Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Efficient reduction of finite state model checking to reachability analysis[*]

Viktor Schuppan, Armin Biere

Computer Systems Institute, ETH Zürich, CH-8092 Zurich, Switzerland, e-mail: {schuppan|biere}@inf.ethz.ch

January 27, 2004

Abstract. Two types of temporal properties are usually distinguished: safety and liveness. Recently we have shown how to verify liveness properties of finite state systems using safety checking. In this article we extend the translation scheme to typical combinations of temporal operators. We discuss optimizations that limit the overhead of our translation. Using the notions of predicated diameter and radius we obtain revised bounds for our translation scheme. These notions also give a tight bound on the minimal completeness bound for simple liveness properties. Experimental results show the feasibility of the approach for complex examples. For one example even an exponential speed-up can be observed.

Key words: Liveness – Safety – Linear temporal logic – Model checking

1 Introduction

Sequential properties of systems are often formulated in temporal logics such as LTL [12]. These properties fall into two categories: liveness and safety properties. Safety properties are invariants of the system and can be checked fairly easily by reachability analysis. Sophisticated algorithms and implementations exist. On the other hand, many important system properties, for example absence of deadlock or livelock, are more naturally formulated as liveness properties.

Many techniques, and in addition many implementations, target only safety properties or use optimizations that are only applicable to safety checking. Examples of such techniques are invariant checking [3,32], sequential automated test pattern generation (ATPG) [28], and symbolic trajectory evaluation (STE) [31] in its basic form. In this article we extend our translation scheme introduced in [2], that allows an efficient reformulation of liveness checking for finite state systems as safety checking. The translation makes tools and techniques for safety checking applicable to liveness checking as well.

Safety is often characterized as “something bad never happens” while liveness means “something good eventually happens” [25]. A counterexample to a liveness property is an infinite path where something good never happens. Such a path must include a loop in a finite state system. If that loop is extended infinitely, a lasso-shaped counterexample is obtained. Essentially, our translation searches for such a lasso-shaped counterexample. It tries to guess the start of a loop, saves it in a copy of the state variables and checks whether the saved state occurs a second time. When this happens, a loop has been found and the property is checked. Our translation is able to handle fairness. Thus, it is applicable to all LTL properties via a standard automaton construction [13]. For several commonly used LTL properties such as the request/acknowledge template $\mathbf{G}(r \rightarrow \mathbf{F}a)$ we also give a direct translation.

Our translation scheme can be applied even manually on the design entry-level, with the proviso that an observer automaton be added, without changing the behavior of the original system. The user does not need to have access to the source code of the tool, e.g., the model checker, itself. This could be useful in an industrial setting where the source code of a tool is usually not available. To some extent it might also discourage tool vendors to charge extra license fees for liveness support, if compromises with respect to capacity are acceptable.

Some optimizations [23,1] from bounded model checking [4] can be applied to our translation. When combined with a translation-specific optimization, the performance of our approach is also acceptable on more involved examples, in some cases even comparable with standard techniques. We give an example where our approach is exponentially faster.

Radius, diameter and reoccurrence diameter are characteristics of a Kripke structure that are used to give bounds

[*] This is a preprint. The final version appeared in Springer STTT 5 (2004) ?, pp.? – ?. or online at http://www.springerlink.com/

on the number of iterations required for verification [4]. The bounds on the maximal number of image computations necessary to check liveness with our translation in BDD based symbolic model checking [27] as stated in [2] are formulated using these notions. These bounds proved incorrect. In this paper we give revised bounds using the new notions of predicated radius and diameter. In addition, we extend the concept of completeness threshold [23] and prove a tight bound on the completeness bound for liveness properties which can immediately be applied to bounded model checking.

The idea most closely related is the verification of liveness properties in bounded model checking [4]. The double DFS [11] used in on-the-fly model checking [13] is somewhat similar. However, depth first search tends to find counterexamples where the loop starts rather late. Our approach can be used with breadth first search symbolic model checking to find shortest counterexamples and, at least in principle, does not need any change to the model checker.

Our translation can also be viewed as an extension of monitors as used in static or dynamic checking with additional inputs that signal beginning and closing of a loop. Synchronous observers that check properties of a program are proposed in [16] to verify reactive systems. The class of properties is restricted to safety, observers are required to be deterministic. This approach is adapted in [22] to provide a structural translation from past time LTL into program fragments in ESTEREL. In [19] a monitor based on a dynamic programming algorithm is generated for Java from a past time LTL formula. Two different versions that produce monitors for future time LTL adapted to finite traces are presented in [18,14]. There, (non-)occurrence of eventualities is only considered up to the end of a trace.

One standard optimization for BDD based model checking is forward model checking [21,20,5]. It uses a different model checking algorithm which avoids to visit unreachable states and often is able to find counterexamples faster. For safety properties this optimization is implemented in most symbolic model checkers. The algorithm for general properties, in particular liveness properties, is usually not available. Therefore [24] characterizes safety properties as properties with a finite violating prefix. These can be checked with efficient algorithms using reachability analysis. As already noted in [2], our translation allows to use this restricted version of forward model checking for liveness properties as well. For specific examples, checking a simple liveness property with our approach is exponentially faster than forward (and backward) model checking algorithms.

Section 2 presents the state recording translation and describes some optimizations. In Section 3 we revise the necessary formal background. Section 4 introduces the notion of predicated radius and diameter. In Section 5 we extend the completeness threshold to the more general notion of completeness bound. Both notions are then used to give tighter bounds for the minimal completeness bound of simple liveness properties. Correctness of the state recording translation is proved in Section 6. Section 7 proves revised bounds for the verification with our translation. Sections 8 and 9 report on applying the state recording translation to artificial and real-world examples. Section 10 concludes.

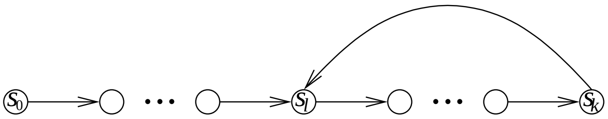

Fig. 1. A generic lasso-shaped counterexample.

2 Translating Simple Liveness into Safety

A counterexample trace for a simple liveness property $\mathbf{F}p$ is an infinite path where p never holds along the path. If the number of states in a system is finite, a counterexample trace to a simple liveness property can be assumed to be lasso-shaped: it consists of a finite prefix and an infinitely repeating loop, see Fig. 1. Such a trace can always be derived from an arbitrary infinite trace by inserting a back loop from the first state occurring the second time. If p was false for every state in the original trace it will also hold nowhere in the lasso-shaped trace.

Thus, simple liveness properties $\mathbf{F}p$ of finite state systems can be verified by finding all lasso-shaped traces and checking whether p has been true somewhere on each trace once the loop is closed. Explicit state algorithms for Büchi Automata [13] and unfolding liveness properties in bounded model checking [4] are examples of model checking algorithms that use this observation. Instead of implementing this observation in a special purpose algorithm we show in the following how it can be used to transform a system and a liveness property such that reachability checking is sufficient to verify that property.

In model checking applications it is often observed that a liveness property $\mathbf{AF}p$ can further be restricted by adding a bound k on the number of steps within which the body p has to hold. The bound is either given in the specification or may be determined by manual inspection. A bounded liveness property $\mathbf{AF}^k p$ is defined as

$$\mathbf{AF}^k p \equiv \mathbf{A}(p \vee \mathbf{X}p \vee \cdots \vee \mathbf{X}^k p), \text{ with } \mathbf{X}^i p \equiv \underbrace{\mathbf{X}\cdots\mathbf{X}}_{i-\text{times}} p \qquad (1)$$

and clearly $\mathbf{AF}^k p$ implies $\mathbf{AF}p$. The reverse direction is also true if the bound is chosen large enough, in particular as large as the number of states $|S|$ in the model, since all states are reachable in $|S|$ steps. A naive translation would just exchange $\mathbf{AF}p$ for $\mathbf{AF}^k p$ with k the number of states. However, the expansion of $\mathbf{AF}^k p$ in (1) results in a very large formula, especially in the context of symbolic model checking.

Assume instead, that the system is extended with a variable *looped* that indicates when a loop is closed and with a variable *live* that remembers whether p has already been true. Then, the liveness property $\mathbf{F}p$ in the original system is equivalent to the safety property $\mathbf{G}(\mathit{looped} \rightarrow \mathit{live})$ in the extended system. Implementing *live* is easy. In the rest of this

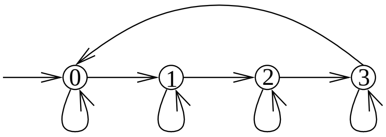

Fig. 2. A 2-bit counter with self-loops

section two implementations for *looped* are discussed. The first *counter-based translation* is based on the verification of bounded liveness only as described above. Our main contribution is the second *state-recording translation* that can be applied to arbitrary finite state systems and general LTL properties and can still be verified efficiently in many cases.

As an example, consider the 2-bit counter with self-loops in Fig. 2. There, $\mathbf{F}\ (s = 3)$ does not hold. A counterexample is given by $\pi = 0, 1, 2, 2, \ldots$. Figure 3 shows a model of the counter in the input language of the model checker *SMV* [27] in its original form and with the counter-based and the state-recording translation applied. Note that all three models explicitly enumerate all possible values of the counter. While this makes the description easier to understand, it is exponential in the number of bits of the counter. A linear description can be obtained by using a binary encoding of s in the declaration of the variables and in the transition relation.

2.1 Counter-Based Translation

Instead of detecting a loop when it is closed, the counter-based translation infers that a loop should have occurred once a sufficient number of transitions has been performed. A counter is added to the system that is incremented at each transition and sets *looped* to true once it reaches a predefined bound.

A trivial bound valid for arbitrary systems and properties is the overall number of states in the original system: any path of that length must include a loop. However, this requires an impractically large number of iterations in a realistic system as the property can only be checked when the counter has reached its bound.

