

$$\delta_{y_i}(X, P) = \bigvee_j (P \equiv j) \wedge e_{i,j}(X),$$

where $j \in \{0, 1, \ldots, K - 1\}$ is a PC location and $e_{i,j}(X)$ is the right-hand side of an assignment to y_i at location j. Note that $e_{i,j} = x_i$ if there is no assignment to y_i at location j.

$$e_{i,j}(X) = \begin{cases} expr(X), & \text{if } x_i = expr(X) \text{ is in block } j \\ x_i, & \text{otherwise} \end{cases}$$

4.3 Symbolic Reachability Computation

In symbolic model checking, the state transition graph of the model is represented symbolically by $\langle T, I \rangle$, where $T(X, P, Y, Q)$ is the characteristic function of the transition relation, and $I(X, P)$ is the initial state predicate. Let $\tilde{X}$, $\tilde{P}$, $\tilde{Y}$, and $\tilde{Q}$ be the valuations of X, P, Y, and Q, respectively. If the valuation of present-state variables makes $I(\tilde{X}, \tilde{P})$ true, the corresponding state $(\tilde{X}, \tilde{P})$ is an initial state. Similarly, $T(\tilde{X}, \tilde{P}, \tilde{Y}, \tilde{Q})$ is *true* if and only if there is a transition from State $(\tilde{X}, \tilde{P})$ to State $(\tilde{Y}, \tilde{Q})$. Note that both the transition relation and sets of states are represented by Boolean functions, which are in turn represented by BDDs.

$T(X, P, Y, Q)$ is the conjunction of all the transition bit-relations

$$T = \bigwedge_{1 \le i \le N} T_{y_i}(X, P, y_i) \wedge \bigwedge_{1 \le l \le M} T_{q_l}(X, P, q_l),$$

where T_{y_i} is the bit-relation for a program variable, and T_{q_l} is the bit-relation for a PC variable. Bit-relations are defined as follows.

$$T_{y_i}(X, P, y_i) = y_i \leftrightarrow \delta_{y_i}(X, P)$$
$$T_{q_l}(X, P, q_l) = q_l \leftrightarrow \delta_{q_l}(X, P)$$

The image of a set of states D, consisting of all the successors of D in the graph $\langle T, I \rangle$, is denoted by $EY_T D$. The image of D is computed as follows.

$$EY_T D = \exists X, P \ . \ T(X, P, Q, Y) \wedge D(X, P)$$

After image computation, we also need to simultaneously substitute all next-state variables $\{Y, Q\}$ with corresponding present-state variables $\{X, P\}$. Since T is the conjunction of many bit-relations, image computation consists of a series of *conjoin-quantify* operations. Different orders of these operations may lead to quite different peak BDD sizes for the intermediate products. Ordering these operations in order to keep the peak memory usage low, also called the quantification scheduling problem, has been the focus of several research projects. A large number of heuristics, as well as matured implementations are publicly available [Ranjan et al. 1995; Moon et al. 2000; Jin et al. 2002].

Our discussion in this article will be focused on checking reachability properties, but the same framework supports full symbolic model checking. In particular, reachability properties can be checked by a least fixpoint computation that starts from the initial states and repeatedly adds the postcondition (or image) of already-reached states. Reachability analysis is then formulated into a fixpoint computation

$$R = \mu Z \ . \ I(X, P) \cup EY_T Z,$$

where μZ represents the *least fixpoint*. In other words, we repeatedly add the successors of already-reached state until the set of reachable states stops growing.

5. DISJUNCTIVE IMAGE COMPUTATION

The best known symbolic algorithms [Ranjan et al. 1995; Moon et al. 2000] do not work well when applied directly to the software models. Figure 4 shows the data of reachability analysis on a C program from a Linux implementation of point-to-point protocol (PPP), whose verification model has 1,435 binary-state variables. We have encoded the model in BLIF-MV format and then ran

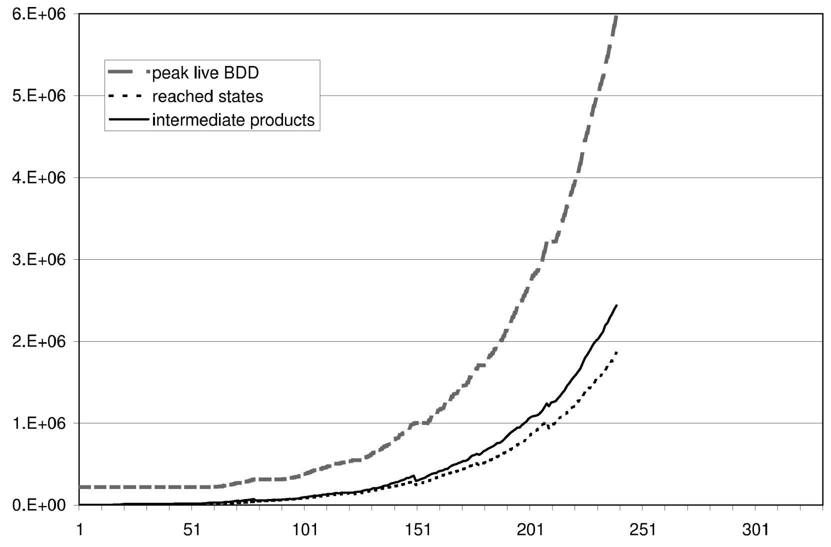

Fig. 4. PPP: BDD sizes at different reachability steps.

reachability analysis with VIS [Brayton et al. 1996]. We have applied the best classic symbolic image computation algorithm [Moon et al. 2000] and with dynamic variable reordering. The three curves represent at each BFS (breadth-first search) step the peak number of live BDD nodes, the BDD size for reachable states, and the maximum BDD size for intermediate products. Among them, the first curve represents total memory usage.

