Context-Bounded Model Checking of LTL Properties for ANSI-C Software

Jeremy Morse[1], Lucas Cordeiro[2], Denis Nicole[1], Bernd Fischer[1]

[1] Electronics and Computer Science, University of Southampton, UK
{jcmm106,dan,bf}@ecs.soton.ac.uk

[2] Electronic and Information Research Center, Federal University of Amazonas, Brazil
lucascordeiro@ufam.edu.br

Abstract. Context-bounded model checking has successfully been used to verify safety properties in multi-threaded systems automatically, even if they are implemented in low-level programming languages like ANSI-C. In this paper, we describe and experiment with an approach to extend context-bounded model checking to liveness properties expressed in linear-time temporal logic (LTL). Our approach converts the LTL formulae into Büchi-automata and then further into C monitor threads, which are interleaved with the execution of the program under test. This combined system is then checked using the ESBMC model checker. Since this approach explores a larger number of interleavings than normal context-bounded model checking, we use a state hashing technique which substantially reduces the number of redundant interleavings that are explored and so mitigates state space explosion. Our experimental results show that we can verify non-trivial properties in the firmware of a medical device.

1 Introduction

Model checking has been used succesfully to verify actual software (as opposed to abstract system designs) [25, 2, 5, 1, 6], including multi-threaded applications written in low-level languages such as ANSI-C [9, 23, 17]. In *context-bounded model checking*, the state spaces of such applications are bounded by limiting the size of the program's data structures (e.g., arrays) as well as the number of loop iterations and context switches between the different threads that are explored by the model checker. This approach is typically used for the verification of safety properties expressed as assertions in the code, but it can also be used to verify some limited liveness properties such as the absence of global or local deadlock.

However, many important requirements on the software behavior can be expressed more naturally as liveness properties in a temporal logic, for example "whenever the start button is pressed the charge eventually exceeds a minimum level". Such requirements are difficult to check directly as safety properties; it is typically necessary to add additional executable code to the program under test to retain the past state information. This amounts to the *ad hoc* introduction of a hand-coded state machine capturing (past-time) temporal formulae.

Here, we instead we use context-bounded model checking to validate multi-threaded C programs directly against (future-time) temporal formulas over the variables and expressions of the C program under test. Thus, if the C variables `pressed`, `charge`, and `min` represent the state of the button, and the current and minimum charge levels, respectively, then we can capture the requirement above with the linear-time temporal logic (LTL) formula `G({pressed} -> F {charge > min})`. We check these formulas following the usual approach [7, 14], albeit with a twist: we convert the negated LTL formula (the so-called *never claim* [13]) into a Büchi automaton (BA), which is composed with the program under test; if the composed system admits an accepting run, the program violates the specified requirement. We check the actual C program however, rather than its corresponding BA. We thus convert the LTL's BA further into a separate C *monitor thread* and check all interleavings between this monitor and the program using ESBMC [9], an off-the-shelf, efficient bounded model checker for ANSI-C. We bound the execution of the monitor thread in such a way that it still searches for loops through accepting states after the program has reached its own bound. We thus consider the bounded program as the finite prefix of an infinite trace where state changes are limited to this finite prefix; this gives us a method to check both safety and liveness uniformly within the framework of bounded model checking.

Our approach avoids any imprecision from translating the C program into a BA, but the monitor has to capture transient behaviour internal to the program under test. The monitor and the program communicate via auxiliary variables reporting the truth values of the LTL formula's embedded expressions. Our tool automatically inserts and maintains these and also uses them to guide ESBMC's thread exploration. Nevertheless, our approach requires that the underlying bounded model checker must be able to accommodate deep interleavings of the monitor thread with the program threads. We have thus implemented a state hashing strategy which eliminates multiple examinations of identical parts of the state space and improves ESBMC's performance.

Our paper makes three main contributions. First, it describes the first mechanism, to the best of our knowledge, to verify LTL properties against an unmodified C code base. Second, since ESBMC is a symbolic model checker based on the satisfiability modulo theory approach, it also describes the first symbolic LTL model checker that does not use binary decision diagrams (BDDs). Third, it is the first application of the concept of state hashing to symbolic model checking.

2 From LTL to Monitor Threads

2.1 Linear-time Temporal logic

Linear-time temporal logic (LTL) is a commonly used specification logic in model checking [3, 15, 16], which extends propositional logic by including temporal operators. The primitive propositions of our LTL are side-effect-free boolean C expressions over the global variables of the C program.

