Deterministic Compilation of Temporal Safety Properties in Explicit State Model Checking

Kristin Y. Rozier and Moshe Y. Vardi *

Rice University, Houston, Texas 77005, {kyrozier, vardi}@cs.rice.edu

Abstract. The translation of temporal logic specifications constitutes an essential step in model checking and a major influence on the efficiency of formal verification via model checking. We devise a new explicit-state translation of Linear Temporal Logic to automata for the class of LTL specifications that describe *safety properties*, arguably the most used formal specifications in real-world systems. By exploiting the inherent determinism in safety specifications, we can build deterministic Promela `never` claims that accept only the bad prefixes of the safety specification. In contrast to previous works, we focus on compilation to `never` claims rather than simply automata and measure Spin model-checking time separately from compilation time and automata size. An extensive experimental evaluation over a space of configurations demonstrates that our new translation consistently results in better model-checking performance, for a large array of benchmarks, over the best current translation.

1 Introduction

In linear-time model checking, the negation of the temporal specification is translated into a nondeterministic Büchi automaton, combined with the system model, and then checked for nonemptiness [33]. The model checker searches for a *lasso-shaped counterexample trace* in this combined model, a trace that starts at an initial system state and reaches a cycle that contains an accepting state. The explicit-state translation of Linear Temporal Logic (LTL) formulas to Büchi automata constitutes an essential step in explicit-state linear-time model checking and has a major influence on the efficiency of model checking [10]. Consequently, this topic has received a significant level of attention over the past two decades and there are many available tools; see [27] for an extensive survey. Most of that research has focused on minimizing the size of the generated automata. The rationale was that minimizing the size of the automaton would minimize the size of the space in the product of the system model and the automaton that the model checker must search. Yet this heuristic approach has no experimental evidence that would demonstrate its efficacy [32]. In fact, the extensive experimental investigation reported on in [27], which focused on *satisfiability checking*, a special

* This research was supported in part by NSF grants CNS 1049862 and CCF-1139011, by NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering", by BSF grant 9800096, by a gift from Intel, and by the Shared University Grid at Rice funded by NSF under Grant EIA-0216467 and a partnership between Rice, Sun Microsystems, and Sigma Solutions, Inc.

case of model checking, shows little correlation between automaton size and model-checking time. It is argued in [9] that larger automata may result in less work for LTL model checking. In this paper we revisit the translation of LTL formulas to automata, which we call *LTL compilation*, specifically focusing on model-checking performance.

We concentrate on model checking *safety properties*, which assert "something bad never happens" [1]. Safety properties are the most often used formal properties in practice, capturing the desired behaviors of a wide variety of real-world systems, such as of fault tolerance [11] and hardware resets [7]. Safety properties can also describe most intended properties of real-time systems, since responses are usually required within bounded intervals [15].

Intuitively, "something bad" only needs to happen once in a computation for the property to be violated. Thus, a violation of a safety property can always be witnessed by a finite prefix of a violating infinite trace. Rather than search the system model for a violating infinite trace, we can search the system model for this *bad prefix*. This insight forms the basis for an alternative automata-theoretic approach for model checking safety properties, proposed in [20]: construct a *deterministic* automaton for the language of bad prefixes, take its product with the system model, and then search for an accepting finite trace. A disadvantage of this approach is that while the translation from LTL to nondeterministic Büchi automata is, in the worst case exponential [34], the translation from safety LTL formulas to deterministic automata for bad prefixes is, in the worst case, doubly exponential [20]. Perhaps because of this additional blow-up, this approach, which we refer to as *deterministic compilation*, has yet to be seriously explored.

There has been recent evidence that deterministic compilation may be a viable approach in spite of the possible additional exponential blow-up. Deterministic compilation proved to be effective for SAT-based model checking [2] and explicit-state hybrid-systems analysis [26]. Determinizing finite automata representing safety formulas has been correlated with smaller system model/automaton products even without minimizing the formula automaton [21]. Intuitively the product–system model times automaton–is simpler when the automaton is deterministic, as nondeterminism in the product stems solely from nondeterminism in the system. Intuitively, in the standard approach the search algorithm has to find both a counterexample trace in the system and an accepting run of the specification automaton. This second search is not needed when the specification automaton is deterministic, as it has a unique run on a given input word. (It has been argued in [30], though without evidence, that "more deterministic" compilation may be an advantageous approach.) Recent work on deterministic compilation in the context of run-time verification demonstrated both that the doubly exponential blow-up rarely appears in practice, and that the resulting deterministic automata are often actually *smaller* than their nondeterministic counterparts since we can minimize deterministic automata efficiently [31, 11].

The main result of this paper is that deterministic compilation is indeed an effective approach to explicit-state model checking of safety properties. To demonstrate this, we build on the theoretical foundations developed in [4, 20]. We show how to use SPOT [6], the best LTL-to-automaton translator (see [27]), and `BRICS Automaton` [23], a tool for determinizing and minimizing finite-word automata, in order to go from a nondetermin-

istic Büchi automaton $\mathcal{A}_\varphi$ representing a safety property φ to a deterministic automaton $\mathcal{A}^d$ that accepts the bad prefixes of φ. This construction uses the fact that determinization of finite-word automata is much simpler than determinization of ω-automata; while nondeterministic finite automata can be determinized with a simple subset construction [14], determinization of nondeterministic ω-automata requires a complex subset-tree-based construction [28].

To use $\mathcal{A}^d$ for model checking, we apply Spin, the canonical explicit-state model checker [12]. We introduce 26 novel encodings of LTL safety properties as deterministic automata in the form of Promela (PROcess MEta LAnguage) `never` claims, describing behaviors that should *not* occur in the system model. We implement these encodings as an extension of the open-source CHIMP tool[1] [31] that creates SystemC monitors for LTL formulas; our extension, CHIMP-Spin, [2] creates Spin `never` claims. Our systematic empirical investigation of the effectiveness of these automata as `never` claims also constitutes a novel contribution since earlier works focused on translation to automata without considering their encodings as `never` claims. We show over a large array of benchmarks that our deterministic encodings for model checking of safety properties consistently result in significantly reduced model-checking times over the SPOT encoding. We also demonstrate that the encoding used to represent deterministic automata as `never` claims has a significant impact on performance and we identify a single encoding that dominates all other encodings.

A key point of our approach is that we concentrate on reducing *model-checking time*, while typical experimental work in LTL model checking measures total time–compile plus model-checking time, e.g., [9]. Since in real-world applications of model checking, properties are written once and then checked against a changing system design multiple times, we find it worthwhile to reduce model-checking time even at the cost of increased property-compilation time. This choice is particularly pertinent for regression testing: when the system is changed to fix a bug or add a new feature it is necessary to recheck all properties checked earlier to ensure that previous checks produce the same results. To streamline regression testing future versions of Spin should not require a recompilation of `never` claims for each run of the model checker, even when they have not changed. Such an adjustment would more accurately reflect industrial applications of model checking and, combined with our reduced model checking times, reduce the amortized cost of model checking.

The structure of the paper is as follows. We detail the theory underlying our construction of deterministic encodings of LTL safety specifications in Section 2 and describe our 26 novel constructions of Promela `never` claims in Section 3. We then describe our experimental methodology in Section 4, and present our experimental results, which demonstrate that we can consistently outperform SPOT, the current best LTL-compilation tool, in Section 5. We conclude with a discussion in Section 6.

[1] `http://sourceforge.net/projects/chimp-rice/`

[2] Our tool extension is released under an open-source license; contact us for a copy.

2 Theoretical Background

We interpret LTL formulas over infinite computations of the form $\pi : \omega \rightarrow 2^{Prop}$, where ω is the set non-negative integers and $Prop$ is a set of atomic propositions. We define $\pi, i \vDash \varphi$ (computation π at time instant $i \in \omega$ satisfies LTL formula φ) as follows [8]:

- $\pi, i \vDash p$ for $p \in Prop$ if $p \in \pi(i)$.
- $\pi, i \vDash g_1 \wedge g_2$ if $\pi, i \vDash g_1$ and $\pi, i \vDash g_2$.
- $\pi, i \vDash \neg g$ if $\pi, i \nvDash g$.
- $\pi, i \vDash \mathcal{X} g$ if $\pi, i+1 \vDash g$.
- $\pi, i \vDash g_1 \mathcal{U} g_2$ if $\exists j \geq i$, such that $\pi, j \vDash g_2$ and $\forall k, i \leq k < j$, we have $\pi, k \vDash g_1$.
- $\pi, i \vDash g_1 \mathcal{R} g_2$ if $\forall j \geq i$, if $\pi, j \nvDash g_2$, then $\exists k$, $i \leq k < j$, such that $\pi, k \vDash g_1$.
- $\pi, i \vDash \Diamond g$ if $\exists j \geq i$, such that $\pi, j \vDash g$.
- $\pi, i \vDash \Box g$ if $\forall j \geq i$, $\pi, j \vDash g$.

We take $models(\varphi)$ to be the set of computations that satisfy φ at time 0: $\{\pi : \pi, 0 \vDash \varphi\}$.

In automata-theoretic model checking, we represent LTL formulas using Büchi automata. A *Nondeterministic Büchi Word Automaton* (NBW) is a quintuple $\mathcal{A} = (Q, \Sigma, \delta, Q^0, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \rightarrow 2^Q$ is a transition function, $Q^0 \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of accepting states. If $q' \in \delta(q, \sigma)$ then we say that we have a transition from q to q' labeled by σ. A run of a Büchi automaton $\mathcal{A}$ over an infinite computation $\pi = \pi_0, \pi_1, \pi_2, \ldots \in \Sigma$ is a sequence $q_0, q_1, q_2, \ldots$ of states such that $q_0 \in Q_0$, and $\langle q_i, \pi_i, q_{i+1} \rangle \in \delta$ for all $i \geq 0$. $\mathcal{A}$ accepts π if the run over π visits states in F infinitely often. We denote the set of infinite words accepted by $\mathcal{A}$ by $\mathscr{L}_\omega(\mathcal{A})$. Computations are infinite words over the alphabet $\Sigma = 2^{Prop}$.