Figure 4 is typical for applying conventional symbolic model checking techniques to software models. All three curves grow exponentially with the BFS step, indicating that as the reachability analysis goes on, the BDDs grow rapidly in size. Due to the large number of program variables in software models, such an exponential growth can quickly deplete the memory. CPU time also grows in a similar fashion, since the complexity of BDD operations depends on the size of the BDDs. To mitigate the memory blowup, the size of BDDs representing the transition relation and reachable states must be reduced.

It is known that disjunctive partitioned representation of the transition relation can reduce the BDD size. Applying disjunctive partition-based image computation to hardware models has not been very successful, for computing the disjunctive partition itself is a nontrivial task. For sequential programs, however, a natural decomposition of next-state functions does exist. In particular, the transition function δ_{y_i} as shown in Figure 3 is the union of a set of expressions, only one term of which is active at any program location. This can be used to decompose the transition relation T into the union of disjunctive components.

5.1 Decomposition of Transition Relation

The transition relation T of a software model can be decomposed naturally into a union of disjunctive components, one for each program location. Since existential quantification $\exists$ distributes over disjunction $\vee$, we can compute individual images with smaller transition relation components. This can significantly reduce the peak memory usage at each reachability step.

Let (j/P) represent the substitution of P with the integer value j; similarly, let $f(X/Y)$ represent the substitution of Y variables with X variables inside Function f. By definition, we have

$$\delta_{y_i}(X, j/P) = e_{i,j}(X),$$

where $e_{i,j}(X)$ is actually the cofactor of $\delta_{y_i}(X, P)$ with respect to $(P \equiv j)$. The cofactors of transition bit-relations with respect to $(P \equiv j)$ are given as follows.

$$\begin{aligned}(T_{y_i})_{(P\equiv j)} &= (y_i \leftrightarrow \delta_{y_i})_{(P\equiv j)}\\ &= (y_i \wedge \delta_{y_i} \vee \neg y_i \wedge \neg\delta_{y_i})_{(P\equiv j)}\\ &= y_i \wedge (\delta_{y_i})_{(P\equiv j)} \vee \neg y_i \wedge \neg(\delta_{y_i})_{(P\equiv j)}\\ &= y_i \wedge e_{i,j} \vee \neg y_i \wedge \neg e_{i,j}\\ &= y_i \leftrightarrow e_{i,j}\end{aligned}$$

and

$$\begin{aligned}(T_{q_l})_{(P\equiv j)} &= (q_l \leftrightarrow \delta_{q_l})_{(P\equiv j)}\\ &= (q_l \wedge \delta_{q_l} \vee \neg q_l \wedge \neg\delta_{q_l})_{(P\equiv j)}\\ &= q_l \wedge (\delta_{q_l})_{(P\equiv j)} \vee \neg q_l \wedge \neg(\delta_{q_l})_{(P\equiv j)}\\ &= q_l \leftrightarrow (\delta_{q_l})_{(P\equiv j)}\end{aligned}$$

By definition, $T = \bigvee_j (P \equiv j) \wedge (T)_{(P\equiv j)}$, where

$$\begin{aligned}(T)_{(P\equiv j)} &= (T_{y_i})_{(P\equiv j)} \wedge (T_{q_l})_{(P\equiv j)}\\ &= \bigwedge_{1\le i\le N}(y_i \leftrightarrow e_{i,j}) \wedge \bigwedge_{1\le l\le M}\big(q_l \leftrightarrow (\delta_{q_l})_{(P\equiv j)}\big).\end{aligned} \tag{1}$$

Since existential quantification distributes over $\vee$, we have

$$\begin{aligned}\exists X, P\ .\ D(X,P) \wedge T &= \exists X, P\ .\ D(X,P) \wedge \bigvee_j (P \equiv j) \wedge (T)_{(P\equiv j)}\\ &= \bigvee_j \exists X, P\ .\ D(X,P) \wedge (P \equiv j) \wedge (T)_{(P\equiv j)}\\ &= \bigvee_j \exists X\ .\ D(X, j/P) \wedge (T)_{(P\equiv j)}.\end{aligned}$$

Therefore, we can use following formula to compute the successors of D.

$$EY_T D(X,P) = \bigvee_j EY_{T_{(P\equiv j)}} D(X) = \bigvee_j \exists X\ .\ D(X, j/P) \wedge (T)_{(P\equiv j)} \tag{2}$$

There can be one disjunctive component for every PC location j. However, for efficiency purposes, we often merge multiple locations and then build a disjunctive component for each cluster. At the same time, the merit of disjunctive decomposition must be preserved as much as possible. In Section 7 we will give a heuristic algorithm for the merging, which simultaneously minimizes the number of live variables in each cluster.

Note that the decomposition into $(T)_{(P\equiv j)}$ is based on the PC locations, not on individual program variables, as in Barner and Rabinovitz [2003]. The method in Barner and Rabinovitz [2003] builds one transition relation disjunct for each variable, which often defeats the purpose of exploiting variable locality. When each disjunct contains multiple assignment statements scattered at different program locations, the number of live variables with respect to the disjunct will

```
ReachabilityComputation( )
{
    foreach disjunctive submodule $i$
        $new = img(T_i, R_i);$
        foreach disjunctive submodule $j$
            $R_j = R_j \vee (new \wedge P == j);$
}
```

Fig. 5. Disjunctive reachability computation.

increase. Another significant difference is that we create $(T)_{(P \equiv j)}$ directly from the software program, while they rely on the existing conjunctive transition relation and expensive existential quantification operations. In practice, building the conjunctive transition relation itself may be computationally expensive, or even infeasible. We will show later that without variable locality, the BDD representation of reachable states will be significantly larger, and the sequential depth of the finite state model may be longer.

