Verifying Multi-threaded C Programs with SPIN

Anna Zaks and Rajeev Joshi*

[1] New York University
[2] Lab for Reliable Software, Jet Propulsion Laboratory

Abstract. A key challenge in model checking software is the difficulty of verifying properties of implementation code, as opposed to checking an abstract algorithmic description. We describe a tool for verifying multi-threaded C programs that uses the SPIN model checker. Our tool works by compiling a multi-threaded C program into a typed bytecode format, and then using a virtual machine that interprets the bytecode and computes new program states under the direction of SPIN. Our virtual machine is compatible with most of SPIN's search options and optimization flags, such as bitstate hashing and multi-core checking. It provides support for dynamic memory allocation (the `malloc` and `free` family of functions), and for the `pthread` library, which provides primitives often used by multi-threaded C programs. A feature of our approach is that it can check code *after* compiler optimizations, which can sometimes introduce race conditions. We describe how our tool addresses the state space explosion problem by allowing users to define data abstraction functions and to constrain the number of allowed context switches. We also describe a reduction method that reduces context switches using dynamic knowledge computed on-the-fly, while being sound for both safety and liveness properties. Finally, we present initial experimental results with our tool on some small examples.

1 Introduction

A key challenge in applying model checking to software is the difficulty of verifying properties of implementation code, as opposed to checking abstract algorithmic descriptions. Even well understood protocols such as Peterson's protocol for mutual exclusion, whose algorithmic description takes only half a page, have published implementations that are erroneous. This problem is especially acute in our domain of interest – small, real-time embedded systems in use on robotic spacecraft – where limits on memory and processor speeds require implementation ingenuity, but where the risks of failure are high.

* The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Funding was also provided by the NASA ESAS 6G project on Reliable Software Engineering.

Our work extends previous work on *model-driven verification*, in which model checking was applied to the verification of *sequential* C programs [HJ04],[GJ08]. Model-driven verification is a form of software model checking for C programs that works by executing C code embedded in a PROMELA model. The SPIN model checker [Hol03] translates a PROMELA model (along with an LTL property to be checked) into a C program `pan.c` that encodes a model checker that checks the property in question (in a sense, therefore, SPIN is really a *model checker generator*). Because SPIN compiles models into C code, recent versions (since SPIN 4.0) allow fragments of C programs to be embedded within PROMELA models. Each such fragment is treated as a deterministic atomic transition (in SPIN parlance, the equivalent of a "dstep"). This allows SPIN to be used to check C programs against LTL specifications using most[3] of SPIN's search options, including bitstate hashing, and multi-core execution[HB07].

A significant limitation of model-driven verification is that each fragment of embedded C code is executed as an atomic transition by SPIN. This in turn means that it is hard to (a) check properties (such as assertions and invariants) at control points *within* the embedded C code, (b) interrupt the control flow within a C function (to simulate, say, a device interrupt or an asynchronous reset), and (c) explore interleavings of more than one fragment of C code, which is needed in order to check multi-threaded C programs. A discussion of how to address the first two limitations appears elsewhere [GJ08]; in this paper, we address the third limitation of checking multi-threaded C programs. We describe a tool (named "pancam") which implements a virtual machine for executing programs in the LLVM bytecode language [LA04]. Since the state space of even a small C program is typically much larger than that of most PROMELA models, we consider various approaches to combat the space explosion problem. In particular, we describe a technique called *superstep reduction* that can increase the granularity of atomic steps on-the-fly during a model checking run. We also discuss how context-bounding [MQ07] can easily be integrated with our tool, with only a small modification to our virtual machine, and how pancam allows users to define data abstractions to reduce state space still further. Finally, we present initial experimental results with using our tool on some small multi-threaded C programs.

2 Model Checking C programs with pancam

Our approach to checking a concurrent C program with SPIN is to first translate the program into bytecode for the Low Level Virtual Machine (LLVM) compiler infrastructure [LA04]. This bytecode is then checked by executing it within the context of an explicit-state model checker by using a virtual machine interpreter. In a sense, this approach is similar to Java Pathfinder (JPF) [VHB+03]. However,

[3] Two key options not supported for embedded code by SPIN are breadth-first search, which would require too much additional overhead, and partial-order reduction, which is difficult for C programs because computing a nontrivial independency relation is hard.

```c
#include <pthread.h>
struct pa_desc {
  volatile int *f0, *f1 ;
  int last ;
} ;
...
volatile int pa_f0, pa_f1, pa_last ;
...
void pa_desc_lock(struct
pa_desc *d) {
  for (*d->f0=1, pa_last=d->last;
       *d->f1==1 && pa_last==d->last;
      ) ;
}
...
int count = 0 ;
void threadx_critical(void) {
  count++ ;
  ... // critical section
  count-- ;
}
```

```c
void * thread1_main(void *args) {
  struct pa_desc d ;
  pa_desc_init(&d, 1) ;
  for (;;) {
    pa_desc_lock(&d) ;
    threadx_critical() ;
    pa_desc_unlock(&d) ;
  }
  return NULL ; /* NOT REACHED */
}

/* pancam helpers */

void init(void) {
  pa_f0 = pa_f1 = pa_last = 0 ;
}
Bool check_exclusion(void) {
  return (count <= 1) ;
}
```