Theorem 1. [34] *Given an LTL formula φ, we can construct an NBW $\mathcal{A}_\varphi = (Q, \Sigma, \delta, q_0, F)$ such that $|Q|$ is in $2^{O(|\varphi|)}$, $\Sigma = 2^{Prop}$, and $L_\omega(\mathcal{A}_\varphi)$ is exactly $models(\varphi)$.*

In the automata-theoretic approach to model checking [33], to check that a model M under verification satisfies an LTL formula φ, we translate $\neg\varphi$ into the automaton $\mathcal{A}_{\neg\varphi}$ and compose $\mathcal{A}_{\neg\varphi}$ with M, forming the automaton $\mathcal{A}_{M, \neg\varphi}$, which the model checker checks for emptiness. If there is no accepting run of $\mathcal{A}_{M, \neg\varphi}$ (i.e. the language $\mathscr{L}(\mathcal{A}_{M, \neg\varphi}) = \emptyset$), we have proven that $M \models \varphi$.

The automata-theoretic approach can be refined when dealing with *safety properties*. A formula φ is a safety formula if its failure can always be witnessed by a finite prefix [1]; that is, if $\pi \not\models \varphi$ then there there is a finite word $w \in \Sigma^*$ such that $w \cdot \pi \not\models \varphi$ for every infinite computation $\pi \in \Sigma^\omega$. Here w is called a *bad prefix* for φ. The set of bad prefixes for φ is $pref(\varphi)$. It is argued in [19] that $pref(\varphi)$ is a regular language; consequently, we can use automata on finite words for model checking safety properties.

A *Nondeterministic Finite Word Automaton* (NFW) is a quintuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \rightarrow 2^Q$ is a transition function, $Q_0 \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is a set of accepting states. If Q_0 is a singleton, and $\delta(q, a)$ contains at most one state for every state q and letter a, then we say that $\mathcal{A}$ is a *Deterministic Finite Word Automaton* (DFW). A run of $\mathcal{A}$ over a finite word $w \in \Sigma^*$ is accepting if it terminates in an accepting state.

Theorem 2. [19] *Given a safety LTL formula* φ*, we can construct a DFW* $\mathcal{A}^d = (Q, \Sigma, \delta, q_0, F)$ *such that* $|Q|$ *is in* $2^{2^{O(|\varphi|)}}$*,* $\Sigma = 2^{Prop}$*, and* $\mathcal{L}(\mathcal{A}^d)$ *is exactly* $pref(\varphi)$*.*

Therefore, when φ is a safety property, we can opt to form an NFW or a DFW corresponding to $\neg\varphi$ instead of an NBW, since we only need to construct an automaton that accepts the set of finite prefixes that witness violations of φ.

A concrete algorithm to construct automata for bad prefixes was given in [4]. Given a safety formula φ, we first form the NBW $\mathcal{A}_\varphi$. Here we use SPOT [6] for this translation; we showed earlier that SPOT is the best LTL-to-automata translator [27]. Let $empty(\mathcal{A}_\varphi)$ be the set of states in $\mathcal{A}_\varphi$ that cannot appear on an accepting run. SPOT can compute this set of states and remove them from $\mathcal{A}_\varphi$. We now turn this NBW into an NFW $\mathcal{A}_\varphi^f$ by re-labeling all remaining states to be accepting. We now have the NFW $\mathcal{A}_\varphi^f$ defined by the quintuple $(Q', \Sigma, \delta', q_0 \cap Q', F \cap Q')$, where $Q' = Q - empty(\mathcal{A}_\varphi)$ and δ' is restricted to $Q' \times \Sigma$. Note that this approach is not sound for liveness formulas.

Theorem 3. [4] $\mathcal{A}_\varphi^f$ *rejects precisely* $pref(\varphi)$*.*

To model check a safety formula, we need an automaton that accepts $pref(\varphi)$ [31]. If we apply the subset construction to $\mathcal{A}_\varphi^f$ we obtain a DFW $\mathcal{A}_\varphi^d$, where all nonempty sets of states of $\mathcal{A}_\varphi^f$ are accepting states, that rejects $pref(\varphi)$. Its complement $\mathcal{A}_{\neg\varphi}^d$, where only the empty set of states is accepting, accepts $pref(\varphi)$.

3 `Never` Claim Generation

A `never` claim is a Promela code sequence that defines a system behavior that should never happen. Since we use `never` claims to specify properties that should *never* happen, that is, bad properties we wish to assert the system does not have, we create `never` claim corresponding to the negation of the property we wish to hold. In other words, when we create a `never` claim that accepts exactly $\mathcal{L}(\neg\varphi)$ we are stating that it would be a correctness violation of the system if there exists an execution sequence in which $\neg\varphi$ holds. For the system to be considered correct, φ must always hold.

To generate a Promela `never` claim for LTL formula φ, Spin translates $\neg\varphi$ into the NBW $\mathcal{A}_{\neg\varphi} = (Q, \Sigma, \delta, q_0, F)$, enumerates and creates label for the states in Q, labels q_0 with 'init' to designate the state in which the `never` claim starts, labels accepting states with 'accept,' and implements δ by a nondeterministic choice: for each state, nondeterministically choose from among enabled transitions given the set of propositions true in the current state. Currently, all LTL-to-Promela translators follow this high-level construction. (They vary widely in the details of the formation of $\mathcal{A}_{\neg\varphi}$ as described in [27].)

In this paper, we construct Promela `never` claims corresponding to the DFW $\mathcal{A}_\varphi^d$ for bad prefixes of safety formulas. We now describe several novel alternatives for constructing `never` claims for safety properties.

To prove that a system model M satisfies the LTL property $\varphi = (\Box good)$, we create a `never` claim that accepts the negation of this property. Spin can do this automatically using the command `spin -f '![] good'`. Intuitively, the `never` claim generated by

the formula would restrict system behavior to those states where $(\Diamond !good)$ holds. If any such execution of the system is found, Spin reports a violation.

In addition to the infinite-behavior `never` claims produced by Spin, SPOT, and other tools, `never` claims can be also be used to specify finite automata; the distinction is implicit in the structure of the claim rather than explicitly stated. A finite behavior is matched if the claim can reach its closing curly brace while executing in lockstep with the system model [13]. Spin automatically checks for this type of `never` claim termination. A `never` claim corresponding to the NFW that accepts $pref(\varphi)$ simply needs to reach its closing curly brace, for example, when the formula is $\Box good$, if $!good$ is ever true, thus accepting the finite prefix indicating to a correctness violation of the system. Note that we check the finite-behavior `never` claim using different Spin commands than the infinite-behavior version, where the run-time flag `-a` explicitly tells Spin to check for acceptance cycles. Specifically, we check for finite acceptance using the following commands:

```
cat Model > pan_in
cat finite_never_claim >> pan_in
spin -a pan_in
gcc -w -o pan -D_POSIX_SOURCE -DMEMLIM=1550 -DSAFETY -DXUSAFE -DNOFAIR
              -DNXT pan.c
./pan -v -X -m10000 -w19  -A -E -c1
```

3.1 Determinization and Minimization

As in [31], there are two approaches to constructing the DFW $\mathcal{A}_\varphi^d$. First, we can explicitly determinize the NFW $\mathcal{A}_\varphi^f$ using an NFW-to-DFW translator (`BRICS Automaton`[23]), which we refer to as the `det` construction. Second, we can construct a `never` claim directly from $\mathcal{A}_\varphi^f$, essentially performing the subset construction on-the-fly. For consistancy with previous work [31], we refer to this as the `nondet` construction, because determinism is delayed. The advantage of pre-compilation determinization is the ability to minimize $\mathcal{A}_\varphi^d$ before constructing the `never` claim; we use `BRICS Automaton` to produce a minimal equivalent DFW. We refer to this as the `min` construction. The additional steps of determinization and minimization may incur a nontrivial computational cost during the construction of the `never` claim. The trade-off between property-compilation time and model checking time is a key issue in this paper.

To use `BRICS Automaton`, we have to find a way to represent the alphabet of the automata [31]. SPOT labels transitions with Boolean formulas over the set *Prop* of atomic propositions, while `BRICS Automaton` represents the alphabet of the automaton as Unicode characters. Therefore, we adapt the techniques of [31] for describing the alphabet in terms of 16-bit integers. We have two alphabet representations: OBDD-based and assignment-based.

We can represent Boolean formulas using *Ordered Binary Decision Diagrams (OBDDs)* [3]. We implement this approach as follows. First, we obtain references to all Boolean formulas that appear as transition labels in the automaton using SPOT's `spot::tgba_reachable_iterator_breadth_first::process_link()` function. Second, we assign a unique integer label to the OBDD representation of each Boolean formula (up to $2^{|\varphi|}$ in the worst case) using SPOT's `spot::tgba_succ_iterator::current_condition()` function. The formulas labeling automaton transitions can now be replaced by the corresponding integers.