Definition 1. *Our LTL syntax is defined over primitive propositions, logical operators and temporal operators as follows:*

$$\begin{aligned}\phi ::= &\ \texttt{true} \mid \texttt{false} \mid \{p\} \mid \texttt{!}\,\phi \mid \phi_1\ \texttt{\&\&}\ \phi_2 \\ &\mid \phi_1\ \texttt{||}\ \phi_2 \mid \phi_1\ \texttt{->}\ \phi_2 \mid \texttt{(}\phi\texttt{)} \\ &\mid \texttt{F}\phi \mid \texttt{G}\phi \mid \phi_1\texttt{U}\phi_2 \mid \phi_1\texttt{R}\phi_2\end{aligned}$$

The logical operators include *negation* (!), *conjunction* (&&), *disjunction* (||) and *implication* (->). The temporal operators are "in some future state (eventually)" (F), "in all future states (globally)" (G), "until" (U) and "release" (R). Here, p is a side-effect-free boolean C expression, and $\phi_1\texttt{U}\phi_2$ means that ϕ_1 must hold continuously until ϕ_2 holds; ϕ_2 must become true before termination. The other temporal operators can be defined in terms of U, as shown below.

We are only interested in temporal formulae which are closed under stuttering; following Lamport [19], we thus do not provide an explicit "next state" operator X. Our LTL expressions are thus insensitive to refinements of the timestep to intervals less than those required to capture the ordering of changes in the global state. The timesteps however only need to be sufficiently fine to resolve any potentially dangerous interleavings of the program. For efficiency reasons we assume interleavings only at statement boundaries and assume sequential consistency [18], but options to ESBMC allow us also to use a finer-grained analysis to detect data races arising from interleavings within statement evaluations.

We use a *linear-time* rather than a *branching-time* approach and thus there are no explicit path quantifications (i.e., CTL*-style operators A and E). There is, however, an implicit universal quantification over all possible interleavings and program executions. In this formulation we have the following identities: [1]

$$\begin{aligned}\phi &= \texttt{false}\ \texttt{U}\ \phi \\ \texttt{F}\ \phi &= \texttt{true}\ \texttt{U}\ \phi \\ \texttt{G}\ \phi &= \ \texttt{!F}\ \texttt{!}\phi \\ \phi_1\ \texttt{R}\ \phi_2 &= \texttt{!}\ (\ \texttt{!}\ \phi_1\ \texttt{U}\ \texttt{!}\ \phi_2\ )\end{aligned}$$

We interpret a possibly multi-threaded C program as a Kripke structure whose state transitions are derived from the possibly interleaved execution sequence of C statements and whose valuations are the possible values of the program's global variables. Since we have implemented a *bounded* model checker, all (bounded) programs will either deadlock or terminate in finite time. We use a separate run of ESBMC to assure deadlock freedom and formally extend the behaviour of deadlock-free programs with an infinite sequence of timesteps which leave all global variables unchanged. Thus every program that is scheduled generates an infinite sequence of states. We finally describe the desired liveness property ϕ as an LTL expression in the above syntax and then check that there are no possible infinite sequences of program states for which ! ϕ holds.

[1] This differs from the notation of [20], which has $\texttt{X}\ \phi = \texttt{false}\ \texttt{U}\ \phi$

2.2 Büchi Automata

Büchi automata (BA) are finite-state automata over infinite words first described by Büchi [4]. We follow Holzmann's presentation [14] and define a BA as a tuple $B = (S, s_0, L, T, F)$ where S is a finite set of states, $s_0 \in S$ the initial state of the BA, L a finite set of labels, $T \subseteq (S \times L \times S)$ a set of state transitions and $F \subseteq S$ a set of final states. B may be deterministic or non-deterministic. A *run* is a sequence of state transitions taken by B as it operates over some input. A run is accepted if B passes through an accepting state $s \in F$ infinitely often along the run.

A number of algorithms exist for converting an LTL formula to a BA accepting a program trace [11, 24, 12]. We use the `ltl2ba` [11] algorithm and tool, which produces smaller automata than some other algorithms. Figure 1 illustrates the BA produced from the example LTL formula in the introduction. Input symbols are propositions composed from the primitive C-expressions used in the LTL.

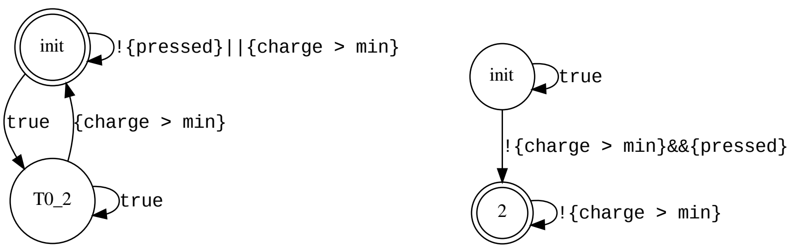

Fig. 1. The left BA accepts the example from the introduction, `G({pressed} -> F {charge > min})`. The right BA is its negation, used for the never claim in our monitor

2.3 Monitor threads

In our context, a monitor is some portion of code that inspects program state and verifies that it satisfies a given property, failing an assertion if this is not the case. A monitor thread is a monitor that is interleaved with the execution of the program under test. This allows it to verify that the property holds at each particular interleaving of the program, detecting any transient violations between program interleavings.

