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

Applying Symbolic Bounded Model Checking to the 2012 RERS Greybox Challenge

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

[1]Department of Computer Science, University of Bristol, UK
`jeremy.morse@bristol.ac.uk`
[2]Electronic and Information Research Center, Federal University of Amazonas, Brazil
`lucascordeiro@ufam.edu.br`
[3]Electronics and Computer Science, University of Southampton, UK
`dan@ecs.soton.ac.uk`
[j]Division of Computer Science, Stellenbosch University, South Africa
`bfischer@cs.sun.ac.za`

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

Abstract We describe the application of ESBMC, a symbolic bounded model checker for C programs, to the 2012 RERS Greybox Challenge. We checked the reachability properties via reachability of the error labels, and the behavioral properties via a bounded LTL model checking approach. Our approach could solve about 700 properties for the small and medium problems from the offline phase, and scored overall about 5000 marks but still ranked last in the competition.

1 Introduction

Model checking has been used successfully to verify abstract system designs as well as actual software; applying it to the RERS Greybox Challenge is thus an obvious idea. Model checking comes in a variety of different techniques, but we use symbolic bounded software model checking, as implemented by our ESBMC model checker [7,8]. That is,

- we analyze the challenge programs directly (specifically, the C versions), not an abstract model that has been extracted separately;
- we (arbitrarily) bound the number of iterations of the main loop that we analyze and unroll the program accordingly;
- we generate a number of verification conditions (VCs) from the unrolled program, which we pass to a Satisfiability Modulo Theories (SMT) solver, instead of explicitly exploring the reachable state space of the original program.

We use this approach both for the reachability properties (in the usual way via checking the reachability of the error labels) and the behavioral properties (via our bounded LTL model checking approach [15,16]). However, it seems to be clear that symbolic bounded software model checking is not the optimal technique for the Challenge: the programs implement finite state machines with a relatively small state space, but bounding and unrolling under-approximates the reachable state space while at the same time the structure of the VCs over-approximates it. Similarly, the programs are much simpler than those typically encountered in software model checking (e.g., the offline problems only use integer equality and contain no other operations or data structures) while at the same time the large programs (approximately 70,000 to 180,000 lines of code) are too large to unroll them sufficiently often.

We only participated in the offline phase of the Challenge, and only attempted the small and medium problems (i.e., `Problem1` to `Problem6`). As expected, we did thus not score well, and came last in the competition, with a total score of 5061 marks. However, our main motivation for participating in the Challenge was to evaluate our bounded LTL model checking approach over a large, external benchmark set. Here, we fared well on the four problems we attempted: we correctly analyzed 385 out of the 400 properties, and scored close to 3000 marks on these properties alone.

The remainder of this paper is organized as follows. In the next section we briefly describe the applied tools and the challenge problems. In Sections 3 and 4 we give details of our approach to solving the reachability problems and the behavioral problems, respectively. The complete results are given in the appendix.

2 Experimental Set-Up

2.1 ESBMC

ESBMC is a context-bounded symbolic model checker that allows the verification of single- and multi-threaded C code with shared variables and locks. ESBMC supports full ANSI-C (as defined in ISO/IEC 9899:1990), and can verify programs that make use of bit-level operations, arrays, pointers, structs, unions, memory allocation and 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-specified assertions, although none of ESBMC's built-in checks are useful for the Challenge.

As a bounded model checker ESBMC checks (the negation of) a given property at a given depth: given a program, a property φ, and a bound k, BMC unrolls the program k times and translates it into a VC ψ such that ψ is satisfiable if and only if φ has a counterexample of length less than or equal to k. ESBMC uses a modified CBMC [5] frontend to unroll the program, to convert it into static single assignment (SSA) form, and to generate the VC(s), but it uses different background theories and passes them to an SMT solver, rather than a pure satisfiability (SAT) solver. ESBMC natively supports Z3 [10] and Boolector [4] but can also output the VCs using the SMTLib format. However, due to the simple structure of the challenge programs (see Section 2.3) the use of SMT solvers is of little advantage over plain propositional satisfiability solvers. For the Challenge we used ESBMC 1.21.1, which is available from `www.esbmc.org`.

2.2 Bounded LTL model checking

We have also extended (see [16] for details) context-bounded model checking to validate multithreaded C programs directly against linear-time temporal logic (LTL) formulae over expressions in the global variables of the C program under test. The key problem here is that a *bounded* model checker only explores finite prefixes of any possibly infinite traces produced by the program, while the LTL standard semantics is defined over infinite traces. We cannot simply cut the traces because the standard interpretation of the *next*-operator X requires the existence of a next state to hold. One possible approach is to systematically extend the finite traces, e.g., by infinite stuttering of their last state [14]. However, in a two-state logic, we cannot then distinguish between a formula that (truly) holds because we have seen a *good prefix* [13] and so *all* possible infinite continuations of the observed finite trace will be models as well, and one that (presumably) holds because we have merely not seen a *bad prefix* (i.e., a finite trace that *cannot* be prefix of a model) because we stutter the final state infinitely often. In order to achieve this distinction, we need to use a larger truth domain. Our extension is based on a four-valued domain which uses two additional truth values to interpret inconclusive (i.e., neither good nor bad) prefixes [3].

