Noname manuscript No.
(will be inserted by the editor)

Model Checking LTL Properties over C Programs with Bounded Traces

Jeremy Morse[1], Lucas Cordeiro[2], Denis Nicole[1], Bernd Fischer[1,3]

[1] Electronics and Computer Science, University of Southampton, UK
{jcmm106,dan,b.fischer}@ecs.soton.ac.uk
[2] Electronic and Information Research Center, Federal University of Amazonas, Brazil
lucascordeiro@ufam.edu.br
[3] Division of Computer Science, Stellenbosch University, South Africa

The date of receipt and acceptance will be inserted by the editor

Abstract Context-bounded model checking has been used successfully to verify safety properties in multi-threaded systems automatically, even if they are implemented in low-level programming languages such as C. In this paper, we describe and experiment with an approach to extend context-bounded software model checking to safety and liveness properties expressed in linear-time temporal logic (LTL). Our approach checks the actual C program, rather than an extracted abstract model. It converts the LTL formulas into Büchi automata (BA) for the corresponding never claims and then further into C monitor threads, which are interleaved with the execution of the program under analysis. This combined system is then checked using the ESBMC model checker. We use an extended, four-valued LTL semantics to handle the finite traces that bounded model checking explores; we thus check the combined system several times with different acceptance criteria to derive the correct truth value. In order to mitigate the state space explosion, we use a dedicated scheduler that selects the monitor thread only after updates to global variables occurring in the LTL formula. We demonstrate our approach on the analysis of the sequential firmware of a medical device and a small multi-threaded control application.

1 Introduction

Model checking has been used successfully to verify actual software (as opposed to abstract system designs) [3,9,11,12,46], including multi-threaded applications written in low-level languages such as C [15,31,40]. 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 deadlocks [15].

Many important requirements on the software behaviour can, however, be expressed more naturally as liveness properties in a temporal logic [18], 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 even on finite program executions; it has typically been necessary to add additional executable code to the program under analysis to retain the past state information. This amounts to the *ad hoc* introduction of a hand-coded state machine capturing (past-time) temporal formulas.

Here, we instead use context-bounded model checking to validate multi-threaded C programs directly against (future-time) temporal formulas over expressions in the global variables of the C program under test. Thus, continuing the previous example, 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 with the LTL formula $\mathsf{G}(\{\texttt{pressed}\} \rightarrow \mathsf{F}\,\{\texttt{charge>min}\})$.[1] In principle, we follow the usual approach [13,24] to check these formulas; we convert the negated LTL formula (the so-called *never claim* [23]) into a Büchi automaton (BA) which is composed with the program under analysis. If the composed system admits an accepting run, the program violates the specified requirement. Our approach differs, however, in two key aspects. First, we check the actual C program, rather than an extracted and abstracted model. We thus convert the LTL formula's BA further into a separate C *monitor thread* and check the interleavings between this monitor and the program using ESBMC [15], our context-bounded model checker for C. Second, we extend the truth values of the LTL expressions to a four-valued lattice describing the least truth values over various possible future behaviours of a C program with possibly infinite state space. In particular, we consider the explored traces to be finite prefixes of infinite traces and our four-valued logic describes the accepting behaviour of the BA for different infinite extensions of the explored finite traces. In practice, the resulting never claim BA obtained from commonly used specifications is rather small. The small size allows us to analyse which states are accepting under the different infinite extensions of the finite traces. We then check the combined system several times, with different assertions corresponding to the different acceptance criteria, to derive the correct truth value for the LTL formula. The program's overall "correctness" value in the lattice is the weakest truth value for which the model checker can find a witness that violates the corresponding assertion. This

[1] Here and throughout the paper we enclose the embedded C expressions in curly brackets and typeset them in typewriter font.

gives us a method to analyse both safety and liveness within the framework of bounded software model checking.

Our approach avoids the inherent imprecision from abstracting the C program into a BA, but the monitor has to capture transient behaviour internal to the program under analysis. The monitor and the program communicate via auxiliary variables reporting the truth values of the LTL formula's inner subexpression. Our tool automatically inserts these variables on-the-fly, maintains them, and also uses them to guide ESBMC's thread exploration. In order to support this addition efficiently, we have extended ESBMC's scheduler so that the monitor thread is scheduled only after updates to global variables.

Our paper makes two main contributions, one theoretical, one practical. On the theoretical side, it describes techniques that allow a bounded model checker to give meaningful information about liveness properties of potentially non-terminating programs. On the practical side, it describes the first mechanism, to the best of our knowledge, to verify LTL properties against an unmodified C code base, which can include multi-threaded code using the standard pthreads library [26].

Organisation. This article is a substantially revised and extended version of our SEFM 2011 contribution [37]. The major differences are that: (*i*) we now use a four-valued LTL semantics to make judgements based on the finite traces that bounded model checking explores and check the system several times with different BA acceptance criteria to derive the correct truth value; (*ii*) we now handle multi-threaded code; (*iii*) we implemented a dedicated scheduler which speeds up the analysis dramatically; and (*iv*) we extended our evaluation with new examples. The remainder of the paper is organised as follows: in the next two sections we give the necessary background, first on the ESBMC context-bounded model checker (Section 2), and then on LTL (Section 3). In Section 4, we then describe our approach to characterizing the runs of a BA according to our four-valued semantics. In Section 5 we demonstrate by examples how this approach can be used to handle different classes of LTL properties. We then describe in more detail the implementation (Section 6) and two case studies (Section 7), before we finally discuss related work and conclude.

2 Bounded Model Checking with ESBMC

ESBMC is a context-bounded symbolic model checker for C software, which allows the verification of single- and multi-threaded programs with shared variables and locks [15,17]. ESBMC can verify programs that make use of bit-operations, arrays, pointers, structs, unions, memory allocation and some floating-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-defined assertions. The latter can be specified at arbitrary program locations using the usual C `assert`-statements.

In ESBMC, the program to be analysed 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 counters of all threads and the values of all program variables. An initial state s_0 assigns the initial program location of the CFG to the program counter of the main thread. 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 counters and the program variables.

Given the transition system M, a proposition ϕ, a context switch bound C and a bound k, ESBMC builds a reachability tree (RT) that represents the program unfolding for C, k and ϕ. It then derives a verification condition ψ_k^π for each given interleaving of statements (or computation path) π such that ψ_k^π is satisfiable if and only if ϕ has a counterexample of depth less than or equal to 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 characterises the set of initial states of M and $\gamma(s_j, s_{j+1})$ is the transition relation of M between steps j and $j+1$, as above. 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 counterexample or witness. A counterexample for a property ϕ is a sequence of states $s_0, s_1, \ldots, s_k$ with $s_0 \in S_0$, $s_{i+1} \in S$, 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 $\psi_k = \bigvee_\pi \psi_k^\pi$ 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 several 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 LTL over Infinite and Finite Traces

3.1 Linear-time Temporal Logic

LTL is a specification logic commonly used in model checking [10,25,28], which extends propositional logic by including temporal operators.

Definition 1 *LTL formulas are defined over primitive propositions, logical operators and temporal operators as follows:*

$$\begin{aligned}\varphi, \psi ::= {} & \mathit{true} \mid \mathit{false} \mid p \mid \neg\varphi \mid \varphi \vee \psi \\ & \mid \mathsf{X}\varphi \mid \mathsf{F}\varphi \mid \mathsf{G}\varphi \mid \varphi\,\mathsf{U}\,\psi \mid \varphi\,\mathsf{R}\,\psi\end{aligned}$$

Here, p is a C expression over the global variables of the program under analysis; p must not have side effects. In ESBMC's LTL notation, these expressions must be enclosed in curly brackets and are treated as truth values according to C semantics. We define the remaining logical operators in the usual way. We use the mathematical notation for LTL formulas, and C style notation inside the embedded C expressions. The temporal operators are "in the next state" or *next* (X), "in some future state" or *eventually* (F), "in all future states" or *globally* (G), *until* (U), and *release* (R). $\varphi\,\mathsf{U}\,\psi$ means that φ must hold continuously until ψ holds; ψ must eventually become true. $\varphi\,\mathsf{R}\,\psi$ means that ψ must hold now and continue to hold either until φ becomes true as well, or forever (if φ never becomes true). All temporal operators can be defined in terms of X and U [36], but we use the full set of operators here.

In the standard semantics [39], LTL formulas are interpreted over *traces* over a given alphabet Σ of symbols, i.e., possibly infinite words $a_0 a_1 \cdots$, with $a_i \in \Sigma$. In LTL model checking, it is common to consider a non-empty set of atomic or primitive propositions *Prop* and to define $\Sigma = 2^{Prop}$. Each symbol $a \in \Sigma$ denotes a valuation, the set of Boolean expressions over the global variables of the C program that hold at a given time; it can be seen as a possible world in a Kripke structure. We use $u \in \Sigma^*$ to denote finite traces, $w \in \Sigma^\omega$ to denote infinite traces, and ϵ to denote the empty trace. We further use $w^i = w_i w_{i+1} \ldots$ to denote the suffix of an infinite trace; for a finite trace of length n, $u^i = u_i u_{i+1} \cdots u_{n-1}$ if $i < n$ and ϵ otherwise. We finally use the notation a^ω to denote the infinite trace consisting of the letter $a \in \Sigma$ only.