For most systems and properties smaller bounds exist that still ensure correct results, see the examples given in Sect. 5. A smaller bound adds fewer state bits and should lead to faster verification. Presently, a practically efficient method to compute a minimal bound is not known for arbitrary systems and properties. Also note that, in general, the counter-based translation will not produce shortest counterexamples.

In Fig. 3 (b) the state variables and the transition relation of the original system are left unchanged. The *loop detection part* implements a counter for the number of transitions performed. The *property observing part* adds the flag *live*. Finally, the specification is modified as described.

Note, that in our definition the last state in the loop must have already been seen and does not add new information regarding the truth of the liveness property. Therefore, the result could be determined one cycle before this bound is actually reached. This optimization has not been applied in Fig. 3 (b) to keep the presentation of both translations uniform.

A more general form of the counter-based translation can use a flag *finished* instead of *looped*. That flag becomes true once a sufficient number of transitions has been performed to ensure that p would have occurred on a path if $\mathbf{F}p$ were true.

2.2 State-Recording Translation

In principle, state space search is memory-less. Detecting a loop as soon as it is closed can not be expressed directly in temporal logic. Instead, we add copies of all variables to the model, enabling us to save a state that has previously been visited. Reoccurrence of a state can now be detected by comparing the present state to the saved copy. As the start of a loop is not known beforehand, an oracle *save* is used to indicate when a copy of the present state should be saved. An additional flag, *saved* is needed to prevent overwriting a previously saved copy.

For simple liveness properties the counter-based and the state-recording translation differ only in the loop detection part, see Fig. 3 (c). Here, it consists of a *save* oracle, a copy of the original state variables *l2s_s*, and a *saved* flag to ensure that the state is saved only once on a path. The *on_loop* flag indicates whether the presumed loop has started. It is used in translations for more complex formulae, see Tab. 1.

When the loop closing condition *looped* becomes true, this means that current state was visited earlier. Therefore, the transformed specification does not need to take the current value of the property p into account. It suffices that the *live* flag remembers whether p has been true in the past. Figure 4 illustrates a run of the state-recording translation for the generic counterexample from Fig. 1.

2.3 Translating Fairness and Hierarchy

Fairness conditions can be incorporated similar to liveness properties. A fairness condition is a set of states in the original model. A path is fair if it passes infinitely often through a state in each fairness condition. An additional state variable $fair_i$ is introduced for fairness condition i that observes, similarly to *live*, whether one of its fair states has been seen. It is initially set to false and becomes true when a fair state occurs on the loop. The specification is required to hold at the end of a loop only if all fairness conditions hold as well.

No special precautions are required for hierarchical models that can be flattened. If hierarchy should be preserved the *save*, *saved*, and *on_loop* signals are forwarded to each submodule. The submodules perform detection of loops and observing of fairness and specification properties locally. The results are sent back to the main module that computes global values for *looped* and *fair* and then checks the specification. This enables translating models (possibly by hand) without separate flattening before. For an example, see Appendix A.1.

2.4 General LTL

Generalized Büchi automata and thus LTL [13] can be translated into fair Kripke structures. Therefore, our translation

```
MODULE main


VAR
  s: {0, 1, 2, 3};
ASSIGN
  init(s) := 0;
  next(s) := case
    s = 0: {1, s};
    s = 1: {2, s};
    s = 2: {3, s};
    s = 3: {0, s};
  esac;


SPEC
  AF s = 3
```

(a) original

```
MODULE main

-- unmodified part of the
-- original system
VAR
  s: {0, 1, 2, 3};
ASSIGN
  init(s) := 0;
  next(s) := case
    s = 0: {1, s};
    s = 1: {2, s};
    s = 2: {3, s};
    s = 3: {0, s};
  esac;

-- loop detection part
VAR
  counter: 0..4;
ASSIGN
  init(counter) := 0;
  next(counter) := case
    counter < 4: counter + 1;
    1: counter;
  esac;
DEFINE
  looped := counter = 4;


-- property observing part
VAR
  live: boolean;
DEFINE
  found := s = 3;
ASSIGN
  init(live) := 0;
  next(live) := live | found;

-- transformed specification
SPEC
  AG (looped -> live)
```

(b) counter-based

```
MODULE main

-- unmodified part of the
-- original system
VAR
  s: {0, 1, 2, 3};
ASSIGN
  init(s) := 0;
  next(s) := case
    s = 0: {1, s};
    s = 1: {2, s};
    s = 2: {3, s};
    s = 3: {0, s};
  esac;

-- loop detection part
VAR
  save: boolean;
  saved: boolean;
  l2s_s: {0, 1, 2, 3};
ASSIGN
  init(saved) := 0;
  next(saved) := on_loop;
  init(l2s_s) := s;
  next(l2s_s) := case
    save & !saved: s;
    1: l2s_s;
  esac;
DEFINE
  looped := saved & (s = l2s_s);
  on_loop := save | saved;

-- property observing part
VAR
  live: boolean;
DEFINE
  found := s = 3;
ASSIGN
  init(live) := 0;
  next(live) := live | found;

-- transformed specification
SPEC
  AG (looped -> live)
```

(c) state-recording

Fig. 3. Original and transformed SMV code of 2-bit counter with self-loops

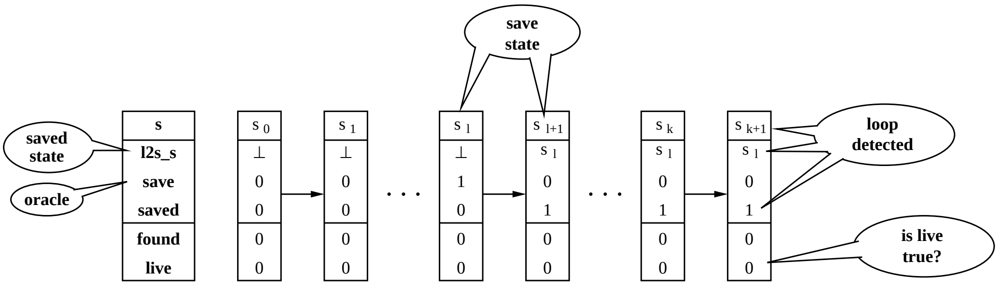

Fig. 4. A run of the state-recording translation for the generic counterexample

applies to LTL model checking in general performing the following steps:

1. Translate the negated LTL formula into a generalized Büchi automaton. Algorithms for this purpose include [13, 11], some of them are also described in [10,29]. Various tools are available that implement more advanced algorithms, see e.g. [34,33].
2. Build the cross-product of the model and the formula automaton. Each acceptance set is represented as a fairness constraint. The property that is actually verified states that no fair path exists in the model.
3. Apply the state-recording translation.

An example is given in Appendix A.2.

2.5 *Templates for Frequently Occurring Specifications*

A large fraction of the specifications found in practice can be covered by a limited number of temporal formulae. Table 1 provides templates for the translation of some frequently used

LTL formulae. This avoids having to use the explicit translation described above. Column 1 gives the (universally quantified) LTL formula represented, column 2 states its (existentially quantified) negation. For the former the translation searches a counterexample, a witness for the latter. The property observing part and the specification of the translation are shown in the language of the model checker *SMV* in the last column. The loop detection part is independent of the formula and can be found in Fig. 3. In each template p and q can be replaced with arbitrary propositional formulae.

A formal proof for the translation of $\mathbf{F}p$ follows in Sect. 6. We do not prove the other translations but rather try to give an intuitive understanding. The translation of $\mathbf{F}p$ is repeated for reference. While not necessary for a simple safety property, the translation of $\mathbf{G}p$ shows the symmetry to finite liveness. The translations of $\mathbf{FG}p$ and $\mathbf{GF}p$ are similar whereby the former considers, the latter requires p to hold on the loop only. Intuitively, for $\mathbf{FG}p$ to be true, p must hold on each state of a loop while the prefix of the loop does not influence the truth of the formula. $\mathbf{GF}p$ can only be true in a finite state system if p holds on at least one state of a loop. The translation of $p\,\mathbf{U}\,q$ combines the representations for a safety and a liveness property: q must become true as long as p is true or when p is false for the first time. The translation of the request-response property $\mathbf{G}(p \rightarrow \mathbf{F}q)$ directly reflects that one or more p-states must be met or followed by at least one q-state, where any q-state on the loop is sufficient. Finally, the next-time operator $\mathbf{X}$ is handled by shifting the point of view one step forward in time first and then applying the translation of $\mathbf{F}$.

2.6 Optimizations

Two optimizations can help to improve the performance of verifying a translated model. They are source to source and are applied after the translation has been performed.

Bounded model checking and the state-recording translation prove or disprove a liveness property by searching for a lasso-shaped counterexample. Not all variables need to be considered when comparing states in the search for a loop. Both techniques can use the same static set of variables for loop detection. Kroening and Strichman [23] proved in the context of bounded model checking that input variables can be ignored when comparing states in the search for a loop. Baumgartner et al. [1] observed that the diameter of a model need only be computed for the variables in the cone of influence of the property. Thus, input variables and variables not in the cone of influence of the property under consideration need neither be copied nor compared in our translation. The correctness of the latter fact can also be seen in our context by first applying cone of influence reduction to the original model and then performing the state recording translation. Obviously, variables known to remain constant after initialization can also be ignored. The combination of these three optimizations is referred to as *variable optimization*. Note, that finding a shortest counterexample is not guaranteed if variable optimization is enabled.

The second optimization is based on the monotonicity of a simple liveness property $\mathbf{F}p$ — once p has been found true on a path, the value of the *live*-flag remains constant further down on this path, i.e., the truth of the formula $\mathbf{G}(\mathit{looped} \rightarrow \mathit{live})$ will be true from then on. This fact allows to stall the state machine of the original model completely once *live* is true in a state. Thus, from that state no further states of the original model are reachable. For hardware systems, this corresponds to adding a stall signal to each flip-flop that keeps its output at the current state. In effect, the radius of the translated system is reduced (see Sect. 7). In addition, the reachable state space might be cut. We call this *halt optimization*.