Formally, we consider the set of atomic propositions $Prop$ over the global variables of the C program and define $\Sigma = 2^{Prop}$. We use $u \in \Sigma^*$ to denote finite traces, $w \in \Sigma^\omega$ to denote infinite traces, and $a^\omega \in \Sigma$ to denote the infinite trace consisting of the letter $a \in \Sigma$ only. We can define the standard semantics of LTL [17] formulas via an interpretation function $[\_ \models \_]_\omega : \Sigma^\omega \times LTL \to \mathbb{B}_2$, where $\mathbb{B}_2 = \{\bot, \top\}$ is the standard truth domain. 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. Finite traces can be extended systematically by infinite stuttering of their last state [14] to extend the standard semantics to finite traces, i.e., $[u \models \varphi]_\infty = [uu_{n-1}^\omega \models \varphi]_\omega$ for a finite trace u of length n. However, this so-called the *infinite extension* semantics [2] cannot handle inconclusive prefixes properly, as sketched above. In order to achieve this, we introduce the larger truth domain $\mathbb{B}_4 = \{\bot, \bot^p, \top^p, \top\}$, with $\bot \sqsubseteq \bot^p \sqsubseteq \top^p \sqsubseteq \top$ [3], and then use the infinite extension semantics to resolve the inconclusive prefixes into *presumably good* (i.e., $\top^p$) or *presumably bad* (i.e., $\bot^p$). This *bounded trace semantics* is thus 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 traces $\mathcal{T}(P)$ of a program P 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., $[U \models \varphi]_B = \bigsqcap_{u \in U} [u \models \varphi]_B$ and say that φ *holds* (resp. *presumably holds*) *for a C program* P if $[\mathcal{T}(P) \models \varphi]_B = \top$ (resp. $\top^p$). Finally, we call P *good* (resp. *succeeding*, *failing*, or *bad*) wrt. φ if $[\mathcal{T}(P) \models \varphi]_B = \top$ (resp. $\top^p$, $\bot^p$, or $\bot$).

The usual approach [6, 12] to check LTL formulas converts their negation (the so-called *never claim*) into a non-deterministic Büchi automaton (BA), which is composed with the program; if the composed system admits an accepting run, the program violates the specified requirement. However, in order to implement the bounded trace semantics, we need to modify the approach. We thus first precompute a complete static analysis to determine which states are accepting under the different infinite extensions of the observed finite traces. This is feasible due to the relatively small size of the BAs produced by the `ltl2ba` [11] algorithm and tool[1] (which we modified to produce C code). We then check the combined system several times, with different assertions corresponding to the different acceptance criteria, based on different infinite extensions of the observed traces, to derive the correct truth value for the LTL. For each of these assertions our model-checker searches for a witness which violates the assertion; our program's overall "correctness" value is the weakest such value in $\mathbb{B}_4$ for which a witness can be found that violates the corresponding assertion.

2.3 Challenge problems

The challenge problems are all so-called event-condition-action systems, which are finite state transducers where the states are not given explicitly, but only implicitly by the possible valuations of a number of state variables. The implementations consist of a main loop, which in each iteration reads an input (i.e., event) from the standard input, updates the state variables, and possibly writes an output (i.e., action) to the standard output; the latter two are guarded by conditionals over the input, and over the values of the state variables.

The challenge problems all work with relatively small alphabets, and use five or (in most cases) six different input symbols, and between three and nine different output symbols. Easy and moderate problems have between four and eight state variables, while large problems have 30. The offline problems (`Problem1` to `Problem9`) have a much simpler structure than either the validation or the online problems. The programs for the offline problems only assign between two and five different integer constants to the state variables, and only use the equality and propositional operators in the guards. In contrast, the remaining programs (`Problem10` to `Problem19`) use arithmetic operators to update the state variables (but from their old values only, i.e., the new value does not depend on any of the other variables), and use the other operators in the guards.

The classification of the problems as *easy*, *moderate*, or *hard* remains opaque to us, although all three hard problems have substantially more state variables. However, from a *bounded* model checking point of view the primary issue is the

[1] with further improvements by Babiak et al. [1]

length of the shortest counterexample traces for the reachability properties, as this determines the necessary unwinding bounds.[2] In this view, at least some of the offline problems seem to be mis-classified. For the simple problems all reachable error labels require three to seven loop iterations, but the (supposedly) easy medium problem (`Problem4`) requires 17 to 21 loop iterations, while the moderate (`Problem5`) and hard (`Problem6`) versions require only eight and six iterations, respectively.