Alternatively, we can represent Boolean formulas in terms of their satisfying truth *assignments*. By selecting an order for $Prop = \{p_1, \ldots, p_n\}$, we can represent an assignment as an n-bit vector $\mathbf{a} = [a_1, a_2, \ldots, a_n]$. Every such bit vector corresponds to an integer $I(\mathbf{a})$ in the domain $\{0, \ldots, 2^n - 1\}$; $I(\mathbf{a}) = a_1 2^{n-1} + a_2 2^{n-2} + \ldots + a_n 2^0$. We can use this domain as a new alphabet, replacing a transition labeled by a Boolean formula α by several transitions labeled by the integers corresponding to truth assignments satisfying α. Once we have used `BRICS Automaton` to form a DFW, we convert transition labels back to a Boolean formula that we use to construct Promela `never` claims.

The assignment-based approach sometimes creates a large number of transitions. For example, the Boolean formula **true** corresponds to 2^n assignments. We introduce an *edge-abbreviation* technique to merge separate transitions. When we have several transitions with the same source and destination states, we can remove these transitions and replace them by a single transition labeled by the disjunction of the labels of the removed transitions. For each such disjunction, we utilize SPOT's built-in `formula_to_bdd()` function to create a BDD representing the disjunction, extract a simplified formula from the BDD via the reverse `bdd_to_formula()` function, and then label the associated transition by this new formula. A related optimization is to replace all `else` branches in the Promela `never` claims by explicit Boolean formulas corresponding to the negation of the conjunction of the labels of all of the other transitions (reduced using SPOT's built-in BDD functions). This enables us to eliminate redundant trap states and reduce `never` claim code size.

3.2 `Never` claim encodings

Inspired by the work in [31], we introduce 26 ways of encoding automata for safety properties as Promela `never` claims. We form these encodings by combining our `never` claim adaptations of the constructions for transition direction (`front` vs `back`), determinism (`det` vs `nondet`), state minimization (`min` vs `nomin`), and alphabet representation (`bdd` vs `abr`) from [31] with the options to encode `never` claim states either using Promela state labels or integer state numbers (`state` vs `number`), to employ either finite or infinite acceptance conditions (`fin` vs `inf`), and to reduce the size of the generated `never` claim via edge abbreviation and trap-state elimination (`ea`). We illustrate our encodings in Appendix A for benchmark safety formula 4 from Table 3.

Nondeterministic encodings We introduce 12 novel Promela encodings that perform determinization on-the-fly. In `nondet never` claims we maintain an array used to describes sets of states of $\mathcal{A}_\varphi^f$. An array that corresponds to an empty set indicates that $\mathcal{A}_\varphi^f$ got stuck, which means that we have discovered a violation of φ. We can encode the transition relations either in a `front` fashion, where for any state q we enumerate the outgoing transitions from q, or in a `back` fashion, where for any state q we enumerates the incoming transitions that lead to q.

The `front_nondet` encoding uses an `if` statement to check each outgoing transition from each possible current state and marks all possible next states in the `next_state` array. If there is no possible next state, the automaton fails. For `never` claims with finite acceptance conditions, this is accomplished by breaking from the `do` loop and coming

to the end } of the claim. The `back_nondet` encoding works similarly, but the branching is over incoming transitions rather than over outgoing transitions. See Listing 1.4 and and 1.5 for examples.

Deterministic encodings In contrast to `nondet` encodings, where we determinize on the fly, in `det` encodings we already have the states of $\mathcal{A}_\varphi^d$ and we can encode them directly. We introduce 14 novel deterministic Promela encodings that presume $\mathcal{A}^d$ has been minimized and determinized using assignment-based encoding. We use two ways to encode the states. First, we can encode states by using Promela variable, whose value, a (`number`), refers to the current states. Second, we can use Spin's standard state-label format coupled with `goto` statements to transition between states. We illustrate each of these two state representations in turn.

The `back_det` encoding uses state numbers. The `never` claim first calculates the `system_state_index`, the integer corresponding to the current valuation of the system variables. Like its `back_nondet` counterpart, it transitions by checking for an enabled incoming transition from the current state. The `front_det_switch_number_fin` encoding uses a series of `if` statements, the closest Promela construction to a C-like `switch` statement, to check for enabled outgoing transitions from the current state. See Listing 1.6, Listings 1.7, and 1.8 for examples.

Alternatively, we can encode the `never` claim without using any state numbers, by taking advantage of Promela's constructs for representing automata states. The `front_det_switch_state_inf` encoding transitions to program labels corresponding to the names of the states in $\mathcal{A}_\varphi^d$. The initial state is labeled "`init`" and appears first, the accepting state is labeled "`accept`," and all other states are assigned unique names. See Listings 1.9 and 1.9 for examples.

<table>
<tr><th>State Minimization</th><th>Alphabet Representation</th><th>Automaton Acceptance</th><th>Never Claim Encoding</th><th>State Representation</th></tr>
<tr><td>no</td><td>BDDs</td><td rowspan="4">finite

infinite</td><td>front_nondet
back_nondet</td><td rowspan="3">number</td></tr>
<tr><td rowspan="3">yes</td><td rowspan="2">assignments</td><td>front_nondet
back_nondet
back_det
front_det_memory_table</td></tr>
<tr><td>front_det_switch</td><td>state
number</td></tr>
<tr><td>assignments with edge abbreviation</td><td>back_det</td><td>number</td></tr>
</table>

Table 1. The configuration space for generating `never` claims. Each row in the table represents an encoding configuration. Componenets of the winning encoding are bolded.

Look-Up Tables The above encodings represent automaton transition functions as `if` statements. Alternatively, we can declare a state look-up table in memory storing

the next state as a function of the current state and the `system_state_index`. This forms very compact `never` claims and the next state can be found in one operation. The `front_det_memory_table` encoding declares the table directly as a one-dimensional, row-major array. See Listing 1.11 for an example.

Configuration space The different options allow 26 possible combinations for generating `never` claims, summarized in Table 1.

4 Experimental Method

Platform We ran all tests on the Shared University Grid at Rice (SUG@R), an Intel Xeon compute cluster.[3] SUG@R is comprised of 134 SunFire x4150 nodes, each with two quad-core Intel Xeon processors running at 2.83GHz and 16GB of RAM per processor. The OS is Red Hat Enterprise 5 Linux, 2.6.18 kernel. Each test was run with exclusive access to one node. Times were measured using the Unix `time` command.

4.1 Model-Scaling Benchmarks

We chose a set of 14 typical safety formulas, taken from related literature, listed in Table 3. We model checked them against scaled linearly-sized Universal Models (UM) from [27]. (See also Appendix B.) By scaling up the size of these UMs to dwarf the sizes of the safety formulas, we create difficult model-checking benchmarks.

0	$\Box\neg bad$	"Something bad never happens."
1	$\Box(request \rightarrow \mathcal{X}\, grant)$	"Every request is immediately followed by a grant"
2	$\Box(\neg(p \wedge q))$	Mutual Exclusion: "p and q can never happen at the same time."
3	$\Box(p \rightarrow (\mathcal{X}\,\mathcal{X}\,\mathcal{X}\, q))$	"Always, p implies q will happen 3 time steps from now."
4*	$\mathcal{X}\,((p \wedge q)\mathcal{R}\, r)$	"Condition r must stay on until buttons p and q are pressed at the same time."
5*	$\mathcal{X}\,(\Box(p))$	slightly modified *intentionally safe* formula from [19]
6*	$\mathcal{X}\,(\Box(q \vee \mathcal{X}\Box p) \wedge \Box(r \vee \mathcal{X}\Box\neg p))$	slightly modified *accidentally safe* formula from [19]
7*	$\mathcal{X}\,([\Box(q \vee \Diamond\Box p) \wedge \Box(r \vee \Diamond\Box\neg p)] \vee \Box q \vee \Box r)$	slightly modified *pathologically safe* formula from [19]
8	$\Box(p \rightarrow (q \wedge \mathcal{X}\, q \wedge \mathcal{X}\,\mathcal{X}\, q))$	safety specification from [31]
9	$((((((p0\mathcal{R}(\neg p1))\mathcal{R}(\neg p2))\mathcal{R}(\neg p3))\mathcal{R}(\neg p4))\mathcal{R}(\neg p5))$	Sieve of Erathostenes [13, 21]
10	$(\Box((p0 \wedge \neg p1) \rightarrow (\Box\neg p1 \vee (\neg p1\,\mathcal{U}\,(p10 \wedge \neg p1)))))$	G.L. Peterson's algorithm for mutual exclusion algorithm [25, 22, 13, 24, 21]

[3] `http://rcsg.rice.edu/sugar/`

11	$(\Box(\neg p0 \rightarrow ((\neg p1 \mathcal{U} p0) \vee \Box\neg p1)))$	CORBA General Inter-Orb Protocol [17, 21]
12	$((\Box(p1 \rightarrow \Box(\neg p1 \rightarrow (\neg p0 \wedge \neg p1)))) \wedge (\Box(p2 \rightarrow \Box(\neg p2 \rightarrow (\neg p0 \wedge \neg p1)))) \wedge (\Box\neg p2 \vee (\neg p2 \mathcal{U} p1)))$	GNU i-protocol, also called iprot [5, 24, 21]
13	$((\Box(p1 \rightarrow \Box(\neg p1 \rightarrow (\neg p0 \wedge \neg p1)))) \wedge (\Box(p2 \rightarrow \Box(\neg p2 \rightarrow (\neg p0 \wedge \neg p1)))) \wedge (\Box\neg p2 \vee (\neg p2 \mathcal{U} p1)))$	Sliding Window protocol [16, 21]

Table 3. Industrial safety formulas used in model-scaling benchmarks.

For each of the formulas in Table 3, we model checked against a series of linearly-sized UMs, described in [27], starting with the 10-variable UM and scaling up the number of variables in the model, thereby exponentially increasing its state space. We used two configurations of UMs; starred formulas are checked against UMs that set all variables to *true* first; see Appendix B.