We follow the exposition by Bauer et al. [6] and use finite deMorgan lattices as truth domains. A *deMorgan lattice* is a distributive lattice $(\mathcal{L}, \sqcup, \sqcap, \top, \bot)$ where every element $x \in \mathcal{L}$ has a *dual* element $\overline{x} \in \mathcal{L}$ such that $\overline{\overline{x}} = x$ and $x \sqsubseteq y$ implies $\overline{y} \sqsubseteq \overline{x}$; here, $\sqsubseteq$ is the partial order induced by the lattice structure. Note that not every deMorgan lattice is a Boolean lattice, because duals are not proper complements (i.e., $x \sqcap \overline{x} = \bot$ is not necessarily true), but the converse holds, and in particular the Boolean lattice over the standard two-valued truth domain $\mathbb{B}_2 = \{\bot, \top\}$ is a deMorgan lattice with $\bot \sqsubseteq \top$.

Propositional constants.

$$[w \models \mathsf{true}]_\omega = \top \qquad [w \models \mathsf{false}]_\omega = \bot \qquad [w \models p]_\omega = \begin{cases} \top \text{ iff } p \in w_0 \\ \bot \text{ iff } p \notin w_0 \end{cases}$$

Propositional operators.

$$[w \models \varphi \vee \psi]_\omega = [w \models \varphi]_\omega \sqcup [w \models \psi]_\omega \qquad [w \models \neg\varphi]_\omega = \overline{[w \models \varphi]_\omega}$$

Temporal operators.

$$[w \models \mathsf{X}\varphi]_\omega = [w^1 \models \varphi]_\omega$$

$$[w \models \mathsf{F}\varphi]_\omega = \begin{cases} \top \text{ iff } [w^i \models \varphi]_\omega = \top \text{ for some } i \geq 0 \\ \bot \text{ otherwise} \end{cases}$$

$$[w \models \mathsf{G}\varphi]_\omega = \begin{cases} \top \text{ iff } [w^i \models \varphi]_\omega = \top \text{ for all } i \geq 0 \\ \bot \text{ otherwise} \end{cases}$$

$$[w \models \varphi \mathsf{U} \psi]_\omega = \begin{cases} \top \text{ iff } [w^i \models \psi]_\omega = \top \text{ for some } i \geq 0 \\ \qquad \text{and } [w^j \models \varphi]_\omega = \top \text{ for all } 0 \leq j < i \\ \bot \text{ otherwise} \end{cases}$$

$$[w \models \varphi \mathsf{R} \psi]_\omega = \begin{cases} \top \text{ iff } [w^i \models \psi]_\omega = \top \text{ for all } i \geq 0 \\ \qquad \text{or } [w^i \models \varphi]_\omega = \top \text{ for some } i \geq 0 \\ \qquad\quad \text{and } [w^j \models \psi]_\omega = \top \text{ for all } 0 \leq j \leq i \\ \bot \text{ otherwise} \end{cases}$$

Fig. 1 Standard LTL semantics over infinite traces.

We can then define the standard semantics of LTL formulas via the interpretation function $[\_ \models \_]_\omega : \Sigma^\omega \times LTL \to \mathbb{B}_2$, as shown in Figure 1 [6]. We call $w \in \Sigma^\omega$ a *model* of φ iff $[w \models \varphi]_\omega = \top$ and also say that w satisfies φ, or that φ holds for w. For each LTL formula the set of all its models is an ω-regular language that is accepted by a corresponding Büchi automaton [44, 45].

We interpret a possibly multi-threaded C program P as a Kripke structure whose state transitions are derived from the possibly interleaved execution sequence of C statements and whose valuations are given by the possible values of the program's global variables; in the current configuration we consider interleavings only at statement boundaries and assume sequential consistency [32], but options to ESBMC allow us also to use a finer-grained analysis. P can be non-deterministic, so the transition relation can branch even for single-threaded programs. As C's semantics gives a defined (zero) value to all global variables not initialised explicitly at their declaration, all valuations are completely defined in each possible world, including the initial world. This also gives us a well-defined interpretation of the next operator: $\mathsf{X}\varphi$ holds for P if φ holds after the next update of a global variable used in the LTL C-expressions. In many situations this interpretation of X is not directly useful in assessing program correctness; it is often appropriate to write X-free *stutter-invariant* formulas, following

P_1:
```
int s=0;
while(true){
  s=1-s;
};
```

P_2:
```
int s=0;
while(true){
  s=1;
  s=0;
};
```

P_3:
```
int s=0;
s=1;
while(true){
  s=0;
  s=1;
};
```

Fig. 2 Programs with identical infinite traces but different behaviour on finite unwindings for $\gamma \equiv \mathsf{G}(\{\texttt{s=0}\} \rightarrow \mathsf{F}\{\texttt{s=1}\})$.

Lamport [33]. We identify a C program P with the set of all traces $\mathcal{T}(P)$ that correspond to this Kripke structure, and say that an LTL formula φ holds for P if φ holds for all $w \in \mathcal{T}(P)$.

Note that 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.

3.2 LTL over Finite Traces

The standard LTL semantics is defined over infinite traces, but as we are using a *bounded* model checker to analyse the program, we explore only finite traces. Like other bounded model checkers [11], ESBMC bounds the program executions by limiting the number of times a loop is unrolled, rather than limiting the length of traces.[2] This guarantees that loop invariants are respected over the traces. If the program contains at most one potentially unbounded loop then the finite traces explored by ESBMC are proper prefixes of the potentially infinite traces of the original program. If the program contains several potentially unbounded loops then we can still analyse it, using the `--partial-loops` option. In this case, however, the observed finite traces are not necessarily proper prefixes of the original program traces, and our approach can produce false results, as the symbolic execution can continue past unsatisfied loop termination conditions.

We use the notation $P_{|k}$ to denote the k-fold loop unwinding of the program P. Consider for example the three programs shown in Figure 2 and the request-response formula $\gamma \equiv \mathsf{G}(\{\texttt{s==0}\} \rightarrow \mathsf{F}\{\texttt{s==1}\})$. Since all three programs alternate infinitely often between `s==0` and `s==1`, the single infinite trace produced by each program satisfies γ under the standard LTL semantics. The situation, however, looks different for the traces produced by finite unwindings (with increasing loop bounds) of the program loops, as the loop structure of the program determines the lengths of the finite

[2] Unstructured `goto` C code is also handled; every execution of a backward jump counts as a "loop iteration" associated with that `goto`.

prefixes which are considered. P_1 ends with `s==1` (i.e., responds to the request) if we unwind the loop an odd number of times, and with `s==0` (i.e., a pending request) otherwise, P_2 always ends with `s==0`, while P_3 always ends with `s==1`. P_3 is thus intuitively better behaved than both P_2 (which is consistently wrong) and P_1 (which behaves erratically). The standard (infinite trace) LTL semantics does not distinguish between the programs.

There is a fundamental problem with applying LTL to finite traces. It is caused by extending the standard interpretation of X as a strong (or existential) next operator [29] to finite traces, which requires the existence of a next state to hold. This is counter-intuitive for finite traces, since $\mathsf{X}\,\mathsf{true}$ is now no longer a tautology, as $\models_F$ (i.e., the standard interpretation applied to finite traces) gives us for all formulas φ, $[u \models \mathsf{X}\varphi]_F = \bot$ if $u^1 = \epsilon$ [6].

Several approaches tweak the syntax or semantics of LTL to remedy this situation. Since G and F can be defined relatively straightforwardly on finite traces, Giannakopolou and Havelund [21] suggested removing X and work with an X-free subset of LTL. The syntax can instead be extended by adding an additional weak (or universal) next operator $\overline{\mathsf{X}}$ [35], which complements the strong next and holds if there is no next state: $[u \models \overline{\mathsf{X}}\varphi]_F = \top$ if $u^1 = \epsilon$. Hence, $\overline{\mathsf{X}}\,\mathsf{true}$ *is* a tautology. This also gives unwinding laws for F and G, namely $\mathsf{F}\varphi \equiv \varphi \vee \mathsf{X}\mathsf{F}\varphi$ and $\mathsf{G}\varphi \equiv \varphi \wedge \overline{\mathsf{X}}\mathsf{G}\varphi$. Alternatively, the distinction between strong and weak next can be encoded into the semantics rather than the syntax, via two different semantic functions which coincide on the temporal and most Boolean operators, but differ on negation (which flips between both functions) and the atomic propositions, where they reflect the behaviours of strong and weak next, respectively [19]. Finally, the finite traces can be systematically extended, e.g., by infinite stuttering of their last state [33], to allow the use of standard semantics, i.e., defining $[u \models \varphi]_\infty = [uu_{n-1}^\omega \models \varphi]_\omega$ for a finite trace u of length n. This is also called the *infinite extension* semantics [4], and we say that a BA corresponding to φ *stutter-accepts* u if $[u \models \varphi]_\infty = \top$.