We only participated in the offline phase of the Challenge, and only attempted the small and medium problems (i.e., `Problem1` to `Problem6`); the large problems (`Problem7` to `Problem9`) are too large and broke our frontend. For the behavioral properties we only attempted `Problem1` to `Problem4`. We ran ESBMC on the C versions of the Challenge programs; the only modifications were to replace the input (`scanf`) by an appropriately constrained non-deterministic choice, and to prune (by means of an assumption on the computed output) executions that use invalid inputs.

2.4 Execution of Experiments

We ran all experiments on the Southampton IRIDIS compute cluster,which comprises about 1000 nodes, each with 12 2.4Ghz Intel Westmere cores and 22Gb of memory, running Red Hat Enterprise Linux Server release 5.3 (Tikanga). We submitted batches of 60 jobs, which where scheduled by IRIDIS' own job scheduling system. We set no time or memory limits for the jobs corresponding to the reachability properties, and a time limit of one hour (but no memory limit) for the jobs corresponding to the behavioral properties.

3 Checking the reachability properties

3.1 Approach

The very simple structure of the programs (i.e., no arithmetic, array, or memory operations) means that we only need to check the reachability of the explicit error labels to solve the reachability problems. Since ESBMC supports error labels, this is straightforward; for each label *lab*, we called ESBMC as follows (with unwind bound *n* dependant on the problem category):

```
esbmc --no-assertions --no-unwinding-assertions
      --unwind n --error-label lab problem.c
```

We ran ESBMC for each label separately, although this requires repeated unrolling and conversion of the same program; we suspect that we could improve our overall performance substantially if we checked for all labels in one batch (e.g., using Z3's context stack mechanism). Table 1 summarizes the results.

[2] Note that the internal program structure still plays a role: for the same unwinding bound the hard problems take one to two orders of magnitude longer than the easy or moderate ones; see Table 1 for details.

3.2 Small problems

The relatively small size of these programs (approximately 600 to 1600 lines of code) allowed us to unroll them quite aggressively. We iteratively deepened the unrolling bound until the results stabilized at $n = 7$ and then used a larger bound to double-check the results. For the easy and moderate programs (`Problem1` and `Problem2`) we were able to run ESBMC with $n = 50$, but the hard program (`Problem3`) produced larger and harder VCs requiring substantially longer SMT solver times, so we only used $n = 20$ here.

For the labels identified as reachable, ESBMC produces a counterexample trace, as usual in bounded model checking, from which a test input could be extracted; due to the simple structure of the challenge programs, we did not execute these inputs, but simply assumed them to be true counterexamples. We associated the maximum weighting of 9 marks with each of these labels.

For the other labels we interpreted the failure to reach this label within the given bound as sufficient evidence that it is indeed unreachable. We also used this strategy successfully in the TACAS software verification competition [9]. However, strictly speaking we should not make an equally strong claim, despite the large bounds we were able to explore (representing at least a 3-fold increase over the size of the counterexamples found with smaller bounds). We therefore "wagered" only a weighting of 6 marks for each of these lables. In the end, this turned out to be too cautious, since all of our results here where correct, and we achieved 408, 390, and 408 out of 549 possible marks for the three problems, respectively.

3.3 Medium problems

The substantially larger size of these programs (approximately 4800 to 9500 lines of code) means that we could unroll them only to much smaller bounds. We were able to analyze the easy problem (`Problem4`) at a bound of $n = 20$ and the moderate and hard problems at $n = 7$ before the calculations became intractable. However, for the moderate problem (`Problem5`) we were unable to find any reachable error labels for this bound. This is in marked contrast to the hard problem (`Problem6`), where we found 26 reachable error labels. We discounted the moderate results as an anomaly, because we were unable to resolve this situation during the Challenge, and submitted solutions only for the easy and hard problems (i.e., `Problem4` and `Problem6`). After the results were released, we realized that all reachable labels in `Problem5` require counterexample traces with at least eight inputs, which is just outside our chosen unwinding bound.

We used the same marking scheme as for the small problems; in particular, we kept a weighting of 6 marks for the problems where we did not find a counterexample within the bounds. This time our caution proved slightly more justified, as one of the labels (`error12`) of `Problem4` is reachable with 21 inputs, just outside chosen unwinding bound of $n = 20$. However, this was the only wrong result we produced, and we achieved 420 and 444 out of 549 possible marks, respectively.

3.4 Abstraction into Boolean programs

By default, ESBMC uses Z3, a satisfiabilty solver *modulo theories*, as backend engine. Z3 supports a wide variety of different theories, including uninterpreted functions, arrays, and linear integer arithmetics, which are very useful for general software verification. However, the offline challenge programs are very simple, and require none of these operations. In particular, all `int`-typed state variables are only assigned a small number of of different integer values, and the only operations on them are assignment and equality comparison, both with constant integer operands. We thus experimented with a Boolean abstraction, in which the state variables were replaced by the appropriate number of Booleans. However, this turned out to be counter-productive: the larger number of assignments led to larger VCs and longer solver times. We suspect that Z3's built-in bit-blasting implements the same approach more efficiently.