5.2 Decomposition of Reachable States

Reachable states can also be represented disjunctively as the union of many reachable state subsets, one for each submodule

$$R(X, P) = \bigvee_j (P \equiv j) \wedge R(X, j/P).$$

Under the partitioned representation of reachable states, image computation results need to be redistributed at every reachability step. Conceptually, this redistribution step is shown by the pseudocode in Figure 5. In the figure, R_i represents the reachable state subset associated with block i, while T_i represents its transition relation. Function call `img` computes the standard image computation procedure using conjunctively partitioned transition relation. In some sense, our new image computation algorithm is implemented on top of the standard image algorithm. Given K disjunctive submodules and K reachable subsets, the number of standard image computations at each reachability step is, at most, K. In addition, the result must be redistributed among all submodules. Note that the pseudocode in Figure 5 is for illustration purposes, and a simple optimization based on control flow structure can make the complexity of redistribution become $O(E)$, where E is the number of edges in the control flow graph.

In practice, we do not perform reachability computation at basic block level. Instead, multiple basic blocks are merged to form clusters, which will be discussed in Section 7. When basic blocks are merged, the reachable state subsets should also be merged. Nevertheless, the resulting cluster remains a disjunctive partition. Compared to a monolithic BDD, this partitioned representation may reduce the overall BDD size significantly.

The procedure in Figure 5 computes reachable states frame-by-frame (FBF). Alternatively, we can compute reachable states machine-by-machine (MBM); that is, the analysis is performed on one individual cluster until it converges, after which the result is propagated to other clusters. MBM minimizes the traffic (data transfer) among different clusters and is therefore appealing when a distributed implementation is used. This is analogous to the approximate FSM traversal algorithm of Cho et al. [1996], with the difference that we build the transition relations without approximation and our reachable states are always exact.

There are two different approaches in implementing our disjunctive image computation algorithm. In the first approach, all transition relations and reachable subsets follow the same BDD variable order. In this case, a single BDD manager is allocated to hold all the BDDs. Alternatively, different BDD variable orders can be used for different submodules to build their individual transition relations and reachable state subsets. In the latter case, a separate BDD manager is allocated for every disjunctive submodule.

With multiple BDD managers, BDD size can be kept much smaller by having different variable orders for different BDDs. This makes it possible for image computation to consume much less memory, and potentially speed-up the computation. However, BDDs need to be transferred among different managers while propagating new reachable states. This may introduce problems, since a variable order that is good for one BDD may be extremely bad for another. Therefore, dynamic variable reordering has to be triggered during the transfer process. Frequent reordering may consume a lot of CPU time, as well. In this sense, the multiple-manager-based approach is actually trading CPU time for memory. We have implemented both single and multiple BDD manager-based approaches. The experimental comparison can be found in Section 8.

6. SIMPLIFICATION USING VARIABLE LOCALITY

In this section, we describe simplifications using variable locality specific to software models. Specifically, we describe the removal of irrelevant variables and using variable live scope to simplify reachable state subsets.

6.1 Relevant Variables

Definition 6.1. The set of *relevant* variables with respect to location j, denoted by X_j^R, contains those appearing in either the assignments or conditional expressions of block j. The set $X_j^I = X \backslash X_j^R$ consists of *irrelevant* variables.

The contribution of an irrelevant variable to the transition relation is of the form $(y_i \leftrightarrow x_i)$; hence $(T)_{(P\equiv j)}$ in Eq. (1) can be replaced by

$$\bigwedge_{x_i \in X_j^I} (y_i \leftrightarrow x_j) \wedge \bigwedge_{x_i \in X_j^R} (T_{y_i})_{(P\equiv j)} \wedge \bigwedge (T_{q_l})_{(P\equiv j)}.$$

Although $(y_i \leftrightarrow x_i)$ can be represented by a BDD with three nodes, conjoining many of them is known to produce BDDs with exponential numbers of nodes in the worst case. On the other hand, a good BDD order for the constraints may be bad for other Boolean formulas encountered in reachability analysis.

Inside image computation, the equality constraints facilitate the substitution of x_i with y_i for all irrelevant variables. Unfortunately, existing quantification scheduling algorithms [Ranjan et al. 1995; Moon et al. 2000] often fail to identify these variables. We choose not to include the constraints $\bigwedge_{x_i \in X_j^I}(y_i \leftrightarrow x_j)$

```
{
    int a, b;

    a = x + 5;
    if (y==10)
      b = 1;
    else
      b = 2;
    z = a + b;
}
...
```

```
{
    x = a;
    z = x + b;
}
...
{
    x = c;
    z = x + d;
}
...
```

Fig. 6. Left: locally defined variables a and b; right: globally defined variable x.

in $(T)_{(P\equiv j)}$ in the first place, to avoid the potential BDD blowup during quantification. Therefore, we can improve Eq. (2) by

$$EY_{T_{(P\equiv j)}} D(X) = \exists X_j^R \;.\; D\big(Y_j^I/X_j^I, j/P\big) \wedge \bigwedge_{x_i \in X_j^R} (T_{y_i})_{(P\equiv j)} \wedge \bigwedge (T_{q_l})_{(P\equiv j)}. \quad (3)$$

Letting $R(Y, Q)$ be the result of Eq. (3), we still need to substitute all the Y and Q with the corresponding present-state variables to get $R(X, P)$. Therefore for irrelevant variables, the substitutions need to be done twice. In our actual implementation, we remove the equality constraints from the transition relation and avoid substitutions in both directions. Our experimental studies show that this significantly reduces the peak BDD sizes of the intermediate products in image computation.