3 Preliminaries

A *Kripke structure* $K = (S, T, I, L)$ consists of a set of states S, a transition relation $T \subseteq S \times S$, a set of initial states $I \subseteq S$ and a labeling function $L\colon S \rightarrow P(A)$, where $P(A)$ is the power set of the set of atomic propositions $A = \{p, q, \ldots\}$. A state $s \in S$ is defined to *have a transition* if there exists $s' \in S$ with $(s, s') \in S$. Then T is called *total* if all $s \in S$ have a transition. For technical reasons we also have to work with non total transition relations, and do not require T to be total as is usually done. An important restriction for the rest of the article is, that we only consider *finite Kripke structures* with $|S| < \infty$.

It is often convenient to describe the state space S of a Kripke structure as product of the valuations of a set of variables $V\colon S = V_1 \times \ldots \times V_n$, $n = |V|$ where V_i is the set of valuations of variable $v_i \in V$. The transition relation is then given as a set of equations each defining the next state of a variable in terms of the current and next state values of a set of variables: $v_i' \in f(U, U')$, $U \subseteq V$. If the next state value of a variable is not constrained by the transition relation it is called an *input variable*.

A *path* $\pi = (s_0, s_1, \ldots)$ of a Kripke structure is, whether finite or infinite, a non-empty sequence of states $s_i \in S$. For a finite sequence $(s_0, \ldots, s_n)$ we define the *length* of π to be $|\pi| = n$ and $|\pi| = \infty$ for an infinite sequence. For any path it is also required that $(s_i, s_{i+1}) \in T$ for $0 \leq i < |\pi|$. Further let $\pi(i)$ denote the i-th state of the sequence. Then π_i is the suffix $(\pi(i), \pi(i+1), \ldots)$ of π with its first i states chopped off. A path π is *maximally expanded* if it is infinite, or if π is finite and the last state $\pi(|\pi|)$ of π does not have a transition. The set of all paths of a Kripke structure is denoted by Π.

A (partial) specification describes desired properties of a system. We consider specifications given as *linear temporal logic* (LTL) formulae. An LTL formula is made of atomic propositions from A and the standard boolean operators for conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$) and implication ($\rightarrow$). Additionally the following temporal operators are used: the unary operators *next-time* ($\mathbf{X}$), *globally* ($\mathbf{G}$), *finally* ($\mathbf{F}$), and the binary temporal operator *until* ($\mathbf{U}$) and its dual *release* ($\mathbf{R}$). The validity of a temporal formula f over a *maximally expanded* path π, written $\pi \models f$, is defined recursively. Let g and h be LTL formulae and $p \in A$.

Formula (counterexample)	Negation (witness)	Translation

$\mathbf{F}p$	$\mathbf{G}\neg p$	`ASSIGN init(live) := 0;`
		`       next(live) := live | p;`
		`SPEC   AG (looped -> live)`

$\mathbf{G}p$	$\mathbf{F}\neg p$	`ASSIGN init(safe) := 1;`
		`       next(safe) := safe & p;`
		`SPEC   AG (looped -> safe)`

$\mathbf{GF}p$	$\mathbf{FG}\neg p$	`ASSIGN init(live) := 0;`
		`       next(live) := live | on_loop & p;`
		`SPEC   AG (looped -> live)`

$\mathbf{FG}p$	$\mathbf{GF}\neg p$	`ASSIGN init(safe) := 1;`
		`       next(safe) := safe & (on_loop -> p);`
		`SPEC   AG (looped -> safe)`

$p\,\mathbf{U}\,q$	$\neg p\,\mathbf{R}\,\neg q$	`VAR    safe_p: boolean;`
		`ASSIGN init(live) := 0;`
		`       next(live) := live | safe_p & q;`
		`       init(safe_p) := 1;`
		`       next(safe_p) := safe_p & p;`
		`SPEC   AG (looped -> live)`

$\mathbf{G}(p \rightarrow \mathbf{F}q)$	$\mathbf{F}(p \wedge \mathbf{G}\neg q)$	`VAR    live_q: boolean;`
		`ASSIGN init(safe) := 1;`
		`       next(safe) := !p & safe | q | live_q;`
		`       init(live_q) := 0;`
		`       next(live_q) := live_q | on_loop & q;`
		`SPEC   AG (looped -> safe)`

$\mathbf{F}(p \wedge \mathbf{X}q)$	$\mathbf{G}(\neg p \vee \mathbf{X}\neg q)$	`VAR    last_p: boolean;`
		`ASSIGN init(live) := 0;`
		`       next(live) := live | last_p & q;`
		`       init(last_p) := 0;`
		`       next(last_p) := p;`
		`SPEC   AG (looped -> live)`

Table 1. Property observing part and specification for frequently used LTL formulae

$$
\begin{array}{lll}
\pi \models p & \text{iff} & p \in L(\pi(0)) \\
\pi \models \neg g & \text{iff} & \pi \not\models g \\
\pi \models g \wedge h & \text{iff} & \pi \models g \text{ and } \pi \models h \\[4pt]
\pi \models \mathbf{X}\, g & \text{iff} & |\pi| > 0 \text{ and } \pi_1 \models g \\
\pi \models \mathbf{F}\, g & \text{iff} & \text{there exists } i \leq |\pi| \text{ with } \pi_i \models g \\
\pi \models \mathbf{G}\, g & \text{iff} & \pi_i \models g \text{ for all } i \leq |\pi| \\[4pt]
\pi \models g\,\mathbf{U}\,h & \text{iff} & \text{there exists } i \leq |\pi| \text{ with} \\
 & & \pi_i \models h \text{ and } \pi_j \models g \text{ for all } j < i \\
\pi \models g\,\mathbf{R}\,h & \text{iff} & \text{for all } i \leq |\pi| \text{ either } \pi_i \models h \text{ or} \\
 & & \text{ there exists } j < i \text{ with } \pi_j \models g
\end{array}
$$

A path π is defined to be *initialized* iff $\pi(0) \in I$. Then an LTL formula f is called *valid*, more precisely *universally* valid, for a Kripke structure K, written $K \models_\forall f$, iff $\pi \models f$ for all initialized and maximally expanded paths π of K. In particular, if $S = \emptyset$, then all LTL formulae are valid. Note, that our notion of (universal) validity matches the classical semantics, if T is total and $I \neq \emptyset$. For the dual notion of *existential validity* we define $K \models_\exists f$ iff there exists an initialized and maximally expanded path π with $\pi \models f$. If no doubt can arise, we write $K \models f$ for $K \models_\forall f$.

Note, that existential validity is slightly different from the CTL semantics [12] of $\mathbf{E}f$, assuming $\mathbf{E}f$ is a CTL formula. Then $K \models \mathbf{E}f$ in CTL semantics holds iff *for all* initial states $s \in I$, the zero length path (s) can be expanded to a fully expanded path π with $\pi \models f$. In our definition of existential validity s is existentially quantified. The two notions only match if there is a unique initial state ($|I| = 1$).

If an LTL formula f does not hold in a Kripke structure K, a maximally expanded path π of K can be found, with $\pi \models \neg f$. If π is infinite we additionally assume that π is *lasso-shaped* as in [4]. A lasso-shaped path has the general structure shown in Fig. 1: starting from an initial state s_0, the loop state s_l is reached after l steps and after $k - l$ steps the loop closing state s_k is reached from which there is a transition back to the loop state s_l. In this case we define the *counterexample* for f representing π to consist of the first $k+1$ states of π and the backward loop position l. The length of the counterexample is defined as k.

A fairness constraint is a subset of S. A path π is called *fair* wrt. *one* fairness constraint $F^i \subseteq S$ iff some state in F^i occurs infinitely often on π. If π is fair, then π is infinite, written $|\pi| = \infty$. Formally we add a fifth component F to a Kripke structure, where F is a possibly empty list of fairness constraints $F = (F^1, \ldots, F^m)$. Then a path is fair for K iff it is fair wrt. every F^i. The semantics of structures with fairness constraints is defined as in the unfair case, except that all quantified paths are required to be fair.

Validity of CTL* formulae (and, hence, also LTL formulae) is preserved under bisimulation equivalence [6,10]. Two Kripke structures $K = (S,T,I,L)$ and $\hat{K} = (\hat{S},\hat{T},\hat{I},\hat{L})$ over the same set of atomic propositions are bisimulation equivalent iff there exists a relation $\sim \subseteq S \times \hat{S}$ with the following properties: Let $s \in S$ and $\hat{s} \in \hat{S}$ with $s \sim \hat{s}$.

1. The labeling has to match, that is $L(s) = \hat{L}(\hat{s})$.
2. For all $s' \in S$ with $T(s,s')$ there has to exist $\hat{s}' \in \hat{S}$ with $\hat{T}(\hat{s},\hat{s}')$ and $s' \sim \hat{s}'$.
3. For all initial states $s \in I$ there has to be an initial state $\hat{s} \in \hat{I}$ with $s \sim \hat{s}$.

The dual properties of (2) and (3) where K and $\hat{K}$ are reversed have to hold as well.

Bisimulation with fairness is defined by expanding the transition based definition stated above to whole fair paths as in [10]: the additional requirement is that for all fair paths $\pi \in \Pi$ there exists a fair path $\hat{\pi} \in \hat{\Pi}$ with $\pi \sim \hat{\pi}$, where $\pi \sim \hat{\pi}$ iff $\pi(i) \sim \hat{\pi}(i)$ for all $i \geq 0$.

