Bounded Model Checking of Concurrent Programs

Ishai Rabinovitz[1,2] and Orna Grumberg[1]

[1] Technion - Israel Institute of Technology
[2] IBM Haifa Research Laboratory, Haifa, Israel
ishai@il.ibm.com, orna@cs.technion.ac.il

Abstract. We propose a SAT-based bounded verification technique, called TCBMC, for threaded C programs. Our work is based on CBMC, which models sequential C programs in which the number of executions for each loop and the depth of recursion are bounded.

The novelty of our approach is in bounding the number of context switches allowed among threads. Thus, we obtain an efficient modeling that can be sent to a SAT solver for property checking. We also suggest a novel technique for modeling mutexes and Pthread conditions in concurrent programs. Using this bounded technique, we can detect bugs that invalidate safety properties. These include races and deadlocks, the detection for which is crucial for concurrent programs.

1 Introduction

In recent years there have been two main trends in formal verification. The first is that SAT-based Bounded Model Checking (BMC) [2] has become the leading technique for model checking of hardware. BMC constructs a propositional formula describing all possible executions of the system of length k, for some bound k. This formula, conjuncted with the negation of the specification, is fed into a SAT solver. If the formula is satisfied, the specification is violated.

The second trend is that software verification using formal methods has become an active research area. Special attention is given to verification of concurrent programs, in which testing tools often fail to find bugs that are revealed only with very specific inputs or timing windows.

However, adopting the BMC technique for software causes a severe problem. This technique is sensitive to the length of the error trace, i.e., the number of execution steps until an error state is reached. In software, error traces are typically quite long, and therefore a large bound k is needed. This, in turn, may result in a propositional formula that is too large to be handled by a SAT solver. Ivancic et al. [6] try to shorten the trace length by compressing multiple statements within one basic block into one complex statement. However, the resulting traces may still be too long.

C-Bounded Model Checking (CBMC) [4] presents a different approach to utilizing a SAT solver in order to verify software. CBMC translates a program

K. Etessami and S.K. Rajamani (Eds.): CAV 2005, LNCS 3576, pp. 82–97, 2005.

with no loops and no function calls into *single assignment form* (SSA form). In this form, variables are renamed so that each variable is assigned only once. As a result there is no need for a notion of state. Such a program can be viewed as a set of constraints and solved using a SAT solver. This technique is less sensitive to the length of a trace.

CBMC can also deal with pointers, arrays, and real size integers rather than just their restricted abstractions. This distinguishes it from other model checkers, which use abstractions in order to cope with size problems. Still, most if not all interesting programs include functions and loops. CBMC handles this by bounding the number of times each loop may be executed and unwinding the loop to this bound. It is then possible to inline function calls and even handle recursion (after bounding its depth as well). As in ordinary Bounded Model Checking, the bounds over the loops can be increased iteratively until a bug is found or the SAT solver explodes.

Each variable in a bounded program has a bounded number of assignments that can be indexed statically in an increasing order. CBMC translates the program in such a way that each indexed assignment is to a fresh variable, yielding a program in SSA form. This is very simple for sequential programs and was proven effective for some real-life examples [7].

However, it is not straightforward to extend this approach to concurrent programs. This is because it is not possible to index assignments to global variables statically. When there are assignments in two different threads to the same global variable, we cannot determine the order in which they will be executed.

In this paper we propose an extension of CBMC to concurrent C programs, called TCBMC (Threaded-C Bounded Model Checking). Concurrent C programs have shared memory and several threads that run concurrently. Each thread has its own local variables, whereas global variables are shared. Only one thread is executed at any given time, until, after an unknown period, a *context switch* occurs and another thread resumes its execution (see Figure 2). A set of consecutive lines of code executed with no intervening context switch is called a *context switch block*.

To obtain a bounded concurrent C program, TCBMC bounds the number of allowed context switches. This strategy is reasonable since most bug patterns have only a few context switches [5]. For each context switch block i and each global variable x, TCBMC adds a new variable val_x_i, which represents the value of x at the end of block i. It then models the concurrent program in SSA form, where the value of x in block $i+1$ is initialized to val_x_i.

The technique of bounding the number of context switches was independently suggested in [8], However [8] uses this idea on Boolean programs using pushdown automata.

Next we show how synchronization primitives such as mutexes and conditions can be modeled efficiently within TCBMC. We present a novel approach which, instead of modeling the internal behavior of a mutex, eliminates all executions in which a thread has waited for a mutex to unlock. We show that any bug which can be found in the naive model will also be found in our reduced model.

