

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

Jeremy Morse, Lucas Cordeiro,
Bernd Fischer, Denis Nicole



UFAM

Model Checking C

Model checking:

- normally applied to formal state transition systems
- checks safety and temporal properties

Software model checking:

- models are abstractions, not necessarily precise
- no guarantee that model and software agree

BUT: C is difficult to model check:

- weakly typed \Rightarrow conversion increase model complexity
- pointers \Rightarrow indirections increase model complexity
- infinite state
- parts deliberately undefined, implementation- or host-specific
 \Rightarrow need to handle useful or common interpretations

SMT-based bounded model checker for C, based on CBMC:

- symbolically executes C into SSA, produces QF formulae
- unrolls loops up to a maximum bound
- assertions
 - safety properties (e.g., pointer dereferences, overflows,...)
 - user-specified properties

Goal: support LTL formulas in properties

Multi-threaded programs:

- produces one SSA program for each possible thread interleaving
- interleaves only at “visible” instructions
- optional context bound

LTL – Linear Temporal Logic

Supported operators:

- U: p holds **until** q holds $p \text{ U } q$
- F: p will hold eventually in the **future** $F p$
- G: p **always** holds in the future $G p$
- X is not well defined for C
 - no notion of “next”
- C expressions used as atoms in LTL:

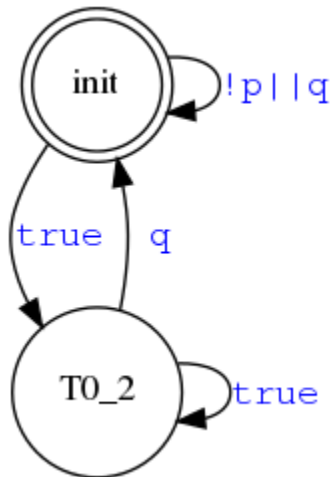
$\{\text{keyInput} == 1\} \rightarrow F \{\text{displayKeyUp}\}$

$(\{\text{keyInput} != 0\} \mid \{\text{intr}\}) \rightarrow G\{\text{numInputs} > 0\}$

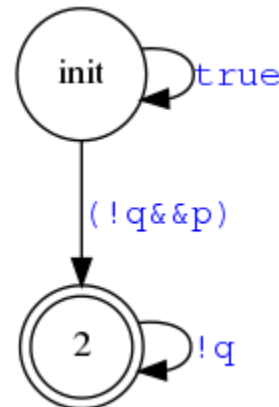
“event”: change of global variable used in LTL formula

Büchi Automata (BA)

- non-deterministic FSM over propositional expressions
- inputs infinite length traces
- acceptance == trace passes through an accepting state infinitely often
- can convert from LTL to an equivalent BA
 - use `l2ba`, modified to produce C



$p \rightarrow Fq$



$!(p \rightarrow Fq)$

Using BAs to check the program

- Theory: check product of model and *never claim* for accepting state
- SPIN: execute *never claim* in lockstep with model
- ESBMC:
 - technically difficult to alternate between normal program and *never claim* program
 - instead: run *never claim* program as a monitor thread concurrently with other program thread(s)
 - ⇒ no distinction between monitor thread and other threads

Ensuring soundness of monitor thread

Monitor thread will miss events:

- interleavings will exist where events are skipped (monitor thread scheduled out of sync)
- ⇒ can cause false violations of the property being verified
- ⇒ monitor thread must be run immediately after events

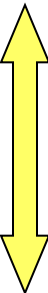
Solution:

- ESBMC maintains (global) current count of events
 - monitor checks it processes events one at a time (using assume statements)
- ⇒ causes ESBMC to discard interleavings where monitor does not act on relevant state changes

Example monitor thread



```
bool cexpr_0; // "pressed"
bool cexpr_1; // "charge > min"
```

```
typedef enum {T0_init, accept_s2 } lt12ba_state;
lt12ba_state state = T0_init;
unsigned int visited_states[2];
unsigned int trans_seen;
extern unsigned int trans_count;
```

 State transition and "event" counter setup

```
void lt12ba_fsm(bool state_stats) {
    unsigned int choice;
    while(1) {
        choice = nondet_uint();
        /* Force a context switch */
        yield();
        atomic_begin();
        assume(trans_count <= trans_seen + 1);
        trans_seen = trans_count;
```

