Symbolic Model Checking of Concurrent Programs Using Partial Orders and On-the-Fly Transactions

Vineet Kahlon[1], Aarti Gupta[1], and Nishant Sinha[2]

[1] NEC Laboratories America, Princeton, NJ 08540, USA
[2] Carnegie Mellon University, Pittsburgh, PA 15213, USA

Abstract. The state explosion problem is one of the core bottlenecks in the model checking of concurrent software. We show how to ameliorate the problem by combining the ability of partial order techniques to reduce the state space of the concurrent program with the power of symbolic model checking to explore large state spaces. Our new verification methodology involves translating the given concurrent program into a circuit-based model which gives us the flexibility to then employ any model checking technique of choice – either SAT or BDD-based – for verifying a broad range of linear time properties, not just safety. The reduction in the explored state-space is obtained by statically augmenting the symbolic encoding of the program by additional constraints. These constraints restrict the scheduler to choose from a minimal *conditional stubborn set* of transitions at each state. Another key contribution of the paper, is a new method for detecting transactions *on-the-fly* which takes into account patterns of lock acquisition and yields better reductions than existing methods which rely on a lockset based analysis. Moreover unlike existing techniques, identifying on-the-fly transactions does not require the program to follow a lock discipline in accessing shared variables. We have applied our techniques to the Daisy test bench and shown the existence of several bugs.

1 Introduction

The widespread use of concurrent software in modern day computing systems necessitates the development of effective verification methodologies for multi-threaded programs. However, subtle interactions between threads makes multi-threaded software behaviorally complex and hard to analyze necessitating the use of formal methodologies for their debugging. It is not surprising then that the use of model checking – both symbolic and explicit state – for the verification of concurrent software has recently been an active area of research.

Explicit state model checkers, such as Verisoft [God97] rely on exploring an enumeration of the states and transitions of the concurrent program at hand. Additional techniques such as state hashing for compaction of state representations, and partial order methods are typically used to avoid exploring all interleavings of transitions of the constituent threads. While these techniques are powerful tools for state space reduction, they still do not fully address the scalability issues that arise due to state explosion when model checking large-scale concurrent programs.

Symbolic model checkers, on the other hand, avoid an explicit enumeration of the state space by using symbolic representations of sets of states and transitions. One of

© Springer-Verlag Berlin Heidelberg 2006

the first successful approaches in this regard was the use of BDDs to succinctly represent large state spaces for the purpose of model checking [McM93]. More recently, SAT-based techniques [BCCY99] have become popular both for finding bugs using SAT-based Bounded Model Checking (BMC) and for generating proofs via SAT-based Unbounded Model Checking (UMC).

One of the contributions of this paper is that we have proposed a new methodology to leverage the synergy that results from combining the ability of partial order techniques to reduce the state space of the system to be explored with the power of symbolic model checking techniques to explore large state spaces that has many advantages over existing techniques that attempt to achieve the same goals. Indeed, methods different from ours that combine partial order reductions with the use of BDDs were given in [ABH$^+$01, LST03]. However, the use of BDDs requires one to first symbolically encode the entire state space of the given concurrent program thereby running into the state explosion problem. Our technique gives us the freedom to use any technique of choice, either SAT or BDD-based. This is crucial as SAT-based BMC techniques tend to be much more scalable on larger programs than the ones based on the use of BDDs.

We start by translating a given concurrent program into a circuit-based (finite-state) model. Building upon the F-Soft framework [ISGG05] for translating sequential programs with bounded data and bounded recursion into circuits, we first obtain a finite model for each individual thread wherein each variable of the thread is represented in terms of a vector of binary-valued *latches* and a boolean next-state function (or relation) for each latch. Then using a scheduler, we compose the circuits for the individual threads into one single circuit for the entire concurrent program. Verification is then carried out on this circuit. Partial order techniques are incorporated into the framework by statically augmenting the circuit-based boolean encoding of the given concurrent program with additional constraints. These constraints restrict the transitions explored from each global state to a minimal *conditional stubborn set* of that state.

Another contribution of this paper is that we have proposed a new provably better method for identifying transactions *on-the-fly* that is based on analyzing patterns of lock acquisition as opposed to existing techniques [Sto02, FQ03] which rely on a lockset based analysis. Lockset based methods for state space reduction essentially exploit the ability of locks to enforce mutually exclusive access to regions of code encapsulated between the locking and unlocking operations on the same lock. They rely on the assumption that the given concurrent program follows a *lock discipline* in accessing shared variables, i.e., all accesses to a shared variable sh are protected by the same lock l_{sh} [Sto02, FQ03]. Then we can cut down on the number of interleavings that need to be explored by essentially allowing context switches only before the acquire and after the release operations on l_{sh} and prohibiting them before access to sh. Disallowing context switches increases the granularity of transitions and cuts down on the number of possible interleavings resulting in a reduced state space to be explored.

On the other hand, by analyzing concurrent programs for patterns of lock acquisition rather than for locksets, we can identify not only those transactions which lockset based method do but also some that they don't. This makes our new technique provably better. In fact, the lockset based technique for identifying transactions turns out to be a special case of the one based on lock acquisition patterns that we propose here.