Our approach to modeling synchronization primitives is general, and as such, it is applicable to explicit and BDD-based symbolic model checkers as well. There, it decreases the number of interleavings and hence gains efficiency.

We next suggest how the TCBMC model can be altered to detect synchronization bugs such as races and deadlocks. Different extensions to the model are needed for each one. Thus, it will be more efficient to apply TCBMC three times: for detecting "regular" bugs, races, and deadlocks.

We implemented a preliminary version of TCBMC. This version supports only two threads. It supports mutexes and conditions, but it cannot detect deadlocks. Preliminary experiments show that TCBMC can handle a real representation of integers and that it performs well for data-dependent bugs.

The rest of this paper is organized as follows. The next section presents the preliminaries and explains CBMC. Section 3 presents TCBMC. Section 4 extends TCBMC to model mutexes and conditions. Section 5 describes another extension of TCBMC that allows for detection of races and deadlocks. Section 6 presents our experiments with TCBMC, and Section 7 outlines future work.

2 Preliminaries

A statement *uses* a variable when it reads its value, and it *defines* a variable when it writes a value to it. A statement *accesses* a variable when it either uses it or defines it.

In this paper we consider a *concurrent program* to be a program with several threads that share global variables. An *execution* of such a program starts to execute statements from a certain thread, after which it performs a context switch and continues to execute statements from another thread. It keeps track of the last statement executed in each thread and, when performing a context switch back to this thread, it continues the execution from the next statement.

A statement is *visible* if it accesses a global variable[1], and it is *invisible* otherwise. A *visible block* is a block of consecutive lines in which only the first statement is visible. A statement is *atomic* if no context switch is allowed during its execution. A sequence of consecutive statements is atomic if each statement is atomic and no context switch is allowed between them.

The *assert* function receives a Boolean predicate that should evaluate to True in all executions. Evaluation of assert to False indicates a bug in the program.

In this paper x and x_i are global variables, and y, y_i, w, w_i, z and z_i are local variables.

2.1 CBMC: C-Bounded Model Checking

CBMC [4] is a tool that gets a C program and an integer bound. It translates the program into a bounded program by unrolling each loop to the given

[1] A local variable that can be pointed by a global pointer is considered to be global for this definition.

bound, inlining functions (bounding also the number of recursion calls with the given bound). CBMC then takes the bounded C program and generates a set of constraints. There is a one-to-one mapping from the possible executions of the bounded program to the satisfying assignments of the set of constraints.

CBMC automatically generates cleanness specifications such as no access to dangling pointers, no access out of array bounds, and no assert violations. It adds a constraint which requires that one of these specifications be violated. It then activates a SAT solver over these constraints. If it finds a satisfying assignment to all the constraints, then it follows that there exists a valid execution that violates one of the specifications.

CBMC generates the constraints by translating the code to SSA form, in which each variable is assigned no more than once. To this aim CBMC generates several copies of each variable, indexed from zero to the number of assignments to this variable.

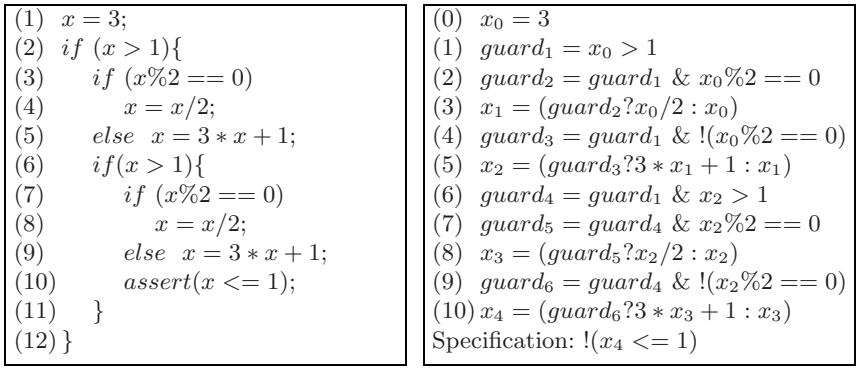
Each statement in a C program is executed only if all the "if" conditions that lead to it are evaluated to True. In order to reflect this in the generated constraint, CBMC also has several guard variables. Each guard variable is associated with the conjunction of all the conditions in the "if"s that lead to a certain statement in the code (If the statement is in the "else" clause of an "if" condition, the negation of the condition is used). Note that several statements may have the same guard.