4 Checking the behavioral properties

4.1 Approach

The challenge rules allow different approaches to handle the behavioral properties, but we interpret and verify them as LTL formulae over the program's variables. We thus first converted the given formulae into our LTL notation, replacing the propositional shorthand notation by explicit comparisons involving `input` and `output` (e.g., `iB` becomes `input==2`), and eliminating the WU operator along the way. We then converted these formulae further into C monitor code and model-checked the combined system (i.e., original program and monitor). An early version [15] of our system ran the program and monitor as concurrent threads, but we now have an optimized scheduling scheme for this case. This scheduler only triggers a step of the BA monitor when any of the variables used in the LTL formula are assigned a value.

Originally, we checked only for the validity of the behavioral properties encoded in the LTL formulae and ignored the reachability properties; more specifically, we ignored the `assert(0)`-statements at the error labels. This means that we allowed the underlying finite state machine to ignore the invalid input that led to the invalid state, so that it could even transition out of it again (more precisely, resume from the last valid state). However, when we tested this approach against the evaluation examples (specifically `Problem10`), it became clear that a different way of interpreting the interaction between the error labels and the LTL formulae was assumed, that of pruning away such behaviors. We thus replaced the `assert(0)`-statements at the error labels by `assume(0)`-statements.

4.2 Interpretation of results

It is rarely possible to verify an LTL property by only exploring finite traces. A simple *co-safety* property such as $\mathsf{F}p$ might be verifiable, but only if every execution of the program sets p to true within the unwinding bound. A *safety* property such as $\mathsf{G}p$ cannot be verified using finite traces, but a witness may be found to its failure. A *liveness* property such as $(p \rightarrow \mathsf{F}\neg p) \wedge (\neg p \rightarrow \mathsf{F}p)$ cannot be shown to be true or false using finite traces although, for this expression, evidence of toggling of p might be reassuring.

Our approach computes its outcome by determining the *worst* (i.e., closest in the domain $\mathbb{B}_4$ to satisfying the never claim) behavior of any explored finite trace of the program. The four cases correspond to traces as follows:

- *P is bad wrt. φ*: At least one trace guarantees the satisfaction of the never claim, i.e. the BA is able to visit an accepting state infinitely often regardless of the future behavior of the program. The extracted BMC counterexample is a true counterexample of the safety property.
- *P is failing wrt. φ*: At least one trace will satisfy the never claim if the program stutters, i.e., continues infinitely without changing any observed state.
- *P is succeeding wrt. φ*: For at least one trace, there exists some future evolution of the BA's observable state in which the never claim is satisfied, but no such evolution is stuttering.
- *P is good wrt. φ*: For no trace can the never claim be satisfied by any future extension. Typically, every trace has resulted in the (non-deterministic) BA reaching a set of states, where no state has a successor. The extracted BMC counterexample is a true witness of the co-safety property.

Note that the two definitive cases (i.e., bad and good) are "sticky" in the sense that increasing the unwind bound for the underlying C program cannot change the outcome.

As demonstrated in the example below, not all LTL formulae are able to exhibit all these behaviors, regardless of the program to which they are coupled. Our static analysis of the BA allows us to catalogue the available behaviors for each LTL expression.

4.3 An example

We take as an example the LTL formula for the first behavioral property for the small/easy case, i.e., the output U occurs before output Z:

```
(! oZ WU (oU & ! oZ))
```

After translation into our input format, the never claim becomes

```
!(({output != 26} U ({output == 21} && {output != 26}))
  || (G {output != 26}))
```

We can see that our direct translation of the LTL has the potential to investigate unreachable states; the program state `{output == 21} && !{output != 26}` is potentially explored by the BA although it is clearly unreachable by the C program. In this particular case, however, the automaton as shown in Fig. 1 does not have explicit transitions on such unreachable states. If there were transitions enabled only on unreachable states, they would introduce additional possible behaviors of the BA. These would never be explored by ESBMC as the monitor BA

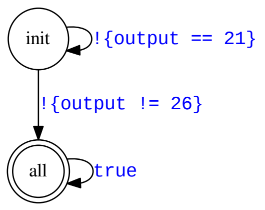

Fig. 1 The BA generated for the never claim of the property *output U occurs before output Z.*

is coupled to the C program. They would, however, show up in the "optimistic" analysis of the possible future behaviors of the BA after the unwound bound limit is reached. This in turn could lead to excessively cautious conclusions about the program's correctness wrt. to the LTL formulae. Program runs may be labelled succeeding when a more carefully constructed BA would show them as good. We could have used auxiliary C variables to ensure that no such redundant transitions were generated but, in order to avoid extensive rewriting of the programs, we have taken the naïve approach.