The state space of a Kripke structure is usually constructed as the product of the valuations of a set of variables. For a large fraction of the models occurring in practice the transition relation can be written as a set of functions such that each function defines the next state value of a variable in terms of the current valuations of some set of variables. Cone of influence reduction [10] removes all variables from a model that do not influence its behavior with respect to a given specification. The cone of influence is defined as the smallest set of variables that includes all variables mentioned in the specification and, recursively, each variable mentioned in the next state function of any variable in the cone of influence. Formally, let $K = (S,T,I,L)$ be a Kripke structure, f be a property, U the set of variables mentioned in f, and $dep(v)$ be the set of variables defining the next state of v. Then, $coi(K, f)$ is the smallest set such that $U \subseteq coi(K, f)$ and if $v \in coi(K, f)$ then $dep(v) \subseteq coi(K, f)$.

4 Radius and Diameter

A path π of a Kripke structure $K = (S,T,I,L)$ *leads* from state s to state t, if it is finite, $\pi(0) = s$ and $\pi(|\pi|) = t$. In this case t is called *reachable* from s. Similarly we say t is reachable from a set $\hat{S} \subseteq S$, if there is an $s \in \hat{S}$ and t is reachable from s.

The *distance* $\delta(K,s,t)$ between s and t in K is the length $|\pi|$ of a path π in K with minimal length that leads from s to t. The distance is infinite, written $\delta(K,s,t) = \infty$, if t is not reachable from s. The *diameter* $d(K)$ of a Kripke structure K is the maximal distance between two reachable states in K. The distance $\delta(K,\hat{S},t)$ of a state t from a set of states $\hat{S} \subseteq S$ is the minimum $\delta(K,s,t)$ of all $s \in \hat{S}$. Finally the *radius* is defined as the maximal $\delta(K,I,t)$ of all t reachable from I.

Thus the diameter is the maximal number of transitions it takes to reach all states reachable from a state, or just the length of the longest shortest finite path. The radius is the maximal number of transitions it takes to reach a state reachable from the initial states. As an example consider the Kripke structure of Fig. 5. It models a 2-bit counter with initial state 0 from which all the other states can be reached in one step. As usual, the initial states are marked by an incoming edge without source state. L is represented by marking states with sets of atomic propositions, e.g., state 3 is the only state in which p holds. In this example the diameter is 3, which is maximal, since the only path that leads from state 1 back to the initial state 0 has a length of 3 transitions. The radius however is only 1. This example can be generalized to an n-bit counter with diameter $2^n - 1$ and constant radius 1.

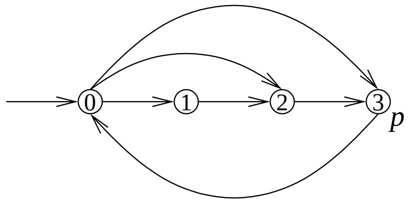

Fig. 5. Kripke structure with constant radius but large diameter.

The set of reachable states $R(K)$ is defined as all states that can be reached from an initial state. Since the validity of an LTL formula is always defined with respect to initialized paths, it is clear that we can simply remove all non reachable states from a Kripke structure without affecting the validity of LTL formulae. Formally, for any $\hat{S} \subseteq S$ we define $K_{\hat{S}} \equiv (\hat{S},\hat{T},\hat{I},\hat{L})$ with $\hat{T} \equiv T \cap (\hat{S})^2$, $\hat{I} \equiv I \cap \hat{S}$ and $\hat{L} \equiv L|_{\hat{S}}$. Then it is easy to see that $K \models f$ iff $K_{R(K)} \models f$. Since the maximal distance of *all* states may be much larger than the distance of reachable states, it is often advantageous in practice to restrict model checking to $K_{R(K)}$.

Let $L^{-1}: A \rightarrow P(S)$ be the reverse of L, e.g., $s \in L^{-1}(p)$ iff $p \in L(s)$. Then L^{-1} is lifted to arbitrary boolean expressions f over A by defining $L^{-1}(f) \equiv L^{-1}(g) \cap L^{-1}(h)$ for $f = g \wedge h$ and $L^{-1}(f) \equiv S \backslash L^{-1}(g)$ for $f = \neg g$ etc. and we write K_f for $K_{L^{-1}(f)}$.

Then we call $d(K_f)$ the *predicated diameter* of K with respect to f, or just f-diameter. Similarly the *predicated radius* of K with respect to f is $r(K_f)$. In particular we are interested in $K_{\neg p}$ which is obtained from K by deleting all states in which p holds. Then the '$\neg p$'-diameter of K turns out to be the longest shortest finite path in K on which p does not hold. In the n-bit counterexample generalized from Fig. 5, where p only holds in state $2^n - 1$ the '$\neg p$'-diameter is $2^n - 3$, for $n > 1$, which is the length of the single path that leads from state 1 to state $2^n - 2$.

The Kripke structure of Fig. 6 models a variant of a 2-bit counter with an additional set-state. The counter starts in state 0 and increments the state index up to 3 and then wraps back to 0. Additionally, at any instant of time, the counter may transition to the set-state *, from which after the following time step any other state can be reached. This example can again be generalized for arbitrary n. The diameter and the radius are both constant, because every state can be reached

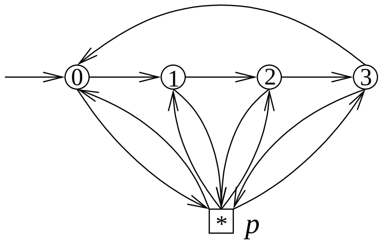

Fig. 6. Kripke structure with constant diameter but larger '$\neg p$'-diameter.

after at most two steps from any other state, by going over the set-state if necessary.

It is particular interesting about this example, that the '$\neg p$'-diameter is $2^n - 1$ and thus exponential in n. It is obtained by calculating the diameter after removing the set-state, the only state in which p holds. The opposite is also possible: consider a Kripke structure in which states far away from the initial states are only reachable through states, that are close to the initial states and in which p holds. Then removing those close states will cut of all the far away states, making them unreachable. This will result in a much smaller radius, which is the '$\neg p$'-radius of the original Kripke structure.

Note, in the construction of $K_{\hat{S}}$, which is essential for the proof of our diameter bounds further down, T being total does not imply the totality of $\hat{T}$. This is the reason, why we could not assume a total transition relation in the beginning and had to give non standard semantics.

In the rest of the article we assume that the *original* Kripke structure under consideration has a total transition relation. Then all maximally expanded paths are infinite and all counterexamples can be assumed to be lasso-shaped. Only for $K_{\neg p}$ we have to take maximally expanded finite paths into account.

5 Completeness Bounds for Simple Safety and Liveness

Some verification algorithms work in an iterative manner by increasing the value of a parameter until truth or falsity of a formula can be concluded. For example, bounded model checking [4] searches for counterexamples up to a specified length. This parameter is increased until a bound on the maximal length of a potential counterexample has been reached. The number of image computations performed in BDD-based reachability checking [27] is a similar example. Performing only a minimal number of iterations while still ensuring a correct result can help to limit the resources required for verification. This idea is captured by the notion of *completeness threshold* introduced by Kroening and Strichman in [23] in the context of bounded model checking. In the following we generalize their notion to a broader class of verification algorithms and parameters. The new notion is then used to rephrase the well-known fact that the radius of a Kripke structure is a sufficient bound to verify safety properties and a new bound is derived for simple liveness properties.

5.1 Completeness Bound

Let K be a Kripke structure, let f be an LTL formula, let i, m, n be natural numbers. A *semantic approximation* is a function

$$V : (K, f, i) \mapsto \{false, true\}.$$

V is *stable at n* iff

$$\forall m \,.\, m \geq n \Rightarrow (V(K, f, m) \Leftrightarrow V(K, f, n)).$$

If a semantic approximation V is stable at some n then the limit $lim(V, K, f) := \lim_{i \to \infty} V(K, f, i)$ exists. V is *correct* for K and f, iff it is stable at some n and

$$lim(V, K, f) \Leftrightarrow K \models f.$$

Finally, we call V a *verification function* for K and f iff it is a correct semantic approximation for K and f. Informally, a verification function converges to the correct answer to $K \models f$ for any increasing sequence of parameter values.

A parameter value n is a *completeness bound* for V, K and f, denoted $cb(V, K, f)$, iff V is correct and stable at n. We are particularly interested in the minimal completeness bound for K and f, denoted $cb_{min}(V, K, f)$.

Remark 1. In other words, we know that we have reached a completeness bound for $cb(V, K, f)$, and thus, that $V(K, f, n)$ is the correct answer to $K \models f$, if further increasing the value of the parameter n will not lead to a different result.

Most verification functions are monotonically increasing or decreasing in n for the order $false < true$ for a given model and property. Then, a correct result is obtained as soon as V changes from *true* to *false* or vice versa:

Corollary 1. *Let $V(K, f, i)$ be a verification function monotonic in i. Then*

$$(\exists m, n \,.\, m < n \wedge (V(K, f, m) \Leftrightarrow \neg V(K, f, n))) \Rightarrow \\ (V(K, f, n) \Leftrightarrow K \models f)$$

The notion introduced above can be extended to sequences of other partially ordered sets of parameter values, e.g., to the set of variables used for detection of loops in the state-recording translation (see also Sect. 2.6). Note, that if the order is not linear a minimal completeness bound of a model and a property might not be unique.

5.2 Safety Properties

Given a concrete Kripke structure the (universal) validity of *simple safety properties* of the form $\mathbf{G}p$ can be checked by traversing all reachable states and checking whether p holds for each state reached. In symbolic model checking [27] the search is usually organized as breadth first search (BFS), starting with the set of initial states and adding images. An image is calculated as the set of states that can be reached in one step from the set of states reached so far. This process is continued until no new states can be added or a state violating p is found.