4.2 Formula-Scaling Benchmarks

For our formula-scaling benchmarks, we model checked each formula against a universal model with 30 variables and 1,073,741,824 states. We employed two types of formula-scaling benchmarks: random and syntactically safe random. We scaled each of the formulas until model checking became unachievable within machine bounds of timeout/spaceout.

We generated two sets each of 500 m-length safety specifications over n atomic propositions, for m in $\{5,10,15,20,25\}$ and n in $\{2..6\}$ (25,000 random formulas in these two benchmark sets, combined). The probability of each temporal operator was $P = 0.5$. For the first set, we generated syntactic safety formulas, allowing negation only directly before atomic propositions and limiting the temporal operators to $\{X, G, R\}$. For the second set, we generated each specification randomly over the full syntax of LTL. We then checked if the generated specification represented a safety property using the SPOT command `ltl2tgba -O`, adding the specification to our test set if so and rejecting it if not.

Test Method We encoded every benchmark LTL formula as a set of Promela `never` claims using SPOT and our novel encodings. We experimented with scheck [21] encodings; that tool produced too many bugs to be included. However, it is reasonable to assume that the results would not be comparable to our best encoding since the algorithm implemented by scheck constructs a nondeterministic finite automaton from the restricted closure of the formula that accepts precisely the informative prefixes of the formula and then determinizes as a last step without employing optimizations that we found particularly influential, such as minimization or edge abbreviation. Each `never` claim, was model checked by Spin.[4]

We measured model checking time separately from the times for various compilation stages. This is important for two reasons. It is relevant for regression testing and system debugging applications where the system is repeatedly changed but model checked against the same specifications. It is also essential for demonstrating our claim that deterministic encoding of LTL safety formulas can reduce model checking time; it is clear that we are not, for example, encoding LTL formulas in a manner that compiles more quickly but requires the same or more time to model check than the equivalent SPOT-encoding.

Figure 1 depicts the Spin model checking process. Unlike previous works, which report only the total time required for analysis via Spin, we measure the time required for compilation of

[4] We also investigated using the SPOT back-end; SPOT is unable to analyze Promela `never` claims at the time of this writing.

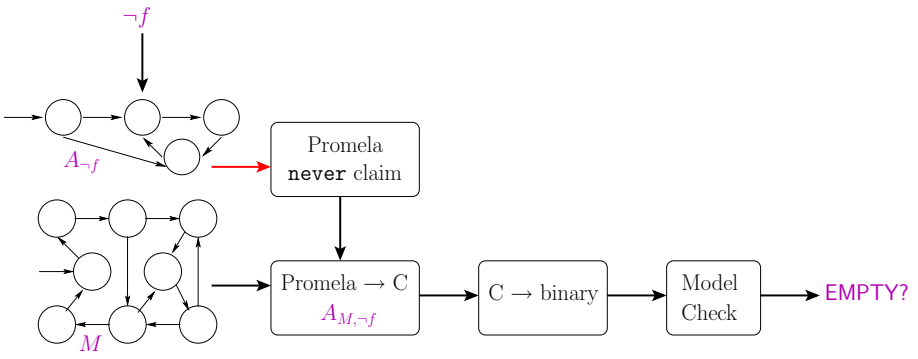


Fig. 1. System Diagram illustrating the Spin model checking process. We present an improved encoding for the LTL formula $\neg f$ to the Promela `never` claim $A_{\neg f}$.

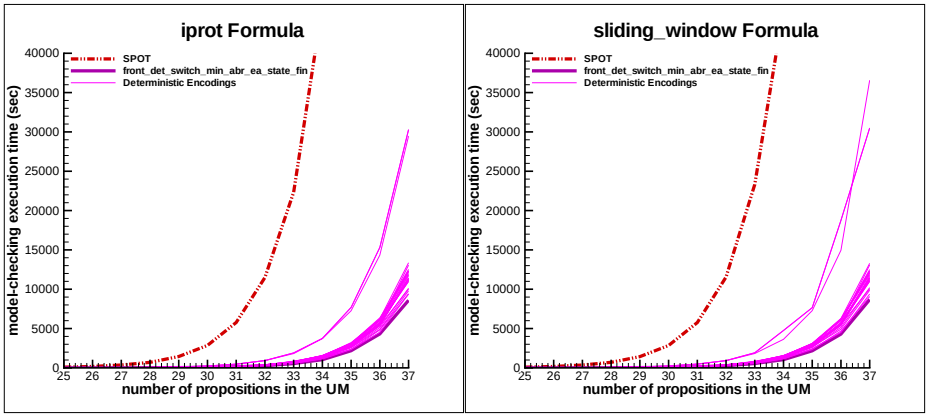
LTL-to-`never` claim (by either SPOT or CHIMP-Spin), `never` claim-to-C (via the `spin` command), and C-to-binary (via `gcc`) separately. In the following plots, we refer to the sum of these three times as *compile time* and separate this sum from from *model checking time*, or the time required to run the `pan` executable produced by Spin. Because we ran SPOT as a step in the creation of each of our new encodings, the specification automaton generation times incurred by our algorithm will always be greater than running SPOT alone. (It is important to note that our automaton generation times are consistently dwarfed by the corresponding model checking times.) To streamline regression testing, we argue that future versions of Spin should not require us to recompile `never` claims for each run of the model checker, even when they have not changed. Such an adjustment would more accurately reflect industrial applications of model checking and, combined with our reduced model checking times, reduce the amortized cost of model checking.

5 Experimental Results

Our experiments demonstrate that the new Promela `never` claims we have introduced significantly improve the translation of LTL safety formulas into explicit automata, as measured by model checking time. We found that one of our encodings is always best: `front_det_switch_min_abr_ea_state_fin`. Using this encoding, we can consistently improve on the model checking time required for SPOT encodings. We recommend using our `front_det_switch_min_abr_ea_state_fin` encoding for safety formulas and the standard SPOT encoding for non-safety formulas. (Recall that SPOT can test for safety formulas.)

We found certain encoding aspects to be always better. This helps explain why the `front_det_switch_min_abr_ea_state_fin` encoding is always the fastest: it is the encoding that combines all of the fastest `never` claim components. We found the following trends to hold: deterministic (`det`) `never` claims are faster than determinized-on-the-fly (`nondet`) `never` claims; finite acceptance (`fin`) is faster than infinite acceptance (`inf`); state labels (`state`) are faster than state numbers (`number`); minimized automata (`min`) are faster than not (`nomin`); edge abbreviation (`ea`) always equates to better performance. Note that deterministic encoding (`det`) enables faster features such as state minimization and edge abbreviation and that, all other encoding aspects being equal, there seems to be a positive correlation between the code size of a given `never` claim and the required model checking time, explaining the efficiency of this encoding. Also note that the (`front_det_switch`) encoding enables the faster state labels representation (`state`).

5.1 Model-Scaling Experimental Results



(a) Benchmarks for the iprot specification (formula 12).

(b) Benchmarks for the sliding window specification (formula 13).

Fig. 2. Model scaling benchmarks, showing the model-checking times based on the number of propositions in the UM.

Figure 2 demonstrates empirically that our deterministic automata require less time to model check than SPOT's nondeterministic automata. For some benchmarks, we found that all of our encodings, whether they determinized $\mathcal{A}^d$ up front or on the fly, required less model checking time than the equivalent nondeterministic SPOT `never` claims.[5] For example, for the iprot and sliding window benchmarks, pictured in Figures 2(a) and 2(b), all of our new encodings performed better than SPOT, though our `front_det_switch_min_abr_ea_state_fin` encoding was best. In these figures, the SPOT encoding is shown in red, our best encoding is shown in purple, and our 25 other encodings are shown in magenta. Note also that these plots demonstrate the orthogonality of automata size and model-checking time: all of our encodings represent the same automaton so the differences in model-checking times in these graphs stem entirely from the type of encoding and not the number of states in the automaton. Deterministic encodings can result in significant improvements in model checking performance by reducing calls to the internal nested depth-first search algorithm in the model checker; see Appendix A.1.

Figure 3 shows a speedup of a factor of two when using our best CHIMP-Spin encoding to model check our 14-formula workload against a 34-variable UM. Since we terminated the plot when the first benchmark formula exceeded machine bounds, this plot does not show instances where our encoding was able to scale to larger model checking benchmarks than the equivalent SPOT encoding. For example, Figure 2 demonstrates that our encoding was more scalable than SPOT's when model checking formulas 12 and 13.

Out of all of our benchmarks, the formula 4 benchmark displayed the smallest difference between our encoding and SPOT. For the 36-variable universal model, the SPOT `never` claim took 4606.94 seconds, or roughly 77 minutes whereas our `never` claim took 4281.22 seconds, or roughly 71 minutes Still, our `front_det_switch_min_abr_ea_state_fin` encoding encoding enabled Spin to scale to model check a 40-variable model whereas model checking the SPOT `never` claim timed out at 39 variables.

[5] Note that not all SPOT `never` claims are nondeterministic; for other benchmarks SPOT produced deterministic `never` claims.