Monitor threads have been employed in SPIN to verify LTL properties against the execution of a program [14]. A non-deterministic BA representing the negation of the LTL property, the so-called never claim, is implemented in a Promela process which will accept a program trace that violates the original LTL property. SPIN then generates execution traces of interleavings of the program being verified, and for each step in each trace runs the Promela BA. This is called a *synchronous interleaving*.

In this work we employ a similar mechanism to verify LTL properties by interleaving the program under verification with a monitor thread, detailed in Section 3.2.

3 Model-Checking LTL Properties with ESBMC

3.1 ESBMC

ESBMC is a context-bounded model checker for embedded ANSI-C software based on SMT solvers, which allows the verification of single- and multi-threaded software with shared variables and locks [10, 9]. ESBMC supports full ANSI-C, and can verify programs that make use of bit-level, arrays, pointers, structs, unions, memory allocation and fixed-point arithmetic. It can reason about arithmetic under- and overflows, pointer safety, memory leaks, array bounds violations, atomicity and order violations, local and global deadlocks, data races, and user-specified assertions.

In ESBMC, the program to be analyzed is modelled as a state transition system $M = (S, R, s_0)$, which is extracted from the control-flow graph (CFG). S represents the set of states, $R \subseteq S \times S$ represents the set of transitions (i.e., pairs of states specifying how the system can move from state to state) and $s_0 \subseteq S$ represents the set of initial states. A state $s \in S$ consists of the value of the program counter *pc* and the values of all program variables. An initial state s_0 assigns the initial program location of the CFG to *pc*. We identify each transition $\gamma = (s_i, s_{i+1}) \in R$ between two states s_i and s_{i+1} with a logical formula $\gamma(s_i, s_{i+1})$ that captures the constraints on the corresponding values of the program counter and the program variables.

Given the transition system M, a safety property ϕ, a context bound C and a bound k, ESBMC builds a reachability tree (RT) that represents the program unfolding for C, k and ϕ. We then derive a VC ψ_k^π for each given interleaving (or computation path) $\pi = \{\nu_1, \ldots, \nu_k\}$ such that ψ_k^π is satisfiable if and only if ϕ has a counterexample of depth k that is exhibited by π. ψ_k^π is given by the following logical formula:

$$\psi_k^\pi = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \neg\phi(s_i) \tag{1}$$

Here, I characterizes the set of initial states of M and $\gamma(s_j, s_{j+1})$ is the transition relation of M between time steps j and $j+1$. Hence, $I(s_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ represents executions of M of length i and ψ_k^π can be satisfied if and only if for some $i \leq k$ there exists a reachable state along π at time step i in which ϕ is violated. ψ_k^π is a quantifier-free formula in a decidable subset of first-order logic, which is checked for satisfiability by an SMT solver. If ψ_k^π is satisfiable, then ϕ is violated along π and the SMT solver provides a satisfying assignment, from which we can extract the values of the program variables to construct a counter-example. A counter-example for a property ϕ is a sequence of states $s_0, s_1, \ldots, s_k$ with $s_0 \in S_0$, $s_k \in S$, and $\gamma(s_i, s_{i+1})$ for $0 \leq i < k$. If ψ_k^π is unsatisfiable, we can conclude that no error state is reachable in k steps or less along π. Finally, we can define $\psi_k = \bigwedge_\pi \psi_k^\pi$ and use this to check all paths. However, ESBMC combines symbolic model checking with explicit state space exploration; in particular, it explicitly explores the possible interleavings (up to the given context bound) while it treats each interleaving itself symbolically. ESBMC implements different variations of this approach, which differ in the way they exploit the RT. The most effective variation simply traverses the RT depth-first, and calls the single-threaded BMC procedure on the interleaving whenever it reaches an RT leaf node. It stops when it finds a bug, or has systematically explored all possible RT interleavings.

3.2 Checking LTL properties against a C program

As discussed in Section 2.3, an LTL property can be verified against a program by interpreting the corresponding BA over the program states along the execution path. We apply this approach to a C code base by implementing the BA in C which is then executed as a montior thread, interleaved with the execution of the program. This involves three technical dimensions: the conversion of the BA to C, the interaction of the monitor thread with the program under test, and the control of the interleavings.

The monitor thread itself is not interleaved with the program in a special manner as in SPIN, but instead is treated as any other program thread. We use a counting mechanism to ensure that the BA thread operates on the program states in the right sequential order. This approach can be slower than a synchronous composition, but it requires no fundemental changes to the way that ESBMC operates as it uses only existing features.

Implementing a Büchi automata in C. We follow the SPIN approach of inverting the LTL formula being verified so that the BA accepts execution traces which violate the original formula. We then modified the `ltl2ba` tool to convert its usual Promela output to C, which uses some ESBMC built-ins.