Under the infinite extension semantics, γ (see Figure 2 again) now holds for all unwindings of P_3, but not for the unwindings of P_2 or P_1. However, in a two valued logic, we cannot distinguish between a formula that (truly) holds because we have seen a *good prefix* [30] and so *all* possible continuations of the observed finite trace will be models as well, and one that (presumably) holds because we have not yet seen a *bad prefix* (i.e., a finite trace that *cannot* be prefix of a model) or because it holds if we stutter the final state infinitely often. In order to realize this distinction, we use a larger truth domain. Bauer et al. [5–7] have proposed and analysed two different domains, $\mathbb{B}_3 = \{\bot, ?, \top\}$, with $\bot \sqsubseteq ? \sqsubseteq \top$, $\overline{\bot} = \top$, and $\overline{?} = ?$, and $\mathbb{B}_4 = \{\bot, \bot^p, \top^p, \top\}$, with $\bot \sqsubseteq \bot^p \sqsubseteq \top^p \sqsubseteq \top$, $\overline{\bot} = \top$, and $\overline{\bot^p} = \top^p$. Under $\models_3$, finite traces are mapped to $\top$ (resp. $\bot$) iff they are good (resp. bad) prefixes; all other finite traces are considered "ugly" and are mapped to the inconclusive truth value ? [5,7]. In $\mathbb{B}_4$, ? is refined into the two truth values $\bot^p$ ("presumably false") and $\top^p$ ("presumably true"). The interpretation function $\models_4$ then uses the finite trace semantics with weak next to distin-

guish between the two cases (i.e., $[u \models \varphi]_4 = \bot^p$ iff u is an ugly prefix and $[u \models \varphi]_F = \bot$, and similarly for $\top^p$) [6].

Our analysis here is based on $\mathbb{B}_4$ as well, but we use a different interpretation function from Bauer et al. [6]. In particular, we use the infinite extension semantics to resolve ugly prefixes into presumably good or presumably bad. The advantage of this approach is that we do not need to resort to the weak-next operator and can define the finite trace semantics in terms of the standard semantics only. The use of stutter extension in this way is naturally compatible with the stutter-invariant semantics introduced by Lamport [33] for computer programs but does not require it.

Definition 2 *The* bounded trace semantics *of LTL formulas is given by*

$$[u \models \varphi]_B = \begin{cases} \top & \text{iff } \forall w \in \Sigma^\omega \cdot [uw \models \varphi]_\omega = \top \\ \top^p & \text{iff } [uu_{n-1}^\omega \models \varphi]_\omega = \top \wedge \exists w \in \Sigma^\omega \cdot [uw \models \varphi]_\omega = \bot \\ \bot^p & \text{iff } [uu_{n-1}^\omega \models \varphi]_\omega = \bot \wedge \exists w \in \Sigma^\omega \cdot [uw \models \varphi]_\omega = \top \\ \bot & \text{iff } \forall w \in \Sigma^\omega \cdot [uw \models \varphi]_\omega = \bot \end{cases}$$

for a finite trace $u \in \Sigma^*$ of length $n > 0$ and an LTL formula φ.

In our case, all program traces are guaranteed to be non-empty, because all global variables have defined initial values, which then form the initial state. We extend the interpretation to sets of traces by taking the meet over all elements, i.e., $[U \models \varphi]_B = \bigsqcap_{u \in U} [u \models \varphi]_B$. We say that φ *holds* (resp. *presumably holds*) *for a C program* P if $[\mathcal{T}(P) \models \varphi]_B = \top$ (resp. $\top^p$). We finally say φ *holds* (resp. *presumably holds*) if $[\Sigma^\omega \models \varphi]_B = \top$ (resp. $\top^p$) and define the notion of *failing* resp. *presumably failing* correspondingly.

The bounded trace semantics is an extension of our earlier work [37] where we only used the infinite stutter semantics. Consequently, we were effectively working only with the inconclusive truth values $\bot^p$ and $\top^p$, while we add the definitive truth values $\bot$ and $\top$ here.

3.3 LTL Model Checking vs. LTL Runtime Verification

Finite LTL semantics similar to the bounded trace semantics we are using here have been developed largely for run-time monitoring and verification purposes [34], and due to the focus on finite traces, our approach has some similarities with run-time verification, but one key difference remains. Runtime verification only considers actually observed behaviours, one at a time, while we analyse all possible behaviours at the same time. This difference becomes prominent with non-determinism, even for single-threaded programs. Consider for example the program Q

```
int p=0, q=0; p=1; if(*){p=0}; if(*){q=1};
```

where "`*`"denotes a non-deterministic choice and `p` and `q` are zero-initialised global variables. Q can produce four finite traces, depending on the particular non-deterministic choices:

(*i*) [{p==0}∧{q==0}, {p==1}∧{q==0}],
(*ii*) [{p==0}∧{q==0}, {p==1}∧{q==0}, {p==1}∧{q==1}],
(*iii*) [{p==0}∧{q==0}, {p==1}∧{q==0}, {p==0}∧{q==0}], and
(*iv*) [{p==0}∧{q==0}, {p==1}∧{q==0}, {p==0}∧{q==0}, {p==0}∧{q==1}].

Now consider the LTL formula $\psi \equiv \mathsf{X}(\{\texttt{p==1}\}\,\mathsf{U}\,\{\texttt{q==1}\})$. Clearly, ψ does not hold for the traces (*iii*) and (*iv*), and over these, $\models_3$, $\models_4$, and $\models_B$ all map ψ to $\bot$. However, in run-time verification, there is no guarantee that we ever observe these traces, so the assurance we gain from its results is limited. Our approach, however, will work out that $[\mathcal{T}(Q) \models \psi]_B = \bot$ and hence Q can fail ψ. Moreover, if we consider Q' to be the variant of Q where `q` is initialised with one, we find $[\mathcal{T}(Q') \models \psi]_B = \top$ as well. Finally, if we change Q to Q''

```
int p=0, q=0; p=1; if(*){q=1};
```

then (*iii*) and (*iv*) become impossible, and our approach will calculate $[\mathcal{T}(Q'') \models \psi]_B = \bot^p$, meaning that no finite trace produced by Q'' is a definitive counter-example but, on stuttering, ψ does not hold for all traces.

4 Characterizing Program Behaviours Using $\mathbb{B}_4$

Definition 2 characterises the truth value in $\mathbb{B}_4$ of an LTL formula φ with respect to a single finite trace u. In this section we now show how we can use the Büchi automaton for the never claim to effectively calculate the truth value of the formula with respect to the finite traces of a program P. In Section 4.1, we briefly recall the basic notions of Büchi automata. In Section 4.2 we characterise the relationship between truth values in $\mathbb{B}_4$ and validity of never claims over $\mathbb{B}_2$, while we describe the high-level structure of our algorithm in Section 4.3.

4.1 Büchi Automata

Büchi automata (BA) are finite-state automata over infinite words first described by Büchi [8]. We follow Holzmann's presentation [24] 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, $T\colon S \times L \to 2^S$ a state transition function and $F \subseteq S$ a set of accepting states. A *run* is a sequence of state transitions taken by B as it operates over some input. A run is *accepted* if B can pass through an accepting state $s \in F$ infinitely often along the run. B may be deterministic or non-deterministic but in the following, we will consider only non-deterministic BAs, since deterministic BAs need a more complicated acceptance condition in order to model LTL. A BA is in *reduced form* [1] if it has no rejecting traps, i.e., if the BA has a possible next state, then there is some extension of the trace that is accepted.

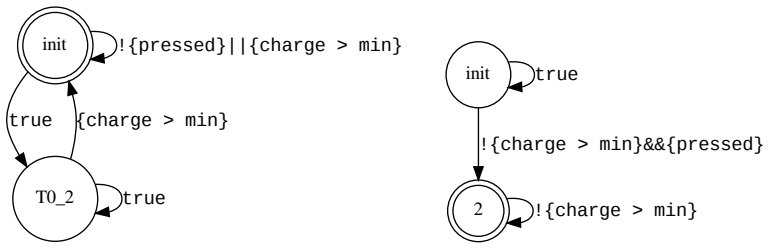

Fig. 3 The left BA accepts the example from the introduction, G({pressed} → F{charge > min}). The right BA is its negation, used for the *never claim* in our monitor.

A number of algorithms exist for converting an LTL formula to a BA accepting a program trace [20,22,42]. We use the `ltl2ba` [20] algorithm and tool. Figure 3 illustrates the BA produced from the example LTL formula in the introduction. Its labels (i.e., input symbols) are propositions composed from the primitive C-expressions used in the LTL formula.