Fig. 1. Excerpt of C implementation of Peterson's Algorithm using pthreads, from the Wikipedia. The two highlighted occurrences of `volatile` were missing, causing a potential race condition.

unlike JPF, we do not integrate the model checker with the bytecode interpreter. Instead, our virtual machine executes bytecode as directed by SPIN, by providing a method `pan_step(i)` that is called by SPIN to execute the next transition of thread `i`. In a sense, therefore, SPIN *orchestrates* the search by deciding which thread to execute next, by storing visited states in its hash table, and by restoring a previous state during a backtracking step. This division of labor allows us to freely benefit from SPIN's unique abilities, notably its scalability, search heuristics, and, lately, the capability to deploy it on multi-core CPUs [HB07]. The C language does not have any built-in primitives for concurrency, so our framework provides support for the constructs from the standard `pthreads` library such as mutexes and condition variables. Even though the dynamic thread creation is not yet fully implemented in pancam, the extension can be organically incorporated into the framework. The only limitation would be on the total number of threads, which should not exceed 255 (the bound imposed by SPIN).

To illustrate how our tool works, Fig. 1 shows the C program that appears in the Wikipedia entry[4] for Peterson's mutual exclusion protocol [wik]. The

[4] To simplify the statement of the mutual exclusion property, we have added an additional variable `count` as shown.

property to be checked is mutual exclusion, which is defined by the boolean valued function `check_exclusion`.

Our tool first compiles this program into LLVM bytecode, using the `llvm-gcc` compiler (an extension of the GNU `gcc` compiler that generates LLVM bytecode). LLVM bytecode is like typed assembly language; a sample appears in Fig. 2, which shows the bytecode corresponding to the `pa_desc_lock` function shown in Fig. 1.

```
define void @pa_desc_lock(%struct.pa_desc* %d) {
entry:
        %tmp1 = getelementptr %struct.pa_desc* %d, i32 0, i32 0
        %tmp2 = load i32** %tmp1
        volatile store i32 1, i32* %tmp2
        %tmp4 = getelementptr %struct.pa_desc* %d, i32 0, i32 2
        %tmp5 = volatile load i32* %tmp4
        volatile store i32 %tmp5, i32* @pa_last
        %tmp8 = getelementptr %struct.pa_desc* %d, i32 0, i32 1
        %tmp9 = load i32** %tmp8
        br label %bb6
bb6:
        %tmp10 = volatile load i32* %tmp9
        %tmp11 = icmp eq i32 %tmp10, 1
        br i1 %tmp11, label %cond_next, label %return
cond_next:
        %tmp15 = volatile load i32* %tmp4
        %tmp16 = volatile load i32* @pa_last
        %tmp17 = icmp eq i32 %tmp15, %tmp16
        br i1 %tmp17, label %bb6, label %return
return:
        ret void
}
```

Fig. 2. LLVM bytecode for function `pa_desc_lock`

To check the C code for Peterson's algorithm with our tool, we use a PROMELA model to make appropriate calls to schedule the threads via our virtual machine. Fig. 3 shows a SPIN model for checking the program in Fig. 1. The `c_decl` primitive is used to declare external C types and data objects that are used in the embedded C code. For simplicity, we assume the declarations needed by our model are in the header file `pancam_peterson.h`. Next, the `c_track` declarations are *tracking* statements, which are discussed below. The PROMELA process `init` defines the initialization steps for the SPIN model: as shown, they consist of initializing the interpreter (by calling `pan_setup()`), registering an invariant (defined by the C function `check_exclusion`) with the interpreter,

```
c_decl {
\#include "pancam_peterson.h"
}
c_track "csbuf" "CS_SIZE" "Matched";
init() {
  c_code {
    pan_setup() ;
    pan_invariant("check_exclusion") ;
    pan_run_function("init") ;
    pan_start_thread(0,
        "thread0_main", NULL) ;
    pan_start_thread(1,
        "thread1_main", NULL) ;
  } ;
  run thread0() ;
  run thread1()
}
```

```
proctype thread0() {
  do
  :: c_expr{pan_enabled(0)}
     -> c_code{pan_step(0);}
  od
}
proctype thread1() {
  do
  :: c_expr{pan_enabled(1)}
     -> c_code{pan_step(1);}
  od
}
```

Fig. 3. Spin driver for executing pancam on Peterson's Algorithm

performing one-time initialization of the C program (`pan_run_function()`), creating and starting the threads, and, finally, starting one PROMELA process for each thread. As shown, each PROMELA process then consists of repeatedly executing a single step of the associated thread (by calling `pan_step()`) provided that the thread is enabled.

The `c_track` declarations provide the essential ingredient that allows us to use the SPIN model checking engine in conjunction with our interpreter. During its depth first search[5], whenever SPIN reaches a state with no new successors, it backtracks to the most recent state that has not been fully explored. For PROMELA variables, restoration of earlier values when backtracking is automatic, since they are stored in the state vector maintained by SPIN. However, the state of the pancam virtual machine is not part of the PROMELA model. Thus the model checker needs explicit knowledge of the region of memory where this state is stored, so that it can copy and restore this memory during its backtracking search. This knowledge is provided through the `c_track` declarations. In our framework, the bytecode interpreter maintains its state in a single contiguous region of memory starting at address `csbuf` and occupying `CS_SIZE` bytes; this corresponds to the `c_track` declaration shown in the figure.