Using the notation introduced above we can define a function smc_{safe} that yields *false* iff a violating state is reachable from an initial state in at most i steps. Let $K = (S,T,I,L)$ be a Kripke structure, let $f = \mathbf{G}p$ be a simple safety property. Then

$$smc_{\mathrm{safe}}(K,f,i) = \begin{cases} \mathit{false} & \text{if } \exists s \in I \; \exists t \in S \,.\, \delta(K,s,t) \leq i \wedge p \notin L(t) \\ \mathit{true} & \text{otherwise} \end{cases}$$

Clearly, smc_{safe} is a verification function monotonic in i. If f holds in K, smc_{safe} is *true* for each i. If $K \not\models f$, then smc_{safe} is *true* as long as i is smaller than the distance of the closest violating state, *false* for any other i. Therefore, for simple safety properties the minimal completeness bound is 0 if the property holds, the distance of the closest violating state otherwise.

Neither the truth of the property nor the distance of the closest violating state are usually known in advance. Without additional information verification terminates when either a violating state is found (Cor. 1) or all reachable states have been traversed (Rem. 1). In the latter case the states with the largest distance to the set of initial states determine the number of image computations. This number turns out to be exactly the radius of the Kripke structure. It is a completeness bound for smc_{safe} and any Kripke structure and simple safety property. In the same manner the radius can be used as maximal bound for bounded model checking of simple safety properties.

5.3 Liveness Properties

In bounded model checking, a generic counterexample of length k is represented symbolically by a boolean formula. The formula is a conjunction of k copies of the symbolic representation of the (total) transition relation, an initial state constraint and a loop closing condition. Then, to falsify a simple liveness property, of the form $\mathbf{F}p$, e.g., disprove its validity, with respect to the universal semantics, the states are further restricted to fulfill $\neg p$. From a satisfying assignment for the resulting boolean formula a counterexample can be extracted.

However, if the liveness property is valid for the given Kripke structure, then for any k the generated boolean formula remains unsatisfiable. Since we can not test infinitely many values of k, the question is, up to which k do boolean formulae have to be generated and checked for unsatisfiability before validity of the liveness property can be concluded.

We can define a monotonic verification function bmc_{live} similar as above that yields *false* if a lasso-shaped counterexample of length $\leq i$ exists. The minimal completeness bound is 0 if the property holds for a model, the length of the shortest counterexample otherwise.

In [4,23] it has been observed that the *recurrence diameter*, which is the longest cycle free path, and its initialized variant, the *recurrence radius*, are upper bounds for the minimal completeness bound of simple liveness properties. Note, that the diameter is *not* an upper bound for the minimal completeness bound of $\mathbf{F}p$, as the example in Fig. 6 shows, where the diameter is constant, but the length of the single counterexample is linear in 2^n, the number of states. As this example shows the search for a lasso-shaped counterexample has to be restricted to $K_{\neg p}$. This leads to one of our main results: the minimal completeness bound for simple liveness properties $\mathbf{F}p$ is linear in the '$\neg p$'-predicated diameter. The exact relation is stated in the following:

Theorem 1.

$$cb_{min}(bmc_{\mathrm{live}},K,\mathbf{F}p) \;\leq\; r(K_{\neg p}) + d(K_{\neg p}) \;=\; \mathbf{O}(d(K_{\neg p}))$$

Proof. $r(K_{\neg p}) + d(K_{\neg p})$ is a sufficient bound on the length of counterexamples for $\mathbf{F}p$ in K. The proof works as follows: given an arbitrary counterexample of length k, which represents an infinite initialized path π of K with $\pi \models \neg\mathbf{F}p$, we construct an infinite initialized path π^* of K with $\pi^* \models \neg\mathbf{F}p$. Then we show that π^* is represented by a counterexample of maximal length $r(K_{\neg p}) + d(K_{\neg p})$.

For the construction, let $\pi = (s_0,\ldots,s_l,\ldots,s_k,\ldots)$ with $0 \leq l \leq k$ and $(s_k,s_l) \in T$. Without loss of generality we assume $s_l \neq s_k$ if $l < k$. Otherwise k is decremented until the assumption is fulfilled. Clearly p does not hold in any of the states of π and, since $K_{\neg p}$ still contains all states violating p, this implies that π is also an initialized path in $K_{\neg p}$. Therefore there exists an initialized path $\hat{\pi}$ in $K_{\neg p}$ of maximal length $r(K_{\neg p})$ with $\hat{\pi}(|\hat{\pi}|) = s_l$.

If in π there is a self loop at the looping state, e.g., $s_l = s_k$, then the infinite initialized path $\pi^* \equiv \hat{\pi} \cdot (s_l)^\omega$, is still a path in $K_{\neg p}$ and $\pi^* \models \neg\mathbf{F}p$. It consists of the prefix $\hat{\pi}$ and an infinite repetition of the looping state s_l and can be represented by a counterexample of length $|\hat{\pi}| \leq r(K_{\neg p})$. Otherwise let $l < k$ and thus $s_l \neq s_k$ after the assumption above. Then we can find a second path $\hat{\hat{\pi}}$ in $K_{\neg p}$, with $\hat{\hat{\pi}}(0) = s_l$ and $\hat{\hat{\pi}}(|\hat{\hat{\pi}}|) = s_k$. This path leads us from s_l to s_k and is not necessarily initialized. Its length can only be bounded by $d(K_{\neg p})$.

There is a transition back from s_k to s_l. Therefore $\pi^* \equiv \hat{\pi} \cdot \hat{\hat{\pi}}_1 \cdot (\hat{\hat{\pi}})^\omega$ is an initialized infinite path of $K_{\neg p}$ and thus $\pi^* \models \neg\mathbf{F}p$. It consists of the prefix $\hat{\pi}$ concatenated with $\hat{\hat{\pi}}_1$, which is $\hat{\hat{\pi}}$ with its first state, the looping state s_l, chopped off, and an infinite repetition of $\hat{\hat{\pi}}$. The length of the counterexample representing π^* is $|\hat{\pi}| + |\hat{\hat{\pi}}| \leq r(K_{\neg p}) + d(K_{\neg p})$. The rest follows from $r(K_{\neg p}) \leq d(K_{\neg p})$. □

As a corollary we obtain, that the maximal bound for checking $\mathbf{F}p$ in BMC is $r(K_{\neg p}) + d(K_{\neg p})$. Note again, that $d(K)$ and $d(K_{\neg p})$ are in general not comparable and there are examples, see above, where either one is much larger than the other.

6 Correctness

In this section we formally establish the correctness of the state recording translation for simple liveness properties. For the proof we show that adding the loop detection part and the property observing part preserves bisimulation equivalence between the original and the transformed system. For this purpose we introduce the notion of an observer extension. Then, it remains to construct a counterexample for the original system and specification from one for the transformed system and specification and vice versa.

6.1 Observer Extensions

Both, the loop detection part and the property observing part add state variables to a system. The newly added variables determine their next-state values in terms of the variables of the original system but do not interfere with the original system. In particular, they neither change the transition relation of the original system nor do they introduce dead ends. This is called an observer [16] or monitor [19].

Let $K = (S,T,I,L)$ be a Kripke structure, let O be a set of states. $K^{\mathbf{O}} = (S^{\mathbf{O}},T^{\mathbf{O}},I^{\mathbf{O}},L^{\mathbf{O}})$ is an *observer extension* of K with O, iff

1. $S^{\mathbf{O}} = S \times O$
2. $(\,(s,o),(s',o')\,) \in T^{\mathbf{O}} \Rightarrow (s,s') \in T$
3. $\forall (s,s') \in T \,.\, \forall o \in O \,.\, \exists o' \in O \,.\, (\,(s,o),(s',o')\,) \in T^{\mathbf{O}}$
4. $s \in I \Leftrightarrow (\exists o \in O \,.\, (s,o) \in I^{\mathbf{O}})$
5. $L^{\mathbf{O}}(\,(s,o)\,) = L(s)$

Requirements (2) and (3) ensure that the transition relation of the original system is respected and that the enhanced system can proceed if the original system can. The fourth requirement guarantees that each initial state of the enhanced system has a counterpart in the original system and vice versa. The labeling of the states in the enhanced system is defined by the component from the original system.

Let $K = (S,T,I,L)$ be a Kripke structure, O be a set of states, and $K^{\mathbf{O}} = (S^{\mathbf{O}},T^{\mathbf{O}},I^{\mathbf{O}},L^{\mathbf{O}})$ be an observer extension of K with O. Let ρ be the projection of $S^{\mathbf{O}}$ on S, i.e., $\rho(\,(s,o)\,) = s$. Then

Lemma 1. *K and $K^{\mathbf{O}}$ are bisimulation equivalent.*

Proof. Consider $\sim \subseteq S \times S^{\mathbf{O}}$ with $s \sim s^{\mathbf{O}} \Leftrightarrow s = \rho(s^{\mathbf{O}})$. □

6.2 Adding Loop Detection

The loop detection part is common for all translations. Let $K = (S,I,T,L)$ be a Kripke structure. Then we construct

$$K^{\mathbf{L}} = (S^{\mathbf{L}},T^{\mathbf{L}},I^{\mathbf{L}},L^{\mathbf{L}})$$

with

$$\begin{aligned}
S^{\mathbf{L}} &= S \times (S \cup \{\bot\}) \\
T^{\mathbf{L}} &= \{(\,(s,l),(s',l')\,) \mid \\
&\qquad (s,s') \in T \wedge \\
&\qquad ((l = \bot \wedge l' = s) \vee l' = l)\} \\
I^{\mathbf{L}} &= \{(s,\bot) \mid s \in I\} \\
L^{\mathbf{L}}(\,(s,l)\,) &= L(s)
\end{aligned}$$

which operates on the first state component like the original transition relation. In the second state component a previously reached original state may be recorded, nondeterministically, but at most once. We further assume that $\bot$ is a new state that does not already occur in S. It is easy to see that $K^{\mathbf{L}}$ is an observer extension of K and therefore we have

Lemma 2. *K and $K^{\mathbf{L}}$ are bisimulation equivalent.*

Note that $T^{\mathbf{L}}$ is monotonic in its second component for the order $\leq^{\mathbf{L}} \subseteq (S \cup \{\bot\})^2$ with $s \leq^{\mathbf{L}} t$ iff $s = t$ or $s = \bot$.