CBMC is best understood by example. Assume CBMC gets the following C program with bound two.

```
x = 3;
while  (x > 1){
       if (x%2 == 0)  x = x/2;
       else           x = 3 * x + 1;
}
```

It first unrolls the loop, resulting in the program in Figure 1(a). It also adds an assert that ensures sufficient unrolling (This assert will fail in our example). Figure 1(b) presents the constraints representing this bounded code. Consider Constraint (3). It describes the behavior of Line (4) in the bounded code. This is the second assignment to x and therefore the constraint is on x_1. This statement is executed only if the two "if"s leading to it are True, i.e, $guard_2$ is True. The constraint presents the following behavior: if $guard_2$ is True, then $x_1 = x_0/2$; otherwise (the statement is not executed) $x_1 = x_0$ (x does not change).

As mentioned previously, CBMC supports the assert function and detects bugs in which an assert is violated. CBMC also supports the assume function. This function informs CBMC that all legal executions of the program must satisfy a certain constraint. Assume is the opposite of assert in the sense that when the constraint does not hold in a certain execution, CBMC ignores this execution without complaining.



(a) Bounded C code (b) Constraints

Fig. 1. Translation from bounded code to constraints

Pointers and Arrays. In CBMC, every assignment to a dereference of a pointer is actually instantiated to several assignments, one for each possible value of the pointer. The instantiations are limited to values that the pointer might have gotten in the previous assignments. Here is the code for the statement $*p_1 = 3$; where the indexes are for a possible program in which this statement appears.

$$x_{12} = (p == \&x)?3 : x_{11};$$
$$y_7 = (p == \&y)?3 : y_6;$$
$$z_4 = (p == \&z)?3 : z_3;$$
$$\ldots$$

Every assignment to an array cell is treated similarly, by instantiating it for each possible value of the array index. Statements that include the use of a dereference of a pointer or of an array cell are treated in a similar manner.

Since the program is bounded, the number of malloc calls is bounded as well. CBMC treats each allocated memory as a regular global variable. CBMC also supports pointer arithmetic inside array bounds.

3 Bounded Model Checking for Concurrent C Programs

In this section we describe how a concurrent program can be efficiently translated to a set of constraints.

The main idea is to bound the number of context switches in the run while allowing them to be anywhere in the code. We denote this bound by n. This strategy is reasonable since most bug patterns have only a few context switches [5]. This strategy is also consistent with the main idea of CBMC. We first present our method for programs with two threads. Later, we describe the required changes for more than two threads.

We note that it is possible to limit the places in which a context switch can occur. There is no advantage in allowing a context switch before an invisible statement [3]; allowing context switches only before visible statements decreases the number of possible executions.

Similarly to CBMC, our goal is to translate concurrent C programs into a set of constraints. As in CBMC, these constraints will be conjuncted with those representing the negation of the specification, and checked for satisfiability.

The translation process consists of three stages.

Stage 1 - Preprocessing. A C statement is not always executed as an atomic statement. Consider the code generated by a compiler for a C statement of the form $x_1 = x_2 + x_3$. The generated assembly code is:
$\boxed{r_a \leftarrow x_2;\ r_b \leftarrow x_3;\ r_c \leftarrow r_a + r_b;\ x_1 \leftarrow r_c;}$ (Where each r is a register). A context switch may occur between these instructions. Statements that involve at most one global variable are not affected by this. To allow such context switches in statements that access more than one global variable we need to break statements just as a compiler does. For example, the statement $x_1 = x_2 + x_3$; (in which each x_i is a global variable) is translated to the following code (in which each y_i is a new temporary local variable): $\boxed{y_1 = x_2;\ \ y_2 = x_3;\ \ x_1 = y_1 + y_2;}$ "if" and loop statements, in which the condition accesses more than one global variable, are treated similarly. Note that the order of execution of an expression is not guaranteed under C semantics. Since we assume that this order is consistent for a compiler, we can configure CBMC for compatibility with any given compiler. This preprocessing can also be avoided if we are not interested in examining such interleavings.

Stage 2 - Applying CBMC Separately on Each Thread. In this stage the first phase of CBMC is applied on each thread, and a list of constraints is obtained for each. We refer to this set of constraints as a *template*. In this template each variable has several copies, and each copy appears only once on the left-hand side of a constraint.

We can think of this template as either a list of constraints or as a program in which each constraint is an assignment and each variable is assigned only once. In the rest of this section we use the latter interpretation, and refer to the template as being executed.