6.2 Live Variables

Disjunctive decomposition and early quantification of irrelevant variables are aimed at simplifying the transition relation and intermediate products in image computation. However, the BDDs of reachable state subsets can still be very large. Note that even a reachable state subset may have all the program variables in its support, making it hard to find a compact BDD with dynamic reordering (a major reason for the blowup in Figure 4). However, many variables are local to certain program locations and their values are meaningless at others. This is completely different from state variables in hardware models, whose values are updated concurrently in every clock cycle. Locally defined variables, for instance, should be considered as state-holding only inside the blocks where they are defined, since elsewhere their values neither affect the control flow nor the data path. However, by default, their values are carried on as Boolean functions in the reachable state subsets, which makes the BDD representation of reachable states unnecessarily large.

Local variables can be easily identified and removed from state subsets. This is illustrated by the example in the program on the left in Figure 6, in which a and b are defined as local variables, while x, y, and z are global variables. After the program exits this block, values of a and b do not need to be carried on.

The live scope of a certain variable, defined as the set of basic blocks in which its value should be retained, can be obtained directly from the software model. Note that the live scope is often more accurate than the syntactic scope of a variable—even global variables are often used only within a limited number of blocks. In the program on the right in Figure 6, for instance, if the global variable x appears only in the two blocks, it should be considered dead in all other blocks. In this sense, the only *truly global* variables are the PC variables, since their values are required in every block.

Definition 6.2. Block j is within the live scope of Variable x_i if and only if there exists a path from block j to the block k where the value of x_i is used, and x_i is not assigned along the path.

The live scope of a variable x_i can be computed using a variant of the standard technique in control flow analysis [Aho et al. 1986]. Letting K be the number of program locations, E the number of transitions in the control flow, and N the number of state variables, the complexity of this process is $O((K + E) \times N)$. Compared to model checking, the overhead is negligible. Although the algorithm is not new, our contribution is the novel way of using it to reduce the complexity of symbolic image/fixpoint computation.

Live scope has also been used in Ball and Rajamani [2000], but only on variables at function boundaries during the computation of function summaries. We use the live variable analysis for the following optimization. During the redistribution of image results, all variables that are not alive in that destination location can be existentially quantified out. Letting D_j be the dead variables at cluster j, $R_j = R_j \vee (new \wedge P == j)$ in Figure 5 is replaced by

$$R_j = R_j \vee ((\exists D_j.new) \wedge P == j).$$

Note that live variable analysis can achieve significantly more size reduction of the transition relation than simple program slicing. In Figure 6, for instance, we can remove the impact of implicit assignments like $x' = x$ from lines 4–6, where x is not live. However, a property-dependent program slicing alone cannot achieve this. Also, our experimental study has shown that in practice, the number of live variables with respect to any individual basic block is relatively small; live variables typically comprise less than 30% of the entire set of state variables in X.

6.3 Early Convergence of Reachability Fixpoint

Removing variables that are not in their live scopes from reachable subsets is important because this not only reduces the BDD sizes, but also leads to a potentially faster convergence of the reachability fixpoint computation.

Take the piece of C code in Figure 7 as an example. Here we assume that each statement is a basic block, and x, y are global variables initialized at L1. For any existing model checker, the values of x and y are carried on until the next assignment. However, the analysis outlined in previous subsections indicates that x is live only in L2, while y is live only in L4 if variables x and y are not used anywhere else in the program. In other words, their values

Table II. New Reachable States Discovered After Each BFS Step

Line	With all program variables	With live variables only
1	$P \equiv 2 \wedge x \equiv 0 \wedge y \equiv 0$	$P \equiv 2$
2	$P \equiv 3 \wedge x \equiv 7 \wedge y \equiv 0$	$P \equiv 3 \wedge x \equiv 7$
3	$P \equiv 4 \wedge x \equiv 7 \wedge y \equiv 0 \wedge s \equiv 7$	$P \equiv 4 \wedge s \equiv 7$
4	$P \equiv 5 \wedge x \equiv 7 \wedge y \equiv 8 \wedge s \equiv 7$	$P \equiv 5 \wedge s \equiv 7 \wedge y \equiv 8$
5	$P \equiv 6 \wedge x \equiv 7 \wedge y \equiv 8 \wedge s \equiv 8$	$P \equiv 6 \wedge s \equiv 8$
6	$P \equiv 2 \wedge x \equiv 7 \wedge y \equiv 8 \wedge s \equiv 8$	$\emptyset$
2’	$P \equiv 3 \wedge x \equiv 7 \wedge y \equiv 8 \wedge s \equiv 8$	$\emptyset$
3’	$\emptyset$	$\emptyset$

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. 7. Earlier convergence of reachability analysis.

should be preserved after the execution of L2 and L4, but nowhere else. Taking this information into account, we can declare the termination of reachability analysis after going from L1 through L6 only once. This is because the state discovered after L6, namely ($P \equiv 2 \wedge s \equiv 8$), is already covered by ($P \equiv 2$) (in which s can take any value). However, if x and y are assumed to be live globally, reachability analysis cannot converge after the first pass. After L6, State ($P \equiv 2 \wedge x \equiv 7 \wedge y \equiv 8 \wedge s \equiv 8$) is not covered by an already-reached state, since earlier the state was ($P \equiv 2 \wedge x \equiv 0 \wedge y \equiv 0$). Therefore, reachability analysis needs two more steps to converge.

By identifying for each disjunctive submodule the dead variables and removing them from the reachable state subset, we can significantly reduce the BDD variables that are actually involved in image computations. This translates into more compact BDDs, faster image computation, and the potential early termination of reachability analysis.