Moreover, our technique does not rely on the given concurrent program following a locking discipline in accessing shared variables. An important advantage of the non-reliance of our method on lock discipline is that one of the main reasons for the existence of data races in threads is an unprotected/wrongly protected access to a shared variable. The requirement of lock discipline precludes the application of these powerful reductions to programs where such commonly occurring bugs are present. Thus our method enables the use of lock-based reductions for a broader class of concurrent programs, viz., that need not follow lock discipline, to catch a frequently occurring class of bugs.

Another, important feature of the lock-pattern based transactions is that they can be transparently incorporated into partial order reduction by improved conditional dependency detection via addition of extra constraints that are incorporated into the transition relation not a priori but dynamically while unrolling the executions of the threads. We show that the increased granularity of transitions due to transactions can be captured as a reduction in the sizes of the conditional stubborn sets of states.

We believe that our decision to build circuit-based models for concurrent programs gives us many unique advantages. Indeed, in this sense, the work most closely resembling ours are the approaches presented in [RG05, CKS05] that involve translating a `C` program directly into a SAT formula for model checking using SAT-based BMC. However [RG05] does not incorporate partial order reductions and neither technique leverages on-the-fly transactions. Circuit based models make it easy to incorporate static space reduction techniques like partial order reductions, on-the-fly-transactions as well as lightweight static analysis techniques like range analysis to reduce model sizes. Another advantage of our approach lies in the separation of the model building and verification phases. Once we have built a circuit for the concurrent program at hand, it affords us the flexibility to tackle the verification problem using any model checking technique of choice for a broad range of linear time temporal properties, not just safety. Unlike [RG05, CKS05], we can employ a suite of model checking tools for a rich class of *linear-time* temporal properties, which can be used *both* for finding bugs and generating proofs. These include SAT-based BMC and UMC as well as BDD based model checking. We believe this flexibility is important as software generated circuits are not as well structured as hardware circuits and hence no one strategy can be expected to be universally effective. Thus we have presented a new approach for model checking concurrent programs that combines the power of symbolic techniques with partial order reduction and on-the-fly transactions while at the same time retaining the flexibility to employ a broad arsenal of model checking techniques – both SAT and BDD-based – for checking not just reachability but a richer classes of linear-time temporal properties.

In the rest of the paper, Section 2 introduces the system model while on-the-fly transactions are defined in section 3. The details for modeling concurrent programs as circuits are provided in section 4 and the Daisy case study in section 5. Finally, we conclude with some remarks in section 6 along with a comparison with related work.

2 System Model

We consider concurrent systems comprised of a finite number of processes or threads where each thread is a deterministic sequential program written in a language such as

C. Threads interact with each other using communication/synchronization objects like shared variables, locks and semaphores.

Formally, we define a concurrent program $\mathcal{CP}$ as a tuple $(\mathcal{T}, \mathcal{V}, \mathcal{R}, s_0)$, where $\mathcal{T} = \{T_1, ..., T_n\}$ denotes a finite set of threads, $\mathcal{V} = \{v_1, ..., v_m\}$ a finite set of shared variables and synchronization objects with v_i taking on values from the set V_i, $\mathcal{R}$ the transition relation and s_0 the initial state of $\mathcal{CP}$. Each thread T_i is represented by the control flow graph of the sequential program it executes, and is denoted by the pair (C_i, R_i), where C_i denotes the set of control locations of T_i and R_i its transition relation. A global state s of $\mathcal{CP}$ is a tuple $(s[1], ..., s[n], v[1], ..., v[m]) \in \mathcal{S} = C_1 \times ... \times C_n \times V_1 \times ... \times V_m$, where $s[i]$ represents the current control location of thread T_i and $v[j]$ the current value of variable v_j. The global state transition digram of $\mathcal{CP}$ is defined to be the standard interleaved composition of the transition diagrams of the individual threads. Thus each global transition of $\mathcal{CP}$ results by firing a local transition of the form (a_i, g, u, b_i), where a_i and b_i are control locations of some thread $T_i = (C_i, R_i)$ with $(a_i, b_i) \in R_i$; g is a guard which is a Boolean-valued expression on the values of local variables of T_i and global variables in $\mathcal{V}$; and u is function that encodes how the value of each global variable and each local variable of T_i is updated. A transition $t = (a_i, g, u, b_i)$ of thread T_i is enabled in state s iff $s[i] = a_i$ and guard g evaluates to true in s. If $s[i] = a_i$ but g need not be true in s, then we simply say that t is *scheduled* in s. We write $s \xrightarrow{t} s'$ to mean that the execution of t leads from state s to s'. Given a transition $t \in \mathcal{T}$, we use $proc(t)$ to denote the process executing t. Finally, we note that each concurrent program $\mathcal{CP}$ with a global state space $\mathcal{S}$ defines the global transition system $A_G = (\mathcal{S}, \Delta, s_0)$, where $\Delta \subseteq \mathcal{S} \times \mathcal{S}$ is the *transition relation* defined by $(s, s') \in \Delta$ iff $\exists t \in \mathcal{T} : s \xrightarrow{t} s'$; and s_0 is the initial state of $\mathcal{CP}$.