The C implementation of the BA (see Figure 2 for the code corresponding to the BA in Figure 1) consists of an infinite loop (which will be unrolled the appropriate number of times, see below for details) around a switch statement on the state variable that branches to code which atomically (lines 18, 46) evaluates the target state of the transition. The non-deterministic behavior is simulated by attempting all transitions from a state non-deterministically (lines 24, 27, 36), after which guards on each transition evaluate whether the transition can be taken (lines 25, 28, 37). These guards use ESBMC's *assume* statements, which ensure that transitions not permitted by the current state of the program under test are not explored.

To determine when the BA has accepted a program trace, we first await a time where the program has terminated — given that we operate in the context of bounded model checking this is guarenteed as any infinite loop is unrolled only to the length of the bound. Detection of thread deadlock is already performed by ESBMC [9]. Once the program terminates, its state never changes. The BA loop is run a second time with the final program state as input, recording the number of times it passes through each state (lines 44-45). If a loop through an accepting state exists it will be visited more than once and an assertion in the monitor thread will be fired showing that the BA accepted the trace. This technique places a constraint on the unwind bound of the BA loop, that it is sufficient for any such loop to be detected. Setting this bound to twice the number of states in the BA permits it to pass through every state twice on the largest possible loop.

This acceptance criteria operates on the principle that, should some program state need to be reached for the LTL formula to hold, then it needs to have happened by the time that the program bound has been reached. This can be an overapproximation of the program being verified, as there can be circumstances where that program state could be reached if the program bound were higher.

Several practical issues are addressed to ensure the BA is sound. The evaluation of the next state to transition to is executed atomically, ensuring that the BA always perceives a consistent view of program state. We also yield execution (line 17) before the BA inputs a program state to force new interleavings to be explored. Certain utility functions are provided to allow a program test harness to start the BA and check for acceptance at the end of execution (not shown).

Interacting with the existing code base. LTL formulas allow verification engineers to describe program behavior with propositions about program state. To describe of the state of a C program, we support the use of C expressions as propositions within LTL formulas. Any characters in the formula enclosed in braces are interpreted as a C expression and as a single proposition within LTL. The expression itself may use any global variables that exist within the program under test as well as constants and side-effect free operators. The expression must also evaluate to a value that can be interpreted as a boolean under normal C semantics.

For example, the following liveness property verifies that a certain input condition results in a timer increasing:

```
!G(({pressed_key == 4} && {mstate == 1}) -> F{stime > ref_stime})
```

and the following safety property checks a buffer bound condition:

```
!G({buffer_size != 0} -> {next < buffer_size})
```

Within the BA (Figure 2) these expressions are required for use in the guards preventing invalid transitions from being explored. We avoid using the expressions directly in the BA and instead ESBMC searchs the program under verification for assignments to global variables used in a C expression, then inserts code to

```
char __ESBMC_property___cexpr_0[] = "pressed";
bool __cexpr_0_status;
char __ESBMC_property___cexpr_1[] = "charge > min";
bool __cexpr_1_status;

typedef enum {T0_init, accept_S2 } ltl2ba_state;
ltl2ba_state state = T0_init;
unsigned int __visited_states[2];
unsigned int __transitions_seen;
extern unsigned int __transitions_count;

void ltl2ba_fsm(bool state_stats) {
  unsigned int choice;
  while (1) {
    choice = nondet_uint();
    /* Force a context switch */
    __ESBMC_yield();
    __ESBMC_atomic_begin();
    __ESBMC_assume(__transition_count <=
                    __transitions_seen + 1);
    __transitions_seen = __transition_count;
    switch(state) {
    case T0_init:
      if (choice == 0) {
        __ESBMC_assume((1));
        state = T0_init;
      } else if (choice == 1) {
        __ESBMC_assume((!__cexpr_1_status &&
                        __cexpr_0_status));
        state = accept_S2;
      } else {
        __ESBMC_assume(0);
      }
      break;
    case accept_S2:
      if (choice == 0) {
        __ESBMC_assume((!__cexpr_1_status));
        state = accept_S2;
      } else {
        __ESBMC_assume(0);
      }
      break;
    }
    if (state_stats)
      __visited_states[state]++;
    __ESBMC_atomic_end();
  }
}
return;
```

Fig. 2. C implemention of the Büchi automaton for the formula `!G({pressed} -> F {charge > min})`.

update a Boolean variable corresponding to the truth of the expression (lines 2, 4) immediately after the symbol is assigned to. In case multiple propositions update on the same variable, re-evaluations are executed atomically. All modifications are performed on ESBMC's internal representation of the program and do not alter the code base.

Synchronous Interleaving. An impediment of operating the monitor thread containing the BA as a normal program thread is that it is not guarenteed always to receive a consistent series of input states—that is, it is entirely possible for the BA not to be scheduled to run after an event of interest, and thus not perform a state transition it should have. This is clearly an invalid action when one considers it in terms of a BA skipping an input symbol. While such interleavings can occur it is also guarenteed that there will be interleavings where the BAa is run often enough to observe all relevant program states.