7. OPTIMIZING THE DISJUNCTIVE PARTITION

We now explain how to merge basic blocks into disjunctive clusters. Although considering each block as a separate cluster maximizes the number of dead and irrelevant variables, the often large number of basic blocks encountered in practice prevents us from doing so. Instead, we merge basic blocks into larger groups, for each of which a transition relation is built.

Ideally, we want to make as few groups as possible and at the same time, retain the benefit of variable locality. We approach for merging the basic blocks is to use the actual BDD size of the transition relation disjunct as an indicator.

We can keep merging BDDs of the transition relations until the size of the resulting BDD is above the threshold. However, it would be too expensive to build transition relations for all individual basic blocks and then start merging them. Instead, we choose to precompute a good partition before any BDD is created, and then build transition relation clusters directly for individual groups.

In our approach we achieve this goal by formulating a multiway hypergraph partitioning problem. Initially, all basic blocks are treated as a single group, starting from which recursive bipartitioning is performed. An inexpensive cost function needs to be defined so that bipartitioning stops whenever the cost of a subset drops below a predetermined threshold. We choose the total number of relevant variables as such an indicator. Our motivation is that the number of support variables of a BDD is often a good indicator of its size.

We define the partitioning optimization criteria as one of the following numbers:

—*liveVar*: the number of shared live variables,

—*asgnVar*: the number of shared assigned variables, or

—*cfgEdge*: the number of edges in the CFG across the partitions.

Shared live variables are those that are alive in both partitions, while shared assigned variables are those whose values are changed in both partitions. Note that a smaller number of shared live variables means a higher degree of variable locality, since more dead variables can be quantified out from the reachable state subsets. A smaller number of shared assigned variables means that bit-relations with similar supports are grouped together. Merging BDDs with similar support variables is less likely to cause blowups. Less shared CFG edges means less traffic among different submodules. When two partitions are connected by a CFG edge, new reachable states may need to be transferred from one to the other.

All three optimization criteria can be represented uniformly as a generic multiway hypergraph partitioning problem. A hypergraph is a generalization of a graph in which a hyperedge can connect more than two hypernodes. Hypergraphs arise naturally in many important practical problems, including circuit layout, Boolean satisfiability, numerical linear algebra, etc. Given a hypergraph H, a k-way partitioning of H assigns hypernodes of H to k disjoint nonempty partitions. The k-way partitioning problem seeks to minimize a given cost function of such an assignment. The standard cost function is net cut, which is the number of hyperedges that span more than one partition, or more generally, the sum of weights of such edges. Constraints are typically imposed on the solution: For example, the total hypernode weight in each partition may be limited to a predetermined threshold. Although the problem of optimally partitioning a hypergraph is known to be NP-hard, heuristic algorithms have been developed with near-linear runtime in practice [Karypis and Kumar 1998].

Our three optimization criteria differ only in the way hyperedges are defined. As illustrated in Figure 8:

(1) Hypernodes correspond to individual basic blocks, with one node for each block; and

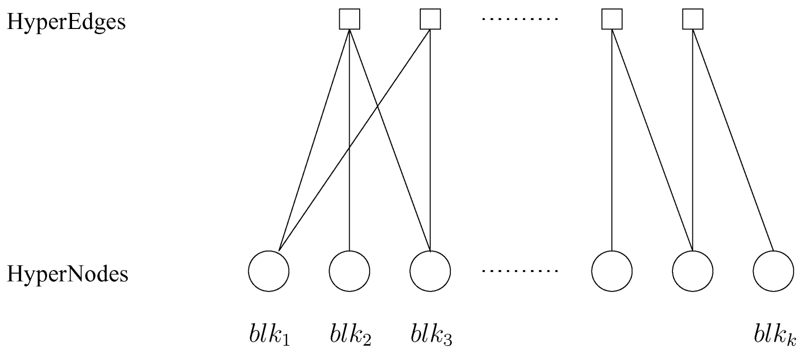

Fig. 8. The hypergraph.

(2) hyperedges correspond either to program variables or edges in the CFG:
—*liveVar*: one edge is added for each variable so as to connect blocks in which it is alive;
—*asgnVar*: one edge is added for each variable so as to connect blocks in which it is assigned; and
—*cfgEdge*: one edge is added for each CFG edge so as to connect the head and tail blocks.

We want to compute a multiway partition of this hypergraph such that the cost of each group is below a predetermined threshold, while the number of hyperedges across different groups is minimized. In our experiments, we use the number of relevant variables in a group as the cost function (with the threshold empirically set to 200).

Note that although our live-variable-based partitioning method is similar to the MLP (minimum lifetime permutation) algorithm of Moon et al. [2000], they are designed for different applications. MLP is applied to a set of conjunctively partitioned transition relations to compute a good schedule for the conjoin-quantify operations. Our algorithm, on the other hand, is applied to a set of disjunctively partitioned transition relations so as to group individual disjunctive transition relations into larger clusters. It is worth mentioning that, similar to the impact of quantification scheduling on classic image computation, a good disjunctive partition is also important for the performance of disjunctive image computation.