In using our tool to verify the Wikipedia C implementation of Peterson's protocol, we discovered a bug in the implementation. The bug is interesting because it manifests itself when the code is compiled with optimization enabled. The problem arose from the fact that certain global variables were not originally marked as `volatile` (as indicated by the shaded keywords in Fig. 1). As a

[5] SPIN currently supports execution of embedded C code only when using depth first search mode.

result, the optimized bytecode reused stale values read earlier. For example, in the procedure `pa_desc_lock` from Fig. 2, all the instructions that occur after the second store were removed, leading to scenarios where mutual exclusion was violated. We have since fixed the Wikipedia entry.

3 Addressing State Space Explosion

Not surprisingly, the biggest challenge in using a tool such as ours is the problem of state space explosion. Even though our main interest is in checking small embedded C programs, the typical state vectors we encounter are much larger (of the order of hundreds or even thousands of bytes) as compared to typical PROMELA models (whose state vectors are smaller by one or two orders of magnitude). In addition, because a single line of C may translate into many steps of bytecode, a naive exploration of all interleavings of a set of threads would quickly make even the smallest of problems intractable. To address these issues, pancam uses three techniques: (a) it allows users to provide data abstraction functions, (b) it provides the ability for the user to enforce context-switch bounding (see below), and (c) it employs an algorithm that performs a kind of partial order reduction on-the-fly to reduce the number of context switches without losing soundness of checking. We describe the first two of these techniques in the rest of this section; our reduction method is described in Section 4.

3.1 Abstraction

The ability of our tool to support abstractions is derived from the distinction between *tracked* and *matched* objects in SPIN. As discussed in Section 2, a tracked data object is stored on the stack used by SPIN's depth first search (DFS), so that an earlier state of that object can be restored on each backtracking step during the DFS. In almost all cases[6], any data that changes during an execution should be tracked. A matched object, on the other hand, is one that is part of the *state descriptor* that SPIN uses to determine if a state has been seen before. By declaring an object to be tracked but not matched, we can therefore exclude it from the state descriptor. Support for this is provided by the `"Matched"` and `"UnMatched"` keywords in SPIN. (These keywords were introduced in SPIN version 4.1.)

The ability to separate tracked and matched data allows us to use data abstraction to reduce the size of the state space [HJ04]. A simple but effective scheme is to define a new auxiliary variable `abs` for storing the abstract state, and provide a function `update_abs()` which updates the value of `abs` based on the (current) values of the concrete program variables. Then, to make SPIN search the abstract state space, we declare all concrete program variables as tracked but `"UnMatched"`, and declare the abstraction variable `abs` as tracked

[6] There are valid reasons for not tracking certain data even though it changes during an execution [GJ08]; see Section 4.2.

and "Matched", and we ensure that the function update_abs() is called after every transition that changes concrete state.

Our tool supports this scheme for data abstraction by providing a buffer abs. The user provides the function update_abs, which computes the data abstraction and writes it to the buffer. Our tool ensures that this function is invoked if any of the variables that appear in the body of this function changes during a transition.

3.2 Context-Bounded Checking

The idea in context-bounded model checking [QR05,MQ07,MQ08] is to avoid state space explosion in multi-threaded programs by enforcing an upper bound on the number of allowed preemptive context switches. A context switch from process p to process q is called preemptive if process p is enabled (and could therefore continue execution if the context switch did not occur). Experience with context-bounded model checking suggests that, in most cases, errors in multi-threaded software typically have shortest counterexample traces that require only a small number of context switches [MQ07]. Thus exhaustive exploration of runs with a small budget of allowed context switches has a good chance of finding errors.

To extend our tool with support for context-bounded search, we change the top-level SPIN model that orchestrates the run by replacing calls to pan_enabled(p) (which check if thread p is enabled) by calls to the function pan_enabled_cb(p) (which additionally checks the condition for context-bounding). Fig. 4 shows the C code for the function pan_enabled_cb. As shown, we add two additional integers last_proc and nswitch to the state space (but note that these variables are only tracked, and not matched). It is not hard to show that by using it to replace the original pan_enabled function, (and by appropriately updating last_proc and nswitch whenever a thread is executed) we achieve the desired effect of limiting the number of preemptive context switches to the user-provided bound of MAX_SWITCH.

4 On-the-fly Superstep Reduction

As described in Section 2, our tool uses a SPIN model to orchestrate the state space search by choosing, at each step, a thread to execute, and executing its next transition by invoking the virtual machine. An exhaustive search along these lines would require exploring all possible interleavings of the threads in the program, which is intractable for all but the smallest of programs. A common technique used to deal with the problem is partial order reduction [Pel93,CGP00]. Intuitively, partial order reduction works by reducing the number of context switches, exploiting the fact that transitions in different threads are often independent (in the sense that the order in which they occur does not affect visible program behavior).