To handle this the BA discards interleavings where the propositions have changed more than once but the BA has not had opportunity to run and interpret them (lines 19–21 in Figure 2). We maintain a global variable (line 10) counting the number of times that the C expressions forming propositions in the LTL formula have been re-evaluated, keep a corresponding counter (line 9, 21) within the BA, and use an assume statement to only consider traces where the the global counter has changed at most once since the last time the BA ran.

4 Optimizing State Space Exploration

The context-bounded approach has proven to be effective for model checking multi-threaded software, with a small number of context switches allowing us to explore much of the system behavior. However, our approach to verifying programs against LTL properties requires frequent context-switching between monitor and program threads, which makes a greater context bound necessary. We implemented *state hashing* in ESBMC to eliminate redundant interleavings and reduce the state space to be explored. This is also the first work to our knowledge where state hashing has been used in conjunction with symbolic model checking.

The driving force behind our approach to state hashing is that during the exploration of the RT of multi-threaded software many interleavings pass through identical RT nodes, i.e., nodes that represent the same global and thread-local program states, respectively, and differ only in the currently active thread. Only one of these nodes need be explored, as the reachability subtrees of all other nodes will be identical. As an example, consider a simple multithreaded C program shown in Figure 3 and its corresponding RT shown in Figure 4. The RT consists of the nodes ν_0 to ν_{16}, where each node is defined as a tuple $\nu = (A_i, C_i, s_i, \langle l_i^j, G_i^j \rangle_{j=1}^n)_i$ for a given time step i. Here, A_i represents the currently active thread, C_i the context switch number, and s_i the current (global and local) state. Further, for each of the n threads, l_i^j represents the current location of thread j and G_i^j represents the control flow guards accumulated in thread j along the path from l_0^j to l_i^j (although these are not shown in Figure 4). Notice how the transitions originating from node ν_1 as those originating from ν_7, produce the same program states. When we explore the node ν_7, we can simply eliminate the transitions that originate from it — provided that we realise that we have already explored another identical RT node. We thus maintain a set of hashes representing the states of RT nodes that we have already explored.

```
#include <pthread.h>

int x=0, y=0;

void* t1(void* arg) {
  x++;
  return NULL;
}

void* t2(void* arg) {
  x++;
  return NULL;
}

void* t3(void* arg) {
  y++;
  return NULL;
}

int main(void) {
  pthread_t id1, id2, id3;
  pthread_create(&id1,NULL,t1,NULL);
  pthread_create(&id2,NULL,t2,NULL);
  pthread_create(&id3,NULL,t3,NULL);
  return 0;
}
```

Fig. 3. A simple multi-threaded C program.

4.1 Hashing symbolic states

In explicit-state model checking, state hashing takes a state vector containing the current *values* of all program variables, and applies a hash function to compute a hash value that can then be stored to indicate a particular state has been explored.

Unfortunately, state hashing is not so simple for symbolic model checking, as the state vector does not simply contain values but is defined symbolically by the calculations and constraints that make up the variable assignments in

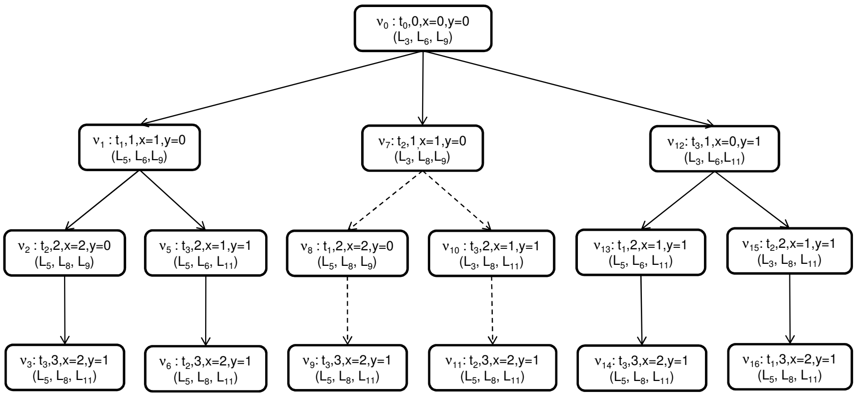
v_0 : t_0,0,x=0,y=0 (L_3, L_6, L_9)
v_1 : t_1,1,x=1,y=0 (L_5, L_6,L_9)
v_7: t_2,1 x=1,y=0 (L_3, L_8,L_9)
v_{12}: t_3,1,x=0,y=1 (L_3, L_6,L_{11})
v_2 : t_2,2,x=2,y=0 (L_5, L_8, L_9)
v_5 : t_3,2,x=1,y=1 (L_5, L_6, L_{11})
v_8 : t_1,2,x=2,y=0 (L_5, L_8, L_9)
v_{10} : t_3,2,x=1,y=1 (L_3, L_8, L_{11})
v_{13}: t_1,2,x=1,y=1 (L_5, L_6, L_{11})
v_{15}: t_2,2,x=1,y=1 (L_3, L_8, L_{11})
v_3: t_3,3,x=2,y=1 (L_5, L_8, L_{11})
v_6 : t_2,3,x=2,y=1 (L_5, L_8, L_{11})
v_9: t_3,3,x=2,y=1 (L_5, L_8, L_{11})
v_{11}: t_2,3,x=2,y=1 (L_5, L_8, L_{11})
v_{14}: t_3,3,x=2,y=1 (L_5, L_8, L_{11})
v_{16}: t_1,3,x=2,y=1 (L_5, L_8, L_{11})