3 Lock Synchronization Based Reductions

We start by using some examples to motivate our technique. Consider the concurrent program $\mathcal{CP}$ shown in figure 1. Here x, which is the only variable shared among the threads, is unprotected at control location 5b and protected by lock lk at all other locations. Since x is not protected at all locations where it is accessed, it does not satisfy lock discipline in the sense of [Sto02, FQ03], which will therefore force a context switch before locations 3a and 3b. Consider, however, a global state s of $\mathcal{CP}$ with threads T_1 and T_2 at control locations 3a and 1b, respectively. The key observation is that starting at global state s of $\mathcal{CP}$, 3a does not interfere with 3b and 5b even though 5b is unprotected. This is because for T_2 to execute 3b it has to acquire lk currently

```
1a: a = 0;              1b: z = 5;
2a: lock(lk);           2b: lock(lk);
3a: x = 1;              3b: x = 2;
4a: unlock(lk);         4b: unlock(lk);
5a: a = 4;              5b: x = 6;
          (a)                     (b)
```

Fig. 1. Threads T_1(a) and T_2(b) with unprotected access to x

held by T_1. But in order for T_1 to release `lk`, it has to first execute `3a`. Thus starting at s, $\mathcal{CP}$ is forced to execute `3a` before `3b`. As a result no context switch is required before `3a`. However, in the global state s' with T_1 and T_2 at control locations `3a` and `5b`, respectively, the transitions `3a` and `5b` do interfere with each other thus forcing a context switch before `3a`. The bottom line is that even when shared variables do not follow locking discipline globally, we can still identify local portions of the state space where locking discipline is followed. Thus a context driven analysis allows us to define transactions locally *on-the-fly* where existing methods [Sto02, FQ03], because of their reliance on a global analysis, fail to do so.

```
1a: a = 1;              1b: b = 0;
2a: lock(lk1);          2b: lock(lk2);
3a: lock(lk2);          3b: lock(lk1);
4a: y = 1;              4b: z = 2;
5a: unlock(lk2);        5b: unlock(lk1);
6a: x = 0;              6b: x = 1;
7a: unlock(lk1);        7b: unlock(lk2);
        (a)                     (b)
```

Fig. 2. Threads T_1(a) and T_2(b) with unprotected access to x

Taking the above discussion further, we next show that transactions can be identified even in the absence of lock discipline–local or global. Let $\mathcal{CP}$ be the concurrent program comprised of the two threads T_1 and T_2 sharing variable `x` shown in figure 2. Consider a global state s of $\mathcal{CP}$ with threads T_1 and T_2 in control locations `6a` and `1b`, respectively. Observe that starting at s, the transitions at control locations `6a` and `6b` cannot interfere with each other even though they access the same shared variable `x`. This is because in order for thread T_2 to reach location `6b` from `1b` it has to traverse the local path `1b,2b,3b,4b,5b`, along which it has to acquire (and release) lock `lk1` currently held by T_1. In order for that to happen, T_1 must release `lk1` for which it must execute transition `6a`. This forces transition `6a` to be executed before `6b`. Thus no context switch is required before location `6a`. The key observation is that even though disjoint sets of locks were held at locations `6a` and `6b`, it was the set of locks that needed to be acquired by T_2 in order to transit from `1b` to `6b` (even though some of these locks were released before reaching `6b`) that prevented `6a` and `6b` from interfering with each other. A traditional lockset based analysis as given in [Sto02, FQ03] would treat `6a` and `6b` as conflicting transitions (as `x` does not follow locking discipline) and force a context switch before these locations. Thus a conflict analysis based on lock acquisition patterns is more refined than one based on locksets. Indeed, a lockset based analysis is a special case of lock-pattern based analysis since the set of locks held at a location would have to be acquired and thus would be tracked in the lock acquisition pattern.

Transactions Via Persistent Sets. We now show how to integrate lock-pattern based on-the-fly transactions with partial order reduction in a transparent fashion by capturing the increased granularity of transitions due to transactions as a reduction in the sizes of the conditional stubborn sets of states. This is accomplished by ensuring that if in a global state s, a thread T_i is in the process of executing a transaction, then in the persistent set of s, we include only one transition, viz., the transition of T_i that fires next along the transaction being executed. This ensures that once the first transition of a transaction is executed, by a thread T_i then no other process can be scheduled unless all transitions of the transaction finish firing.

State space reduction using partial order techniques is obtained by exploring from each state only those transitions that belong to a persistent set of that state instead of all the enabled transitions. Although there are many ways to compute persistent sets, the method of computing conditional stubborn sets usually generates those with small cardinality. In this paper, we use standard terminology from the theory of partial order reductions and the algorithm for computing conditional stubborn sets from [God96], which we denote by $Algo_1$. We recall the following definition from [God96].

Might-be-first-to-interfere. *Let op and op′ be two operations on the same object O and s be a reachable state. The relation* $op \rhd_s op'$ *holds if there exists a sequence* $s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} ... \xrightarrow{t_n} s_{n+1}$ *of transitions in* A_G *such that* $\forall 1 \leq i < n : \forall op''$ *on O used by* t_i*: op and op″ are independent in state* s_i, t_n *uses op′, and op and op′ are dependent in* s_n.