Most partial order methods described in the literature are *static* in the sense that they determine independence of transitions by analyzing program text.

```
c_decl {
  int last_proc = -1 ;
  int nswitch = 0 ;
  int MAX_SWITCH = -1 ;

  Bool pan_enabled_cb(int p) {
    int i ;

    if (!pan_enabled(p)) /* thread p is disabled */
      return FALSE ;

    if (last_proc == p) /* no context switch */
      return TRUE ;

    /* Check if bound not specified, or not reached */
    if ((MAX_SWITCH < 0) || (nswitch < MAX_SWITCH))
      return TRUE ;

    /* Check if any other thread is enabled */
    for (i=0 ; i<ThreadCount ; i++)
      if ((i != p) && pan_enabled(i))
        return FALSE ;
    /* all other threads are disabled, so don't preempt */
    return TRUE ;
}

c_track "&nswitch"    "sizeof(int)"  "UnMatched";
c_track "&last_proc"  "sizeof(int)"  "UnMatched" ;
```

Fig. 4. Code for implementing context bounding with pancam

Such analyses are, however, not terribly effective with C programs, and typically allow only very simple and conservative independence relations to be computed. For C programs, therefore, it is more instructive to look at *dynamic* partial order reduction methods[FG05],[GFYS07], in which independence relationships are computed dynamically, during a model checking run. For example, one of the simplest approaches to dynamic partial order reduction is to only allow a context switch after an update or an access to a global memory location.

In the context of our tool, however, there is one additional complication caused by the fact that the model checking engine (SPIN) treats the model as having a single transition (denoted by the function `pan_step`). In particular, this means that support for partial order reduction therefore requires either exposing additional pancam state (which would require modification of SPIN, which we hope to avoid), or for the reduction to be implemented entirely within pancam. We adopt the latter strategy. Pancam performs partial order reduction on the state space by allowing a thread i to execute a sequence of more than one instruction as part of a single SPIN transition from a state s. We refer to such a sequence of instructions as a "superstep" and denote it by the notation Λ_i^s. Since the model checker only sees the first and last states of a superstep, the intermediate states are hidden from the model checker, which in turn reduces

the number of interleavings to be explored (and therefore the number of states and transitions).

Of course, as with traditional partial order reduction, there are certain conditions that must be satisfied by such supersteps in order to preserve soundness of model checking. In the next subsection, we describe a set of conditions under which we can preserve the soundness of next-time free LTL properties.

4.1 Correctness of Superstep Reduction

For convenience, we consider programs with k deterministic threads (or processes), where the only source of nondeterminism comes from thread scheduling. We also assume that each instruction can access at most one global memory location. This assumption is safe to make about the LLVM bytecode, which uses designated instructions *store* and *load* to access memory.

We say that two transitions α and β are *independent* if neither enables nor disables the other, and for any state, execution of α followed by execution of β results in the same state as execution of β followed by α. We say that two transitions *conflict* if both access a common memory location and at least one of them is a write. Under the assumption that one thread may enable or disable another only by means of mutexes, which are a type of a shared object, the absence of a conflict between transitions implies independence as long as the transitions do not belong to the same thread.

Claim 1 *The soundness and completeness of next-time free LTL model checking is preserved as long as for every thread i enabled in state s, superstep sequence Λ_i^s satisfies the following three requirements:*

1. **Superstep Size** *Λ_i^s must be finite and contain at least one transition. The check for finiteness can be implemented conservatively by setting an upper bound on the number of transitions in a superstep sequence or the number of loop heads within the sequence.*
2. **Independence** *Only the very last transition of the path Λ_i^s conflicts with any of the transitions in Λ_k^s for any thread $k \neq i$.*
3. **Visibility** *At most one transition which changes the value of any of the atomic propositions is allowed in Λ_i^s. If exists, it must be the very last transition of the superstep sequence.*

The formal proof of the claim is similar to the one presented in [CGP00] and is beyond the scope of this paper. Here we only present some intuition about the correctness of the superstep reduction.

All the paths outgoing from a state s can be partitioned into sets, where each set is covered by one of the superstep sequences as following. If, on the path θ^s, all the transitions of the superstep sequence Λ_i^s precede the last transitions of the superstep sequences that correspond to all the other threads, θ^s is said to be covered by Λ_i^s. Consider the example in Fig. 5(a) that depicts the superstep sequences of the three program threads (Th_1, Th_2, and Th_3) from the program

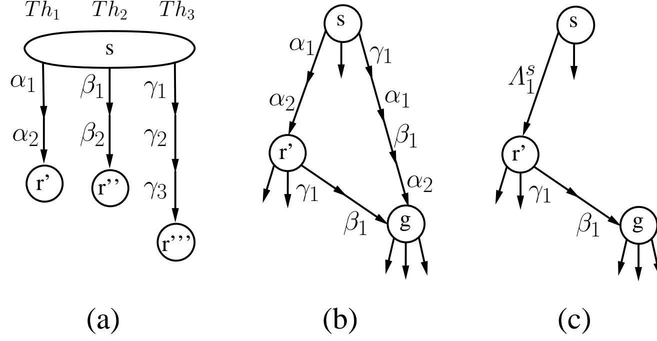


Fig. 5. Superstep POR

state s. In Fig. 5 (b), the transitions $\gamma_1, \alpha_1, \beta_1, \alpha_2$ form the prefix of a number of the program paths outgoing from state s. All these paths correspond to the program runs in which, from the state s, the threads are scheduled in the following order. First, one transition of Th_3 is scheduled, followed by a transition from Th_1, a transition from Th_2, and another transition from Th_1. We say that all these paths are covered by the superstep sequence of Th_1 since α_2 occurs before β_2 and γ_3 on each of the paths. At each state s, the superstep reduction prunes away all the program paths which do not have a superstep sequence as a prefix and substitutes the superstep sequence with just one summary transition as shown in Fig. 5(c).