As a result of the preprocessing, this template has four types of statements:

1. An assignment of an expression defined over local variables to a local variable, e.g., $w_k = (guard_r ? y_c * 2 : w_{k-1})$
2. An assignment of an expression defined over local variables to a global variable, e.g., $x_k = (guard_r ? y_c * 2 : x_{k-1})$
3. An assignment of a global variable to a local variable, e.g., $y_c = (guard_r ? x_k : y_{c-1})$
4. An assignment to a guard variable. The guard is local and there may be at most one copy of a global variable on the right-hand side, e.g., $guard_r = guard_{r-1} \&\& x_k > y_c$

After CBMC is applied, each variable has several copies, one for each assignment, where x_j refers to the j-th assignment to x.

We will denote by m the number of constraints in this template and enumerate them from 0 to $m-1$. We will use the notation l_{x_j} to refer to the number of the constraint in which x_j is assigned.

Each thread may have its own code and therefore its own template. We translate each thread into a set of constraints. In the following description we will refer to thread t. To avoid name collision, we add the prefix $thread_t$ to each variable.

Stage 3 - Generating Constraints for Concurrency. The main idea of this stage is to associate with each line l in the template a variable $thread_t\_cs(l)$. The value of this variable indicates the number of context switches that occurred before this line was executed.

We induce the following constraints on the values of the $thread_t\_cs(l)$ variables:

- **Monotonicity:** The value of $thread_t\_cs$ must increase monotonically: $\forall_{0 \leq l < m-1}\ thread_t\_cs(l) \leq thread_t\_cs(l+1)$. [2]
- **Interleaving bound:** There is a bound on the number of context switches. If the bound is n, then the maximum value of $thread_t\_cs$ is n. This is described as follows: $thread_t\_cs(m-1) \leq n$
- **Parity:** Each context switch changes the thread that runs. Having only two threads (see extension at the end of Section 3), the values of $thread_t\_cs(l)$ can be restricted to be even for $t = 0$ and odd for $t = 1$. This is described as follows: $\forall_{0 \leq l < m-1}\ (thread_t\_cs(l) mod\ 2) = t$.

Any assignment to the $thread_t\_cs(l)$ variables determines a concurrent execution over the thread templates: first the block of lines for which $thread_0\_cs(l) = 0$ is executed, then those that have $thread_1\_cs(l) = 1$, then those that have $thread_0\_cs(l) = 2$, and so on. Figure 2 illustrates such an execution.

It will be useful to extend the definition of $thread_t\_cs$ in the following manner: $thread_t\_cs(v_j) = thread_t\_cs(l_{v_j})$. This definition maps a copy of a variable to the value of $thread_t\_cs(l_{v_j})$, where l_{v_j} is the line in which v_j was assigned. Thus $thread_t\_cs(v_j)$ is the context switch block number in which v_j gets its value.

Up to this point we added the $thread_t\_cs(l)$ variables that determine where the context switches occurs. We still need to generate constraints for the values of global variables. Because the global variables are shared among the threads, their behavior is not fully covered by the constraints in the templates.

In order to correctly model the global variables in a concurrent program, we define n new variables x_val_i for each global variable x, and $0 \leq i < n$. Variable x_val_i is the value of variable x at the end of the i-th context switch block. We

[2] Our implementation of the tool is more efficient: When two lines are in the same visible block (the assignment in line $l+1$ is invisible), the constraint can be $thread_t\_cs(l) = thread_t\_cs(l+1)$. As a result, these two variables can become one. In this paper we disregard this improvement for better readability.

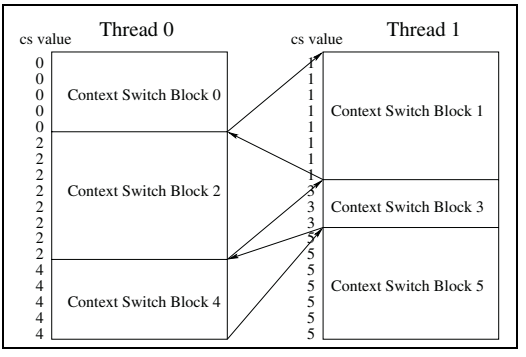


Fig. 2. Context Switch Blocks

can think of x_val_i as the thread interface. This is because in our model, threads can influence each other only through these variables.

Before we define the constraint over x_val_i, we remind the reader that x has several copies in a template, one for each assignment. These assignments are numbered from 0 to $p-1$ (assuming p is the number of assignments to x). The assignment to x_j is in context switch block $thread_t\_cs(x_j)$. Note that the template of thread zero sets x_val_i for even values of i, and the template of thread one sets x_val_i for odd values of i.