This particular LTL formula does not fall into any of the three simple types of property, safety, co-safety, or liveness. A finite prefix[3] can be good (e.g. $\langle \texttt{oV}, \texttt{oV}, \texttt{oU} \rangle$, where the BA fails) or bad (e.g. $\langle \texttt{oV}, \texttt{oV}, \texttt{oZ} \rangle$, where the BA is guaranteed to be able to remain in an accepting loop). It can also be succeeding (e.g. $\langle \texttt{oV}, \texttt{oV}, \texttt{oV}, \texttt{oV} \rangle$, where both success and failure remain possible but an infinite stutter extension would be good). This particular BA cannot, however, show failing behavior.

We are thus able to use an automatic analysis of the available behaviors of the BA to guide our confidence in the finite-trace results obtained from coupling the BA to the C program using ESBMC.

4.4 Analysis results

We were only able to achieve useful unwind bounds on the three small problems and the medium/easy problem (i.e., `Problem1` to `Problem4`). Table 2 summarizes the resuls. For all small problems, all outcomes are the same for unwind bounds 9–14. We thus have reasonable confidence in our results for the small problems.

For the medium/easy problem there are a few properties (#0, #14, #17, #77, #98) where the outcome changes with increasing unwind bounds. However, in all cases the change is from failing to good, corresponding to finally reaching the co-safety witness with the next iteration of the program's loop.

Overall, the number of definitive (i.e., good and bad) and inconclusive (i.e., succeeding and failing) outcomes are roughly equally common. However, we find substantially more counterexamples (200) than witnesses (12).

[3] Since this specific LTL formula only uses `output` the traces (and thus prefixes) consist of `output`-literals only. However, the corresponding `input` values can still be extracted from the BMC counterexamples.

We used program (`Problem10.c`) to validate our analysis results. For the 100 given LTL properties, our approach produced, with the scheme outlined above, only two false results (for #13 and #30). In both cases, we claim that the formula is succeeding, while the validation suite claims an explicit counterexample. However, in both cases the counterexample involves invalid inputs, which we have explicitly ruled out.

We thus submitted every good (bad) case as a success (failure) with a weighting of 9, since we get explicit witnesses (counterexamples). The succeeding and failing cases are more problematic; based on the results we achieved over the validation suite, we have chosen to report them, even for the medium/easy code, as success and failure with weightings of 7 and 9 respectively.

4.5 Discussion

For the 400 properties we analyzed we returned 385 (96.3%) correct results, which gives us, with the weights as explained above, a total score of 2991 marks. This compares well to the results achieved by the teams from Twente (3492 marks, 99.0% correct) and Paris (3069 marks, 98.1% correct).

The 15 wrong results fall into two different categories. In five cases, we find that the program is failing (resp. succeeding) wrt. the property, but the failure (resp. success) result that we report is wrong, because our unwind bounds are too small. In the remaining cases we find that the program is bad wrt. the property, but the counterexample trace goes through an error state; this trace should eventually be pruned away (using an `assume(0)`-statement) at an error label, but the automaton accepts a number of additional inputs sufficient to push this error label over the unwinding bound.

5 Conclusions

Clearly, if symbolic bounded model checking is a hammer it is doubtful whether the Challenge problems are the right nails. For the reachability problems, ESBMC is orders of magnitude slower than Java Pathfinder, an explicit-state model checker for Java [18], and we failed to process the large problems. However, we expect that our relative performance would improve with larger sets of inputs and outputs. On the other hand, we are encouraged that ESBMC, a general-purpose multi-threaded C model checker, has been able to generate useful analyses of these large and somewhat unusual systems. For the reachability properties we only produced one wrong result, despite the fact that we are using a bounded analysis. For the behavioral properties, we produced 15 wrong results and achieved a success rate of 96.3%, which is relatively close to the winner's success rate of 99.0%. We believe that our software model checking approach will become more competitive as the programs become more complicated (e.g., use of larger alphabets, arithmetic operations in the state updates, or data structures), and plan to participate in future Challenges with such problems.

Acknowledgements. The authors acknowledge the use of the IRIDIS High Performance Computing Facility, and associated support services at the University of Southampton, in the completion of this work.