Let Θ^s be the minimal prefix of θ^s such that it contains all the transitions of Λ_i^s. Then, all the states reachable after following path Θ^s can also be reached after following Λ_i^s due to the fact that all the transitions of Θ^s which are not in Λ_i^s do not conflict with the transitions in Λ_i^s and can be commuted out. Going back to our example, since transitions β_1 and γ_1 do not conflict with α_2, the state g, reachable by following transitions $\gamma_1, \alpha_1, \beta_1, \alpha_2$, can also be reached by following the superstep sequence α_1, α_2.

It only remains to show that the intermediate states of the paths Θ^s and Λ_i^s do not have to be exposed to a model checking algorithm. The paths are finite; and, by definition of Θ^s, its last transition is equal to the last transition of Λ_i^s. Following the visibility requirement, only the very last transition of Λ_i^s and, consequently, only the very last transition of Θ^s may change the values of the predicates participating in the LTL property being checked. Thus, all the states on the paths except for the very last ones are indistinguishable from the state s. Moreover, since the transitions of Θ^s that do not occur in Λ_i^s do not modify the values of the predicates, the new values of the predicates in the last state of Θ^s are the same as in the last state of Λ_i^s. Thus, if a transition changes one of the predicates, it will always be visible to the model checker. In the example on Fig. 5, only the transition α_2 can be visible. All the states can be partitioned into two groups depending on the values of the predicates: the

states undistinguishable from the state s and the states undistinguishable from the state g.

Notice that the listed requirements are general enough to allow for different choices of superstep sequences. However, as long as they are satisfied, the soundness and completeness of LTL model checking is preserved.

4.2 Implementation of Superstep Reduction in pancam

Next, we describe how superstep reduction is implemented as part of our tool, which piggybacks the nested depth-first search algorithm used by SPIN. One of the attractions of using SPIN's nested depth-first search is that, unlike the case with breadth-first search [GFYS07], our implementation is fully compatible with checking of liveness properties. (And, although, we do not describe it here, our method can be straightforwardly extended to cooperate with breadth-first search, if desired.)

During its state exploration, SPIN issues calls to `pan_step(i)`, which, given the current state s, computes the state s' obtained by executing one or more instructions of thread i. The executed instructions form the superstep sequence Λ_i^s. The superstep size requirement guarantees that at least one instruction would be executed; consequently, unless there is a loop in the state space, $s \neq s'$. Due to the nature of depth-first search, `pan_step` will be called multiple times on the same state s. In particular, after exploring the state space in which thread i is executed from state s, SPIN backtracks and attempts to execute the thread $i+1$ from the same state s in response to which pancam computes Λ_{i+1}^s.

The pseudocode of `pan_step` is presented in Fig. 6. If the state s is visited by the depth-first search for the very first time, `pan_step` executes initialization routines. Further, each time SPIN calls `pan_step(i)`, we compute the superstep sequence for thread i by interpreting the enabled instructions of thread i one by one. On each iteration, we check that addition of the corresponding instruction to the sequence does not violate any of the requirements stated above (in practice, the checks are only required for the instructions that access a global program location).

The most non-trivial check is the verification of the independence condition for which one could use various static and dynamic methods. Fig. 7 presents the dynamic independence check employed by pancam. Due to the nature of the independence requirement, the superstep of one thread depends on the transitions that constitute the supersteps of the other threads. An eager approach to this problem is to compute the supersteps for every thread the very first time the state s is visited (with the request to take step on thread one) and use the precomputed supersteps on all the subsequent visits to the same state (when SPIN backtracks to take step on the other threads). However, this solution leads to inefficiencies since computing the supersteps effectively entails computation of the successors of the state s. Storing the successor states along with the current state leads to a large space overhead. Recomputing the successor states, on the other hand, would impair the running time.

```
ConflictType = { CONTINUE, POST_STOP, PRE_STOP }

pan_step(ThreadID i) {
  superstep_length = 0;

  if (not backtracking) {
    init_independence_tester();
  }

  while (true)  {
    tr_i = get_next_instruction(i);
    ConflictType error = test_for_independence(i, tr_i);
    if (error == PRE_STOP) break;
    execute_instruction(tr_i);
    superstep_length++;
    if (superstep_length ≥ MAX_SUPERSTEP_LENGTH) break;
    if (error == POST_STOP) break;
    if (is_proposition_modifying(tr_i)) break;
} }
```

Fig. 6. Pseudocode of `pan_step` with superstep reduction.

The solution we present computes the supersteps lazily - whenever `pan_step`(`i`) is called, it only computes the superstep for thread i. To convey the information about the supersteps which have already been computed, we store additional information along with the program state on the depth-first search stack. For each state s and each thread i, we store $\mathtt{AccessTable}_\mathtt{i}^\mathtt{s}$ - the list of location and access type pairs. Each instruction of Λ_i^s that accesses a global is represented by a pair $(\mathtt{l}, \mathtt{ty})$; it records the global location `l` that is accessed and the flag `ty` stating whether the transition is a read or a write. `AccessTable` is not stored as the part of the state tracked by SPIN but maintained externally within pancam VM since the data it stores is updated each time the state is visited.