For each local transition $a \xrightarrow{g} b$ of a thread, we let $used(t)$ denote the set of operations on variables and synchronization objects executed during the execution of t. A conditional stubborn set of state s of A_G can then be calculated as follows:

```
1. Initialize $T_s = \{t\}$, where $t$ is some enabled transition in $s$.
2. For each $t = a \xrightarrow{g} b \in T_s$
   (a) If $t$ is disabled in $s$,
       i. if $T_j = proc(t)$ and $s[j] \neq a$, then add to $T_s$ all transitions $t'$ of $T_j$ of the form
          $c \xrightarrow{g'} a$, or
       ii. choose a condition $c_j$ in the guard $g$ of $t$ that evaluates to false in $s$; then, for
           all operations $op$ used by $t$ to evaluate $c_j$, add to $T_s$ all transitions $t'$ such that
           $\exists op' \in used(t') : op \rhd_s op'$.
   (b) If $t$ is enabled in $s$ add to $T_s$ all transitions $t'$ such that
       i. $proc(t) \neq proc(t')$ and $\exists op \in used(t), \exists op' \in used(t') : op \rhd_s op'$.
3. Repeat step 2 until no more transitions can be added in $T_s$. Then return all transitions in $T_s$
   that are enabled in $s$.
```

Fig. 3. $Algo_1$ for Computing Conditional Stubborn sets

In $Algo_1$ dependencies between transitions, arising out of operations on shared communication objects are captured using the $\rhd_s$ relation which captures for each operation *op* used by a transition in a state s which other operations *might be first to interfere with op from the current state* s. In practice, to avoid exploration of the state space of the program at hand, static analysis is employed in order to compute a relation, $\rhd_s^{st}$, which is an over-approximation of $\rhd_s$. Towards that end, we say that two operations *op* and *op′* are *statically dependent* if they access a common shared variable such that at least one of the accesses is a write operation. Then $\rhd_s^{st}$, is defined as follows.

Definition. *Let op and op′ be two operations on a common shared variable and s a reachable state of* A_G. *The relation* $op \rhd_s^{st} op'$ *holds iff there exist distinct threads* T_i

and T_j such that there exists (1) a transition of T_i scheduled (not necessarily enabled) at s using op, and (2) a local path $x : p_0 \xrightarrow{t_1} ... \xrightarrow{t_n} p_n$ of T_j such that p_0 is the local state of T_j in s, $\forall 1 \leq k < n$: $\forall op''$ used by t_k: op and op$''$ are not statically dependent, t_n uses op$'$, and op and op$'$ are statically dependent.

To incorporate on-the-fly transactions, we modify the above definition of $\rhd_s^{st}$ to get a new relation $\rhd_s^{lp} \subseteq \rhd_s^{st}$ by adding (in accordance with our discussion above), the extra constraint that none of the locks held by T_i in s is acquired (and possibly released) by T_j along x. Note that since $\rhd_s^{lp}$ is more constrained it enforces fewer dependencies between operations than $\rhd_s^{st}$ thus resulting in smaller conditional stubborn sets. The effect is to weed out certain interleavings to get the effect of executing transactions. Indeed, in the example given in fig 2, in global state s, if *op* and *op$'$* are the operations $x = 0$ and $x = 1$ at locations `6a` and `6b`, respectively, then $op \rhd_s^{st} op'$ but $\neg(op \rhd_s^{lp} op')$. Thus, using $\rhd_s^{lp}$ instead of $\rhd_s^{st}$ to compute conditional stubborn sets removes transition `1b` from the conditional stubborn set of s thus preventing a context switch before `6a`. Formally, $\rhd_s^{lp}$ is defined as follows.

Definition (might-be-the-first-to-interfere-modulo-lock-acquisition). *Let op and op$'$ be two operations on a common shared variable and s a reachable state of A_G. The relation op $\rhd_s^{lp}$ op$'$ holds iff there exist distinct threads T_i and T_j such that there exists (1) a transition of T_i scheduled (not necessarily enabled) at s using op, and (2) a local path $x : p_0 \xrightarrow{t_1} ... \xrightarrow{t_n} p_n$ of T_j such that $\forall 1 \leq k < n$: $\forall op''$ used by t_k: op and op$''$ are not statically dependent, t_n uses op$'$, and op and op$'$ are statically dependent and no lock held by T_i in s is acquired by T_j along x.*

Let $Algo_2$ be the result of replacing $\rhd_s$ in $Algo_1$ by $\rhd_s^{st}$ and $Algo_3$ the result of replacing $\rhd_s^{st}$ in line 2.(b).i of $Algo_2$ by $\rhd_s^{lp}$. Then the following two results state that $Algo_3$ does indeed compute a conditional stubborn set and that, in fact, it computes smaller conditional stubborn sets than $Algo_2$. Note that although we used a specific relation $\rhd_s^{st}$ for computing dependencies statically, one can, of course, incorporate on-the-fly-transactions with any other *implementation* of $\rhd_s$ by merely adding the extra condition regarding lock acquisition patterns, as above.

Theorem 1. *All sets T_s that are computed by $Algo_3$ are conditional stubborn sets of s.*

Proof Sketch. Let $t = a \xrightarrow{g} b$ executed by thread T_i belong to T_s. Let $w = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} ... \xrightarrow{t_n} s_{n+1}$ be a sequence of transitions of A_G such that t is dependent with t_n in s_n. We need to show that at least one of t_1,...,t_n is in T_s. Without loss of generality, we may assume that for $1 \leq i < n$, t is independent with t_i in s_i and t_n is dependent with t in s_n, else we can pick an appropriate prefix of w.