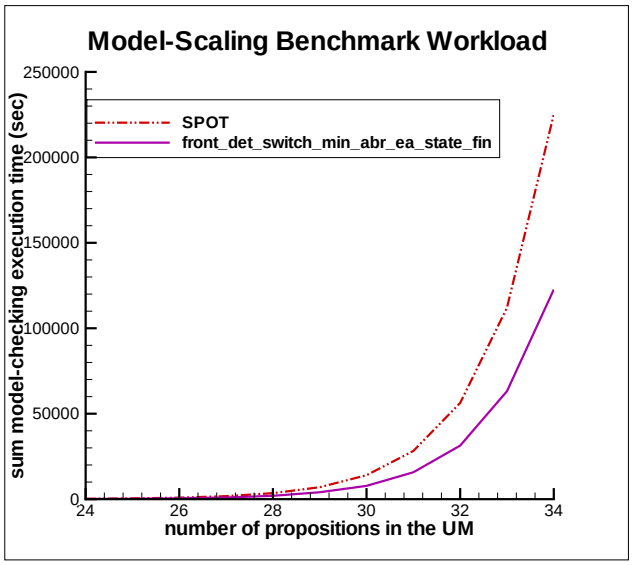


Fig. 3. Sums of the model-checking times for all model-scaling benchmark instances, based on the number of propositions in the UM.

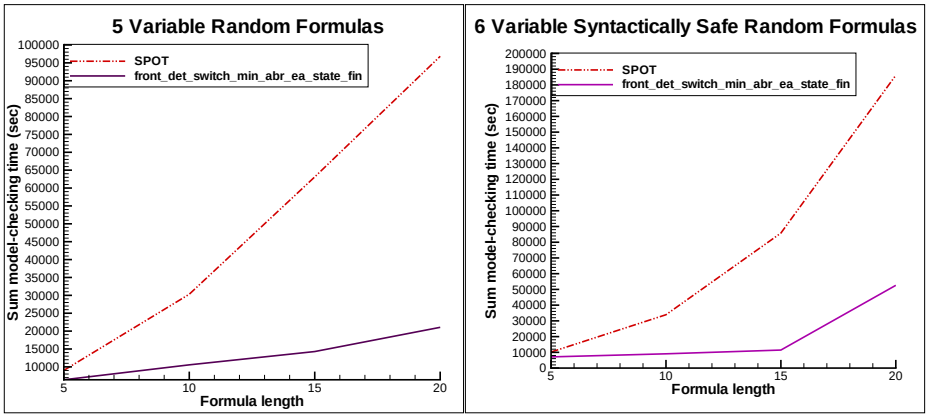
Since we call SPOT as a step in our encoding, our automaton generation times must always be higher than SPOT but compile times were consistently dwarfed by model checking times. Our total compile times were comparable to SPOT for our model-scaling benchmarks. For the set of 14 safety formulas in our workload, when model-checking against a 34-variable UM as shown in Figure 3, the sum of our compile times was 6.01 seconds (that breaks down into a sum of LTL-to-`never` claim times of 1.74 seconds, a sum of Promela-to-C times of 0.05 seconds, and a sum of C-to-binary times of 4.22 seconds), while the sum of our model-checking times was 122662.78 seconds. For SPOT encodings, the sum of compile times was 4.53 seconds (including a sum of LTL-to-`never` claim times of 0.14 seconds, a sum of Promela-to-C times of 0.06 seconds, and a sum of C-to-binary times of 4.33 seconds) with a sum of model-checking times of 225132.7 seconds. Note that the unix `time` command is not accurate to hundredths of a second so there is a potential for some error contributions in these sums.

5.2 Formula-Scaling Experimental Results

Figures 4(a) and 4(b) show the sums of the model checking times of randomly-generated safety formulas: completely randomly generated in Figure 4(a) and syntactically safe in Figure 4(b). Model checking times summed over all non-trivial randomly generated formulas for our best encoding were significantly lower than for SPOT encodings.

Since we call SPOT as a step in our encoding, our automaton generation times were always higher than SPOT but were consistently dwarfed by model checking times. This trend holds for syntactically safe random formulas as well. See Figure 5.2.

`BRICS Automaton` experienced some errors when encoding some randomly generated formulas. These were rare enough as to not significantly impact our timing results, i.e. for the set of 500 5-variable, 15-length random formulas in Figure 4(a), `BRICS Automaton` experienced



(a) Sum of model-checking times for 5 variable random formula benchmark.

(b) Sum of model-checking times for 6 variable syntactically safe benchmark.

Fig. 4. Graphs of sums of model-checking times for both categories of randomly-generated formulas, showing that our model checking times were consistently lower than SPOT.

nine errors. We summed data only for formulas where both the SPOT and CHIMP-Spin model checking runs completed without an error or timeout.

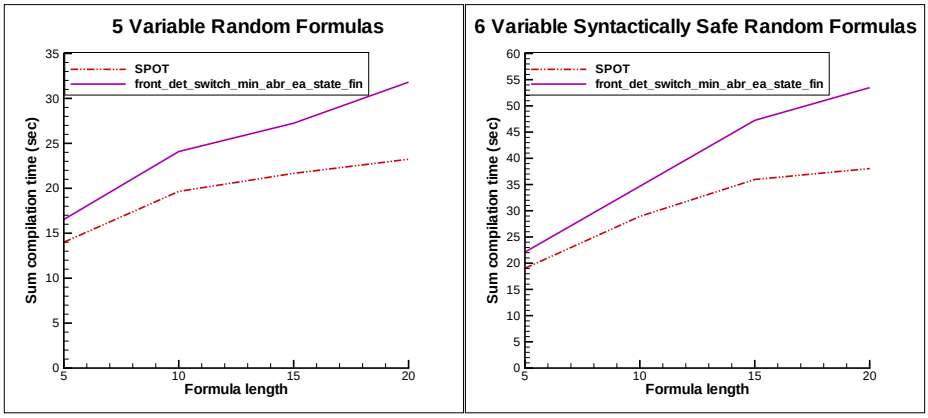
The difference in model checking time is not directly correlated with other statistics we measured, such as the length of counterexamples returned for formula violations. Across all of the randomly-generated formulas, we found that the number of states and the lengths of counterexamples associated with our `front_det_switch_min_abr_ea_state_fin` `never` claims and with SPOT's were usually very close, within a few states of each other. In general, the number of transitions had a higher variance between these two encodings; in the median cases, we ended up with less than or equal to the number of transitions in the equivalent SPOT `never` claim.

6 Discussion

In this paper we brought attention to the benefit of deterministic compilation for safety LTL properties. We defined novel encodings of safety LTL properties as deterministic `never` claims and showed that one encoding consistently leads to faster model-checking times than the state-of-the-art SPOT encoding or any of our other new encodings. Therefore, we recommend a multiple-pronged property-compilation approach to the Spin model checker: use SPOT for the compilation of non-safety properties and use deterministic compilation with our new `front_det_switch_min_abr_ea_state_fin` encoding for safety properties. This approach is *extensible*; different encodings of `never` claims may be appropriate for different types of LTL formulas, see [29].

Determinizing `never` claims for safety properties up front, rather than on-the-fly, seems to have a major effect on model-checking performance. While either method of determinizing yields better performance due to the simpler structure of the product search space, determinizing up front enables the use of other optimizations that improve performance: state labels (rather than numbers), state minimization, edge abbreviation. There is also a consistent time savings associated with model checking using finite acceptance conditions.

In general, deterministic compilation is more time consuming than nondeterministic compilation due to the need to determinize and minimize, though this overhead is dwarfed by the improvements in model-checking time. Still, our experiment revealed the `BRICS Automaton` tool



(a) Sum of compilation times for 5 variable random formula benchmark.

(b) Sum of compilation times for 6 variable syntactically safe benchmark.

Fig. 5. Sums of compilation times for both categories of randomly-generated formulas, showing that compilation times were dwarfed by model checking times. Note that the unix `time` command is not accurate to hundredths of a second; the times presented here may contain substantial error contributions. These graphs simply show that the sum of compile times over all formulas in a test set was always under a minute, for both SPOT and the best CHIMP-Spin encoding.

to be a slow link in our tool chain; improving this link is a subject for future research. In particular, we plan to investigate replacing the `BRICS Automaton` tool by (currently undocumented) determinization functions provided by SPOT. Also, for this paper we implemented our encoding as an extension of the CHIMP tool. However, in the future we would like to implement our best encoding more efficiently rather than relying on a modification of a tool created for a different purpose.

Finally, Kupferman and Lampert [18] developed an alternative approach to model checking of safety properties, which involves the construction of a nondeterministic finite-word automaton for bad prefixes. That approach may yield longer counterexamples, but it does not involve the theoretical additional exponential blow-up that is involved in the approached pursued here. A comparison with that approach is another subject for future research.

References

1. B. Alpern and F.B. Schneider. Recognizing safety and liveness. *Dist.Comp.*, 2:117–126, 1987.
2. R. Armoni, S. Egorov, R. Fraer, D. Korchemny, and M. Y. Vardi. Efficient LTL compilation for SAT-based model checking. In *ICCAD*, pages 877–884. IEEE, 2005.
3. R.E. Bryant. Symbolic Boolean manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
4. M. d'Amorim and G. Rosu. Efficient monitoring of ω-languages. In *CAV*, pages 364–378, 2005.
5. Y. Dong, X. Du, G. J. Holzmann, and S. A. Smolka. Fighting livelock in the GNU i-protocol: a case study in explicit-state model checking. *STTT*, 4(4):505–528, 2003.
6. A. Duret-Lutz and Denis Poitrenaud. SPOT: An extensible model checking library using transition-based generalized Büchi automata. In *MASCOTS*, pages 76–83. IEEE, 2004.