References

1. T. Babiak, M. Křetínský, V. Rehák, and J. Strejček. LTL to Büchi Automata Translation: Fast and More Deterministic. *TACAS*, *LNCS* 7241, pp. 95–109, 2012.
2. A. Bauer and P. Haslum. LTL goal specifications revisited. ECAI'10, *Frontiers in Artificial Intelligence and Applications*, vol. 215, pp. 881–886, 2010.
3. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *J. Log. Comput.*, 20(3):651–674, 2010.
4. R. Brummayer and A. Biere, Boolector: An efficient SMT solver for bit-vectors and arrays, *TACAS*, *LNCS* 5505, pp. 174–177, 2009.
5. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. *TACAS*, *LNCS* 2988, pp. 168–176, 2004.
6. E. Clarke, F. Lerda. Model Checking: Software and Beyond. *J. Universal Computer Science*, 13:639–649, 2007.
7. L. Cordeiro and B. Fischer. Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking. *ICSE*, pp. 331–340, ACM, 2011.
8. L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *IEEE Trans. Software Eng.*, 38(4):957–974, 2012.
9. L. Cordeiro, J. Morse, D. Nicole, and B. Fischer. Context-Bounded Model Checking with ESBMC 1.17 *TACAS*, *LNCS* 7214, pp. 533–536, 2012.
10. L. M. de Moura and N. Bjørner, Z3: An efficient SMT solver, *TACAS*, *LNCS* 4963, pp. 337–340, 2008
11. P. Gastin and D. Oddoux Fast LTL to Büchi Automata Translation. *CAV*, *LNCS* 2102, pp. 53–65, 2001.
12. G. Holzmann. The SPIN Model Checker - Primer and Reference Manual. Addison-Wesley, 2004.
13. O. Kupferman and M. Vardi. Model checking of safety properties. *Formal Methods in System Design,* 19(3):291–314, 2001.
14. L. Lamport. What Good is Temporal Logic? *Information Processing*, 83:657–668, 1983.
15. J. Morse, L. Cordeiro, D. Nicole, and B. Fischer. Context-Bounded Model Checking of LTL Properties for ANSI-C Software. *SEFM*, *LNCS* 7041, pp. 302–317, 2011.
16. J. Morse, L. Cordeiro, D. Nicole, and B. Fischer. Model Checking LTL Properties over ANSI-C Programs with Bounded Traces. *J. Software and Systems Modeling*. Online first, July 2013.
17. Pnueli, A.: The temporal logic of programs. *FOCS*, pp. 46–57, 1977.
18. J. Visser. Personal communication.