6.3 Adding Property Observing

The next step adds a flag that remembers whether p has ever been valid on the path so far:

$$K^{\mathbf{S}} = (S^{\mathbf{S}},T^{\mathbf{S}},I^{\mathbf{S}},L^{\mathbf{S}})$$

with

$$\begin{aligned}
S^{\mathbf{S}} &= S^{\mathbf{L}} \times \{0,1\} \\
T^{\mathbf{S}} &= \{(\,(s,live),(s',live')\,) \mid \\
&\qquad (s,s') \in T^{\mathbf{L}} \wedge \\
&\qquad (\text{if} \;\; p \in L^{\mathbf{L}}(s) \\
&\qquad\;\; \text{then} \;\; live' = 1 \\
&\qquad\;\; \text{else} \;\; live' = live)\} \\
I^{\mathbf{S}} &= I^{\mathbf{L}} \times \{0\} \\
L^{\mathbf{S}}(\,(s,live)\,) &= L^{\mathbf{L}}(s)
\end{aligned}$$

$K^{\mathbf{S}}$ is an observer extension of $K^{\mathbf{L}}$. With Lemma 2 and transitivity of bisimilarity we have

Lemma 3. *K and $K^{\mathbf{S}}$ are bisimulation equivalent.*

Note, that although $K^{\mathbf{S}}$ depends on the property being verified, the translations for all other formulae in Table 1 are also observer extensions. Since validity of CTL* formulae is preserved under bisimulation equivalence [6,10], we obtain the equivalence of $K \models \mathbf{F}p$ and $K^{\mathbf{S}} \models \mathbf{F}p$.

$T^{\mathbf{S}}$ is also monotonic in its second component, in this case for $0 < 1$.

6.4 Proving Equivalence for Simple Liveness

The final step in our translation for simple liveness consists of adding a new atomic proposition q with

$$q \in L^{\mathbf{S}}(\,((s,t),live)\,) \quad \Leftrightarrow \quad s = t \,\rightarrow\, live = 1 \tag{2}$$

Theorem 2.

$$K \models \mathbf{F}p \quad \Leftrightarrow \quad K^{\mathbf{S}} \models \mathbf{G}\,q$$

Proof. It only remains to show the equivalence of $K^{\mathbf{S}} \models_{\exists} \mathbf{G}\neg p$ and $K^{\mathbf{S}} \models_{\exists} \mathbf{F}\neg q$. First assume $K^{\mathbf{S}} \models_{\exists} \mathbf{G}\neg p$. Then there exists an infinite initialized path $\pi \in \Pi^{\mathbf{S}}$ with $p \notin L^{\mathbf{S}}(\pi(i))$ for all $i \geq 0$. Since the number of states of $S^{\mathbf{S}}$ is finite, there have to exist indices $k \geq l \geq 0$ with $\pi(k+1) = \pi(l)$. Let $\pi(i) = ((s_i,t_i),live_i)$ for $i \geq 0$ and define $\hat{\pi}(i) = ((s_i,\hat{t}_i),live_i)$ with $\hat{t}_i = \bot$ for $0 \leq i \leq l$ and $\hat{t}_i = s_l$ for $l < i \leq k+1$.

Clearly $\hat{\pi}$ is an initialized legal path of $K^{\mathbf{S}}$. By definition we have $s_{k+1} = \hat{t}_{k+1} = s_l$ and $live_i = 0$ for $0 \le i \le k+1$, since $p \notin L^{\mathbf{S}}(\hat{\pi}(j)) = L(s_j) = L^{\mathbf{S}}(\pi(j))$ for $0 \le j \le k$. From (2) we get $q \notin L^{\mathbf{S}}(\hat{\pi}(k+1))$ and $\hat{\pi}$ proves to be a witness for $\mathbf{F}\neg q$, assuming $\hat{\pi}$ is extended to an infinite path in the obvious way. Note that $T^{\mathbf{S}}$ is total since T is assumed to be total and our translation does not introduce dead ends.

For the reverse direction assume $\mathbf{F}\neg q$ holds in $K^{\mathbf{S}}$. Without loss of generality we find an initialized path $\pi \in \Pi^{\mathbf{S}}$ with $|\pi| = k+1$ and $\pi(k+1) \models \neg q$. With $\pi(i) = ((s_i, t_i), live_i)$ we deduce from (2) that $s_{k+1} = t_{k+1}$ and $live_{k+1} = 0$. From the monotonicity of $T^{\mathbf{S}}$ in its second state component, we obtain an l with $0 \le l \le k$, such that $\perp = t_0 = \ldots = t_l$ and $s_l = t_{l+1} = \ldots = t_{k+1}$. Now we construct an infinite path $\hat{\pi}$ with $\hat{\pi}(i) = ((\hat{s}_i, \hat{t}_i), live_i)$ as follows: for $0 \le i \le k$ we simply set $\hat{\pi}(i) = \pi(i)$. If $i > k$ we define $\hat{t}_i = t_{k+1}$, $\widehat{live}_i = live_{k+1}$ and $\hat{s}_i = s_{l+c}$ with $c = (i-l) \bmod (k+1-l)$. From the monotonicity of $T^{\mathbf{S}}$ in its second state component, we have $live_{k+1} = \ldots = live_0 = 0$, which implies $s_i \models \neg p$ for $0 \le i \le k$. Since these original states determine the non-validity of p for every $\hat{\pi}(i)$, and $\hat{\pi}$ is a legal initialized infinite path, it serves as witness for $\mathbf{G}\neg p$. □

6.5 *Adding Fairness*

Our translation is able to incorporate fairness. To handle a fair Kripke structure $K(S, I, T, L, F)$ we construct $K^{\mathbf{S}}(S^{\mathbf{S}}, I^{\mathbf{S}}, T^{\mathbf{S}}, L^{\mathbf{S}}, F^{\mathbf{S}})$ where $S^{\mathbf{S}}$, $I^{\mathbf{S}}$, $T^{\mathbf{S}}$, and $L^{\mathbf{S}}$ are defined as above and F is extended to

$$F^{\mathbf{S}} = (F^1 \times (S \cup \{\perp\}) \times \{0,1\}, \ldots, F^m \times (S \cup \{\perp\}) \times \{0,1\}).$$

We define $K_F^{\mathbf{S}} = (S_F^{\mathbf{S}}, I_F^{\mathbf{S}}, T_F^{\mathbf{S}}, L_F^{\mathbf{S}})$ with $S_F^{\mathbf{S}} = S^{\mathbf{S}} \times \{0,1\}^m$ and $I_F^{\mathbf{S}} = I^{\mathbf{S}} \times \{(0, \ldots, 0)\}$ by replacing each fairness constraint F^i with a state bit that remembers whether a loop state in F^i has been reached. Let $L_F^{\mathbf{S}}$ be the natural extension of $L^{\mathbf{S}}$ as before. Let (s,t,x,v), $(s',t',x',v') \in S_F^{\mathbf{S}}$ with $s, s' \in S$, $t, t' \in S \cup \{\perp\}$, $x, x' \in \{0,1\}$ and $v, v' \in \{0,1\}^m$. The transition relation $T_F^{\mathbf{S}}$ is satisfied for (s,t,x,v) and (s',t',x',v') as current and next state iff

$$\begin{aligned} & T^{\mathbf{S}}(\ ((s,t),x)\ ,\ ((s',t'),x')\ ) \ \wedge \\ & \quad \textstyle\bigwedge_{i=1}^{m} \big(v'(i) = v(i) \ \vee\ (t' \neq \perp \ \wedge\ s \in F^i \ \wedge\ v'(i) = 1)\big) \end{aligned}$$

which is again monotonic in the new fairness components of the state space. We further add a new atomic proposition q_F with

$$\begin{aligned} & q_F \in L_F^{\mathbf{S}}(\ (s,t,x,v)\ ) \ \Leftrightarrow \\ & \quad (v(1) = \ldots = v(m) = 1) \ \rightarrow\ q \in L^{\mathbf{S}}(\ ((s,t),x)\ ) \end{aligned}$$

where q is defined as for $K^{\mathbf{S}}$. We can prove a correctness result like before, now including fairness.

Theorem 3.

$$K \models \mathbf{F}p \quad \Leftrightarrow \quad K_F^{\mathbf{S}} \models \mathbf{G}\, q_F$$

The number of added state bits grows linearly in the number m of fairness constraints. This directly corresponds to the increase in size of the input for symbolic model checking. The state space $K_F^{\mathbf{S}}$ itself grows exponentially. So does the diameter and the radius. The approach seems to be feasible, at least for explicit model checking, only for a small number of fairness constraints. However, checking $\mathbf{G}\, q_F$ will always find shortest counterexamples.

An alternative approach counts the number of fairness constraints satisfied so far, similar to the well known translation of generalized Büchi automata into ordinary Büchi automata. It produces a liveness property with a single fairness constraint, which in turn is translated into a safety property. This approach is more space efficient. It requires only a logarithmic number of additional state bits. However, it fails to generate counterexample traces of minimal length. In addition, it is not clear how this *binary* encoding performs for symbolic model checking versus the *one-hot* encoding discussed before.

7 Complexity

After correctness has been established, we can now state the theoretical bounds on the overhead for verification that is introduced into a model by our translation. Our objective was to enable checking liveness properties with techniques and tools previously only used for reachability calculation or safety checking. The impact of our translations on the complexity for model checking or reachability calculation is quite reasonable.