7. C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout. Reasoning with temporal logic on truncated paths. In *CAV*, pages 27–39, 2003.
8. E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 997–1072. Elsevier, MIT Press, 1990.
9. J. Geldenhuys and H. Hansen. Larger automata and less work for LTL model checking. In *SPIN*, volume 3925 of *LNCS*, pages 53–70. Springer, 2006.
10. R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of Linear Temporal Logic. In *PSTV*, pages 3–18. Chapman & Hall, 1995.
11. K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *TACAS*, pages 342–356. Springer, 2002.
12. G.J. Holzmann. The model checker Spin. *IEEE TSE*, 23(5):279–295, May 1997.
13. G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
14. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
15. L. Jategaonkar Jagadeesan, C. Puchol, and J. E. Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunications software. In *CAV*, volume 939 of *LNCS*, pages 127–140. Springer, 1996.
16. R. Kaivola. Using compositional preorders in the verification of sliding window protocol. In *CAV*, volume 1254 of *LNCS*, pages 48–59. Springer, 1997.
17. M. Kamel and S. Leue. Validation of a remote object invocation and object migration in CORBA GIOP using Promela/Spin. In *SPIN*, 1998.
18. O. Kupferman and R. Lampert. On the construction of fine automata for safety properties. In *ATVA*, pages 110–124, 2006.
19. O. Kupferman and M.Y. Vardi. Model checking of safety properties. *FMSD*, 19(3):291–314, Nov 2001.
20. O. Kupferman and M.Y. Vardi. Weak alternating automata are not that weak. *ACM TOCL*, 2(2):408–429, Jul 2001.
21. T. Latvala. Efficient model checking of safety properties. In *SPIN*, pages 74–88, 2003.
22. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.
23. A. Møller. dk.brics.automaton. http://www.brics.dk/automaton/, 2004.
24. R. Pelánek. BEEM: benchmarks for explicit model checkers. In *SPIN*, pages 263–267, 2007.
25. G. L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.
26. E. Plaku, L. E. Kavraki, and M. Y. Vardi. Falsification of LTL safety properties in hybrid systems. In *TACAS*, pages 368–382. Springer, 2009.
27. K.Y. Rozier and M.Y. Vardi. LTL satisfiability checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(2):123 – 137, March 2010.
28. S. Safra. On the complexity of ω-automata. In *FOCS*, pages 319–327, 1988.
29. K. Schneider. Improving automata generation for linear temporal logic by considering the automaton hierarchy. In *LPAR*, pages 39–54, London, UK, 2001. Springer-Verlag.
30. R. Sebastiani and S. Tonetta. "more deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In *CHARME*, volume 2860 of *LNCS*, pages 126–140. Springer, 2003.
31. D. Tabakov, K. Y. Rozier, and M. Y. Vardi. Optimized temporal monitors for SystemC. *Formal Methods in System Design*, page online, 2012.
32. M. Y. Vardi. From monadic logic to PSL. In *Pillars of Comp. Sci.*, volume 4800 of *LNCS*, pages 656–681. Springer, 2008.
33. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Comp. Sci.*, pages 332–344, Cambridge, Jun 1986.
34. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, Nov 1994.

Appendix A: Promela Code Examples

We show examples of our new Promela encodings of automata below.

A.1 Examples of the Winning Encoding: `front_det_switch_min_abr_ea_state_fin`

The encodings without edge abbreviation have as many as $2^{|\Sigma|}$ transitions per state, sometimes fewer if multiple valuations of Σ lead to automaton acceptance. We can improve model-checking performance of `never` claims for `det` encodings utilizing state labels by abbreviating the transitions. For finite `never` claims, our edge abbreviation algorithm can take advantage of the Promela semantics property that transitioning to a terminal error state and failing to find such a transition are equivalent. This enables us to further reduce the code size for finite-acceptance `never` claims by employing trap state elimination as we are abbreviating the edges. The `never` claim for our winning encoding, `front_det_switch_min_abr_ea_state_fin`, corresponding to benchmark formula 4 appears in Listing 1.1.

```
1  /*LTL formula: (!(X ((p0 & p1) R p2)))*/
2  never {
3  init_S2:
4    atomic {
5      if
6        :: (1) -> goto S0;
7      fi;
8    }
9  S0:
10   atomic {
11     if
12       :: (!p2) -> goto done;
13       :: ((!p0 && p2) || (!p1 && p2)) -> goto S0;
14     fi;
15   }
16 done: /*signal property violation by landing here*/
17   skip;
18 }
```

Listing 1.1. Illustrating the `front_det_switch_min_abr_ea_state_fin` `never` claim encoding of the benchmark formula 4

Deterministic encodings can result in significant improvements in model checking performance by reducing calls to the internal nested depth-first search algorithm in the model checker. Take for example the following variant of benchmark formula 6: $\square(q \vee x\square p) \wedge \square(r \vee x\square\neg p)$. The SPOT encoding for the corresponding `never` claim appears in Listing 1.2. As we increase the size of the universal model, the time required to model check this `never` claim increases exponentially.

```
1  never { // F((!p1 & XF!p0) | (!p2 & XFp0))
2  T0_init:
3    if
4    :: (!(p2)) -> goto accept_S2
5    :: ((1)) -> goto T0_S3
```

```
6     :: (!(p1)) -> goto accept_S4
7     fi;
8   accept_S2:
9     if
10    :: ((p0)) -> goto accept_all
11    :: (!(p0)) -> goto T0_S6
12    fi;
13  T0_S3:
14    if
15    :: (!(p2)) -> goto accept_S2
16    :: ((1)) -> goto T0_S3
17    :: (!(p1)) -> goto accept_S4
18    fi;
19  accept_S4:
20    if
21    :: (!(p0)) -> goto accept_all
22    :: ((p0)) -> goto T0_S7
23    fi;
24  T0_S6:
25    if
26    :: ((p0)) -> goto accept_all
27    :: (!(p0)) -> goto T0_S6
28    fi;
29  T0_S7:
30    if
31    :: (!(p0)) -> goto accept_all
32    :: ((p0)) -> goto T0_S7
33    fi;
34  accept_all:
35    skip
36  }
```

Listing 1.2. Illustrating the SPOT `never` claim for the original *accidentally safe* formula from [19], which we modified to form our benchmark formula 6.

However, if we encode this same `never` claim deterministically, the time required to model check this `never` claim remains near zero as we increase the size of the universal model. For comparison, the `front_det_switch_min_abr_ea_state_fin` encoding of the same formula from Listing 1.2 appears in Listing 1.3. Examine the initial state, `init_S1`, in Listing 1.3. In this case, Spin initially explores the valuation where the variables $p0$, $p1$, and $p2$ are false, in which case this `never` claim transitions directly to `done`, causing Spin to skip the NDFS in the emptiness check. It is the NDFS that causes the SPOT `never` claim to require exponentially increasing time to model check: note that the initial state in Listing 1.2 has no equivalent deterministic path to termination.

```
1  // LTL formula: (!([](p1 | (X [] p0)) & [](p2 | (X ([] ! p0)))))
2  never {
3  init_S1:
4    atomic {
5      if
6        :: (p2 && !p1 ) -> goto   S2;
```

```
7         :: (p1 && p2 )  -> goto   init_S1;
8         :: (p1 && !p2 ) -> goto   S0;
9         :: else -> goto done;
10      fi;
11    }
12  S0:
13    atomic {
14      if
15        :: (p1 && !p0 ) -> goto   S0;
16        :: else -> goto done;
17      fi;
18    }
19  S2:
20    atomic {
21      if
22        :: (p0 && p2 ) -> goto   S2;
23        :: else -> goto done;
24      fi;
25    }
26  done: // signal property violation by landing here
27    skip;
28  }
```

Listing 1.3. Illustrating the `front_det_switch_min_abr_ea_state_fin never` claim for the original *accidentally safe* formula from [19], which we modified to form our benchmark formula 6.

A.2 Examples of Nondeterministic Encodings

```
1  /*LTL formula: (!(X ((p0 & p1) R p2)))*/
2  int i = 0;
3  bool not_stuck = false;
4
5  /*Declare state arrays; they are automatically initialized to 0*/
6  bool current_state [3];
7  bool next_state [3];
8  never {
9    /*This next line happens in time -1; one step before the first
10     step of the system model*/
11   next_state[2] = 1; /*initialize current to the initial state*/
12
13   do
14     :: atomic{
15       /*First, swap of current_state and next_state*/
16       i = 0;
17       do
18         :: (i < 3 ) ->
19            current_state[i] = next_state[i];
20            i++;
```

```
21          :: (i >= 3) -> break;
22        od;
23        /*reset next_state*/
24        i = 0;
25        do
26          :: (i < 3) ->
27             next_state[i] = 0;
28             i++;
29          :: (i >= 3) -> break;
30        od;
31        /*Second, fill in next_state array*/
32        if
33          :: current_state[2] ->
34             if
35               :: (1  )
36                  -> next_state[1] = 1;
37               :: else -> skip;
38             fi;
39          :: else -> skip;
40        fi;
41        if
42          :: current_state[0] ->
43             if
44               :: (1  )
45                  -> next_state[0] = 1;
46               :: else -> skip;
47             fi;
48          :: else -> skip;
49        fi;
50        if
51          :: current_state[1] ->
52             if
53               :: (p0 && p1 && p2  )
54                  -> next_state[0] = 1;
55               :: else -> skip;
56             fi;
57             if
58               :: ((p2 && !p0) || (p2 && !p1)  )
59                  -> next_state[1] = 1;
60               :: else -> skip;
61             fi;
62          :: else -> skip;
63        fi;
64        /*Third, check if we're stuck*/
65        i = 0;
66        not_stuck = false;
67        do
68          :: (i < 3) ->
69             not_stuck = not_stuck || next_state[i];
70             i++;
```

```
        :: (i >= 3) -> break;
      od;
      if
        :: (! not_stuck) -> break;
        :: else -> skip;
      fi;
    }
  od;
}
```