A Detailed Results

	Program1						Program2						Program3						Program4				Program6			
			$n = 7$		$n = 50$				$n = 7$		$n = 50$				$n = 7$		$n = 20$				$n = 20$				$n = 7$	
|VC|			8284		59239				8216		59343				47815		136852				286837				324421	
Label		w	Time	t	Time	t		w	Time	t	Time	t		w	Time	t	Time	t		w	Time	t		w	Time	t
global	−	9	2.4	2	220.5	3	−	9	2.4	6	229.7	5	−	9	45.5	4	773	4	−	9	5105	5	−	9	4102	6
error0	+	6	0.8	-	256.0	-	+	6	14.2	-	0.7	-	+	6	270.5	-	1373	-	+	6	11598	-	−	9	4668	7
error1	+	6	0.8	-	279.0	-	+	6	0.7	-	223.4	-	+	6	16.9	-	1752	-	+	6	8039	-	−	9	4918	6
error2	+	6	0.8	-	291.8	-	+	6	0.7	-	224.0	-	+	6	6.7	-	359	-	+	6	6839	-	−	9	4785	7
error3	+	6	0.8	-	281.8	-	+	6	0.7	-	220.9	-	+	6	17.7	-	2588	-	+	6	5120	-	+	6	1026	-
error4	+	6	0.7	-	170.3	-	+	6	0.7	-	215.6	-	+	6	16.4	-	1821	-	−	9	19375	18	−	9	4685	6
error5	+	6	0.6	-	292.1	-	+	6	0.8	-	197.0	-	+	6	8.1	-	907	-	+	6	6524	-	−	9	4664	6
error6	+	6	0.7	-	369.7	-	+	6	1.0	-	526.5	-	+	6	5.9	-	478	-	−	9	16984	19	+	6	1592	-
error7	+	6	0.7	-	338.9	-	+	6	0.8	-	228.1	-	+	6	15.7	-	2278	-	+	6	2903	-	+	6	860	-
error8	+	6	0.7	-	303.3	-	+	6	0.7	-	250.4	-	+	6	6.1	-	332	-	+	6	6778	-	+	6	880	-
error9	+	6	0.7	-	220.5	-	+	6	0.7	-	229.5	-	−	9	60.8	6	949	6	−	9	12809	17	−	9	5046	6
error10	+	6	0.7	-	281.2	-	+	6	0.6	-	198.0	-	+	6	5.7	-	254	-	+	6	4768	-	−	9	4084	6
error11	+	6	0.7	-	336.3	-	+	6	0.7	-	190.0	-	+	6	11.7	-	1982	-	−	9	17920	18	−	9	4467	6
error12	+	6	0.7	-	360.3	-	+	6	0.7	-	154.0	-	+	6	14.5	-	2081	-	+	6	10706	-	−	9	4420	6
error13	+	6	0.6	-	240.0	-	−	9	2.4	4	223.0	3	−	9	49.6	5	900	5	−	9	15734	17	+	6	1271	-
error14	+	6	0.7	-	329.2	-	+	6	0.7	-	364.8	-	+	6	8.2	-	198	-	−	9	14698	17	+	6	1021	-
error15	−	9	2.7	5	223.1	5	+	6	0.6	-	202.5	-	+	6	17.4	-	2451	-	−	9	14747	17	−	9	4453	6
error16	+	6	0.7	-	377.6	-	−	9	2.4	5	214.7	8	+	6	10.6	-	600	-	+	6	8277	-	+	6	942	-
error17	+	6	0.7	-	278.6	-	+	6	0.7	-	134.9	-	+	6	8.8	-	1366	-	−	9	13803	17	+	6	919	-
error18	+	6	0.7	-	220.2	-	+	6	0.7	-	236.1	-	+	6	8.2	-	548	-	−	9	17097	17	+	6	1175	-
error19	+	6	0.6	-	236.0	-	+	6	0.7	-	124.3	-	+	6	15.0	-	2426	-	−	9	17687	20	+	6	1183	-
error20	−	9	2.6	7	254.9	14	+	6	0.7	-	177.1	-	+	6	10.2	-	1181	-	+	6	8114	-	−	9	4272	6
error21	−	9	2.8	5	248.1	5	+	6	0.7	-	151.1	-	+	6	13.2	-	1854	-	+	6	11178	-	−	9	4023	6
error22	+	6	0.7	-	563.8	-	+	6	0.8	-	226.3	-	+	6	7.6	-	775	-	+	6	3689	-	+	6	990	-
error23	+	6	1.0	-	271.0	-	+	6	0.8	-	222.9	-	+	6	19.0	-	2732	-	+	6	1788	-	+	6	1162	-
error24	+	6	0.9	-	197.6	-	+	6	0.7	-	194.6	-	+	6	5.8	-	246	-	+	6	4610	-	−	9	4474	6
error25	+	6	0.9	-	158.8	-	+	6	0.7	-	234.4	-	+	6	9.1	-	782	-	+	6	7123	-	+	6	927	-
error26	+	6	0.9	-	245.9	-	+	6	0.7	-	159.8	-	−	9	46.9	4	849	4	−	9	14603	18	+	6	1133	-
error27	+	6	0.8	-	291.8	-	+	6	0.7	-	144.2	-	−	9	50.5	5	893	5	−	9	15280	17	−	9	3853	7
error28	+	6	0.8	-	263.0	-	+	6	0.8	-	212.6	-	−	9	50.7	5	930	5	+	6	6154	-	+	6	998	-
error29	+	6	0.7	-	247.6	-	+	6	0.7	-	108.2	-	+	6	7.7	-	273	-	+	6	6181	-	−	9	4700	6
error30	+	6	0.7	-	215.6	-	+	6	0.7	-	144.3	-	+	6	7.1	-	348	-	+	6	3350	-	+	6	1298	-
error31	+	6	0.8	-	230.7	-	+	6	0.7	-	129.5	-	−	9	47.5	5	962	5	−	9	16239	19	+	6	1346	-
error32	−	9	2.6	7	236.1	10	+	6	0.8	-	107.0	-	+	6	6.0	-	241	-	−	9	14997	17	+	6	1133	-
error33	−	9	2.5	6	235.4	9	+	6	0.7	-	155.4	-	+	6	7.9	-	339	-	+	6	7294	-	−	9	4698	6
error34	+	6	0.7	-	279.8	-	+	6	0.8	-	128.2	-	+	6	11.7	-	567	-	+	6	6322	-	+	6	1282	-
error35	−	9	2.7	5	240.1	13	+	6	0.8	-	119.4	-	−	9	45.3	6	1435	6	−	9	15567	17	+	6	1123	-
error36	+	6	0.6	-	240.3	-	+	6	0.7	-	177.3	-	+	6	8.1	-	384	-	−	9	17728	17	−	9	4451	7
error37	−	9	2.7	6	215.8	6	+	6	0.7	-	191.9	-	−	9	47.5	5	823	6	+	6	3629	-	−	9	4452	7
error38	−	9	2.6	5	239.5	5	+	6	0.7	-	128.0	-	+	6	6.9	-	480	-	−	9	17833	18	−	9	4917	7
error39	+	6	0.7	-	251.8	-	+	6	0.7	-	345.9	-	−	9	48.8	6	1046	6	−	9	20083	19	+	6	1186	-
error40	+	6	0.9	-	339.4	-	+	6	0.7	-	203.1	-	+	6	7.7	-	236	-	−	9	16729	18	+	6	922	-
error41	+	6	0.7	-	185.6	-	+	6	0.8	-	223.4	-	+	6	6.4	-	504	-	+	6	5259	-	+	6	954	-
error42	+	6	0.7	-	254.4	-	+	6	0.7	-	222.9	-	+	6	7.5	-	254	-	+	6	3488	-	+	6	808.8	-
error43	+	6	0.7	-	297.6	-	−	9	2.5	7	338.3	3	−	9	45.4	7	914	5	+	6	3534	-	+	6	1169	-
error44	−	9	2.4	5	239.1	12	−	9	2.4	7	222.9	9	+	6	13.4	-	2001	-	+	6	4395	-	−	9	4781	6
error45	+	6	0.7	-	219.7	-	−	9	2.4	7	239.1	5	−	9	49.1	6	1202	6	−	9	15119	18	+	6	966	-
error46	+	6	0.6	-	356.3	-	+	6	0.7	-	165.9	-	+	6	5.8	-	404	-	+	6	4544	-	+	6	1124	-
error47	−	9	2.6	7	241.8	7	+	6	0.8	-	215.6	-	+	6	9.2	-	1719	-	+	6	6282	-	−	9	4274	6
error48	+	6	0.7	-	289.9	-	+	6	0.7	-	182.6	-	+	6	8.7	-	434	-	+	6	3351	-	−	9	4445	7
error49	+	6	0.7	-	379.4	-	+	6	0.7	-	215.9	-	+	6	7.3	-	507	-	+	6	7687	-	+	6	1052	-
error50	−	9	2.5	5	234.7	5	−	9	2.4	4	238.3	6	−	9	50.2	5	979	5	+	6	5118	-	+	6	909	-
error51	+	6	0.7	-	363.1	-	+	6	0.7	-	159.5	-	+	6	14.4	-	3122	-	+	6	9681	-	+	6	1212	-
error52	+	6	0.8	-	525.2	-	+	6	0.6	-	269.4	-	−	9	55.0	6	886	6	−	9	12310	19	+	6	927	-
error53	+	6	0.7	-	270.5	-	+	6	0.8	-	151.9	-	+	6	17.0	-	2047	-	+	6	4776	-	+	6	1139	-
error54	+	6	0.7	-	294.5	-	+	6	0.7	-	166.0	-	+	6	7.4	-	767	-	+	6	5830	-	+	6	926	-
error55	+	6	0.7	-	201.7	-	+	6	0.7	-	189.5	-	+	6	9.3	-	1454	-	−	9	12929	17	+	6	676	-
error56	−	9	2.5	6	244.6	6	+	6	0.6	-	223.6	-	+	6	6.1	-	199	-	+	6	5299	-	−	9	4793	6
error57	−	9	2.6	6	225.7	9	+	6	0.6	-	176.4	-	+	6	7.2	-	277	-	+	6	11463	-	+	6	485	-
error58	+	6	0.6	-	298.1	-	+	6	0.6	-	208.5	-	+	6	6.6	-	812	-	−	9	19451	19	−	9	4559	6
error59	+	6	0.7	-	258.9	-	−	9	2.7	6	236.0	10	+	6	12.1	-	1325	-	+	6	6277	-	−	9	4803	6