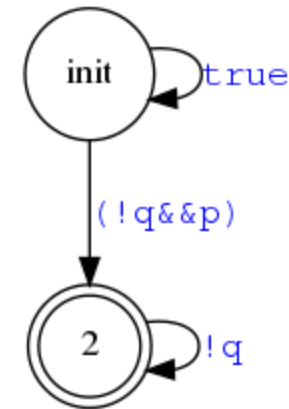
nondeterminism

 only interleave whole block
 reject unsafe interleavings

Example monitor thread

```
switch(state) {  
case T0_init:  
    if(choice == 0) {  
        assume((1));  
        state = T0_init;  
    } else if (choice == 1) {  
        assume(!cexpr_1 && cexpr_0);  
        state = accept_S2;  
    } else assume(0);  
    break;  
case accept_S2:  
    if(choice == 0) {  
        assume(!cexpr_1);  
        state = accept_S2;  
    } else assume(0);  
    break;  
}  
atomic_end();  
}  
}
```

automata transitions
representing the
formula $!(p \rightarrow Fq)$



Infinite traces and BMC?

BMC forces program execution to eventually end
– but BA are defined over infinite traces...

Solution:

- follow SPINs stuttering acceptance approach:
pretend final state extends infinitely
- re-run monitor thread after program termination,
with enough loop iterations to pass through each state twice
- if an accepting state is visited at least twice while stuttering,
BA accepts extended trace
 - LTL property violation found

Experiments

- checked properties of medical device firmware
- mostly of the form $p \rightarrow Fq$ or $(!p \ \&\& \ Fp) \rightarrow Fq$
- tested against original code base,
and code with seeded errors
- all properties shown to hold on original code,
all seeded errors were found

Test name	Interleavings	Elapsed time(s)
start_btn	7764	199
up_btn	3775	83

approach requires large context switch bounds

serial_rx	5454	324
-----------	------	-----

unwind bound:1, context bound: **40**

State Hashing

- used to counter the state explosion problem in explicit-state model checking:
 - variable assignments concatenated into state vector
 - hash values used to record which states have been explored
 - hash collisions prevent unique parts of the state space from being explored
- cannot be applied directly to symbolic model checking:
variable assignments can contain non-deterministic values with constraints

Symbolic State Hashing

Exploit SSA form:

- normalize RHS of each assignment in SSA form
 - compute hash value and associate with LHS variable
 - replace variable occurrences in RHS by variable hashes
 - ... and re-hash
- ⇒ variables with same set of constraints hash to same values
- ⇒ independent of non-deterministic choices
- variable hashes and thread program counters concatenated into state vector
 - rest as before...
 - hash algorithm not important, we use SHA256

Symbolic State Hashing – Limitations

- Equivalent states can have different hash values if:
 - constraints are arranged in different orders
 - (semantically) different sets of constraints
- ⇒ not all redundant states are removed
- However, we are primarily interested in reducing symmetry

State Hashing Experiments

- same experiments, with state hashing enabled
- all tests decreased total runtime
- observable increase in amount of runtime per interleaving

Test name	Interleavings	w / hashing	Elapsed time(s)	w / hashing
start_btn	7764	2245	199	71
up_btn	3775	1385	83	37
keyb_start	92795	49017	9796	4489
baud_conf	485	419	17	16
serial_rx	5454	3108	324	212

Relation to partial order reductions

- Partial order reductions are the more common way to reduce number of redundant states explored
 - demonstrably optimal method of doing this exists...
 - ... but incurs additional complexity in detecting which context switches are redundant
- state hashing only eliminates the most obvious and immediate duplicate states...
- ... but only at the cost of extra overhead in symbolic execution
- detailed comparison remains future work

Conclusions

- BMC framework can be extended to check ANSI-C software against an LTL formula (with reasonable efficiency)
- State hashing can be extended to symbolic model checking
- Runtime performance is improved by a modest amount by the use of state hashing

Future Work

- Full comparison of state hashing with POR
- Evaluate how effective such optimisations are when run on a distributed system