4.2 Truth Values in $\mathbb{B}_4$ and Standard Validity of Never Claims

As noted above, Definition 2 characterises the truth value in $\mathbb{B}_4$ of an LTL formula φ with respect to a single finite trace u. However, for model checking φ over a program P this is not yet suitable. First, we need to express the truth value in $\mathbb{B}_4$ in terms of the validity of the never claim under the two-valued standard semantics. This allows us to use the BA for the never claim directly, and avoids the need to define an explicit acceptance criterion for the four-valued logics. The following lemma addresses this problem. Note that we do not need a complete characterisation of all truth values in $\mathbb{B}_4$.

Lemma 1

(*i*) $[u \models \varphi]_B = \top$ iff $\nexists w \in \Sigma^\omega \cdot [uw \models \neg\varphi]_\omega = \top$

(*ii*) $[u \models \varphi]_B \sqsupseteq \top^p$ iff $[uu_{n-1}^\omega \models \neg\varphi]_\omega = \bot$

(*iii*) $[u \models \varphi]_B = \bot$ iff $\forall w \in \Sigma^\omega \cdot [uw \models \neg\varphi]_\omega = \top$

Proof (*i*) Since the standard semantics $\models_\omega$ (cf. Figure 1) is defined over $\mathbb{B}_2$, $\nexists w \in \Sigma^\omega \cdot [uw \models \neg\varphi]_\omega = \top$ is equivalent to $\forall w \in \Sigma^\omega \cdot [uw \models \neg\varphi]_\omega = \bot$, and thus to $\forall w \in \Sigma^\omega \cdot [uw \models \varphi]_\omega = \top$, which gives us the claim.
(*ii*) Similarly, $[uu_{n-1}^\omega \models \neg\varphi]_\omega = \bot$ is equivalent to $[uu_{n-1}^\omega \models \varphi]_\omega = \top$, which holds if and only if $[u \models \varphi]_B = \top$ or $[u \models \varphi]_B = \top^p$.
(*iii*) This follows directly from the definitions of $\models_\omega$ and $\models_B$.

Second, the program P may be non-deterministic and produce more than one trace. We thus need to consider the minimum truth value attained over all of its possible traces $\mathcal{T}(P)$. The following lemma addresses this problem.

Lemma 2

(*i*) $[U \models \varphi]_B = \top$ iff $\nexists u \in U, w \in \Sigma^\omega \cdot [uw \models \neg\varphi]_\omega = \top$

(*ii*) $[U \models \varphi]_B \sqsupseteq \top^p$ iff $\nexists u \in U \cdot [uu_{n-1}^\omega \models \neg\varphi]_\omega = \top$

(*iii*) $[U \models \varphi]_B = \bot$ iff $\exists u \in U \cdot \forall w \in \Sigma^\omega \cdot [uw \models \neg\varphi]_\omega = \top$

Proof Recall that $\bigsqcap_{u \in U} [u \models \varphi]_B = [U \models \varphi]_B$. Then:
(*i*) By Lemma 1 $\nexists u \in U, w \in \Sigma^\omega \cdot [uw \models \neg\varphi]_\omega = \top$ is equivalent to $\forall u \in U \cdot [u \models \varphi]_B = \top$; hence, $[U \models \varphi]_B = \top$.
(*ii*) By definition of $\models_\omega$, $\nexists u \in U \cdot [uu_{n-1}^\omega \models \neg\varphi]_\omega = \top$ is equivalent to $\forall u \in U \cdot [uu_{n-1}^\omega \models \varphi]_\omega = \top$, which by defnition of $\models_B$ means that $\forall u \in U \cdot [u \models \varphi]_B \sqsupseteq \top^p$, and thus $[U \models \varphi]_B \sqsupseteq \top^p$.
(*iii*) By the definitions of $\models_\omega$ and $\models_B$ we have that $\exists u \in U \cdot \forall w \in \Sigma^\omega \cdot [uw \models \neg\varphi]_\omega = \top$ is equivalent to $\exists u \in U \cdot [u \models \varphi]_B = \bot$ and thus $[U \models \varphi]_B = \bot$ as well.

4.3 Algorithm Structure

Lemma 2 rephrases the definition of validity in $\mathbb{B}_4$ into a form that is suitable for model checking a program against a standard non-deterministic never claim BA. In particular, in all but the inner clause of the test for $\bot$ the quantifiers are existential and are thus compatible with the existential (i.e., optimistic) search for accepting traces.

In the following we use $\text{BA}_{\neg\varphi}$ to denote the never claim BA for the LTL formula φ. Moreover, we assume that all the accepting traps have been replaced with a single accepting state with a transition on *true* to itself and that $\text{BA}_{\neg\varphi}$ is in reduced form [2]. These assumptions make the application of the tests below straightforward.

$[\mathcal{T}(P) \models \varphi]_B = \top$: As $\text{BA}_{\neg\varphi}$ is in reduced form, it cannot accept the program trace any longer if it has no transition to a next state, and the trace can be pruned. If and only if all traces are pruned, the program evaluates to $\top$. Note that this cannot happen [2] if φ is a (non-trivial) classical safety property [1].

$[\mathcal{T}(P) \models \varphi]_B = \bot$: If $\text{BA}_{\neg\varphi}$ reaches an accepting trap for any trace, φ evaluates to $\bot$ over the program, with the trace returned as a witness. Note that this cannot happen [2] if φ is a classical liveness property [1].

$[\mathcal{T}(P) \models \varphi]_B = \top^p$: If the property does not evaluate to $\top$ or $\bot$, we check its stutter acceptance. A simple reachability analysis of $\text{BA}_{\neg\varphi}$, given the transitions enabled in the final program state, allows us to check for possible stutter acceptance at the end of each symbolically generated set of traces. If no accepting cycle is found, the property evaluates to $\top^p$, with one of the traces returned as a witness.

$[\mathcal{T}(P) \models \varphi]_B = \bot^p$: If $\text{BA}_{\neg\varphi}$ stutter accepts for at least one trace, the property evaluates to $\bot^p$ and the trace is returned as witness.

Note that the different cases are not independent of each other, due to the inequality in Lemma 2 (*ii*). As we are looking for a witness to the worst bounded behaviour that the program can exhibit when we model check, the actual implementation of the algorithm (cf. Section 6) needs to check the cases in a specific order.

4.4 Example

As an example, consider the BA on the right of Figure 3, i.e., the never claim BA for $\mathsf{G}(\{\texttt{pressed}\} \rightarrow \mathsf{F}\{\texttt{charge > min}\})$. This BA is generated by `ltl2ba` and is already optimised, and in particular in reduced form. Hence, it can accept on *some* infinite suffix from any state, and the set of optimistically accepting states is {init, 2}. There is no explicit trap state and thus, as this is an optimised BA, the set of states which will accept for all infinite suffixes is empty. The interesting behaviour of this request-response liveness condition is, as explained further in Section 5.3, restricted to its behaviour on infinite stutter. There are four possible infinite stutter suffixes and their accepting sets are shown in Table 1. Hence, if {`charge > min`} and {`pressed`} are both false in the final program state, the BA stutter accepts only if it is in state 2, and thus the trace is presumably failing only then.

Final symbol	Stutter-accepting states
$\neg\{\texttt{charge > min}\} \wedge \neg\{\texttt{pressed}\}$	{2}
$\neg\{\texttt{charge > min}\} \wedge \{\texttt{pressed}\}$	{init, 2}
$\{\texttt{charge > min}\} \wedge \neg\{\texttt{pressed}\}$	$\emptyset$
$\{\texttt{charge > min}\} \wedge \{\texttt{pressed}\}$	$\emptyset$

Table 1 Final symbol valuations and their corresponding stutter-accepting states.

5 Checking Safety, Co-Safety, and Liveness Properties

5.1 Safety Properties

In an imperative language such as C, it is common to test the validity of safety or invariant properties at various points in the program execution via `assert`-statements. These may be checked during program execution using the standard C library and, in conjunction with a suitable test suite, allow checking a variety of runs of the code as noted in Section 3.3. They are also recognised and checked during symbolic execution by ESBMC which

```
const int count=6;
int i=count;
int j=0;

...
while(i) {
  i--;
  j++;
  assert(i+j==count);
}
```

```
const int count=6;
int i=count;
int j=0;
int looking=1;
  /* visibility to monitor */
...
while(i) {
  looking=0;
  i--;
  j++;
  looking=1;
}
```

Fig. 4 C program with a safety assertion (left) and a monitor variable for a guarded safety property (right).

gives an exhaustive examination of their validity for all (bounded) execution traces. Thus the code fragment on the left of Figure 4 will be verified successfully as the loop invariant `i+j==count` holds whenever the `assert`-statement is executed. It is, however, often more convenient to assert a safety property everywhere *except* within a specific region in which updates are taking place, rather than just at particular locations. This is particularly attractive in languages such as C with limited support for data encapsulation: data that would be considered a private instance field in an object-oriented language is modifiable in C by a library's clients.