The very first time state s is visited, `init_independence_tester`() initializes the $\mathtt{AccessTable}_\mathtt{i}^\mathtt{s}$ of each enabled thread i with the access information derived from the very first instruction to be executed on the thread i. Further, before adding an instruction $\mathtt{tr_i}$ to the superstep sequence of thread i, `pan_step` consults with `test_for_independence`$(\mathtt{i}, \mathtt{tr_i})$ to ensure that the independence condition is met. `test_for_independence` may return three different values. `CONTINUE` means that the instruction can be added to the superstep Λ_i^s since it does not conflict with any instructions in Λ_k^s for all threads $k \neq i$. `POST_STOP` means that $\mathtt{tr_i}$ introduces a conflict with some other thread, but adding it to Λ_i^s does not violate the independence requirement as long as it is the very last transition of Λ_i^s. Finally, `PRE_STOP` means that adding $\mathtt{tr_i}$ to Λ_i^s leads to a violation since the transition with which it conflicts is not the very last transition of thread k for some $k \neq i$; thus, $\mathtt{tr_i}$ must not be executed. Due to the initialization of $\mathtt{AccessTable}_\mathtt{i}^\mathtt{s}$, it is not possible to have a `PRE_STOP`

```
init_independence_tester() {
  for (every enabled thread k) {
    AccessTable^s_k.add( get_access_pair( get_next_instruction(k) ) );
  }
}
test_for_independence(ThreadID i, Instruction tr_i) {
  ai = get_access_pair(tr_i);
  for ( all threads k : k ≠ i ) {
    for (all ak ∈ AccessTable^s_k) {
      if (conflict(ai, ak)) {
        if (ak ≠ last_of(AccessTable^s_k)) {
          return PRE_STOP;
        } else {
          if (tr_i ≠  first_of(Λ^s_i)) AccessTable^s_i.add(ai);
          return POST_STOP;
  } } } }
  if (tr_i ≠  first_of(Λ^s_i)) AccessTable^s_i.add(ai);
  return CONTINUE;
}
```

Fig. 7. Pseudocode of the independence condition tester.

on the very first transition of any of the threads; thus, the superstep size requirement is met - `pan_step` always executes at least one transition. Finally, `test_for_independence`($\mathtt{i}, \mathtt{tr_i}$) updates the $\mathtt{AccessTable}^{\mathtt{i}}_{\mathtt{s}}$ with the access pair derived from $\mathtt{tr_i}$ if the instruction is to be added to Λ^s_i and if it is not the very first instruction of Λ^s_i. Recall that the `AccessTable` is updated with the access pairs corresponding to the very first instructions of each thread as part of the initialization routine.

The above technique requires no space overhead when used as part of breadth-first search state exploration. However, when used with depth-first search, the `AccessTable` must be stored on the search stack. In cases when the sets are quite large, one could use approximations. For instance, one idea is to use a *coloring* abstraction, in which the memory is partitioned into regions with distinct colors, and each transition is associated with the set of colors it reads and writes.

Example 1. Let us demonstrate the algorithm on an artificial example from Fig. 8 that depicts the instructions that the three threads can execute from the state s. We assume that the variables k, x, y, and m are global variables; t, l, x_ptr, k_ptr are local; x_ptr and k_ptr are the pointers to x and k, respectively.

When the state s is visited for the very first time, `init_independence_tester` initializes the `AccessTable` with the information derived from the very first instructions of each thread as following:

$\mathtt{AccessTable}^{\mathtt{s}}_1 = (\ (k_ptr, read)\ )$
$\mathtt{AccessTable}^{\mathtt{s}}_2 = (\ (ad(y), write)\ )$
$\mathtt{AccessTable}^{\mathtt{s}}_3 = (\ (ad(m), write)\ )$

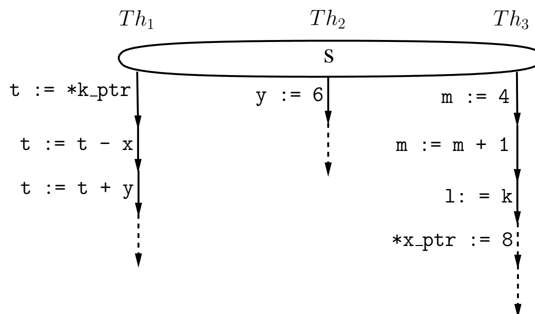


Fig. 8. The example demonstrating the application of the Superstep POR algorithm. The solid arrows represent the instructions that form the supersteps from the state s.