8. EXPERIMENTS

We have implemented our new algorithm in the publicly available symbolic model checker VIS 2.0 [Brayton et al. 1996], which by default uses the CUDD package for BDD operations. The partitioning heuristics were implemented upon the hMeTis hypergraph partitioning package [Karypis and Kumar 1998]. We conducted experimental comparison of our new algorithm with the best known conventional method in VIS 2.0. For the purpose of controlled experiments, all image-computation-related parameters in VIS were kept unchanged, which includes the BDD partition method (set to *frontier*), the BDD partition threshold (set to 2,000), and the image cluster size (set to 5,000). The underlying

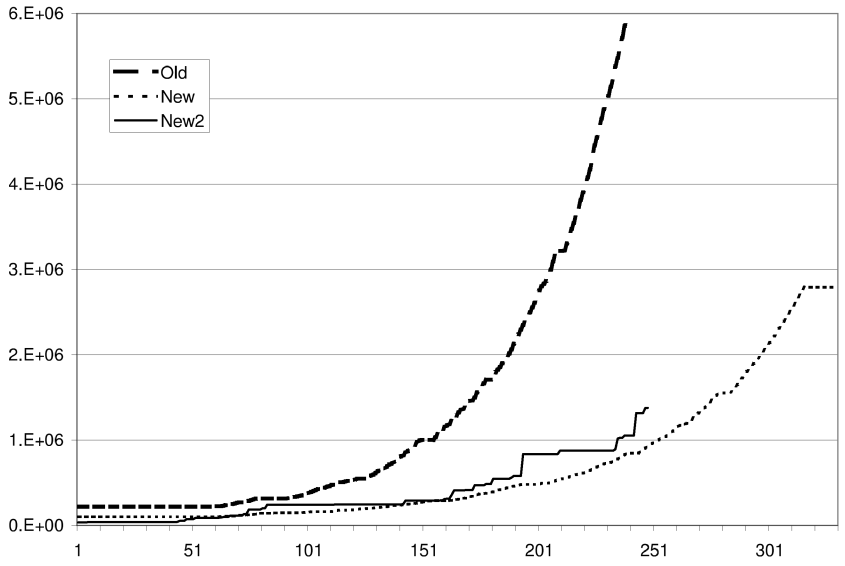

Fig. 9. PPP: peak live BDD nodes in reachability analysis.

conjunction scheduling algorithm was MLP [Moon et al. 2000]. Dynamic variable ordering was enabled throughout all the experiments, with the default reordering method "sift". All the experiments were performed on a workstation with 2.8 GHz Xeon processors and 4GB of RAM running Red Hat Linux 7.2.

The set of benchmarks is composed of typical embedded software, including PPP (point-to-point protocol), Core (embedded software for hand-held devices), and Serial (Linux serial device driver). The properties we are checking are all reachability properties. Range analysis has been applied to reduce the number of bits needed for encoding simple integer/char variables. In order to test the sheer capacity of BDD-based algorithms, verification models are generated without predicate abstraction. In this sense, our software models are significantly more complex than the *Boolean programs* of Ball and Rajamani [2000] (where only Boolean variables are allowed). These software models are also significantly larger than most of the hardware models that can be *directly handled* by symbolic model checkers.

First, we give experimental comparison of our new algorithm and the conventional method on the PPP example. Figure 9 shows the peak memory usage at each reachability step for both methods. Among the three curves, *Old* represents the conventional method, *New* represents our new method with a single BDD manager (and a single BDD order), and *New2* represents the one with multiple managers (and correspondingly multiple BDD orders). The x-axis represents the different BFS steps, while the y-axis represents the peak number of live BDD nodes. All experiments were run with a time limit set to four hours, during which the conventional method completed 238 steps, while *New* and *New2* completed 328 and 247 steps, respectively. The plateaus in the curves are caused by dynamic variable reordering.

Table III. Comparison in Reachability Analysis

Test Cases			Completed		CPU Time (s)		Peak BDD (k)	
name	vars	dep.	Old	New	Old	New	Old	New
ssdf	37	11	Y	Y	0.01	0.02	0.8	0.8
sfi	105	47	Y	Y	0.5	0.6	4	4
sirpp	169	52	Y	Y	8	6	10	10
srb	343	43	Y	Y	2766	146	925	106
core1	416	211	Y	Y	1115	89	155	51
core2	445	192	70	Y	>2h	80	490	70
srr	856	316	Y	Y	5426	151	990	57
smhb	888	104	25	Y	>2h	341	1219	120
siic	967	162	Y	Y	4020	260	1188	79
spr	1001	617	85	Y	>2h	1617	428	430
sdir	1050	209	136	Y	>2h	394	3135	120
srdo	1211	215	128	Y	>2h	496	2482	145
core3	1213	189	41	Y	>2h	205	139	101
ppp	1435	?	208	277	>2h	>2h	3194	540

The results show that our new disjunctive image computation method reduces the peak memory usage significantly: Compared to both *New* and *New2*, the curve of *Old* is much deeper. The reduction in BDD size also translates into reduction in CPU time; both *New* and *New2* can complete more steps than *Old*. Note that although *New2* was initially designed to trade CPU time for memory usage, its advantage over *New* was not clearly demonstrated. We have observed similar results on other test examples, as well. The reason that *New2* is not a clear winner is that BDDs need to be transferred frequently from one BDD manager to another manager while propagating new reachable states, which may trigger frequent dynamic variable reordering. Consuming a lot of CPU time, frequent reordering often offset the benefit of having smaller BDDs inside image computation.

Table III gives the comparison of the conventional method and our new method with single BDD order on a larger set of test cases. The time limit was set to two hours. Columns 1 and 2 show, for each test case, the name and number of binary state variables of the model. Column 3 shows the sequential depth when reachability analysis converges, where a question mark (?) indicates that the sequential depth cannot be obtained within the time limit. Columns 4 and 5 show whether the reachability computation can be completed by the methods. If not, the actual number of completed steps is listed. Columns 6 and 7 compare the CPU time in seconds; columns 8 and 9 compare the peak number of live BDD nodes in thousands.

The results in Table III show that the performance improvement achieved by our new method is significant and consistent. Reachability analysis with disjunctive decomposition is significantly faster, and at the same time consumes less memory. The improvements in both CPU time and memory usage can be by orders of magnitude.