The classical safety property $\mathsf{G}\varphi$ states that φ must hold throughout program execution. However, this is of little practical use as it stands, because φ will typically be violated by any changes to its individual variables. Instead we model the permitted region in which the individual variables can be updated using a global flag `looking` which we set to zero during an update, and use a guarded safety property $\mathsf{G}(\{\texttt{looking}\} \rightarrow \{\texttt{i+j==count}\})$. The listing on the right of Figure 4 shows the modified fragment together with the auxiliary code. In this case, the symbolic execution runs to completion and ESBMC returns $\top^p$.

Since it is in principle always possible for a safety property to be violated at some future time, no finite execution will cause the never claim BA to reject a word outright. In our approach, a terminated program generates an infinite trace by stuttering indefinitely on its last symbol; in other words, the global variables cease changing. Thus stutter rejection of the never claim (i.e., $\top^p$) constitutes correctness for any terminating program. It is precisely our knowledge that the program has terminated (i.e., that the ESBMC run has completed without violating any unwind assertions) that confirms the program correct against the specification.

We can instead modify our LTL specification to capture explicitly the termination of the program; this is a natural use for the U operator. We simply add a second auxiliary variable `done` to capture program termination;

this is initialised to zero, and set to one right before the program finishes. We then use the LTL specification $(\{\texttt{looking}\} \rightarrow \{\texttt{i+j==count}\})\mathsf{U}\{\texttt{done}\}$. In this case, ESBMC reports a successful verification (i.e., $\top$) because the never claim BA fails; the invariant holds until `done` becomes true.

Note that, while accurately expressing a safety property over a terminating program, the second LTL expression does not meet the classical definition of a safety property [1] as finite prefixes can guarantee rejection of the never claim.

5.2 Co-Safety Properties

Co-safety properties [7] often reflect convergence or termination conditions. They are the converse of safety properties; they can be demonstrated to be true by some finite trace. Technically, they are a subset of liveness properties [1] as, whatever the initial trace, there is some future extension that can satisfy them. A co-safety property can never evaluate to $\bot$ in $\mathbb{B}_4$.

If we work again from the example shown in Figure 4, then the LTL formula $\mathsf{F}\{\texttt{j==6}\}$ expresses the termination (co-safety) condition that `j` will eventually reach its final value. When the program runs to completion, the condition is satisfied and ESBMC reports successful verification (i.e., returns $\top$). If we artificially restrict the number of loop interactions by setting the ESBMC flag `--unwindset 1:4` to restrict the program loop to four iterations, ESBMC reports "presumably bad" (i.e., returns $\bot^p$). This is typical of a co-safety property; a gradually extended partial trace will continuously report "presumably bad" (as the necessary event has not happened) until it reports successful verification.

5.3 True Liveness Properties

Safety and co-safety properties have natural definitions over both finite and infinite traces, i.e., for terminating and for non-terminating programs. In contrast, *true* liveness properties[3] are generally regarded as well-defined only over infinite words. It is thus a challenge to use a bounded model checker to explore the true liveness properties of a program.

One of the simplest true liveness properties is a request-response formula of the form $\mathsf{G}(\varphi \rightarrow \mathsf{F}\psi)$. The program is always required to respond to the request φ by producing a response ψ. We may examine this behaviour with the simple program

```
unsigned int i=0; int main() { while(1) i++; };
```

and the property $\mathsf{G}(\{\texttt{i\%2==0}\} \rightarrow \mathsf{F}\{\texttt{i\%3==0}\})$. This property has the typical feature of a true liveness property: no finite trace can determine acceptance

[3] The classical definition of liveness properties [1] includes co-safety properties as well. Here we use the term *true liveness property* to exclude co-safety properties.

or rejection. A simple static analysis which searches for rejecting and accepting traps in the never claim BA already shows that this formula will (regardless of the program) never result in a definitive outcome (i.e., $\bot$ or $\top$).

In general, the regular appearance of "presumably true" as we extend the length of the investigated prefix trace is characteristic of good programs under a request-response liveness property, while bad programs may never result in "presumably true". For this program, as we progressively increase the unwind bound from 1 to 12, the program's behaviour oscillates between "presumably true" (i.e., $\top^p$) and "presumably bad" (i.e., $\bot^p$), and ESBMC reports:

$$\top^p, \bot^p, \top^p, \bot^p, \bot^p, \top^p, \top^p, \bot^p, \top^p, \bot^p, \bot^p, \top^p$$

In this particular case, we have bounded the only program loop in such a way that our trace extends by one symbol with each increase in the bound. More general programs can be more difficult to examine; if, for example, we have to bound several loops, the finite traces we observe may not even be valid prefixes of the real program behaviour. Nevertheless, in well-designed programs loop iterations should independently meet request-response liveness conditions and, as we increase the unwind bounds on the various loops we would expect to see regular appearances of $\top^p$.

A variant of the request-response liveness formula is often used as a fairness formula. The formula $\mathsf{GF}\{\mathtt{p}\}$ expresses that the C expression `p` is true infinitely often at all times in the future. Such conditions can, for example, be conjoined into expressions of the form $(\bigwedge_i \mathsf{GF}\,\rho_i) \rightarrow \mathsf{G}(\varphi \rightarrow \mathsf{F}\psi)$ which are easily handled by our tools. Note that such expressions were the original motivation for the development of the compact BAs produced by `ltl2ba` [20].

Some liveness properties are resistant to an analysis with finite traces. "Toggle" properties such as $\mathsf{G}((\varphi \rightarrow \mathsf{F}\neg\varphi) \wedge (\neg\varphi \rightarrow \mathsf{F}\varphi))$ can be seen from our static analysis to have no stutter-accepting prefixes. The static analysis of the never claim BA for this formula shows that it responds with $\bot^p$ to all (non-empty) finite traces. Unfortunately, our tools are of little further use in this case, other than to confirm the impossibility of the task set in front of them. Thus, checking the formula

$$\mathsf{G}((\{\mathtt{i\%2}\} \rightarrow \mathsf{F}\neg\{\mathtt{i\%2}\}) \wedge (\neg\{\mathtt{i\%2}\} \rightarrow \mathsf{F}\{\mathtt{i\%2}\}))$$

over the above program, as we progressively unwind we see

$$\bot^p, \bot^p, \bot^p, \bot^p, \bot^p, \bot^p, \bot^p, \bot^p, \bot^p, \bot^p, \bot^p, \bot^p, \ldots$$

Overall, our simple reachability analysis of the never claim BA generated from the LTL formula allows us to determine, for any LTL expression, which of the four elements of $\mathbb{B}_4$ can be returned, allowing us to estimate infinite or long-time program behaviours from the data returned by ESBMC. We are, therefore, able to distinguish safety, co-safety, "true" liveness and "toggle" liveness properties and thus to guide the expectations of the ESBMC user.

5.4 Restricted Alphabets

Some symbols of the alphabet $\Sigma = 2^{Prop}$ cannot arise during program execution; this can happen if the various propositions are not independent. As an obvious example, consider a formula which includes both {p} and {!p} as primitive C expressions rather than negating in the LTL using {p} and ¬{p}. This causes no problems with the evolution of the BA during program execution, nor with the computation of stutter-acceptance or rejection for $\bot^p$ or $\top^p$. Our system will, however, explore too large a symbol space when analysing for acceptance over all, or over no, future continuations. We might, in such situations, report $\bot_p$ where a more sensitive analysis would report $\bot$. ESBMC can itself be used, if necessary, to confirm the independence of the C expressions.

6 Implementation

6.1 Monitor Threads for Bounded Trace Semantics

In our context, a monitor is some portion of code that inspects the program state and verifies that it satisfies a given property, causing an assertion to fail if this is not the case. A monitor thread is a monitor that is interleaved with the execution of the program under analysis. This allows the monitor 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 [24]. A non-deterministic BA representing the negation of the LTL property 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.

6.2 Checking LTL Properties Against a C Program

We apply the approach described above to a C code base by implementing the BA in C, which is then executed as a monitor thread, interleaved with the execution of the program. This approach involves two technical dimensions: the conversion of the BA to C, and the interaction of the monitor thread with the program under analysis.

6.2.1 Implementing Büchi automata in C. To implement our BA and monitor thread, we take the `ltl2ba` tool and convert its usual Promela automaton output to C. This output is combined with the output of the reachability analysis information as described in Section 4.3 to produce assertions about the automaton state and program state at the end of the execution. The model checker itself only recognises successful or failed verifications of a program, so we report the discovery of a good prefix (i.e., $\top$) as a successful verification, and all other trace classifications as assertion failures with the assertion message identifying which type of trace has been found.

Listing 1 in Appendix A shows the C implementation of a monitor, with the never claim BA in Figure 3 (see page 11) contained in the function `ltl2ba_fsm` (lines 9 to 39).