Fig. 4. Reachability tree for the program in Figure 3 Dashed edges represent transitions that can be eliminated by the state hashing technique.

the underlying static single assignment (SSA) form of the program. We thus implement a two-level hashing scheme: we use a node-level hash that represents a particular RT node, and a variable level hash that represents the constraints affecting a particular assignment to a variable. Since each new RT node can only change the (symbolic) value of at most one variable, the two-level hashing scheme reduces the compoutational effort, as it allows us to retain the hash values of the unchanged variables.

The node-level hash is created by taking the variable-level hashes of all variables in the current node and concatenating them, together with the program counter values of all existing threads, into a single data vector. This vector is then fed to a hashing function. Variable-level hashes are more complex. Within ESBMC, assignments are defined by an expression representing the calculation and the variable name it is assigned to. For each symbolic assignment encountered in the RT exploration we calculate a hash of the expression and record it with the variable name. This hash is created by serialising each operator and value in the expression to a data representation (i.e., a series of bytes) into a vector, which is then hashed.

For example, Figure 3 contains several assignments to the global variable `x` using the `++` operator (converted to a `+` internally). ESBMC automatically performs constant propagation and effectively converts the example to an explicit state check. We represent the first serialised increment expression as the text:

```
(+,(constant(0)),(constant(1)))
```

This demonstrates one of the simplest encodings of data possible with this method. Any set of operations on constant values can also be expressed in this manner. However such expressions are not yet symbolic — to support this we represent nondeterminstic values with a prefix and unique identifier. We also represent the use of existing variables in expressions with its current variable hash.

To demonstrate this, reconsider Figure 3 and assume the x variable is initialised to a nondeterministic value. The expressions representing the two increments of the x variable then become:

$$(+, (\texttt{nondet}(1)), (\texttt{constant}(1)))$$

$$(+, (\texttt{hash}(\#1)), (\texttt{constant}(1)))$$

where #1 represents the hash value of the first expression. Significantly, no thread specific data is encoded in this representation, meaning that the same serialised representation is produced for whichever order of threads increments the x variable. This means the hash of any assignment is a direct product of all nondeterministic inputs, constant values and operators that represent the constraints on the assignment.

This method is limited however by the ordering of assignments — if the original example in Figure 3 had instead a thread that increased the x variable by 2, and another that increased x by 3, then at the end of execution the variable hash of x would be different depending on the thread ordering, even though the effective constraints on x for every interleaving are identical. This also affects arrays (including the heap, which is modelled as an array) and unions.

4.2 Selection of hash function

As hashing is a lossy abstraction of a tree node, we risk computing two identical hashes for two distinct nodes. Should this occur one node will be successfully explored and its hash stored; but when the other is explored we will discover its hash already in the visited states set, and incorrectly assume it has already been visited. This would cause an unexplored portion of the state space to be discarded.

We require a hash function that takes a stream of characters as input (serialised expressions) and produces a small output. We simply chose SHA256 [22] hashes due to its relatively large output bitwidth (compared to other hash functions) and its certification for use in cryptographic applications, aspects that assure us the likelihood of collisions is extremely low.

5 Experimental Evaluation

We have tested the work described here against a series of properties defining the behavior of a pulse oximeter firmware, which is a piece of sequential software that is responsible for measuring the oxygen saturation (SpO_2) and heart rate (HR) in the blood system using a non-invasive method [8]. The firmware of the pulse oximeter device is composed of device drivers (i.e., display, keyboard, serial, sensor, and timer) that are hardware-dependent code, a system log component that allows the developer to debug the code through data stored on RAM memory, and an API that enables the application layer to call the services provided by the platform. The final version of the pulse oximeter firmware has approximately 3500 lines of ANSI-C code and 80 functions.

Here we report the results of verifying the pulse oximeter code against five liveness properties of the general form

```
G(p -> F q)
```

i.e., whenever an enabling condition `p` has become true, then eventually the property `q` is enabled. We formulated a test harness for each portion of the firmware being tested to simulate the activity that the LTL property checks. We then invoked ESBMC with a variety of loop unwind and context switch bounds to determine the effectiveness of state hashing. We also ran these tests against versions of the firmware deliberately altered to not match the LTL formula to verify that failing execution traces are identified.