As sketched with the example of Fig. 3, the size of a non-canonical symbolic description in program code, increases only by a small constant factor. In global (explicit) model checking [9] the complexity is governed by the number of states, which increases quadratically:

$$|S^{\mathbf{S}}| = |S| \cdot |S \cup \{\perp\}| \cdot |\{0,1\}| = |S| \cdot (|S|+1) \cdot 2 = O(|S|^2)$$

In the case of on-the-fly (explicit) model checking [13] only the size of the reachable state space $R(K^{\mathbf{S}})$ is of interest. A reachable state $(s,t) \in R(K^{\mathbf{L}})$ either contains $\perp$ as second component t, or t is reachable in K since only reachable states are recorded. Therefore $R(K^{\mathbf{L}})$ is bounded by $|R(K)| \cdot (|R(K)|+1)$. This bound is tight: a modulo n counter, like the model in Fig. 2 for $n = 4$, has $|R(K^{\mathbf{L}})| = n \cdot (n+1)$ reachable states. If $n = 4$ then every combination of $\{0, \ldots, 3\} \times \{\perp, 0, \ldots 3\}$ can be reached. Introducing the *live*-recording flag at most doubles the number:

$$|R(K^{\mathbf{S}})| \le 2 \cdot |R(K^{\mathbf{L}})| \le 2 \cdot |R(K)| \cdot (|R(K)|+1) = O(|R(K)|^2)$$

Regarding symbolic model checking with BDDs [27] we have two results. First we relate the size of reduced ordered BDDs for the transition relation of K, $K^{\mathbf{L}}$ and $K^{\mathbf{S}}$. Assuming S is encoded with $n = \lceil \log_2 |S| \rceil$ state bits, we can encode $S^{\mathbf{L}}$ with $2n+1$ boolean variables. It is important to interleave the

	blocked	interleaved	
n	#nodes	#nodes	#nodes/n
10	5146	61	6.1
12	20512	73	6.08333
14	81958	85	6.07143
16	327724	97	6.0625
18	1310770	109	6.05556
20	5242936	121	6.05
32	$*$	193	6.03125
128	$*$	769	6.00781
512	$*$	3073	6.00195
2048	$*$	12289	6.00049

Table 2. BDD sizes for Eqn. (3) ($*$ = memory limit of 1GB reached).

boolean variables for the first and second component. Otherwise the size of the BDD for the term

$$((l = \bot \wedge l' = s) \vee l' = l) \tag{3}$$

in the definition of $T^{\mathbf{L}}$ may explode. With an interleaved order it is linear in n with a factor of approx. 6. The factor has been determined empirically for large state spaces as shown in Table 2. The first column shows the original number n of state bits. The second and third columns contain the number of BDD nodes necessary to represent Eqn. (3) using a non-interleaved (blocked) or interleaved order respectively. The exact number of nodes depends on the details of the encoding of $\bot$.

Thus the size of the BDD for $T^{\mathbf{L}}$ can be bounded roughly by $6 \cdot n$ the size of the BDD for T by using the fact from [7] that computing any boolean binary operation on BDDs will produce a BDD that is linear in size with factor 1 in the size of the argument BDDs. Finally, the size of the BDD for $T^{\mathbf{S}}$ compared to the size of the BDD for $T^{\mathbf{L}}$ may increase by a linear factor in the size of the BDD representing the set of states in which p holds, which in practice is usually very small.

Similar calculations for the set of initial states show that the size of BDDs representing $K^{\mathbf{S}}$ can be bound to be linear in the size of the BDDs representing K, linear in the number of state bits, and linear in the size of the BDD representing the set of states in which p holds.

These *static* bounds do not say anything about the size of the BDDs in the fixpoint iterations. The radius of a Kripke structure is an upper bound for the number of iterations necessary to reach a fixed point (see Sect. 5). The results derived for the radius and the diameter of $K^{\mathbf{S}}$ stated in Theorem 4.4 of [2] are incorrect if $d_{\neg p} > d$.[1] As shown in Sect. 4 the predicated diameter can be much larger than the diameter itself. This is taken into account in cases 1 and 2 below.

To determine the correct radius $r^{\mathbf{S}}$ of $K^{\mathbf{S}}$ consider an initial state $s_0^{\mathbf{S}} = (s_0, \bot, 0)$ and a target state $s_t^{\mathbf{S}} = (s_t, x, y)$ with $s_0^{\mathbf{S}}, s_t^{\mathbf{S}} \in S \times (S \cup \{\bot\}) \times \{0,1\}$. If $s_t^{\mathbf{S}}$ is reachable from $s_0^{\mathbf{S}}$, $s_t^{\mathbf{S}}$ is reachable from $s_0^{\mathbf{S}}$ in at most $r^{\mathbf{S}}$ steps. This is denoted as follows:

$$s_0^{\mathbf{S}} = \begin{pmatrix} s_0 \\ \bot \\ 0 \end{pmatrix} \xrightarrow{\leq r^{\mathbf{S}}} \begin{pmatrix} s_t \\ x \\ y \end{pmatrix} = s_t^{\mathbf{S}}$$

Both enhancements to the original state space are monotonic in the added component. Therefore, depending on x and y four cases can be distinguished: either a state is saved exactly once ($x \in S$) or not ($x = \bot$), and either a state fulfilling p is encountered (y changes to 1 once and remains so) or not ($y = 0$). This gives the following cases:

1. $x = \bot$, $y = 0$: no state is saved, p must be false on each state on the path from s_0 to s_t. The length of such a path is bounded by $r_{\neg p}$:

$$\begin{pmatrix} s_0 \\ \bot \\ 0 \end{pmatrix} \xrightarrow{\leq r_{\neg p}} \begin{pmatrix} s_t \\ \bot \\ 0 \end{pmatrix}$$

2. $x = s_l$, $y = 0$: state s_l is saved, p is false on each state on the path from s_0 to s_t. Now, s_l must be reached first. From its successor, s_l', the target state s_t is reached. No p-state may be visited on the path. This results in a bound of $r_{\neg p} + d_{\neg p} + 1$:

$$\begin{pmatrix} s_0 \\ \bot \\ 0 \end{pmatrix} \xrightarrow{\leq r_{\neg p}} \begin{pmatrix} s_l \\ \bot \\ 0 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} s_l' \\ s_l \\ 0 \end{pmatrix} \xrightarrow{\leq d_{\neg p}} \begin{pmatrix} s_t \\ s_l \\ 0 \end{pmatrix}$$

3. $x = \bot$, $y = 1$: no state is saved, at least one p-state s_p is crossed on the way to s_t. s_p can be reached from s_0 in at most r steps. The "$-$" in the third component denotes a *don't care*: another state $\widehat{s_p}$ with $p \in L(\widehat{s_p})$ may be traversed before s_p is visited, having *live* already made true. From s_p', s_t can be reached in d steps giving a bound of $r + d + 1$:

$$\begin{pmatrix} s_0 \\ \bot \\ 0 \end{pmatrix} \xrightarrow{\leq r} \begin{pmatrix} s_p \\ \bot \\ - \end{pmatrix} \xrightarrow{1} \begin{pmatrix} s_p' \\ \bot \\ 1 \end{pmatrix} \xrightarrow{\leq d} \begin{pmatrix} s_t \\ \bot \\ 1 \end{pmatrix}$$

4. $x = s_l$, $y = 1$: state s_l is saved, at least one p-state s_p is crossed on the way to s_t. If s_p is reached first, this takes at most r steps. From its successor, s_l can be reached in d steps, and s_t in further d steps from s_l'. This gives an overall bound of $r + 2d + 2$:

$$\begin{pmatrix} s_0 \\ \bot \\ 0 \end{pmatrix} \xrightarrow{\leq r} \begin{pmatrix} s_p \\ \bot \\ - \end{pmatrix} \xrightarrow{1} \begin{pmatrix} s_p' \\ \bot \\ 1 \end{pmatrix} \xrightarrow{\leq d} \begin{pmatrix} s_l \\ \bot \\ 1 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} s_l' \\ s_l \\ 1 \end{pmatrix} \xrightarrow{\leq d} \begin{pmatrix} s_t \\ s_l \\ 1 \end{pmatrix}$$

The result is the same if s_l is reached first:

$$\begin{pmatrix} s_0 \\ \bot \\ 0 \end{pmatrix} \xrightarrow{\leq r} \begin{pmatrix} s_l \\ \bot \\ - \end{pmatrix} \xrightarrow{1} \begin{pmatrix} s_l' \\ s_l \\ - \end{pmatrix} \xrightarrow{\leq d} \begin{pmatrix} s_p \\ s_l \\ - \end{pmatrix} \xrightarrow{1} \begin{pmatrix} s_p' \\ s_l \\ 1 \end{pmatrix} \xrightarrow{\leq d} \begin{pmatrix} t \\ s_l \\ 1 \end{pmatrix}$$

Bounds on the diameter $d^{\mathbf{S}}$ can be obtained similarly by starting in an arbitrary state $s_s^{\mathbf{S}} = (s_s, x_s, y_s)$. This leads to the following reformulation of Theorem 4.4 in [2]:

[1] We use the following short-hand notations if no doubt can arise: $d = d(K)$, $d^{\mathbf{S}} = d(K^{\mathbf{S}})$, $d_{\neg p} = d(K_{\neg p})$ and similarly for r.

Theorem 4.

$$r^{\mathbf{S}} \le max\{r+2d+2, r_{\neg p}+d_{\neg p}+1\} \;= \mathbf{O}(max\{d, d_{\neg p}\})$$

and

$$d^{\mathbf{S}} \le max\{3d+2, 2d_{\neg p}+1\} \;= \mathbf{O}(max\{d, d_{\neg p}\}).$$

Note, that if halt optimization is applied, the radius of $K^{\mathbf{S}}$ is reduced by d in cases 3 and 4: $r^{\mathbf{S}} = max\{r+d+2, r_{\neg p}+d_{\neg p}+1\}$.