ESBMC's symbolic execution of the original program then drives the evolution of $\mathrm{BA}_{\neg\varphi}$ through the possible states. However, since the code for $\mathrm{BA}_{\neg\varphi}$ is not actually, but only symbolically executed, we do not model the non-determinism of the BA directly in the C code (e.g., by keeping a set of current states), and can instead represent the current states of the BA as a non-deterministic but properly constrained single integer variable. That is, the C code will transition only from one state to another, not from one subset of states to another. We then rely on the model checker to explore all possible transitions. This makes good use of capabilities of the SMT solver and substantially simplifies the implementation of the monitor. In particular, there is no need to convert the BA into deterministic form, which can lead to an explosion in the number of BA states.

An infinite loop (lines 11–38) encapsulates the state transition code. To model non-deterministic transitions from any particular state, we take a non-deterministic value (line 12) and then attempt all transitions (lines 16 and 19), depending on the non-deterministic value. This allows the model checker to explore all transitions available. Each transition is guarded by an `assume`-statement, which ensures that a transition is only permitted when the current state of program under analysis satisfies the transition's guard.

The test harness generated by our tool calls `ltl2ba_start_monitor` (lines 40–48) when modelling begins and `ltl2ba_finish_monitor` (lines 67–84) when modelling ends in order to identify the start and end of the analysis. Given that we operate in the context of bounded model checking program termination is guaranteed, as any infinite loop is unrolled only to the length of the bound and thread deadlocks which might otherwise prevent termination are separately detected by ESBMC [15].

Once the program terminates, `ltl2ba_finish_monitor` inspects the current automaton state and program state, and determines from the precomputed reachability analysis of the automaton the truth of the particular lattice value being assessed. This precomputed data is held in the arrays `_ltl2ba_stutter_accept_table`, `_ltl2ba_good_prefix_excluded_states`, and `_ltl2ba_bad_prefix_states`. These indicate whether the current symbol and state stutter-accept ($\top^p$), prohibit a $\top$-trace, or indicate a $\bot$-trace, respectively. Multiple runs of ESBMC can be required to determine the

bounded trace interpretation of this (potentially non-deterministic) interleaving as we search for the smallest truth value for which there is a counterexample.

6.2.2 Interacting with the Existing Code Base. LTL formulas allow verification engineers to describe program behaviour using propositions about program states. To describe 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 curly brackets 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 analysis as well as constants and side-effect free operators. The expression must also evaluate to a value that can be interpreted as a truth value under conventional C semantics.

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

$$\mathsf{G}((\{\texttt{press == 4}\} \wedge \{\texttt{mstate == 1}\}) \rightarrow \mathsf{F}\{\texttt{stime > refstime}\})$$

and the following safety property checks a buffer bound condition:

$$\mathsf{G}(\{\texttt{buffer\_size != 0}\} \rightarrow \{\texttt{next < buffer\_size}\})$$

Within the BA (see Listing 1 again) these expressions are required for use in the guards that prevent invalid transitions being explored. We avoid using the expressions directly in the BA; instead ESBMC searches the program under verification for assignments to global variables used in the C expression, then inserts code to update a Boolean variable corresponding to the truth of the expression (lines 2 and 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.

However, this transformation does not handle indirect assignments to variables, i.e., assignments through dereferencing pointers. Neither of our case studies perform such actions—in fact our application domain (embedded software) tends not to feature indirect operations at all, instead preferring to operate on a fixed set of configuration and data variables, due to memory and environment limitations. As a result we have not attempted to extend our approach to handle indirection. If required, it could be implemented by taking all indirect assignments, comparing the pointer being dereferenced to the addresses of variables appearing in the C expression, and updating the relevant Boolean variables if the comparison is true.

6.2.3 Synchronous Interleaving. In our previous work [37] we composed the monitor thread with the program under analysis in the same manner as we would any other thread, with the scheduler giving no special treatment to the monitor. This approach had the benefit of requiring few modifications

to the model checker, but at the expense of performance, with many thread interleavings produced by the scheduler being discarded as they provided the monitor with an inconsistent view of the program state. In turn, this effect resulted in long verification times, even on small programs with no intrinsic use of threads.

We have therefore changed our approach to perform a deterministic and directed interleaving of the monitor with the program under analysis. Code inserted after global variable updates now calls a model checker intrinsic that causes it to context switch to the monitor thread, then context switch back once the monitor has run the BA a single step; the monitor itself no longer behaves as a schedulable thread. This technique effectively inlines the running of the BA at every point of interest. It also ensures that verification of single-threaded programs does not suffer from a multi-threaded state explosion.

7 Case Studies

We have tested the approach described in this paper against a set of behavioural properties of a pulse oximeter firmware and a bicycle monitoring computer. The first application is an embedded firmware that we treat as single-threaded, whereas the second application is a multi-threaded model of a data collection computer for cyclists. All tests were run on an otherwise idle Linux workstation[4] using ESBMC version 1.20[5] and Microsoft Z3 version 2.19, with a time limit of one hour to execute.

7.1 Pulse Oximeter

The pulse oximeter is a medical device responsible for measuring oxygen saturation (SpO_2) and heart rate (HR) in the blood system using a non-invasive method [14]. The firmware of the pulse oximeter is composed of device drivers (i.e., display, keyboard, serial, sensor, and timer), a system log component that allows the developer to debug the code through data stored in RAM, 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 C code and 80 functions.

Here we report the results of verifying the pulse oximeter code against six properties selected from a previous SMV model of the software [16], as shown in Table 2. Note that all six properties hold for the code.

The first four properties are liveness properties of the general form $\mathsf{G}(\varphi \rightarrow \mathsf{F}\psi)$, so that whenever an enabling condition φ has become true, then eventually the property ψ is required to become true as well. The *up_btn* formula checks that when the up button is pressed `(press == 4)`

[4] 2.67Ghz Intel Xeon, 12Gb of memory, running Fedora 16

[5] Available from `www.esbmc.org`

Name	Property
baud_conf	$\mathsf{G}(\{\texttt{brate == 1200}\} \rightarrow \mathsf{F}\{\texttt{TH1 == 0xE8}\})$
keyb_start	$\mathsf{G}(\{\texttt{the\_key == 1}\} \rightarrow \mathsf{F}\{\texttt{command == 1}\})$
serial_rx	$\mathsf{G}((\{\texttt{p\_inDat == 1}\} \vee \{\texttt{flag2 == 1}\}) \rightarrow \mathsf{F}\{\texttt{flag1 == 1}\})$
up_btn	$\mathsf{G}((\{\texttt{press == 4}\} \wedge \{\texttt{mstate == 1}\}) \rightarrow \mathsf{F}\{\texttt{stime > refstime}\})$
start_btn	$\mathsf{G}((\neg\{\texttt{press == 1}\} \wedge \mathsf{F}\{\texttt{press == 1}\}) \rightarrow \mathsf{F}\{\texttt{q\_startCall}\})$
buflim	$\mathsf{G}(\{\texttt{buffer\_size != 0}\} \rightarrow \{\texttt{next < buffer\_size}\})$

Table 2 Properties for verification of pulse oximeter firmware.

and the device is in a particular state `(mstate == 1)`, then eventually an internal counter `stime` becomes larger than its previous value (kept in the variable `refstime` inside the test harness). The formula *start_btn* checks intuitively that whenever there is a transition of `press` to one from any other value then `q_startCall` will also become true now or in the future. Note however that we are not checking for a strict correspondence beetween changes in `press` and the occurences of `q_startCall` becoming true so that for example the former can happen several times before the latter happens. Finally *buflim* is a safety property that ensures a ring-buffer output index does not exceed the allowed limits. This check is similar to buffer overflow checks already supported by ESBMC.

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 different loop unwind bounds. 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.

The results are summarised in Table 3. Here, the *loc* column contains the line count of the source file for the portion of firmware being tested and k the loop unwinding bound specified for the test. The columns t and Result record the elapsed time in seconds that the test took to run and the outcome ESBMC reported for the test. A result of "TO" indicates the test did not complete in the allowed time, and "MO" indicates that ESBMC exhausted the available memory.

We first observe that ESBMC determines the expected result for most test cases. Since the first five properties are liveness properties, ESBMC reports the inconclusive results $\top^p$ and $\bot^p$ instead of the definitive versions. We also observe that the amount of time taken scales roughly linearly with the unwind bound given in most tests. A notable exception is the *buflim* test, which increases dramatically in time and memory requirements. This performance hit is caused by a large amount of program non-determinism in the portion of code being LTL checked, making checking higher unwind bounds unfeasible.