All tests were run on the Iridis 3 compute cluster [2] , with a memory limit of 4Gb and time limit of 4 hours to execute. The results are summarized in Table 5. Here, #L column contains the line count of the source file for the portion of firmware being tested, P/F records whether the test is expected to Pass or Fail, k the loop unwinding bound and C the context-bound specified for the test.

We then report the results for the original version of ESBMC (v1.16, which will be available from www.esbmc.org) and the version with state hashing, respectively. For each version, we report the verification time in seconds, the number #I and #FI of generated and failing interleavings, respectively, and the result. Here, + indicates that ESBMC's result is as expected (i.e. all its interleavings were verified successfully if the test is expected to pass, and at leasat one interleaving is found to violate the LTL property if the test is expected to fail), while − indicates a false negative (i.e., ESBMC fails to find an existing violation of the LTL property). TO indicates the check ran out of time and MO indicates it ran out of memory.

We first observe that ESBMC is generally able to verify all positive test cases, although it sometimes times out with increasing bounds. The situation is less clear for the tests designed to fail. Here, smaller unrolling and context switch bounds allow to correctly identify failing interleavings, but are sometimes not sufficient to expose the error (e.g., up_btn), and small increases in the unrolling bound generally require larger increases in the context bounds to expose the error, leading to time-outs or memory-outs in most cases. However, state hashing improves the situation, and allows us to find even deeply nested errors.

Comparing the figures between tests performed with state hashing and those without, we see that the total number of interleavings generated is often significantly reduced by state hashing. Out of all tests that completed the median reduction was 56%, the maximum 80% and minimum 13%. In all cases the use of state hashing reduced the amount of time required to explore all reachable states.

[2] 1008 Intel Nehalem compute nodes, each with two 4-core processors, up to 45Gb of RAM per node, and InfiniBand communications. For each test we used only one core of one node.

					Original run			With state hashing		
Test name	#L	P/F	k	C	Time (s)	#I / #FI	Result	Time (s)	#FI / #I	Result
start_btn	856	Pass	1	20	207	7764/0	+	67	2245/0	+
		Pass	1	40	199	7764/0	+	71	2245/0	+
		Pass	2	20	2740	55203/0	+	479	11409/0	+
		Pass	2	40	14400	0/0	TO	14400	0/0	TO
		Fail	1	20	236	6719/231	+	81	1919/91	+
		Fail	1	40	244	6719/231	+	94	1919/91	+
		Fail	2	20	1344	29840/0	−	299	6911/0	−
		Fail	2	40	N/A0	0/0	MO	N/A	0/0	MO
up_btn	856	Pass	1	20	78	3775/0	+	32	1385/0	+
		Pass	1	40	83	3775/0	+	37	1385/0	+
		Pass	2	20	2777	102566/0	+	898	41389/0	+
		Pass	2	40	14400	0/0	TO	6012	111335/0	+
		Fail	1	20	90	3775/0	−	35	1385/0	−
		Fail	1	40	82	3775/0	−	33	1385/0	−
		Fail	2	20	2743	102564/0	−	914	40938/0	−
		Fail	2	40	14400	0/0	TO	4832	69275/3422	+
keyb_start	50	Pass	1	20	9668	92795/0	+	4385	49017/0	+
		Pass	1	40	9767	92795/0	+	4489	49017/0	+
		Pass	2	20	14400	0/0	TO	14400	0/0	TO
		Pass	2	40	14400	0/0	TO	14400	0/0	TO
		Fail	1	20	9795	92795/321	+	4836	49017/321	+
		Fail	1	40	9924	92795/321	+	4914	49017/321	+
		Fail	2	20	14400	0/0	TO	14400	0/0	TO
		Fail	2	40	14400	0/0	TO	14400	0/0	TO
baud_conf	178	Pass	1	20	18	485/0	+	16	419/0	+
		Pass	1	40	17	485/0	+	16	419/0	+
		Pass	2	20	2440	39910/0	+	971	17500/0	+
		Pass	2	40	2635	39910/0	+	1078	17500/0	+
		Fail	1	20	18	485/56	+	17	419/56	+
		Fail	1	40	18	485/56	+	16	419/56	+
		Fail	2	20	2583	39910/2002	+	1010	17500/880	+
		Fail	2	40	2851	39910/2002	+	1139	17500/880	+
serial_rx	584	Pass	1	20	334	5454/0	+	194	3108/0	+
		Pass	1	40	324	5454/0	+	212	3108/0	+
		Pass	2	20	10959	62332/0	+	4494	29257/0	+
		Pass	2	40	14400	0/0	TO	70	627/0	+
		Fail	1	20	215	3286/273	+	137	2030/257	+
		Fail	1	40	211	3286/273	+	135	2030/257	+
		Fail	2	20	3768	20917/0	−	1846	11388/0	−
		Fail	2	40	14400	0/0	TO	14400	0/0	TO

Table 1. Timings and results from testing LTL properties against pulse oximeter firmware

6 Related Work

SPIN [13] is a well known software model checker that operates on concurrent program models written in the Promela modelling language. It operates with explicit state and uses state hashing to reduce the quantity of state space it explores. SPIN also allows users to specify a LTL formula to verify against the execution of a model by using BA in a similar manner to this work. While SPIN is well established as a model checker the requirement to re-model codebases in Promela can be time consuming.