Listing 1.4. Illustrating the `front_nondet_nomin_bdd_number_fin never` claim encoding of formula 4. Note this encoding utilizes finite acceptance.

```
/* LTL formula: (!(X ((p0 & p1) R p2)))*/
int i = 0;

/*Declare state arrays
  They are automatically initialized to 0*/
bool current_state [3];
bool next_state [3];
never {

S0_init: /*initialize current here*/
atomic {
  current_state[0] = 1;
  next_state[0] = 0;
  next_state[1] = ( current_state[2] && (p0 && p1 && p2)) ||
                  ( current_state[1] && (1));
  next_state[2] = ( current_state[0] && (1)) ||
                  ( current_state[2] && ((p2&&!p0)||(p2&&!p1)));

  /* if any next state is enabled, loop */
  /* Note that this if-statement will choose nondeterministically
     from among the true guards, but that's OK since multiple
     guards go to the same place*/
  if
    :: next_state[0] -> goto S1;
    :: next_state[1] -> goto S1;
    :: next_state[2] -> goto S1;
    :: else -> goto accept_all;
  fi;
  }

S1: /*loop here forever if property holds*/
atomic {
  /*update: current_state = next_state*/
  i = 0;
  do
      :: (i < 3) ->
          current_state[i] = next_state[i];
```

```
          i++;
      :: (i >= 3) -> break;
  od;

  next_state[0] = 0;
  next_state[1] = ( current_state[2] && (p0 && p1 && p2)) ||
                  ( current_state[1] && (1));
  next_state[2] = ( current_state[0] && (1)) ||
                  ( current_state[2] && ((p2&&!p0)||(p2&&!p1)));

  /* if any next state is enabled, loop */
  if
    :: next_state[0] -> goto S1;
    :: next_state[1] -> goto S1;
    :: next_state[2] -> goto S1;
    :: else -> goto accept_all;
  fi;
  }

accept_all: /*signal property violation by omega-looping here*/
  skip;
}
```

Listing 1.5. Illustrating the back_nondet_min_bdd_number_inf encoding of formula 4. Note this encoding utilizes infinite acceptance.

A.3 Examples of Deterministic Encodings

```
/* LTL formula: (!(X ((p0 & p1) R p2)))*/
int current_state = 2;
int next_state = 2;
int system_state_index = 0;
never {
  next_state = 2; /*initialize current to the initial state here*/

  do
    :: atomic {
      current_state = next_state; /*update state*/
      next_state = -1; /*reset*/

      /*Calculate the system state index*/
      system_state_index = 0; /*reset*/
      system_state_index=system_state_index+ ((p0) -> (1 << 2):0);
      system_state_index=system_state_index+ ((p1) -> (1 << 1):0);
      system_state_index=system_state_index+ ((p2) -> (1 << 0):0);
      if
        :: (((current_state == 2) && (system_state_index == 5)) ||
            ((current_state == 2) && (system_state_index == 7)) ||
            ((current_state == 0) && (system_state_index == 1)) ||
            ((current_state == 0) && (system_state_index == 5)) ||
```

```
            ((current_state == 2) && (system_state_index == 6)) ||
            ((current_state == 2) && (system_state_index == 0)) ||
            ((current_state == 2) && (system_state_index == 3)) ||
            ((current_state == 2) && (system_state_index == 4)) ||
            ((current_state == 0) && (system_state_index == 3)) ||
            ((current_state == 2) && (system_state_index == 1)) ||
            ((current_state == 2) && (system_state_index == 2)))
           -> next_state = 0;
        :: (((current_state == 0) && (system_state_index == 7)) ||
            ((current_state == 1) && (system_state_index == 0)) ||
            ((current_state == 1) && (system_state_index == 1)) ||
            ((current_state == 1) && (system_state_index == 2)) ||
            ((current_state == 1) && (system_state_index == 3)) ||
            ((current_state == 1) && (system_state_index == 4)) ||
            ((current_state == 1) && (system_state_index == 5)) ||
            ((current_state == 1) && (system_state_index == 6)) ||
            ((current_state == 1) && (system_state_index == 7)))
           -> next_state = 1;
        :: else break;
      fi;
    }
  od;
}
```

Listing 1.6. Illustrating the back_det_min_abr_number_fin encoding of formula 4.

```
/* LTL formula: (!(X ((p0 & p1) R p2)))*/
int current_state = 2;
int next_state = 2;
int system_state_index = 0;
never {
  next_state = 2; /*initialize current to the initial state here*/

  do
    :: atomic {
       current_state = next_state; /*update state*/
       next_state = -1; /*reset*/

       /*Calculate the system state index*/
       system_state_index = 0; /*reset*/
       system_state_index=system_state_index+((p0) -> (1 << 2):0);
       system_state_index=system_state_index+((p1) -> (1 << 1):0);
       system_state_index=system_state_index+((p2) -> (1 << 0):0);
       if
         :: (current_state == 2) ->
            if
              :: (system_state_index == 5 )
                 -> next_state =   0;
              :: (system_state_index == 7 )
                 -> next_state =   0;
```

```
25                :: (system_state_index == 6 )
26                   -> next_state =   0;
27                :: (system_state_index == 0 )
28                   -> next_state =   0;
29                :: (system_state_index == 3 )
30                   -> next_state =   0;
31                :: (system_state_index == 4 )
32                   -> next_state =   0;
33                :: (system_state_index == 1 )
34                   -> next_state =   0;
35                :: (system_state_index == 2 )
36                   -> next_state =   0;
37                :: else break;
38             fi;
39          :: (current_state == 0) ->
40             if
41               :: (system_state_index == 7 )
42                  -> next_state =   1;
43               :: (system_state_index == 1 )
44                  -> next_state =   0;
45               :: (system_state_index == 5 )
46                  -> next_state =   0;
47               :: (system_state_index == 3 )
48                  -> next_state =   0;
49               :: else break;
50             fi;
51          :: (current_state == 1) ->
52             if
53               :: (system_state_index == 0 )
54                  -> next_state =   1;
55               :: (system_state_index == 1 )
56                  -> next_state =   1;
57               :: (system_state_index == 2 )
58                  -> next_state =   1;
59               :: (system_state_index == 3 )
60                  -> next_state =   1;
61               :: (system_state_index == 4 )
62                  -> next_state =   1;
63               :: (system_state_index == 5 )
64                  -> next_state =   1;
65               :: (system_state_index == 6 )
66                  -> next_state =   1;
67               :: (system_state_index == 7 )
68                  -> next_state =   1;
69               :: else break;
70             fi;
71       fi;
72       }
73   od;
```

```
}
```

Listing 1.7. Illustrating the `front_det_switch_number_fin never` claim encoding of formula 4.

```
/* LTL formula: (!(X ((p0 & p1) R p2)))*/
int current_state = 2;
int next_state = 2;
int system_state_index = 0;
never {

S0_init:/*initialize current here*/
  atomic {
  current_state = 2;

  /*Calculate the system state index*/
  system_state_index = 0; /*reset*/
  system_state_index = system_state_index + ((p0) -> (1 << 2):0);
  system_state_index = system_state_index + ((p1) -> (1 << 1):0);
  system_state_index = system_state_index + ((p2) -> (1 << 0):0);
  if
      :: (system_state_index == 5 )
        -> next_state =    0; goto S1;
      :: (system_state_index == 7 )
        -> next_state =    0; goto S1;
      :: (system_state_index == 6 )
        -> next_state =    0; goto S1;
      :: (system_state_index == 0 )
        -> next_state =    0; goto S1;
      :: (system_state_index == 3 )
        -> next_state =    0; goto S1;
      :: (system_state_index == 4 )
        -> next_state =    0; goto S1;
      :: (system_state_index == 1 )
        -> next_state =    0; goto S1;
      :: (system_state_index == 2 )
        -> next_state =    0; goto S1;
      :: else
        -> goto accept_stuck;
  fi;
  }

S1: /*loop here forever if property holds*/
  atomic {
  current_state = next_state; /*update state*/

  /*Calculate the system state index*/
  system_state_index = 0; /*reset*/
  system_state_index = system_state_index + ((p0) -> (1 << 2):0);
  system_state_index = system_state_index + ((p1) -> (1 << 1):0);
```

```
46    system_state_index = system_state_index + ((p2) -> (1 << 0):0);
47    if
48      :: (current_state == 2) ->
49        if
50        :: (system_state_index == 5 )
51          -> next_state =    0; goto S1;
52        :: (system_state_index == 7 )
53          -> next_state =    0; goto S1;
54        :: (system_state_index == 6 )
55          -> next_state =    0; goto S1;
56        :: (system_state_index == 0 )
57          -> next_state =    0; goto S1;
58        :: (system_state_index == 3 )
59          -> next_state =    0; goto S1;
60        :: (system_state_index == 4 )
61          -> next_state =    0; goto S1;
62        :: (system_state_index == 1 )
63          -> next_state =    0; goto S1;
64        :: (system_state_index == 2 )
65          -> next_state =    0; goto S1;
66        :: else
67          -> goto accept_stuck;
68        fi;
69      :: (current_state == 0) ->
70        if
71        :: (system_state_index == 7 )
72          -> next_state =    1; goto S1;
73        :: (system_state_index == 1 )
74          -> next_state =    0; goto S1;
75        :: (system_state_index == 5 )
76          -> next_state =    0; goto S1;
77        :: (system_state_index == 3 )
78          -> next_state =    0; goto S1;
79        :: else
80          -> goto accept_stuck;
81        fi;
82      :: (current_state == 1) ->
83        if
84        :: (system_state_index == 0 )
85          -> next_state =    1; goto S1;
86        :: (system_state_index == 1 )
87          -> next_state =    1; goto S1;
88        :: (system_state_index == 2 )
89          -> next_state =    1; goto S1;
90        :: (system_state_index == 3 )
91          -> next_state =    1; goto S1;
92        :: (system_state_index == 4 )
93          -> next_state =    1; goto S1;
94        :: (system_state_index == 5 )
95          -> next_state =    1; goto S1;
```

```
96        :: (system_state_index == 6 )
97          -> next_state =    1; goto S1;
98        :: (system_state_index == 7 )
99          -> next_state =    1; goto S1;
100       :: else
101         -> goto accept_stuck;
102       fi;
103   fi;
104   }
105 accept_stuck: /*signal property violation by omega-looping here*/
106   skip;
107 }
```