Finally, we observe that the *up_btn* property has incorrect results for a number of cases. Here, the seeded error combines a number of (in this case, three) consecutive keypresses into one keypress event. This violates the property that the internal counter `stime` always increases after the

			original		modified	
Property	*loc*	k	t (sec.)	Result	t (sec.)	Result
baud_conf	178	1	1	$\top^p$	1	$\bot^p$
		4	1	$\top^p$	1	$\bot^p$
		10	1	$\top^p$	1	$\bot^p$
		20	2	$\top^p$	2	$\bot^p$
keyb_start	50	1	1	$\top^p$	1	$\bot^p$
		4	2	$\top^p$	2	$\bot^p$
		10	5	$\top^p$	4	$\bot^p$
		20	17	$\top^p$	15	$\bot^p$
serial_rx	584	1	1	$\top^p$	1	$\bot^p$
		4	2	$\top^p$	2	$\bot^p$
		10	7	$\top^p$	5	$\bot^p$
		20	23	$\top^p$	25	$\bot^p$
up_btn	856	1	1	$\top^p$	1	$\top^p$
		4	1	$\top^p$	1	$\top^p$
		10	2	$\top^p$	2	$\top^p$
		20	3	$\top^p$	3	$\bot^p$
start_btn	856	1	1	$\top^p$	1	$\bot^p$
		4	1	$\top^p$	1	$\bot^p$
		10	2	$\top^p$	2	$\bot^p$
		20	3	$\top^p$	2	$\bot^p$
buflim	145	1	1	$\top^p$	1	$\bot$
		4	934	$\top^p$	4	$\bot$
		10	MO	MO	MO	MO
		20	MO	MO	MO	MO

Table 3 Results of testing LTL properties against pulse oximeter firmware.

enabling key press event. However, as every third keypress *does* result in a keypress event, the unwind bounds of 1, 4 and 10 terminate with the most recent keypress having caused a keypress event, thus terminating in a $\top^p$ state. This is an example of a property that oscillates between $\bot^p$ and $\top^p$ as the unwind bounds are changed, as discussed in Section 5.3.

7.2 Bicycle computer

The bicycle computer case study comprises a small C-model of a device designed to gather and display speed and distance information about a cyclist's journey. This case study contains approximately 150 lines of code. The program is multi-threaded and treats user input, display, and data collection as separate processes. We test a number of (valid) properties over the global state of the program, listed in Table 4.

Because this program is multi-threaded, checking it using ESBMC results in a large number of distinct runs of ESBMC's SMT solver, each corresponding to different thread interleavings. These have to be combined

Name	Property
dist_ovfl	G({`cycle_distance_m >= 0`})
tot_dist_ovfl	G({`total_cycle_distance_m >= 0`})
dist_rel	G({`cycle_distance_m <= total_cycle_distance_m`})
state_range	G({`cur_state >= 0`} $\wedge$ {`cur_state <= 3`})

Table 4 Bicycle computer properties.

		$k = 1$		$k = 2$	
Property	C	Time (s)	Result	Time (s)	Result
dist_ovfl	1	1	$\top^p$	1	$\top^p$
	2	7	$\top^p$	34	$\top^p$
	3	56	$\top^p$	379	$\top^p$
tot_dist_ovfl	1	1	$\top^p$	1	$\top^p$
	2	5	$\top^p$	24	$\top^p$
	3	63	$\top^p$	368	$\top^p$
dist_rel	1	1	$\top^p$	2	$\top^p$
	2	7	$\top^p$	32	$\top^p$
	3	59	$\top^p$	542	$\top^p$
state_range	1	1	$\top^p$	2	$\top^p$
	2	7	$\top^p$	42	$\top^p$
	3	62	$\top^p$	478	$\top^p$

Table 5 Results of testing LTL properties against bicycle model.

together to report the worst (in the four-valued lattice) behaviour of any interleaving.

We test the program against the properties with a number of different unwind bounds k and context switch bounds C. Our results (cf. Table 5) show the correct output is determined for each run, for a variety of loop unwind bounds and context switch bounds. We note that verification time increases exponentially with increases in the context bound, which is as expected in multi-threaded verification.

The bicycle computer examples above are all safety properties. Verification of liveness properties in multi-threaded code presents additional difficulties for our approach and is currently practical only for small examples. Multi-threaded safety failures are typically *shallow*, requiring only few interleaves. In contrast, even liveness properties guaranteed by loop invariants require that relatively large interleave bounds be set to ensure that all threads run complete loop iterations. More general liveness properties can depend on scheduling between threads. The default pthreads behaviour provides weak fairness and is accurately modelled by ESBMC. Liveness properties which depend on this weak fairness will, however, inevitably show violations for finite traces.

8 Related Work

Related work in the area of finite LTL semantics has already been discussed in Section 3. We will not discuss general approaches to LTL symbolic model checking (see Rozier [41] for an overview) or to software model checking (see Visser et al. [46] for an overview) here, but focus on approaches that specifically model check software against LTL properties.

SPIN [23] is a well known software model checker that operates on concurrent program models written in the Promela modelling language. SPIN operates with explicit state and uses state hashing to reduce the quantity of state space it explores. SPIN also allows users to specify an LTL formula to verify against the execution of a model by using BA in a similar manner to our 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 formulae can be achieved with the JPF-LTL [38] extension which uses BA and method invocation monitoring to inspect the execution of the model.

Staats and Heimdahl [43] take Simulink models and verify that a prototype Simulink-to-C translator produces code that satisfies the same properties as the Simulink model. A set of predetermined safety properties described in LTL are verified first against the Simulink model, then against the emitted C code. A C monitor is devised, and a feature of the converted model is used to select code locations where the monitor must be inserted. Their approach is not designed to support the checking of liveness or co-safety properties.

Leucker and Schallhart [34] review the field of *run-time verification* and cover its differences from model checking, as well as various LTL-like logics for analysing finite prefixes of traces. More expressive ways of describing system properties are explored, as well as the potential for run-time analysis beyond verification.

9 Conclusions and Future Work

Context-bounded model checking has already been used successfully to verify multi-threaded applications written in low-level languages such as 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 separate C monitor thread and check all interleavings between this monitor and the program using ESBMC. We use a four-valued

LTL semantics to handle the finite traces that bounded model checking explores.

Our results so far 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. There are still considerable opportunities to improve performance and to execute on more capable computer platforms. For multi-threaded simulations, the state hashing reported in our SEFM 2011 contribution [37] has proved to be very useful, cutting verification times by about 50% on average. We expect that an improved hashing implementation, for example removing serialisation, will improve these results further.

We are also keen to embrace the new C language standard's [27] threading support and weak memory model; this should allow us to substantially increase performance by reducing the number of safe interleavings.

Acknowledgement. This work was supported by a Royal Society International Exchange Grant. The reviewers' comments helped us to improve our presentation.

A Sample monitor

Listing 1 C implemention of the Büchi automaton for the formula $\neg\mathsf{G}(\{\texttt{pressed}\} \rightarrow \mathsf{F}\{\texttt{charge > min}\})$.

```
1  char __ESBMC_property___cexpr_0[] = "pressed";
2  _Bool __cexpr_0_status;
3  char __ESBMC_property___cexpr_1[] = "charge > min";
4  _Bool __cexpr_1_status;
5
6  typedef enum {_ltl2ba_state_0,_ltl2ba_state_1} _ltl2ba_state;
7  _ltl2ba_state _ltl2ba_statevar =_ltl2ba_state_0;
8
9  void *ltl2ba_fsm(void  *d) {
10   unsigned int choice;
11   while (1) {
12     choice = nondet_uint();
13     __ESBMC_atomic_begin();
14     switch(_ltl2ba_statevar) {
15     case _ltl2ba_state_0:
16       if (choice == 0) {
17         __ESBMC_assume(1);
18         _ltl2ba_statevar = _ltl2ba_state_0;
19       } else if (choice == 1) {
20         __ESBMC_assume(!_ltl2ba_cexpr_1_status &&
21                        _ltl2ba_cexpr_0_status);
22         _ltl2ba_statevar = _ltl2ba_state_1;
23       } else {
24         __ESBMC_assume(0);
```

```
25        }
26        break;
27      case _ltl2ba_state_1:
28        if (choice == 0) {
29          __ESBMC_assume(!_ltl2ba_cexpr_1_status);
30          _ltl2ba_statevar = _ltl2ba_state_1;
31        } else {
32          __ESBMC_assume(0);
33        }
34        break;
35      }
36      __ESBMC_atomic_end();
37      __ESBMC_switch_from_monitor();
38    }
39  }
40
41  void ltl2ba_start_monitor(void) {
42    pthread_t t;
43    __ESBMC_atomic_begin();
44    pthread_create(&t, NULL, ltl2ba_fsm, NULL);
45    __ESBMC_register_monitor(t);
46    __ESBMC_atomic_end();
47    __ESBMC_switch_to_monitor();
48  }
49
50  _Bool _ltl2ba_stutter_accept_table[4][2] = {
51  {false,true}, {false,false}, {true,true}, {false,true}
52  };
53
54  _Bool _ltl2ba_good_prefix_excluded_states[2] =
55  { true, true };
56
57  _Bool _ltl2ba_bad_prefix_states[2] =
58  { false, false };
59
60  unsigned int _ltl2ba_sym_to_idx(void) {
61    unsigned int idx = 0;
62    idx |= (_ltl2ba_cexpr_1_status) ? 1 : 0;
63    idx |= (_ltl2ba_cexpr_0_status) ? 2 : 0;
64    return idx;
65  }
66
67  void ltl2ba_finish_monitor(void) {
68    __ESBMC_kill_monitor();
69
70    _Bool in_bad_state =
71      _ltl2ba_bad_prefix_states[_ltl2ba_statevar];
72    __ESBMC_assert(!in_bad_state,"LTL_BAD");
73
74    unsigned int cursym = _ltl2ba_sym_to_idx();
```

```
75    _Bool in_accept_state =
76      _ltl2ba_stutter_accept_table[cursym][_ltl2ba_statevar];
77    __ESBMC_assert(!in_accept_state,"LTL_FAILING");
78
79    _Bool not_in_good_prefix =
80      _ltl2ba_good_prefix_excluded_states[_ltl2ba_statevar];
81    __ESBMC_assert(!not_in_good_prefix,"LTL_SUCCEEDING");
82
83    return;
84  }
```