First assume that t is disabled in s. Since t is disabled in s and s_n is the first state along w in which t is dependent (with t_n), we have that t is enabled in s_{n+1}. Since t is disabled in s, either $s[i] \neq a$, or a condition c in guard g evaluates to false in s. In the first case, since t is enabled in s_{n+1}, there exists a transition t_j fired along w, of the form $d \rightarrow a$ labeled with some guard g'. But then executing step 2.(a).i of $Algo_3$, would cause t_j to be included in T_s. In the second case, there exists a transition t_j, that changes the value of c from false to true by changing the output of an operation *op* used to evaluate *c*, i.e., by performing an operation *op$'$* dependent with *op* in s_j. Let t_j be the first such transition occurring along w. Clearly op' is statically dependent with op. By definition of $\rhd_s^{st}$, we have $op \rhd_s^{st} op'$, and so $t_j \in T_s$ by step 2.a.(ii).

Consider now the case when t is enabled in s. From the facts that (i) for $1 \leq j \leq n-1$, t is independent with t_j in s_j, and (ii) t is enabled in s, we have that for $1 \leq j \leq n-1$, t is enabled in s_j. This implies that thread T_i does not execute any transition along w, for otherwise since T_i is deterministic, we can conclude that t is the first transition that T_i executes along w. This which would force T_i out of it current local state thereby disabling t thus contradicting the above observation. Note that here we assumed that executing a transition takes a process out of its current local state, i.e., there are no self loops in a program thread, a reasonable assumption for software programs Now, since t and t_n are dependent in s_n, it implies that $\exists op \in used(t), \exists op' \in used(t_n)$: op and op' are dependent in s_n and hence are also statically dependent. Let t_j be the first transition along w that uses an operation op'' dependent op. Note also that there does not exist a lock l held by T_i at s such that l has to be acquired before t_j is executed along w. For otherwise, l must first be released by T_i thus forcing T_i to execute a transition contradicting our observation above that T_i does not execute any transition along w. Thus we have $op \rhd_s^{lp} op''$. Hence $t_j \in T_s$ by step 2.b.(i). This completes the proof. □

Theorem 2. *For all transitions t that are enabled in s, for all persistent sets $Algo_2(t)$ that can be returned by $Algo_2$, there exists a run of $Algo_3$ that returns a persistent set $Algo_3(t) \subseteq Algo_2(t)$.*

Proof Sketch. From the definition of relation $\rhd_s^{lp}$, it follows that $\rhd_s^{lp}$ is included in $\rhd_s^{st}$. Thus the set T_s returned by $Algo_3$ is always a subset of the one returned by $Algo_2$, provided the same choices are made in case of nondeterminism. □

Note that since $Algo_3$ computes smaller persistent sets than existing lockset-based techniques, it is guaranteed to improve the performance of explicit state model checkers. Even for symbolic model checkers, since the reduction in the number of scheduled transitions results in a pruning of the state space, it leads to a performance boost which, however, may not be directly proportional to the decrease in the size of the state space being explored.

4 Software Modeling for Concurrent `C` Programs

4.1 Translating Individual Threads into Circuits

In this section we briefly describe how, using the F-Soft machinery, we first obtain a circuit-based model of each thread, under the assumption of bounded data and bounded control (recursion) (see [ISGG05] for more details).

We begin with full-fledged `C` and apply a series of source-to-source transformations to simplify complex `C` expressions into smaller but equivalent subsets of `C` . We flatten all arrays and `structs` by replacing them with collections of simple scalar variables, and build an internal memory representation of the program by assigning to each scalar variable a unique number representing its memory address. Variables that are adjacent in `C` program memory are given consecutive memory addresses in our model; this facilitates modeling of pointer arithmetic. We model the heap as a finite array, adding a simple implementation of `malloc()` that returns pointers into this array. For handling pointer accesses, we first perform a points-to analysis to determine the set of variables that a pointer variable can point to. Then, we convert each indirect memory access, through a pointer or an array reference, to a direct memory access. For example, if we determine that pointer `p` can point to variables `a,b,...,z` at a given program location, we rewrite a pointer read `*(p+i)` as a conditional expression of the form `((p+i)==&a ? a : ((p+i)==&b ? b : ...))`, where `&a,&b,...` are the numeric memory addresses we assigned to the variables `a,b,...`, respectively. Non-recursive function calls are handled by inlining exactly once, and replacing the function return by a set of goto-s conditioned upon the unique call site id stored on function entry. Bounded recursive functions are modeled by introducing a bounded call stack. While we aim for accurate modeling of all `C`, practical modeling requires making approximations. We truncate large arrays: writes to elements above a certain index are ignored, and reads from these elements yield non-deterministic values. We currently approximate floating-point values by modeling their integral parts only.

The simplified program consists of scalar variables of simple types (Boolean, enumerated, integer). This is compiled using standard techniques into its control flow graph (CFG). The CFG representation can be viewed as a finite state machine with state vector `(pc,V)`, where `pc` denotes an encoding of the basic blocks, and `V` is a vector of integer-valued program variables. We then construct symbolic transition relations for `pc`, and for each data variable appearing in the program. For `pc`, the transition relation reflects the guarded transitions between basic blocks in the CFG. For a data variable, the transition relation is built from expressions assigned to the variable in various blocks. Finally, we construct a symbolic representation of these transition relations resembling a hardware circuit. For the `pc` variable, we allocate $\lceil \log N \rceil$ latches, where N is the total number of basic blocks. For each `C` program variable, we allocate a vector of n latches, where n is the bit width of the variable. At the end, we obtain a circuit-based model of each thread of the given concurrent program, where each variable of the thread is represented in terms of a vector of binary-valued *latches* and a Boolean next-state function (or relation) for each latch.