Table IV compares the three different partitioning heuristics: *liveVar*, *assignedVar*, and *cfgEdge*. They are all based on multiway hypergraph partitioning, but differ in the optimization criteria. We evaluate their impact on image computation by comparing the performance of reachability analysis. The time

Table IV. Comparing the Three Partitioning Heuristics (with *New*)

Model	CPU time (s)			Peak live BDD (k)			Num. partitions		
name	live	asgn	cfg	live	asgn	cfg	live	asgn	cfg
sssdf	0.02	0.02	0.02	0.8	0.8	0.8	1	1	1
sfi	0.6	0.6	0.6	4	4	4	1	1	1
sirpp	6	6	6	10	10	10	1	1	1
srb	146	>1h	**112**	106	-	**71**	4	4	4
core1	**89**	126	128	**51**	68	68	4	4	4
core2	**80**	98	96	70	**55**	56	5	5	5
srr	151	**115**	132	57	**46**	45	15	17	20
smhb	**343**	2723	1533	**120**	1726	2270	12	10	14
siic	260	**251**	291	**79**	85	105	11	11	12
spr	**1617**	>1h	3156	430	-	**172**	15	14	14
sdir	394	**318**	378	120	150	**86**	17	19	24
srdo	496	**443**	494	145	**97**	98	18	20	25
core3	**205**	314	294	101	**92**	115	12	12	12
ppp*	**3592**	>1h	>1h	**731**	-	-	17	17	28

limit for these experiments was set to one hour. The CPU time and peak memory usage are compared in columns 2–4 and columns 5–7, respectively. Note that the peak memory usage is updated at the end of each image computation. For some experiments the first image computation takes longer than one hour. In this case we use "-" to indicate that we are not able to obtain any data on peak memory usage. Columns 8–10 compare the number of disjunctive partitions created by the different heuristics. All CPU time and memory usage data are compared after reachability analysis converges, except PPP, whose data is compared at step 228. All experiments were conducted with method *New* (i.e., with a single BDD order), and with the partition threshold (i.e., maximum number of live variables) set to 175.

The results in Table IV show that the partitioning heuristic has a significant impact on the performance of disjunctive image computation. The overall reachability analysis runtime with the three partitioning heuristics are 7,379, >15,194, and >10,220 seconds, respectively. The primary goal of this set of experiments was to identify, during partitioning, the most significant factor affecting the overall verification performance. Overall, the *liveVar*-based heuristic is better than the others, especially on most of the harder cases. It is worth pointing out that further improvement may be achieved by defining new cost functions that take into account all three heuristics with different weights.

9. CONCLUSIONS

We have presented a novel disjunctive image computation algorithm for software model checking, by exploiting the unique characteristics of embedded software. It decomposes the software model into a set of submodules and then performs image computation on each. By dividing an expensive computation into a set of easier ones, our new method significantly reduces the peak memory usage required. Program modularity and variable locality are used to get a good disjunctive partition, and to simplify the transition relations and disjunctive sets of reachable states. Experiments on real embedded C programs have shown that our method significantly outperforms the best known conventional method based on conjunctive partitions. Although previous experience with hardware verification shows that symbolic model checking often loses its robustness when the number of state variables exceeds 200, our work demonstrates that by exploiting their unique characteristics, BDD-based algorithms can directly handle software models with thousands of variables.

REFERENCES

Aho, A., Sethi, R., and Ullman, J. 1986. *Compilers: Princeples, Techniques and Tools*. Addison-Wesley, Reading, MA.

Alur, R., Courcoubetis, C., and Dill, D. 1993. Model-Checking in dense real-time. *Inf. Comput. 104,* 1, 2–34.

Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K. 2001. Automatic predicate abstraction of C programs. In *Proceedings of the Programming Language Design and Implementation Conference (PLDI)* (Snowbird, UT).

Ball, T. and Rajamani, S. K. 2000. Bebop: A symbolic model checker for Boolean programs. In *SPIN Workshop on Model Checking of Software*. Lecture Notes in Computer Science, vol. 1885. Springer Verlag, Berlin. 113–130.

Barner, S. and Rabinovitz, I. 2003. Efficient symbolic model checking of software using partial disjunctive partitioning. In *Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*. Lecture Notes in Computer Science, vol. 2860. Springer Verlag, Berlin.

Boujjani, A., Esparza, J., and Maler, O. 1997. Reachability analysis of pushdown automata: Applications to model checking. In *Proceedings of the 8th International Conference on Concurrency Theory*. Lecture Notes in Computer Science, vol. 1243. Springer Verlag, Berlin. 135–150.

Brayton, R. K., Hachte, G. D., Sangiovanni-Vincentelli, A., Somenzi, F., Aziz, A., Cheng, S.-T., Edwards, S., Khatri, S., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R. K., Sarwary, S., Shiple, T. R., Swamy G., and Villa T. 1996. VIS: A system for verification and synthesis. In *Proceedings of the Eighth Conference on Computer Aided Verification (CAV)*, T. Henzinger and R. Alur, eds. Springer-Verlag, Rutgers University, 428–432. LNCS 1102.

Browne, M., Clarke, E. M., and Grumberg, O. 1999. Reasoning about networks with many identical finite state processes. *Inf. Comput. 81*, 13–31.

Bryant, R. E. 1986. Graph-Based algorithms for Boolean function manipulation. *IEEE Trans. Comput. C-35,* 8 (Aug.), 677–691.

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.

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. 1990. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, CA. 1–33.

Cabodi, G., Camurati, P., Lavagno, L., and Quer, S. 1997. Disjunctive partitioning and partial iterative squaring: An effective approach for symbolic traversal of large circuits. In *Proceedings of the Design Automation Conference* (Anaheim, CA). 728–733.