Table 1 Results for the reachability properties. "−" means that the error label is reachable (i.e., the program fails). "+" means that we have not found a counterexample; we thus *claim* that the label is unreachable. w denotes the marks we associate with our results. t is the number of inputs in the counterexample found. Time is given in wall-clock seconds. |VC| gives the size of the VC in assignments.

	Problem1						Problem2						Problem3						Problem4					
$\backslash n$	9	10	11	12	13	14	9	10	11	12	13	14	9	10	11	12	13	14	9	10	11	12	13	14
0	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot^p$	$\boldsymbol{\top}$	$\top$	$\top$	$\top$	$\top$
1	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$	$-$
2	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
3	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$
4	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$
5	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$	$-$
6	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
7	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
8	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$
9	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$
10	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
11	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$
12	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$	$-$
13	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
14	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\boldsymbol{\top}$	$\top$	$\top$	$\top$	$\top$
15	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$
16	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
17	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\boldsymbol{\top}$
18	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$
19	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
20	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
21	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$
22	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
23	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$-$	$\top^p$	$-$
24	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
25	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
26	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$	$-$
27	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
28	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$
29	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
30	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$	$-$
31	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$	$-$
32	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$	$-$
33	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$	$-$
34	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
35	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$
36	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
37	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
38	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
39	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
40	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
41	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$	$-$
42	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
43	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
44	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$
45	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
46	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$	$-$
47	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$	$-$
48	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
49	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$
50	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
51	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$
52	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$	$-$
53	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
54	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
55	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$	$-$
56	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
57	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
58	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
59	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$	$-$
60	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
61	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
62	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
63	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
64	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$
65	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
66	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
67	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
68	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
69	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
70	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
71	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
72	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
73	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$	$-$
74	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
75	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
76	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$-$	$-$	$-$	$-$
77	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$	$\boldsymbol{\top}$
78	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
79	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$-$	$-$	$-$
80	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
81	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
82	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
83	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$
84	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
85	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$
86	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
87	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
88	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
89	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$
90	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$	$-$
91	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
92	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
93	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
94	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
95	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$-$	$-$
96	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$-$	$\top^p$	$-$
97	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$
98	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\boldsymbol{\top}$	$\top$	$\top$	$\top$	$\top$
99	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$-$	$-$

Table 2 Results for behavioral properties. Only claims are shown, for different unwinding bounds ($n = 9$ to $n = 14$). "$-$" denotes a time-out ($t_{\text{max}} = 3600s$). Boldface denotes changes in the outcomes as the unwinding bounds change.