Listing 1.8. Illustrating the `front_det_switch_number_inf never` claim encoding of formula 4. It employs the Promela acceptance-cycle acceptance condition.

```
1  /* LTL formula: (!(X ((p0 & p1) R p2)))*/
2  int system_state_index = 0;
3  never {
4
5  init_S2:
6    atomic {
7      system_state_index = 0; /*reset*/
8      system_state_index= system_state_index + ((p0) -> (1 << 2):0);
9      system_state_index= system_state_index + ((p1) -> (1 << 1):0);
10     system_state_index= system_state_index + ((p2) -> (1 << 0):0);
11     if
12       :: (system_state_index == 5 )
13         -> goto   S0;
14       :: (system_state_index == 7 )
15         -> goto   S0;
16       :: (system_state_index == 6 )
17         -> goto   S0;
18       :: (system_state_index == 0 )
19         -> goto   S0;
20       :: (system_state_index == 3 )
21         -> goto   S0;
22       :: (system_state_index == 4 )
23         -> goto   S0;
24       :: (system_state_index == 1 )
25         -> goto   S0;
26       :: (system_state_index == 2 )
27         -> goto   S0;
28       :: else
29         -> goto accept_stuck;
30     fi;
31   }
32 S0:
33   atomic {
34     system_state_index = 0; /*reset*/
```

```
35      system_state_index= system_state_index + ((p0) -> (1 << 2):0);
36      system_state_index= system_state_index + ((p1) -> (1 << 1):0);
37      system_state_index= system_state_index + ((p2) -> (1 << 0):0);
38      if
39        :: (system_state_index == 7 )
40          -> goto   S1;
41        :: (system_state_index == 1 )
42          -> goto   S0;
43        :: (system_state_index == 5 )
44          -> goto   S0;
45        :: (system_state_index == 3 )
46          -> goto   S0;
47        :: else
48          -> goto accept_stuck;
49      fi;
50    }
51  S1:
52    atomic {
53      system_state_index = 0; /*reset*/
54      system_state_index= system_state_index + ((p0) -> (1 << 2):0);
55      system_state_index= system_state_index + ((p1) -> (1 << 1):0);
56      system_state_index= system_state_index + ((p2) -> (1 << 0):0);
57      if
58        :: (system_state_index == 0 )
59          -> goto   S1;
60        :: (system_state_index == 1 )
61          -> goto   S1;
62        :: (system_state_index == 2 )
63          -> goto   S1;
64        :: (system_state_index == 3 )
65          -> goto   S1;
66        :: (system_state_index == 4 )
67          -> goto   S1;
68        :: (system_state_index == 5 )
69          -> goto   S1;
70        :: (system_state_index == 6 )
71          -> goto   S1;
72        :: (system_state_index == 7 )
73          -> goto   S1;
74        :: else
75          -> goto accept_stuck;
76      fi;
77    }
78  accept_stuck: /*signal property violation by omega-looping here*/
79    skip;
80  }
```

Listing 1.9. Illustrating the `front_det_switch_min_abr_state_inf never` claim encoding of formula 4. This version employs the Promela notion of states and the Promela acceptance-cycle acceptance condition.

```
/* LTL formula: (!(X ((p0 & p1) R p2)))*/
int system_state_index = 0;
never {

init_S2:
  atomic {
    system_state_index = 0; /*reset*/
    system_state_index= system_state_index + ((p0) -> (1 << 2):0);
    system_state_index= system_state_index + ((p1) -> (1 << 1):0);
    system_state_index= system_state_index + ((p2) -> (1 << 0):0);
    if
      :: (system_state_index == 5 )
        -> goto   S0;
      :: (system_state_index == 7 )
        -> goto   S0;
      :: (system_state_index == 6 )
        -> goto   S0;
      :: (system_state_index == 0 )
        -> goto   S0;
      :: (system_state_index == 3 )
        -> goto   S0;
      :: (system_state_index == 4 )
        -> goto   S0;
      :: (system_state_index == 2 )
        -> goto   S0;
      :: (system_state_index == 1 )
        -> goto   S0;
      :: else -> goto done;
    fi;
  }
S0:
  atomic {
    system_state_index = 0; /*reset*/
    system_state_index= system_state_index + ((p0) -> (1 << 2):0);
    system_state_index= system_state_index + ((p1) -> (1 << 1):0);
    system_state_index= system_state_index + ((p2) -> (1 << 0):0);
    if
      :: (system_state_index == 7 )
        -> goto   S1;
      :: (system_state_index == 1 )
        -> goto   S0;
      :: (system_state_index == 5 )
        -> goto   S0;
      :: (system_state_index == 3 )
        -> goto   S0;
      :: else -> goto done;
    fi;
  }
S1:
  atomic {
```

```
51      system_state_index = 0; /*reset*/
52      system_state_index= system_state_index + ((p0) -> (1 << 2):0);
53      system_state_index= system_state_index + ((p1) -> (1 << 1):0);
54      system_state_index= system_state_index + ((p2) -> (1 << 0):0);
55      if
56        :: (system_state_index == 0 )
57          -> goto   S1;
58        :: (system_state_index == 1 )
59          -> goto   S1;
60        :: (system_state_index == 2 )
61          -> goto   S1;
62        :: (system_state_index == 3 )
63          -> goto   S1;
64        :: (system_state_index == 4 )
65          -> goto   S1;
66        :: (system_state_index == 5 )
67          -> goto   S1;
68        :: (system_state_index == 6 )
69          -> goto   S1;
70        :: (system_state_index == 7 )
71          -> goto   S1;
72        :: else -> goto done;
73      fi;
74    }
75  done: /*signal property violation by landing here*/
76    skip;
77  }
```

Listing 1.10. Illustrating the `front_det_switch_min_abr_state_fin never` claim encoding of formula 4.

```
1   /* LTL formula: (!(X ((p0 & p1) R p2)))*/
2   int current_state = 0;
3   int next_state = 0;
4   int system_state_index = 0;
5   int table[24];
6   never {
7
8   S0_init:/*initialize current here*/
9     atomic {
10    table[0] = 2;   table[1] = 2;   table[2] = 2;   table[3] = 2;
11    table[4] = 2;   table[5] = 2;   table[6] = 2;   table[7] = 2;
12    table[8] = 1;   table[9] = 1;   table[10] = 1;  table[11] = 1;
13    table[12] = 1;  table[13] = 1;  table[14] = 1;  table[15] = 1;
14    table[16] = -1; table[17] = 2;  table[18] = -1; table[19] = 2;
15    table[20] = -1; table[21] = 2;  table[22] = -1; table[23] = 1;
16
17    /*Calculate the system state index*/
18    system_state_index = 0; /*reset*/
19    system_state_index = system_state_index + ((p0) -> (1 << 2):0);
```

```
20    system_state_index = system_state_index + ((p1) -> (1 << 1):0);
21    system_state_index = system_state_index + ((p2) -> (1 << 0):0);
22
23    /*Lookup the next state in the table*/
24    next_state = table[current_state * 8 + system_state_index];
25    if
26      :: (next_state == -1) -> goto accept_stuck;
27      :: else -> goto S1;
28    fi;
29    }
30
31  S1: /*loop here forever if property holds*/
32    atomic {
33    current_state = next_state; /*update state*/
34    next_state = -1; /*reset*/
35
36    /*Calculate the system state index*/
37    system_state_index = 0; /*reset*/
38    system_state_index = system_state_index + ((p0) -> (1 << 2):0);
39    system_state_index = system_state_index + ((p1) -> (1 << 1):0);
40    system_state_index = system_state_index + ((p2) -> (1 << 0):0);
41
42    /*Lookup the next state in the table*/
43    next_state = table[current_state * 8 + system_state_index];
44        if
45          :: (next_state == -1) -> goto accept_stuck;
46          :: else -> goto S1;
47        fi;
48    }
49
50  accept_stuck: /*signal property violation by omega-looping here*/
51    skip;
52  }
```

Listing 1.11. Illustrating the `front_det_memory_table_min_abr_inf` encoding of formula 4.

Appendix B: Universal Model from [27]

For each of the formulas in our benchmark sets we model check against univeral models that are linearly-sized in the number of atomic propositions as, described in [27]. For formula-scaling benchmarks we use a universal model with 30 variables and for model-scaling benchmarks we use a series of universal models starting with the 10-variable model and scaling up the number of variables in the model, thereby exponentially increasing its state space.

For all benchmarks, our universal system model is a Promela program that explicitly enumerates all possible evaluations over *Prop* and employs nondeterministic choice to pick a new valuation at each time step. For example, when $Prop = \{p, q\}$, the Promela model is:

```
bool p,q;
active proctype generateValues()
{
  do
   :: atomic{
      if
         :: true -> p = 0;
         :: true -> p = 1;
      fi;
      if
         :: true -> q = 0;
         :: true -> q = 1;
      fi;
   }
  od
}
```

Starred formulas are checked against universal models that set all variables to *true* first like this:

```
   bool p,q;
   active proctype generateValues()
   { do
      :: atomic{
         if
            :: true -> p = 1;
            :: true -> p = 0;
         fi;
         if
            :: true -> q = 1;
            :: true -> q = 0;
         fi;   }
     od }
```