4.2 Building the Circuit for the Concurrent Program

Given the circuit $\mathcal{C}_i$ for each individual thread T_i, we now show how to get the circuit $\mathcal{C}$ for the concurrent program $\mathcal{CP}$ comprised of these threads. In the case where local variables with the same name occur in multiple threads, to ensure consistency we prefix the name of each local variable of thread T_i with `thread_i`. Next, for each thread T_i we introduce a gate `execute_i` indicating whether P_i has been scheduled to execute in the next step of $\mathcal{CP}$ or not.

For each latch `l`, let *next-state*$_i$(`l`) denote the next state function of `l` in circuit $\mathcal{C}_i$. Then in circuit $\mathcal{C}$, the next state value of latch `thread_i_l` corresponding to a local variable of thread T_i, is defined to be *next-state*$_i$(`thread_i_l`) if `execute_i` is true, and the current value of `thread_i_l`, otherwise. If, on the other hand, latch `l` corresponds to a shared variable, then *next-state*(`l`) is defined to be *next-state*$_i$(`l`), where `execute_i` is true. Note that we need to ensure that `execute_i` is true for exactly one thread T_i. Towards that end, we implement a scheduler which determines in each global state of $\mathcal{CP}$ which one of the signals `execute_i` is set to *true* and thus determines the semantics of thread composition.

Conditional Stubborn Sets Based Persistent Sets. To incorporate partial order reduction, we need to ensure that from each global state s, only transitions belonging to a conditional stubborn set of s are explored. Let $\mathcal{R}$ and R_i denote the transitions relations of $\mathcal{CP}$ and T_i, respectively. If $\mathcal{CP}$ has n threads, we introduce the n-bit vector $cstub$ which identifies a conditional stubborn set for each global state s, i.e., in s, $cstub_i$ is *true* for exactly those threads T_i such that the (unique) transition of T_i enabled at s belongs to the same minimal conditional stubborn set of s. Then

$$\mathcal{R}(s, s') = \bigvee_{1 \le i \le n} ((\texttt{execute\_i}) \wedge cstub_i(s) \wedge R_i(s, s')).$$

The $cstub$ vector can be computed in the following way:

1. For each shared variable x and thread T_i, we introduce a latch *touch-now*(T_i, x) which is true at control location pc_i of T_i iff T_i accesses x at control location pc_i. This can be done via a static analysis of the CFG of T_i by determining at which control locations x was accessed and taking a disjunction for those values of pc_i.
2. For each shared variable x and thread T_j, introduce the latch *touch-now-later* (T_j, x), which is true at control location pc_j of T_j if T_j accesses x at some location pc'_j reachable from pc_j. Thus computing *touch-now-later*(T_j, x) involves deciding the reachability of pc'_j, and since we cannot compute it exactly without exploring the entire state space A_G of $\mathcal{CP}$, we over-approximate it by doing a context-sensitive analysis of the control-flow graph of T_j. We set *touch-now-later-pair* (T_j, x) to *true* in control pc_j if for some control pc'_j reachable from pc_j in the control flow graph of T_j, x is accessed at pc'_j.
3. For distinct threads T_i and T_j, the relation $conflict_i(j)$ is then defined as $\vee_{x \in \mathcal{V}_{sh}}$ (*touch-now*$(T_i, x)(pc_i) \wedge$ *touch-now-later*$(T_j, x)(pc_j)$), where pc_i and pc_j are the control locations of T_i and T_j, respectively, in the current global state and $\mathcal{V}_{sh}$ is the set of shared variables of $\mathcal{CP}$.
4. Using a circuit to compute transitive closures, for each i, starting with $J_i = \{i\}$ we compute the closure of J_i under the *conflict* relation defined above.
5. We build a circuit to compute the index min such that the cardinality of J_{min} is the least among the sets $J_1, ..., J_n$. Finally $\forall 1 \le i \le n$, set $cstub_i = 1$ iff $i \in J_{min}$. Note that in the implementation we need to pick only one set with the least cardinality.

Cycle Detection. We first identify *sticky* transitions [KLM$^+$98] for all potential global cycles. We then force a conflict for the process containing the sticky locations with all other processes via the encoding below. Let $sticky(pc)$ be a predicate evaluating to true iff location pc has been marked sticky. Then, for global state s, we define $conflict_i(j)$ = $sticky(pc_i) \vee$ (*touch-now*$(T_i, x)(pc_i) \wedge$ *touch-now-later*$(T_j, x)(pc_j)$), where pc_m is the current control location of T_m in s. In other words, if pc_i is sticky then thread T_i is said to conflict with all other threads.This implies that either a thread T_k, with smaller conflict set J_k, would be chosen for the persistent set computation or a full expansion forced.