References

1. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. **21**(4), 181–185 (1985).
2. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed Computing **2**(3), 117–126 (1987).
3. Barnett, M., DeLine, R., Fähndrich, M., Jacobs, B., Leino, K.R.M., Schulte, W., Venter, H.: The Spec# programming system: Challenges and directions. In: Meyer, B., Woodcock, J. (eds.): Proc. Conf. Verified Software: Theories, Tools, Experiments (VSTTE'05), pp. 144–152. *Lecture Notes in Computer Science*, vol. 4171. Springer (2008).
4. Bauer, A., Haslum, P.: LTL goal specifications revisited. In: Coelho, H., Studer, R., Wooldridge, M. (eds.): Proc. European Conf. Artificial Intelligence (ECAI'10), pp. 881–886. *Frontiers in Artificial Intelligence and Applications*, vol. 215. IOS Press (2010).
5. Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly, but how ugly is ugly? In: Sokolsky, O., Tasiran, S. (eds.): Proc. Workshop Runtime Verification (RV'07), pp. 126–138. *Lecture Notes in Computer Science*, vol. 4839. Springer (2007).
6. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput. **20**(3), 651–674 (2010).
7. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14 (2011).
8. Büchi, J.R.: Symposium on decision problems: On a decision method in restricted second order arithmetic. In: P.S. Ernest Nagel, A. Tarski (eds.): Logic, Methodology and Philosophy of Science, Proceedings of the 1960 International Congress, pp. 1–11. *Studies in Logic and the Foundations of Mathematics*, vol. 44, Elsevier (1966).
9. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. STTT **9**(5-6), 505–525 (2007).
10. Biere, A., Heljanko, K., Junttila, T.A., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. Logical Methods in Computer Science **2**(5) (2006).
11. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.): Proc. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), pp. 168–176. *Lecture Notes in Computer Science*, vol. 2988. Springer (2004).
12. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. Formal Methods in System Design **25**(2-3), 105–127 (2004).

13. Clarke, E.M., Lerda, F.: Model checking: Software and beyond. J. UCS **13**(5), 639–649 (2007).
14. Cordeiro, L., Barreto, R.S., Barcelos, R., Oliveira, M.N., Lucena, V., Maciel, P.R.M.: Agile development methodology for embedded systems: A platform-based design approach. In: Leaney, J., Rozenblit, J.W., Peng, J. (eds.): Proc. Conf. Engineering of Computer Based Systems (ECBS'07), pp. 195–202. IEEE Computer Society (2007).
15. Cordeiro, L., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: Taylor, R.N., Gall, H., Medvidovic, N. (eds.): Proc. Intl. Conf. Software Engineering (ICSE'11), pp. 331–340. ACM (2011).
16. Cordeiro, L., Fischer, B., Chen, H., Marques-Silva, J.: Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In: Chen, T., Serpanos, D.N., Taha, W. (eds.): Proc. Intl. Conf. Embedded Software and Systems (ICESS'09), pp. 396–403. IEEE (2009).
17. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: Grundy, J., Taentzer, G., Heimdahl, M. (eds.): Proc. Conf. Automated Software Engineering (ASE'09), pp. 137–148. IEEE Computer Society (2009).
18. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Boehm, B.W., Garlan, D., Kramer, J. (eds.): Proc. Intl. Conf. Software Engineering (ICSE'99), pp. 411–420. ACM (1999)
19. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Campenhout, D.V.: Reasoning with temporal logic on truncated paths. In: Hunt, W.A., Somenzi, F. (eds.): Proc. Conf. Computer Aided Verification (CAV'03), pp. 27–39. *Lecture Notes in Computer Science*, vol. 2725. Springer (2003).
20. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.): Proc. Conf. Computer Aided Verification (CAV'01), pp. 53–65. *Lecture Notes in Computer Science*, vol. 2102. Springer (2001).
21. Giannakopoulou, D., Havelund, K.: Automata-based verification of temporal properties on running programs. In: Richardson, D., Feather, M.S., Goedicke, M. (eds.): 16th IEEE Intl. Proc. Conf. Automated Software Engineering (ASE'01), pp. 412–416. IEEE Computer Society (2001).
22. He, A., Wu, J., Li, L.: An efficient algorithm for transforming LTL formula to Büchi automaton. In: Proc. Conf. Intelligent Computation Technology and Automation (ICICTA'08), Volume 01, pp. 1215–1219. IEEE Computer Society (2008).
23. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Software Eng. **23**(5), 279–295 (1997).
24. Holzmann, G.J.: The SPIN Model Checker - primer and reference manual. Addison-Wesley (2004).
25. Huth, M., Ryan, M.D.: Logic in computer science - modelling and reasoning about systems (2. ed.). Cambridge University Press (2004).
26. ISO, *ISO/IEC/IEEE 9945:2009 Information technology — Portable Operating System Interface (POSIX) Base Specifications, Issue 7*. Geneva, Switzerland: International Organization for Standardization, December 2009.
27. ISO, *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, December 2011.

28. Jonsson, B., Tsay, Y.K.: Assumption/guarantee specifications in linear-time temporal logic. Theor. Comput. Sci. **167**(1&2), 47–72 (1996).
29. Kamp, H.W.: Tense logic and the theory of linear order. Phd thesis, Computer Science Department, University of California at Los Angeles, USA (1968).
30. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods in System Design **19**(3), 291–314 (2001).
31. Lahiri, S.K., Qadeer, S., Rakamaric, Z.: Static and precise detection of concurrency errors in systems code using SMT solvers. In: Bouajjani, A., Maler, O. (eds.): Proc. Conf. Computer Aided Verification (CAV'09), pp. 509–524. *Lecture Notes in Computer Science*, vol. 5643. Springer (2009).
32. Lamport, L.: A new approach to proving the correctness of multiprocess programs. ACM Trans. Program. Lang. Syst. **1**(1), 84–97 (1979).
33. Lamport, L.: What good is temporal logic? In: IFIP Congress, pp. 657–668 (1983).
34. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. **78**(5), 293–303 (2009).
35. Manna, Z., Pnueli, A.: Temporal verification of reactive systems - safety. Springer (1995).
36. McMillan, K.L.: Symbolic model checking. Kluwer (1993).
37. Morse, J., Cordeiro, L., Nicole, D., Fischer, B.: Context-bounded model checking of LTL properties for ANSI-C software. In: Barthe, G., Pardo, A., Schneider, G. (eds.): Proc. Conf. Software Engineering and Formal Methods (SEFM'11), pp. 302–317. *Lecture Notes in Computer Science*, vol. 7041. Springer (2011).
38. Nguyen, A.C., Khoo, S.C.: Towards automation of LTL verification for Java Pathfinder (2008). In: Proceedings of the 15th National Undergraduate Research Opportunities Programme Congress, Singapore (2010).
39. Pnueli, A.: The temporal logic of programs. In: Proc. Symp. Foundations of Computer Science (FOCS'77), pp. 46–57. IEEE Computer Society (1977).
40. Rabinovitz, I., Grumberg, O.: Bounded model checking of concurrent programs. In: Etessami, K., Rajamani, S.K. (eds.): Proc. Conf. Computer Aided Verification (CAV'05), pp. 82–97. *Lecture Notes in Computer Science*, vol. 3576. Springer (2005).
41. Rozier, K.Y.: Linear temporal logic symbolic model checking. Computer Science Review **5**(2), 163–203 (2011).
42. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. STTT **12**(2), 123–137 (2010).
43. Staats, M., Heimdahl, M.P.E.: Partial translation verification for untrusted code-generators. In: Liu, S., Maibaum, T.S.E., Araki, K. (eds.): Proc. Conf. Formal Methods and Software Engineering (ICFEM'08), pp. 226–237. *Lecture Notes in Computer Science*, vol. 5256. Springer (2008).
44. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Moller, F., Birtwistle, G.M. (eds.): Logics for Concurrency - Structure versus Automata, pp. 238–266. *Lecture Notes in Computer Science*, vol. 1043. Springer (1996).
45. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: Proc. Symp. Logic in Computer Science (LICS'86), pp. 332–344. IEEE Computer Society (1986).
46. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003).