Java PathFinder is a Java Virtual Machine (JVM) that performs model checking on Java bytecode. It also operates with explicit state and uses *state matching* to reduce the search space, but can also operate symbolically for the purpose of test generation and coverage testing. Verification of LTL formula can be achieved with the JPF-LTL extension which uses BA and method invocation monitoring to inspect the execution of the model.

7 Conclusions and future work

Context-bounded model checking has been used successfully to verify multi-threaded applications written in low-level languages such as ANSI-C. However, the approach has largely been confined to the verification of safety properties. In this paper, we have extended the approach to the verification of liveness properties given as LTL formulas against an unmodified code base. We follow the usual approach of composing the BA for the never claim with the program, but work at the actual code level. We thus convert the BA further into a seperate C monitor thread and check all interleavings between this monitor and the program using ESBMC. We use a state thashing scheme to handle the large number of interleavings and counter state space explosion.

The initial results are encouraging, and we were able to verify a number of liveness properties on the firmware of a medical device; in future work, we plan to extend the evaluation to a larger code base and wider variety of properties. The state hashing proved to be very useful, cutting verification times by about 50% on average. We expect that an improved hashing implementation (e.g., removing serialisation) will yield even better results. However, our approach still has some limitations. The (indiscriminate) composition of the monitor thread with the program under test leads to a very large number of interleavings taht need to be explored, despite the improvements that the state space hashing provides. We thus plan to implement a special thread scheduling algorithm in ESBMC that schedules the monitor after changes to the observed variables and so achieves the effect of synchronous composition.

References

[1] Michael Barnett, Robert DeLine, Manuel Fähndrich, Bart Jacobs, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. The Spec# programming system: Challenges and Directions. In *VSTTE*, pages 144–152, 2005.

[2] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST. *STTT*, 9(5-6):505–525, 2007.

[3] Armin Biere, Keijo Heljanko, Tommi Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. Logical Methods in Computer Science 2, pages 1–64, 2006.

[4] J. Richard Büchi On a Decision Method in Restricted Second Order Arithmetic. Studies in Logic and the Foundations of Mathematics 44, pages 1–11, 1966.

[5] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, *LNCS* 2988, pages 168–176, 2004.

[6] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design (FMSD)* 25, pages 105–127, September–November 2004.

[7] E.M. Clarke, F. Lerda Model Checking: Software and Beyond. Journal of Universal Computer Science 13, pages 639–649, 2007.

[8] L. Cordeiro et al. Agile development methodology for embedded systems: A platform-based design approach. *ECBS*, pp. 195–202, 2007.

[9] Lucas Cordeiro and Bernd Fischer. Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking. To appear in *ICSE*, 2011.

[10] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. In *ASE*, pages 137–148, 2009.

[11] P. Gastin and D. Oddoux Fast LTL to Büchi Automata Translation. In *CAV*, LNCS 2102, pages 53-65. 2001.

[12] Anping He, Jinzhao Wu and Lian Li An Efficient Algorithm for Transforming LTL Formula to Büchi Automaton. In *ICICTA*, pages 1215–1219, 2008.

[13] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.* 23, pages 279–295, 1997.

[14] Gerard J. Holzmann. The SPIN Model Checker - primer and reference manual. Addison-Wesley, 2004.

[15] Michael Huth and Mark Ryan Logic in Computer Science: modelling and reasoning about systems. Cambridge University Press, 2004.

[16] Bengt Jonsson and Yih-Kuen Tsay. Assumption/guarantee specifications in linear-time temporal logic. Theor. Comput. Sci., 167(1&2):47–72, 1996.

[17] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. Static and precise detection of concurrency errors in systems code using SMT solvers. In *CAV*, pages 509–524, 2009.

[18] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. ACM Trans. Program. Lang. Syst. 1, pages 84–97, 1979.

[19] Leslie Lamport What Good is Temporal Logic? Information Processing 83, pages 657–668, 1983.

[20] Kenneth L. McMillan Symbolic Model Checking. Kluwer Academic Publishers, 1003, page 15.

[21] Steven S. Muchnick. Advanced compiler design and implementation. Morgan Kaufmann Publishers Inc., 1997.

[22] Secure Hash Standard. National Institute of Standards and Technology. Federal Information Processing Standard 180-2. 2002.

[23] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In *CAV*, *LNCS* 3576, pages 82–97, 2005.

[24] Kristin Y. Rozier and Moshe Y. Vardi LTL Satisfiability Checking. Software Tools for Technology Transfer 12, pages 123-137, 2010.

[25] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.