Chauhan, P. P., Clarke, E. M., Jha, S., Kukula, J., Shiple, T., Veith, H., and Wang, D. 2001. Non-Linear quantification scheduling in image computation. In *Proceedings of the International Conference on Computer-Aided Design* (San Jose, CA). 293–298.

Cho, H., Hachtel, G. D., Macii, E., Poncino, M., and Somenzi, F. 1996. Automatic state space decomposition for approximate FSM traversal based on circuit analysis. *IEEE Trans. Comput. Aided Des. 15,* 12 (Dec.), 1451–1464.

Clarke, E., Grumberg, O., and Peled, D. 2000. *Model checking*. MIT Press, Cambridge, MA.

Clarke, E., Kroening, D., and Lerda, F. 2004. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, K. Jensen and A. Podelski, eds. Lecture Notes in Computer Science, vol. 2988. Springer Verlag, Berlin. 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.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. 1999. *Model Checking*. MIT Press, Cambridge, MA.

COOK, B., KROENING, D., AND SHARYGINA, N. 2005. Symbolic model checking for asynchronous Boolean programs. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software*. 75–90.

CORBETT, J., DWYER, M., HATCLIFF, J., LAUBACH, S., PĂSĂREANU, C., ROBBY, AND ZHENG, H. 2000. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the International Conference on Software Engineering*. 439–448.

COUDERT, O., BERTHET, C., AND MADRE, J. C. 1989. Verification of sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems*, J. Sifakis, ed. Lecture Notes in Computer Science, vol. 407. Springer Verlag, Berlin. 365–373.

DAS, S., DILL, D. L., AND PARK, S. 1999. Experience with predicate abstraction. In *Proceedings of the 11th Conference on Computer Aided Verification (CAV)*, N. Halbwachs and D. Peled, eds. Lecture Notes in Computer Science, vol. 1633. Springer Verlag, Berlin. 160–171.

EDWARDS, S., MA, T., AND DAMIANO, R. 2001. Using a hardware model checker to verify software. In *Proceedings of the 4th International Conference on ASIC (ASICON)*. IEEE, Los Alamitos, CA.

GRAF, S. AND SAÏDI, H. 1997. Construction of abstract state graphs with PVS. In *Proceedings of the 9th Conference on Computer Aided Verification (CAV)*, O. Grumberg, ed. Lecture Notes in Computer Science, vol. 1254. Springer Verlag, Berlin. 72–83.

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

HOLZMANN, G. 1997. The model checker SPIN. *IEEE Trans. Softw. Eng. 23,* 5, 279–295.

HOLZMANN, G. J. AND PELED, D. 1994. An improvement in formal verification. In *Proceedings of the International Conference on Formal Description Techniques (FORTE)*. IFIP Conference Proceedings, vol. 6. Chapman and Hall. 197–211.

IVANČIĆ, F., YANG, Z., GUPTA, A., GANAI, M., AND ASHAR, P. 2004. Efficient SAT-based bounded model checking for software verification. In *1st International Symposium on Leveraging Applications of Formal Methods*.

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

JIN, H., KUEHLMANN, A., AND SOMENZI, F. 2002. Fine-Grain conjunction scheduling for symbolic reachability analysis. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)* (Grenoble, France). Lecture Notes in Computer Science, vol. 2280. Springer Verlag, Berlin. 312–326.

JIN, H., RAVI, K., AND SOMENZI, F. 2002. Fate and free will in error traces. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)* (Grenoble, France). Lecture Notes in Computer Science, vol. 2280. Springer Verlag, Berlin. 445–459.

KARYPIS, G. AND KUMAR, V. 1998. Multilevel algorithms for multi-constraint graph partitioning. Tech. Rep. 98-019, University of Minnesota, Department of Computer Science. May.

KUPFERMAN, O. AND VARDI, M. Y. 2001. Model checking of safety properties. *Formal Methods Syst. Des. 19,* 3, 291–314.

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

MOON, I.-H., HACHTEL, G. D., AND SOMENZI, F. 2000. Border-Block triangular form and conjunction schedule in image computation. In *Formal Methods in Computer Aided Design*, W. A. Hunt, Jr. and S. D. Johnson, eds. Lecture Notes in Computer Science, vol. 1954. Springer Verlag, Berlin. 73–90.

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.

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. Presented at IWLS (Lake Tahoe, CA).

Reps, T., Horwitz, S., and Sagiv, M. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci. 167*, 131–170.

Rugina, R. and Rinard, M. 2000. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM Press, New York. 182–195.

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.

Visser, W., Havelund, K., Brat, G., and Park, S. 2000. Model checking programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*.

Wang, C., Hachtel, G. D., and Somenzi, F. 2003. The compositional far side of image computation. In *Proceedings of the International Conference on Computer-Aided Design*. 334–340.

Wang, C., Yang, Z., Ivancic, F., and Gupta, A. 2006. Disjunctive image computation for emebedded software verification. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)* (Munich, Germany).

Wolper, P. and Leroy, D. 1993. Reliable hashing without collision detection. In *Proceedings of the 5th Conference on Computer Aided Verification (CAV)*, C. Courcoubetis, ed. Lecture Notes in Computer Science, vol. 697. Springer Verlag, Berlin. 59–70.

Zaks, A., Shlyakhter, I., Ivančić, F., Cadambi, H., Yang, Z., Ganai, M., Gupta, A., and Ashar, P. 2006. Range analysis for software verification. In *4th International Workshop on Software Verification and Validation (SVV)*.

Received July 2006; revised November 2006; accepted December 2006