Variable x_val_i should get its value according to the last assignment that was made to x in the i-th context switch block. If x was assigned in the i-th context switch block, x_val_i will be equal to x_j, the last assignment to x in this block. Otherwise, if there were no assignments to x in the i-th context switch block, then x_val_i preserves the value it had at the end of the previous block.

$\forall_{i \text{ s.t. } i \bmod 2=t}$

$$x\_val_i = \text{for } 0 \le j < p \quad \text{if } (thread_t\_cs(x_j) == i) \wedge (i < thread_t\_cs(x_{j+1}))) \quad thread_t\_x_j \quad \text{if } (\forall_{0 \le j < p}\ thread_t\_cs(x_j) \neq i) \quad x\_val_{i-1}$$

For simplicity we define: $x\_val_{(-1)} = init\_value(x)$, $thread_t\_cs(x_p) = n+1$.

After introducing the additional variables needed for concurrent programs and their constraints, we are now ready to translate each statement in the template into a constraint. We present the translation for each of the four statement types in the template:

1. For regular statements, which do not access global variables (e.g. $y_j = (guard_r?(f(z_k, w_c), y_{j-1})$, we simply add the thread prefix: $thread_t\_y_j = (thread_t\_guard_r?f(thread_t\_z_k, thread_t\_w_c) : thread_t\_y_{j-1})$
2. For statements of the form $y_j = (guard_r?x_k : y_{j-1})$, where y is a local variable and x is a global variable, there are two options:
   - If the assignment to x_k is in the same context switch block as the assignment to y_j, the thread prefix can simply be added.
   - Otherwise, the x_val of the previous context switch block should be used.

$$
\begin{array}{l}
thread_t\_y_j = \text{if } (thread_t\_guard_r) \\
\qquad\qquad \text{if } (thread_t\_cs(y_j) == thread_t\_cs(x_k)) \\
\qquad\qquad\quad thread_t\_x_k \\
\qquad\qquad \text{else} \\
\qquad\qquad\quad x\_val_{(thread_t\_cs(y_j)-1)} \\
\qquad\quad \text{else} \\
\qquad\qquad thread_t\_y_{(j-1)}
\end{array}
$$

3. For statements that have a global variable in their left-hand side and in the else clause of their right-hand side (e.g., $x_j = (guard_r ? f(y_k, w_c) : x_{j-1})$, special treatment is required for the else clause. This treatment is similar to that given in the previous item.

$$
\begin{array}{l}
thread_t\_x_j = \text{if } (thread_t\_guard_r) \\
\qquad\qquad f(thread_t\_y_k, thread_t\_w_c); \\
\qquad\quad \text{else} \\
\qquad\qquad \text{if } (thread_t\_cs(x_j) == thread_t\_cs(x_{j-1})) \\
\qquad\qquad\quad thread_t\_x_{j-1}; \\
\qquad\qquad \text{else} \\
\qquad\qquad\quad x\_val_{(thread_t\_cs(x_j)-1)};
\end{array}
$$

4. For an assignment to a guard that does not access a global variable, we simply add the correct prefixes (as in the first item). An assignment to a guard that uses a global variable is treated as in the second item.

Pointers and Arrays. No special treatment is required to support assignments to a pointer dereference or to a cell in an array. Such an assignment is already instantiated into several assignments, one for each possible value, when executing CBMC in Stage 2. Note that for concurrent programs there are more potential values for a global pointer (or a global index of an array), since it may get its value in another thread.

However, we do need to handle dereference of a pointer that may point to a global variable (or a use of a global array cell). These are handled in the preprocessing stage. Their handling is similar to that of expressions with more than one global variable: we break the statement in two. In the first statement, the value of the dereference of the pointer is assigned to a new local variable, y. The second statement is a copy of the original statement, in which the value of the dereference is replaced with y. For example, the statement $v_1 = *p + v_2$; (in which each p may point to a global variable, and each v_i is a local variable) is translated to the following code: $\boxed{y = *p; \quad v_1 = y + v_2;}$

More Than Two Threads. There are two options for extending this algorithm to T threads where $T > 2$: The first is to enforce a round robin among the threads (thread 0 runs first, then thread 1, 2, ..., T-1, and then 0 again and so on). Note that a thread might not perform any statement while running, but the number of context switches still increases. The changes to the constraints are quite trivial. In particular, we change the parity constraint and use *mod* T instead of *mod* 2. This will often require a larger bound over the number of context switches.