Here $ad(x)$ stands for the address in memory where the variable x is stored (`AccessTable` stores the actual addresses of the accessed variables). After the initialization, `pan_step` issues calls to `test_for_independence` passing the instructions of Th_1 one by one. The function returns `CONTINUE` when passed `t := *k_ptr` and `t := t - x`. However, since `t := t + y` conflicts with the very first instruction of Th_2, `POST_STOP` is returned as the result of the third call. The table is updated accordingly:

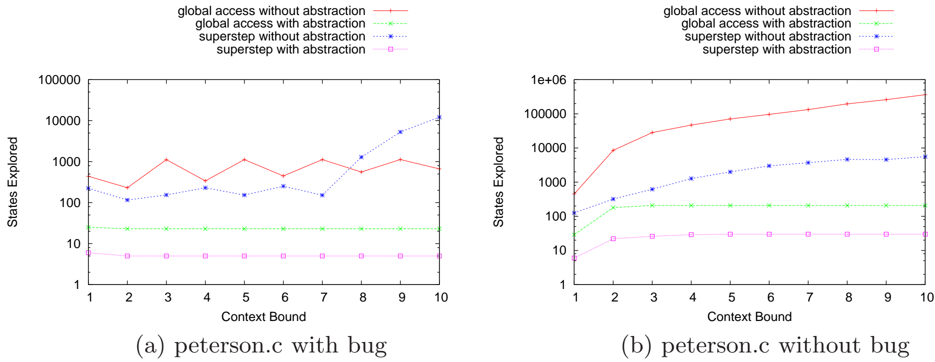
$\texttt{AccessTable}_1^s = (\ (k_ptr, read);\ (ad(x), read);\ (ad(y), read)\ )$

When the depth-first search backtracks to schedule Th_2, `pan_step` calls `test_for_independence` with `y := 6` as the argument. Due to the conflict with the last instruction of Th_1, the function returns `POST_STOP`, making `y := 6` to be the only instruction forming Λ_2^s. The $\texttt{AccessTable}_2^s$ does not need to be updated.

Finally, when `pan_step(3)` is called, the check for independence on the first three instructions of Th_3 returns `CONTINUE`. Even though both the third instruction of Th_3 and the first instruction of Th_1 read from the same memory location: it can be determined at run time that k_ptr equals $ad(k)$, no conflict is reported. However, the fourth instruction, $*\texttt{x_ptr} := 8$, conflicts with the second entry in $\texttt{AccessTable}_1^s$ raising the `PRE_STOP` return code. Since the conflicting transition is not the last transition of Λ_1^s, $*\texttt{x_ptr} := 8$ should not be included in Λ_3^s.

5 Experimental Results

We have gathered some initial experimental results with our prototype on a few small multi-threaded C programs. Fig. 9 shows results from checking two versions of the implementation of Peterson's algorithm in C, described in Section 2. Fig. 9(a) shows the number of states explored against varying context bounds for the version of the program with the missing `volatile` keyword bug, while Fig. 9(b) shows similar results for the version of the program without the bug. The graphs also compare a heuristic that runs a thread until it makes an



(a) peterson.c with bug

(b) peterson.c without bug

Fig. 9. Growth of state space with increasing context switch bound

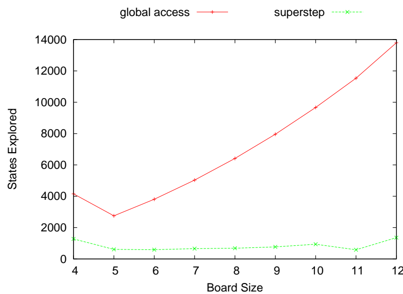


Fig. 10. Robot Example

Benchmark	#states global access	#states superstep
Phil n=2	59	37
Phil n=3	534	380
Phil n=4	4762	3130
Phil n=5	42386	25021
IPC m=1	156863	234
IPC m=2	625359	316
IPC m=3	1529342	479
IPC m=4	!	654
IPC m=15	!	22629

Fig. 11. Other Examples

access to any global state (labeled "global access" in the figure) versus our superstep reduction method (labeled "superstep"). As the graphs indicate, the bug is found fairly easily in all versions, though increasing the context bound beyond a certain point makes it harder to find the bug. (This is likely a consequence of the fact that SPIN uses depth-first search.) The graphs also show the benefit of an abstraction function we used which tracks only the algorithmic state of the protocol (the value of the abstract "flag" and "turn" variables).

Fig. 10 shows results from the "robot" benchmark example [GFYS07]. This example consists of two threads that move across a shared board of size $N \times N$ in slightly different patterns; the program checks that the robots meet only in expected locations. As the graph shows, our superstep method provides a noticeable reduction in the number of states over the global access method, as the size of the board grows.

Fig. 11 compares the improvement of superstep reduction with respect to the global access heuristic on two more examples. The first is a C implementation of the classic dining philosophers algorithm, with varying number of philosophers

(denoted by parameter n). The second example is an inter-process communication module for an upcoming mission. The module consists of around 2800 lines of (non-commented) C source code (including some support modules that it relies on). It implements a communication system that supports prioritized messages and provides thread-safe primitives for sending and receiving messages. To give meaningful results, we restricted the model to a single producer-consumer pair, and forced a bound of 4 context switches, while varying queue depth (denoted by m). Even with the small configuration parameters, the default global access heuristic exhausts memory resources for $m = 4$ (as denoted by the symbol !) on a machine with 32 GBytes of RAM, whereas the superstep method can handle much larger configurations (well over $m = 15$).