This reduction is sound, since (as was shown in [KLM$^+$98]) any cycle in the global state space can be projected on to one or more local cycles in the control flow graph of the individual threads. By forcing a full expansion inside each (potential) local cycle with the help of sticky transitions, we ensure that there is no global cycle such that a thread transition is postponed at each state of the cycle. Therefore this encoding allows the model checker to explore a conservative over-approximation of the representative (minimal) set of interleavings of the given threads. Although the reduced model remains sound, the number of interleavings considered may decrease dramatically with the number of annotated sticky transitions.

So far, we have implemented sticky transitions only for special cases in which cycles can occur locally in threads. In fact, as was noted in [FG05], our experience also has been that acyclic state spaces are very common in software implementations for the purpose of model checking and cycle detection becomes more critical when one is using an abstraction (which introduces cycles) refinement framework. However since (i) we put a lot of effort in modeling programs concretely, (ii) do not use abstraction refinement, and (iii) introduce sticky transitions to cover common trivial cases, the impact of the existence of cycles is reduced. Nevertheless, we are currently in the process of extending the implementation of sticky transitions to the general case.

Encoding Lock Pattern Based Reductions. In order to incorporate transactions *on-the-fly*, we augment the predicate *touch-now-later*, to generate the new predicate *touch-now-later-LS* that also includes lock acquisition pattern information. For control locations pc_i and pc_i', of thread T_i, let $paths(pc_i, pc_i')$ denote the set of paths in the CFG of T_i starting from pc_i that may reach pc_i'. For each $\pi \in paths(pc_i, pc_i')$ of T_i, let $lockPred(\pi)$ be a formula denoting the set of locks acquired (and possibly released) along π, e.g., $lk_1 = T_i \wedge lk_2 = T_i$. Let *touch-now-later-pair*$(T_j, x)(pc_j, pc_j')$ encode all possible sets of locks that can potentially be acquired along local paths in T_i from pc_i to pc_i' accessing x, i.e., *touch-now-later-pair*$(T_j, x)(pc_j, pc_j')$ = *touch-now*$(T_j, x)(pc_j') \wedge AP_x(pc_j, cp_j')$, where $AP_x(pc_i, pc_i') = \bigvee_{\pi \in paths(pc_i, pc_i')} lockPred(\pi)$. Let $CLP(\ T_i, s)$ denote a formula encoding the ownership of locks by T_i in global state s. Then the relation *touch-now-LS*(T_i, x) is obtained from *touch-now-later-pair*(T_i, x) by quantifying out pc_i' and conjoining with the $CLP(T_i, s)$, i.e., *touch-now-LS*(T_i, x) $(pc_i) = (\exists pc_i'$ *touch-now-later-pair*(T_i, x) $(pc_i, pc_i')) \wedge CLP(T_i, s)$. Thus *touch-now-LS*$(T_i, x)$ (pc_i) is true if there is a location pc_i' accessing shared variable x that is reachable from pc_i via a local path π in T_i such that no lock held in s is acquired along π. We evaluate $lockPred(\pi)$ using a context sensitive static analysis of the CFG of T_i.

5 The Daisy Case Study

We have used our technique to find bugs in the Daisy file system which is a benchmark for analyzing the efficacy of different methodologies for verifying concurrent programs [dai]. Daisy is a 1KLOC Java implementation of a toy file system where each file is allocated a unique inode that stores the file parameters and a unique block which stores data. An interesting feature of Daisy is that it has fine grained locking in that access to each file, inode or block is guarded by a dedicated lock. Moreover, the acquire and release of each of these locks is guarded by a ‘token’ lock. Thus control locations in the program might possibly have multiple open locks and furthermore the acquire and release of a given lock can occur in different procedures.

Currently F-Soft only accepts programs written in C and so we first manually translated the Daisy code which is written in Java into C. Furthermore, to reduce the model sizes, we truncated the sizes of the data structures modeling the disk, inodes, blocks, file names, etc., which were not relevant to the race conditions we checked, resulting in a sound and complete *small-domain* reduction. We have shown the existence of the race conditions described below also noted by other researchers (cf. [dai]). The efficacy of our techniques can be judged from the fact that our model checking methodology has been able to detect these race conditions in Daisy in a fully automatic fashion directly on the source code without any code structuring/abstractions beyond redefining the constants as discussed above.

1. Daisy maintains an allocation area where for each block in the file system a bit is assigned 0 or 1 accordingly as the block has been allocated to a file or not. But each disk operation reads/writes an entire byte. Two threads accessing two different files might access two different blocks. However since bytes are not guarded by locks in order to set their allocation bits these two different threads may access the same byte in the allocation block containing the allocation bit for each of these locks thus setting up a race condition. Note that the race condition occurs for any pair of blocks with numbers i and j where $floor(i/8) = floor(j/8)$.

The verification statistics are as follows: We ran our experiments on a machine with an Intel Pentium4 3.20GHz processor and 2GB RAM. Each run was given a timeout of 2 days and had a memout of 2GB. Witnesses for the above race condition were found in two cases, WW_1–corresponding to blocks 0 and 1, and WW_2–due to blocks 1 and 2. Using purely interleaved scheduling, we failed to find either witness because of a memout at depth 15. When only partial order reduction was employed WW_1 was found using SAT-based BMC at unroll depth 122 in 36707 sec and 999MB while incorporating on-the-fly transactions drastically reduced the time and memory usage to 1283sec and 122MB, respectively. The second witness WW_2 was found at depth 151. Using partial order reduction alone took 145176 sec and 1870 MB, while adding transactions reduced it to 5925 sec and 902 MB.