Another option for extending TCBMC to T threads is to add a new set of variables: run_i for $0 \leq i < n$, where run_i is the ID of the thread that runs in the i-th context switch block. The value of run_i is set by the SAT solver, and determines the order in which the threads run. There are some changes in the constraints, which we explain in the full version of this paper. We suggest two methods for extending TCBMC because neither one is better than the other for all input programs.

Note that, when threads are dynamically generated, T can be increased iteratively until a bug is found or the SAT solver explodes.

4 Modeling Synchronization Primitives

Until this point the model we present enables the threads to communicate with each other only via global variables. Concurrent programs usually use synchronization primitives as well. In this section we will describe how we can efficiently model mutexes and the Pthread condition (i.e., the *wait*/*signal* mechanism). The modeling presented in this section interferes with deadlock detection, and will be revisited in subsection 5.2 where deadlocks are handled. We will present the modeling of the synchronization primitives via transformation to C code. It is possible and sometimes even more efficient to directly create the model without changing the C code first. In fact, our implementation actually constructs the model directly from the original code. However, we find the current presentation more readable and easy to understand.

4.1 Modeling Atomic Sections

The first primitive we model is the atomic section, which is not a real programming primitive but is used to model other primitives. Atomic sections are also useful in the verification process. If TCBMC users do not wish to allow context switch along certain sections, they can mark these sections as atomic. This will yield a shorter formula, which will result in a better performance of the SAT solver.

Modeling an atomic section is very simple. We just add constraints that force the $thread_t\_cs$ values of the lines in an atomic section to be identical. Thus, no context switch is allowed along this section.

4.2 Modeling Mutexes

Mutex is the mechanism for implementing mutual exclusion between threads. A mutex has two states, L (locked) and U (unlocked), and at least two basic operations: lock and unlock. Lock waits until the mutex is in U and then changes its state to L. Unlock is applied to a mutex in state L and changes its state to U. There are two common ways to implement the lock operation: The first is to wait until the mutex is in state U. This is done by means of a busy wait. The second is to move the thread to the operating system's sleep state. The thread will return to a ready state when the mutex returns to state U.

A naive approach may model mutexes by including one of these implementations explicitly. The result is a complicated model. In fact, this is not necessary. Our goal is not to verify that the mutex implementation is correct; we assume it is correct. Rather we aim at verifying the programs that use mutexes. We also manage to avoid the main difficulty in modeling mutexes: the modeling of lock operations when the mutex is in state L.

Before explaining how we model mutexes, we present two definitions and a lemma that help us to explain the idea behind our method.

Two executions of a concurrent program are *mutex-free-equivalent* iff they have the same states when ignoring the internal implementation of mutexes (We consider only the state of the mutex U or L). We use the notation $\pi \approx_w \pi'$ to indicate that π and π' are mutex-free-equivalent.

We define *redundant-attempt* as an attempt to lock a mutex that is already in state L. A *wait-free execution* is an execution that has no redundant-attempts.

Lemma 1. *Let P be a concurrent program. For every non-wait-free execution π of P there is a wait-free execution π' that satisfies $\pi \approx_w \pi'$.*

In our modeling all executions are wait-free. If a thread tries to lock a mutex, it either succeeds (the mutex is in state U) or this execution is eliminated. Furthermore, all errors other than deadlock that appear in a non-wait-free execution appear also in a mutex-free equivalent wait-free execution. Thus it is possible to find all the errors.

We model a mutex by implementing special C functions for the lock and unlock primitives, and translate it using CBMC. The lock function uses the assume function. Figure 3 presents the modeling of lock and unlock. Only 1 bit is used for each mutex. In order to improve performance, we maximize the atomic sections in the modeling of lock. We will continue to maximize the atomic sections in other modelings as well.

locking_trd_id can be added to the mutex modeling to ensure that the thread that performs the unlock operation is the same one that locked it earlier. A bounded counter of the number of locks can also be added to support recursive mutexes.

```
atomic{
  assume(*mutex == U);
  *mutex = L;
}
```

(a) lock(mutex)

```
atomic{
  assert(*mutex == L);
  *mutex = U;
}
```

(b) unlock(mutex)

Fig. 3. Modeling of lock and unlock in C

4.3 Modeling Conditions

A condition has 3 primitives: *wait*, *signal* and *broadcast*. $wait(cond, mutex)$ stops the run of the thread until it is awakened by another thread's call to signal or broadcast. $Signal(cond)$ awakens one of the threads that are waiting for this

condition. There is no guarantee as to which of the waiting threads will be awakened. Broadcast awakens all the threads that are waiting for this condition. If a signal is sent and there is no thread waiting for it, then the signal is lost; there is no accumulation of signals. *wait* also receives mutex as parameter. It needs to unlock it before stopping and lock it again before continuing (after the thread has been awakened).