6 Related Work

There has been considerable interest in applying model checking directly to implementation code. The Bandera checker [CDH+00] translates Java programs to the input language for a model checker, while Java Pathfinder (JPF) [VHB+03] uses an approach more similar to ours, in that it interprets bytecode. However, JPF tightly integrates model checking with the virtual machine, whereas our tool uses SPIN to orchestrate the search, using our virtual machine to execute transitions. This allows us to inherit (for free) the various optimizations and features of SPIN (both those that exist, and those yet to be invented). In spite of this loose integration, our approach is flexible; for instance, as shown in Section 2, adding support for bounding context-switches was done fairly easily in our tool. Using Modex [HS99] - a tool which extracts PROMELA models from C implementations provides similar benefits. However, the model extractor is guided by user-defined abstractions, construction of which requires a considerable manual effort.

For verification of multi-threaded C programs, the CMC tool [MPC+02] uses explicit-state model checking. One limitation of CMC, however, is that it requires a manual step by the user to convert an existing C program into a form that can be used by CMC. In contrast, by working directly on bytecode, our tool design is simpler (interpreting typed LLVM bytecode is much easier than interpreting C). In addition, we are able to detect errors introduced during compiler optimization (like the Wikipedia error in Peterson's algorithm, described in Section 2).

The changes introduced by compiler optimizations are also addressed by work in connection with the CodeSurfer tool [BRMT05], in which program analysis is applied to a model constructed from an executable. Advantages of this approach are that, since it deals directly with object code, it is not tied to a specific compiler and it also catches errors introduced by the compiler back-end. However, the constructed model is not precise since it has to recover information about variables and types, which is especially difficult for aggregate types (such as structures and arrays).

Another tool for verifying C programs is VeriSoft [God97], which uses *stateless* model checking. VeriSoft uses static partial order reduction, which typically

results in little reduction when applied to C programs, since the independence relation is hard to compute. More recent work on dynamic partial-order reduction [FG05] addresses this problem, and the modifications have been implemented in the Inspect tool [YCGK07]. However, a limitation of these stateless approaches is that it requires the search depth to be bounded, which poses a challenge for programs whose state graphs have cycles.

More directly related to the superstep reduction presented in Section 4 is the work on “cartesian partial order reduction” [GFYS07], which is a method that dynamically computes independence relationships, and tries to avoid context switching whenever possible. The ideas behind cartesian partial order reduction and superstep reduction are closely related, though there are significant implementation differences. In particular, our reduction is done in the context of SPIN's depth-first search. While this complicates the design somewhat, and incurs some additional memory overhead, it can be applied even when checking liveness properties. (In contrast, the cartesian reduction method was applied only in the context of checking assertion violations and deadlocks.)

Our approach to enforcing context-bounding is directly inspired by the work on the CHESS model checker for concurrent C code [MQ08,MQ07]. One point of departure is that, even with fair scheduling, CHESS only checks livelocks; in contrast, our approach is able to handle general liveness properties.

7 Conclusion

We have described a tool that can be used in conjunction with the SPIN model checker to check multithreaded C programs. Our tool works by generating typed bytecode generated for the Low-Level Virtual Machine (LLVM), which is then interpreted by a virtual machine (named “pancam”). The virtual machine is designed to be used with SPIN, and the resulting tool therefore supports almost all SPIN features such as bitstate verification and multi-core operation. We have also shown we address the state explosion problem by allowing users to specify abstraction functions, context-switching bounds, and by using an on-the-fly algorithm for reducing unnecessary context switches. We are currently working on extending our tool to support checking liveness properties in the context of SPIN nested depth-first search.

References

[BRMT05] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. Wysinwyx: What you see is not what you execute. In *Proceedings of the IFIP Working Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, Oct 2005.

[CDH+00] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, June 2000.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM Symposium on Programming Languages (POPL)*, pages 110–121, 2005.

[GFYS07] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian partial-order reduction. In *SPIN Workshop on Model Checking of Software*, pages 95–112, 2007.

[GJ08] Alex Groce and Rajeev Joshi. Extending model checking with dynamic analysis. In *Proceedings of the Conference on Verification, Model Checking and Abstract Interpretation*, 2008.

[God97] P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

[HB07] Gerard J. Holzmann and Dragan Bosnacki. The design of a multi-core extension of the spin model checker. In *IEEE Transactions on Software Engineering*, volume 33, pages 659–674, October 2007.

[HJ04] Gerard J. Holzmann and Rajeev Joshi. Model-driven software verification. In *SPIN Workshop on Model Checking of Software*, pages 76–91, 2004.

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[HS99] Gerard Holzmann and Margaret Smith. A practical method for verifying event-driven software. In *International Conference on Software Engineering*, pages 597–607, 1999.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[MPC+02] Madanlal Musuvathi, David Park, Andy Chou, Dawson Engler, and David Dill. CMC: A pragmatic approach to model checking real code. In *Symposium on Operating System Design and Implementation*, 2002.

[MQ07] Madan Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 34th ACM Symposium on Programming Languages (POPL)*, pages 446–455, 2007.

[MQ08] Madanlal Musuvathi and Shaz Qadeer. Fair stateless model checking. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, 2008.

[Pel93] Doron Peled. All from one, one for all: on model checking using representatives. In *Proceedings of the 5th Conference on Computer Aided Verification*, pages 409–423. Springer, 1993.

[QR05] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, April 2005.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[wik] Peterson's algorithm. http://en.wikipedia.org/wiki/Peterson's_algorithm.

[YCGK07] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *Proceedings of the 14th International SPIN Workshop*, July 2007.