2. In Daisy reading/writing a particular byte on the disk is broken down into two operations: a seek operation that mimics the positioning of the head and a read/write operation that transfers the actual data. Due to this separation between seeking and data transfer a race condition may occur. For example, reading two disk locations, say n and m, we must make sure that $seek(n)$ is followed by $read(n)$ without $seek(m)$ or $read(m)$ scheduled in between. In this case a witness was found at depth 48. Using partial order reduction alone took 2.99 sec and 5.7 MB while adding transactions reduced it to 2.89 sec and 5.5 MB. For this example also BMC on the completely interleaved model failed to find a witness because of a memout at depth 20.

The bottom line is that, for deep bugs techniques that leverage the use of on-the-fly transactions combined with partial order reduction greatly outperform those which use only partial order reduction – both in terms of time taken and memory used.

6 Concluding Remarks and Related Work

A comparison of our work with [RG05, CKS05], to which it is most closely related, was presented in the introduction. Partial order reduction has been used before for symbolic model checking using BDDs [ABH$^+$01, LST03]. On the other hand, by separating the modeling and verification phases, our methodology gives us the ability to combine partial order reductions with any symbolic model checking technique of choice, either SAT or BDD based. An interesting approach for the verification of concurrent programs using proof-guided under-approximation-widening methodology was presented in [GLST05]. Here constraints are added to the BMC model instance so that only a subset of behaviors of the concurrent system are explored. These constraints are iteratively removed during the widening phase as a result of which, in the worst case, one might end up exploring the entire state space of the concurrent program at hand. In contrast, we add constraints so that we explore a conditional stubborn set at each global state thereby yielding considerable state space reduction. Moreover, [GLST05] does not leverage the use of transactions.

There has also been interesting work ([FQ03, Sto02, SC03, AQR$^+$04, LPQR05]) on the use of lockset based transactions for verifying software and combining it with partial order reductions. These techniques first compute the valid set of transactions in each of the processes and then perform partial order reduction-based state-space exploration. As noted before, such a two-step combination technique may overlook potential reductions related to shared variables which do not always follow a locking discipline. The key reason is that in these approaches a thread-wise global analysis is done to look for potential dependencies between transitions. In contrast, our approach adds information to the model while exploring the state space by detecting dependencies *on-the-fly* via an analysis of patterns of lock acquisition. Our more refined method generates fewer dependencies between transitions resulting in a lesser number of context switches. This gives us better state space reduction than existing lockset based techniques.

To sum up, we have presented a new approach for verifying concurrent programs that combines the power of symbolic model checking with partial order reduction and on-the-fly transactions while at the same time retaining the flexibility to employ a variety of error trace generation/proof techniques – both SAT and BDD-based – for checking not just safety but a broad class of linear time temporal properties. The use of lock acquisition patterns rather than locksets to identify transactions *on-the-fly* is not only a powerful technique in its own right but can also be used in a synergistic manner with both explicit state and BDD-based exploration of concurrent programs as also with dynamic partial order reduction techniques [FG05].

References

[ABH$^+$01] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state-space exploration. *Form. Methods Syst. Des.*, 18(2):97–116, 2001.

[AQR$^+$04] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *CONCUR*, 2004.

[BCCY99] A. Biere, A. Cimatti, E.M. Clarke, and Y.Zhu. Symbolic model checking without BDDs. In *TACAS*, 1999.

[CKS05] Byron Cook, Daniel Kroening, and Natasha Sharygina. Symbolic model checking for asynchronous boolean programs. In *SPIN 2005*, pages 75–90, 2005.

[dai] Joint CAV/ISSTA Special Event on Specification, Verification, and Testing of Concurrent Software. In *http://research.microsoft.com/ qadeer/cav-issta.htm*.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL '05*, pages 110–121, 2005.

[FQ03] C. Flanagan and S. Qadeer. Transactions for software model checking. In *SoftMC 03*, 2003.

[GLST05] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation widening for multi process systems. In *POPL '05*, pages 122–131, 2005.

[God96] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. LNCS 1032. Springer-Verlag, 1996.

[God97] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL '97*, pages 174–186, 1997.

[ISGG05] F. Ivančić, I. Shlyakhter, A. Gupta, and M. Ganai. Model checking c programs using F-Soft. In *ICCD*, 2005.

[KLM$^+$98] Robert P. Kurshan, Vladdimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün. Static partial order reduction. In *TACAS '98*, 1998.

[LPQR05] V. Levin, R. Palmer, S. Qadeer, and S. K. Rajamani. Sound transaction-based reduction without cycle detection. In *SPIN '05*, 2005.

[LST03] F. Lerda, N. Sinha, and M. Theobald. Symbolic model checking of software. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.

[McM93] K.L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. Kluwer Academic Publishers, 1993.

[RG05] I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *CAV '05*, pages 82–97, 2005.

[SC03] Scott D. Stoller and Ernie Cohen. Optimistic synchronization-based state-space reduction. In *TACAS '03*, LNCS, pages 489–504, April 2003.

[Sto02] Scott D. Stoller. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, 4(1):71–91, October 2002.