Here we model each condition *cond* using a vector of flags, one for each thread. To model a wait in thread i we raise the $cond[i]$ flag, allow context switch, and then assume that the $cond[i]$ is down. The idea is similar to the one we used for mutexes; we actually eliminate all the interleavings in which this thread resumes running before it should. To model *signal*, we nondeterministically choose one raised flag and lower it. *Broadcast* is modeled by simply lowering all the flags.

In order to understand why this modeling is similar to that of mutexes, recall that the *wait* operation can be divided into four stages:

1. Raise a flag indicating that this thread is waiting.
2. *Unlock* the mutex.
3. Wait for this flag to reset.
4. *Lock* the mutex again.

We model only wait-free executions: wait operations that do not wait in the third stage.

Figure 4 presents the modeling of wait and signal. broadcast(cond) is simply modeled by lowering all the flags.

```
atomic{
  cond[current thread] = 1;
  unlock(mutex);
}
assume(cond[current thread] == 0);
lock(mutex);
```

(a) $wait(cond, mutex)$

```
atomic{
  i = rand(number of threads);
  assume(cond[i] == 1||cond == 0);
  cond[i] = 0;
}
```

(b) $signal(cond)$

Fig. 4. Modeling of *wait* and *signal* in C

5 Verifying Race Conditions and Deadlocks

When verifying concurrent programs it is important to detect race conditions and deadlocks. This section presents the changes in the model for each of them.

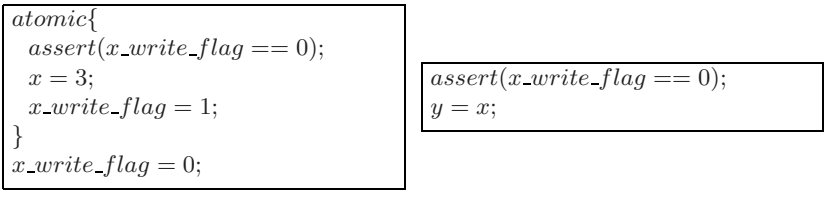
5.1 Detecting Races

A *race condition* is a state in which the next instructions of different threads access the same memory location and at least one of them is a write.

We can identify races by adding to each global variable x a new global bit variable x_write_flag. x_write_flag is raised whenever x is defined (i.e., assigned to) and lowered in the next instruction. There can be a context switch

between these two instructions. In addition, on every access to x we assert that its x_write_flag is low. Figure 5 presents an example of such translation.

Pointers and Arrays. Special treatment is required for pointers and arrays. When there is an assignment to a dereference of a pointer p that points to x, we should change x_write_flag. We have to consider all possible variables which p might point to (see example in subsection 2.1). For each one, we change the corresponding flag while changing the variable itself. Arrays are handled similarly.



(a) Translation of $x = 3$ (b) Translation of $y = x$

Fig. 5. Detecting races

5.2 Finding Deadlocks

Deadlock detection is one of the most interesting issues in concurrent programs. We divide deadlocks into two kinds: a global deadlock is a deadlock in which all the threads are waiting for a mutex or a condition, and a local deadlock is a deadlock in which some of the threads form a waiting cycle (e.g., thread 1 is waiting for mutex m_a which is held by thread 2 which is waiting for mutex m_b which is held by thread 1). In this section we present the extension to TCBMC that allows for detection of global deadlocks. In the full version of this paper we present the extension of TCBMC that allows for detection of local deadlocks [3].

When modeling the code to detect deadlocks, we ignore the existence of other errors. As mentioned before, we encourage TCBMC users to perform three runs: for detecting "regular" errors, for detecting races, and for detecting deadlocks.

In the model we presented in Section 4 we eliminated non-wait-free executions. But this could result in missing global deadlocks because these occur when all threads are in a waiting state. Therefore, we must change the modeling of $lock(m)$, and $wait(cond, mutex)$: We add a new global counter called $trds\_in\_wait$. This counter counts the number of threads in a wait state. When modeling $lock(m)$, if mutex m is already in state L, we increase $trds\_in\_wait$, allow context switch, and then assert that $trds\_in\_wait < T$. If the assertion fails, a global deadlock was detected; otherwise we eliminate this execution, as in the original mutex modeling. When modeling $wait(cond, mutex)$, we increase

[3] We are able to find local deadlocks that involve only mutexes. It is not always possible to find deadlocks that involve conditions when bounding the program.

trds_in_wait after raising the *cond*[*current thread*] flag, allow context switch, and then assert that *trds_in_wait* $< T$. If the assertion fails, a deadlock was detected; otherwise we decrease *trds_in_wait* and continue as in the original modeling of $wait(cond, mutex)$ by assuming that the flag is down.

We also add a global Boolean flag named *dd*(deadlock detected), This flag is raised when a deadlock is found. Once a deadlock has been detected, we know that a bug has been found, and so we can ignore later uses of *lock* and *wait*. More details can be found in the full version.

Figure 6 present the modeling of $lock(m)$ and $wait(cond, mutex)$.

```
if (!dd){
  atomic{
    unlocked = (*m == U);
    if (unlocked) *m = L
    else          trds_in_wait++;
  }
  atomic{
    if (!unlocked){
      dd = (trds_in_wait == T);
      assert(!dd);
      assume(dd);
    }
  }
}
```

(a) $lock(m)$

```
if  (!dd){
    atomic{
      cond[current thread] = 1;
      unlock(mutex);
      trds_in_wait++;
    }
    atomic{
      dd = (trds_in_wait == T);
      assert(!dd);
      assume(dd ∨ cond[current thread] == 0);
      trds_in_wait--;
    }
    lock(mutex);
}
```

(b) $wait(cond, mutex)$

Fig. 6. Modeling of lock and unlock in C when looking for deadlocks

6 Experimental Results

We implemented an initial version of TCBMC that works with two threads and supports mutexes and conditions. Future extensions will support more than two threads, as well as detect deadlocks using the described algorithms. We performed preliminary experiments that checked TCBMC on a naive concurrent implementation of bubble sort. We executed TCBMC over several array sizes, for different values of n (i.e., the number of allowed context switches), and for different integer widths. We used a sufficient loop unwinding bound. The bug in this implementation was dependent on both data and the interleaving.

We also compared it with Microsoft's Zing [1], a state-of-the-art explicit model checker for software that uses various reductions including partial order resuctions. Note that Zing and TCBMC were executed on different platforms [4]. The results are summarized in Table 1.

[4] Zing was executed on the Windows operating system on a Pentium4 1.8Ghz with 1GB memory. TCBMC was executed on the Linux operating system on a Pentium4 2Ghz with 250MB memory.

Table 1. Run time comparison of Zing and TCBMC

array size	Zing		TCBMC					
	8 bit	12 bit	8 bit		16 bit		32 bit	
			n=6	n=10	n=6	n=10	n=6	n=10
3	330.0s	> 1h	0.4s	0.2s	3.6s	4.0s	20.3s	48.3s
4	831.0s	> 1h	11.5s	1.3s	14.6s	58.7s	135.2s	323.0s
5	1496.0s	> 1h	71.0s	94.1s	125.7s	3013.0s	1124.0s	> 1h

From this preliminary experiment we can deduce the following:

- It seems that TCBMC scales better with respect to integer widths. Zing ran for more than an hour for 12 bits, while TCBMC managed to get results even for 32 bits.
- Although tested on different platforms, it seems that TCBMC performs better than Zing for detecting bugs dependent on both data and interleavings.
- Increasing the number of allowed context switches (n) sometimes improves the performance (e.g., array size of 4, and 8 bits). This unexpected behavior can be explained by the fact that for larger n, TCBMC generates a larger formula, but with more satisfying assignments.

7 Conclusions and Future Work

This paper presented an extension of CBMC for concurrent C programs. It explained how to model synchronization primitives and how to detect races and deadlocks. We should complete our implementation to support all of the above.

We also consider changing the template translation into constraints: rather than defining for each line a variable indicating in which context switch block it is executed, we can define, for each context switch block, a variable indicating in which line it begins. This will result in a completely different formula which may be handled better by the SAT solver.

Acknowledgments. We thank Sharon Barner, Ziv Glazberg and Daniel Kroening for many helpful discussions.

References

1. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *CONCUR*, 2004.
2. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. 5^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 1579. Springer-Verlag, 1999.
3. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

4. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS 2004*, pages 168–176. Springer, 2004.
5. E. Farchi, Y. Nir, and S. Ur. Concurrent Bug Patterns and How to Test them. Workshop on Parallel and Distributed Systems: Testing and Debugging, 2003.
6. F. Ivancic, Z. Yang, A. Gupta, M. K. Ganai, and P. Ashar. Efficient SAT-based bounded model checking for software verification, ISoLA, 2004.
7. D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC*, 2003.
8. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.