If breadth-first search is used for reachability analysis of $K^{\mathbf{S}}$ the algorithm will either reach a fixed point or find a counterexample after at most $r^{\mathbf{S}}+1$ iterations. However, if the property under consideration is false there is always a shortest counterexample whose length is equal to the bmc_{live}-minimal completeness bound for K and $\mathbf{F}p$. As the state-recording translation finds a shortest counterexample the fixed point computation may already terminate after $cb_{min}(bmc_{\text{live}}, K, \mathbf{F}p)+1$ iterations. Note that the translated system needs one step to detect a loop and update the *live* flag.

8 Real-World Examples

In this section we report on a series of experiments with examples of non-trivial complexity. Most examples were taken from a collection of benchmarks [35] by Bwolen Yang for *SMV*, one is from previous work of the authors [30]. Three classes of properties were checked: $\mathbf{F}p$, $\mathbf{GF}p$, and $\mathbf{G}(p \to \mathbf{F}q)$.

The experiments were performed with *Cadence SMV* (build 08-20-01) [26] on a PC with an Intel Pentium III at 800 MHz and 1.5 Gbytes RAM running Linux 2.2.19. Model checking was restricted to the reachable states, a variable order was provided explicitly in each case. We set a wall clock limit of $1h$. Tables 3 – 6 give the results. In each table the first column states the class of the property checked. The second column gives the name of the model. Apart from Tab. 6 the third column states whether the property is true or false. The remaining columns list the results. The headings are *live* for the original model using standard liveness checking, *l2s* for the translated, unoptimized model, and *var*, *halt*, and *var + halt* for the translated model with variable, halt, and both optimizations applied. Table 3 shows time and space requirements. Table 4 states the number of iterations performed to check the property, that is, excluding iterations to construct a counterexample. The number of variables in the cone of influence and the size of the reachable state space is given in Tab. 5 for selected examples. Finally, Tab. 6 compares the lengths of counterexamples found for the original and the translated model.

To obtain a good variable order, the original variables were interleaved with their copies introduced by the translation. Some trials showed that a good variable order for the original model also seems to give good results for the translated model. Therefore, we used the variable order of [35] or of [8] if one was provided. A good position for the property observing variables depends on the property being verified. We did not apply further optimizations but placed the variables from the property observing part and the remaining variables from the loop detection part at the end of the variable order.

		Length in # states	
property	model	live	l2s
$\mathbf{F}p$	1394-2-2-false	13	10
	1394-3-2-false	11	11
	1394-4-2-false	19	16
	dme-03-false	1	1
	dme-04-false	1	1
	dme-05-false	—	1
$\mathbf{GF}p$	reactor-bc57-sensors-2	103	103
	reactor-motors-stuck-2	319	315
$\mathbf{G}(p \to \mathbf{F}q)$	prod-cons-1	39	24
	prod-cons-3	27	24

Table 6. Length of counterexamples

Verification of the translated model is feasible. The most optimized version is usually 5 to 50 times slower and requires 3 to 30 times more memory than the usual liveness checking algorithm. Note, that it was not our intention to provide an improved algorithm for liveness checking but to make liveness checking possible if reachability analysis is the only available option. Still, in the optimized translated model a bug in the *dme* model is found much faster than with standard liveness checking.

Both optimizations yield performance improvements in most cases. Variable optimization can speed up verification by more than two orders of magnitude. Within the given resource bounds the *dme* model could not be verified in the translated version without variable optimization. Our translated specification refers to each variable that is copied and compared. All such variables are included in the cone of influence of the translated specification. Variables not in the cone must not be used for loop detection if cone of influence reduction is to be applied. An example is the *abp* model. It contains a data-path of variable width (1 and 4 bits in our experiments), that is not in the cone of influence of the property verified. With variable optimization (and, as enabled by default, cone of influence reduction in Cadence SMV) verification time and space are independent of the number of bits in the data-path, exponential otherwise. Halt optimization shortens the radius of the model if the property is true for all but one model (see Tab. 4). The reachable state space is cut for both, true and false properties (Tab. 5). The resulting speedup is usually between 2 and 10. Most valid instances of the *reactor* model cannot be verified without halt optimization. Both optimizations are independent and may be combined.

Often, a shorter counterexample is produced for the translated model, see Tab. 6. For *1394*, the counterexample given by the original liveness algorithm includes an invocation of a sub-protocol not necessary to falsify the property. The counterexamples obtained for *prod-cons* by the original and the transformed models are semantically different. In addition, the counterexamples produced by the original algorithm contain a number of context switches between processes where the target process cannot act (i.e., nothing changes between two states apart from the *running* variable).

			CPU time [s]					# BDD nodes				
property	model	truth	live	var + halt	halt	var	l2s	live	var + halt	halt	var	l2s
$\mathbf{F}p$	1394-2-2-true	t	0.60	1.13	2.20	1.42	3.27	66584	127148	195691	150631	251357
	1394-3-2-true	t	7.63	11.08	20.59	16.47	32.11	656916	666821	1101901	973403	1740630
	1394-4-2-true	t	382.72	316.92	707.57	731.53	1313.78	12748065	11612358	22156965	20665581	37671970
	1394-2-2-false	f	1.06	1.13	2.16	1.18	2.22	84661	121640	199030	125386	225251
	1394-3-2-false	f	6.73	7.44	14.50	11.25	17.53	538562	552084	804731	725920	1557995
	1394-4-2-false	f	397.73	270.76	536.86	453.28	801.45	12968071	10886232	18262702	20654223	26241966
	dme-03-true	t	112.48	369.47	—	1142.82	—	336311	5987369	—	21848216	—
	dme-04-true	t	414.44	—	—	—	—	1282565	—	—	—	—
	dme-05-true	t	1537.88	—	—	—	—	5116176	—	—	—	—
	dme-03-false	f	404.29	1.48	—	1.37	—	308902	202179	—	198982	—
	dme-04-false	f	2351.86	1.87	—	2.40	—	1079160	284297	—	428931	—
	dme-05-false	f	—	5.39	—	4.18	—	—	687726	—	641885	—
	p-queue	t	0.20	0.23	0.26	1.04	5.37	31214	48630	53176	102627	289409
$\mathbf{GF}p$	abp1	t	0.08	0.49	3.87	0.71	7.83	3573	42782	189528	57747	340689
	abp4	t	0.09	0.51	74.85	0.70	597.89	3573	42782	2880808	57747	21299133
	reactor-base-1	t	1.19	13.56	13.47	517.22	524.07	87849	322271	324984	5751743	5982628
	reactor-base-2	t	1.74	145.93	161.10	525.17	537.21	102961	2604819	2710108	5672905	5419344
	reactor-bc56-sensors-1	t	6.25	107.12	106.88	—	—	373993	1660326	1920631	—	—
	reactor-bc56-sensors-2	t	7.88	2319.06	2454.00	—	—	400194	22702256	23314130	—	—
	reactor-bc57-sensors-1	t	12.35	268.85	280.82	—	—	701793	5150899	5373528	—	—
	reactor-bc57-sensors-2	f	191.28	229.00	266.02	213.72	224.35	1151399	6778798	7314282	6251219	6288691
	reactor-motors-stuck-1	t	12.28	152.90	148.60	—	—	917665	3051950	2954656	—	—
	reactor-motors-stuck-2	f	33.44	670.01	669.09	1309.38	1247.58	1109592	14863244	14367833	28770893	30509639
	reactor-valves-gates-1	t	38.15	939.44	1003.61	—	—	1429572	13039409	10237659	—	—
	reactor-valves-gates-2	t	43.53	—	—	—	—	2001444	—	—	—	—
$\mathbf{G}(p \rightarrow \mathbf{F}q)$	guidance	t	0.46	8.21	95.63	49.43	707.02	41014	504831	2062249	2959929	13377306
	prod-cons-1	f	4.43	7.67	15.07	12.00	27.75	172894	468930	769734	623875	1214313
	prod-cons-3	f	0.66	3.66	7.46	6.98	25.00	39951	219839	437287	432719	1311126
	prod-cons-4	t	0.38	11.72	31.96	1899.24	—	31542	498038	975382	44680697	—
	production-cell-1	t	0.28	2.94	2.96	9.64	9.94	36148	158262	158262	429259	429259
	production-cell-3	t	0.25	0.83	0.90	7.66	7.34	35278	82648	82648	375484	375484

Table 3. Time and memory needed for verification

			# iterations [all (fw + bw)]				
property	model	truth	live	var + halt	halt	var	l2s
$\mathbf{F}p$	1394-2-2-true	t	54 (15 + 39)	15 (15 + 0)	15 (15 + 0)	19 (19 + 0)	19 (19 + 0)
	1394-3-2-true	t	60 (17 + 43)	16 (16 + 0)	16 (16 + 0)	19 (19 + 0)	19 (19 + 0)
	1394-4-2-true	t	116 (27 + 89)	27 (27 + 0)	27 (27 + 0)	31 (31 + 0)	31 (31 + 0)
	1394-2-2-false	f	30 (15 + 15)	10 (10 + 0)	10 (10 + 0)	10 (10 + 0)	10 (10 + 0)
	1394-3-2-false	f	33 (17 + 16)	11 (11 + 0)	11 (11 + 0)	11 (11 + 0)	11 (11 + 0)
	1394-4-2-false	f	61 (27 + 34)	16 (16 + 0)	16 (16 + 0)	16 (16 + 0)	16 (16 + 0)
	dme-03-true	t	13138 (96 + 13042)	247 (247 + 0)	— — — —	301 (301 + 0)	— — — —
	dme-04-true	t	22167 (117 + 22050)	— — — —	— — — —	— — — —	— — — —
	dme-05-true	t	38734 (142 + 38592)	— — — —	— — — —	— — — —	— — — —
	dme-03-false	f	47532 (96 + 47436)	1 (1 + 0)	— — — —	1 (1 + 0)	— — — —
	dme-04-false	f	129681 (117 + 129564)	1 (1 + 0)	— — — —	1 (1 + 0)	— — — —
	dme-05-false	f	— — — —	1 (1 + 0)	— — — —	1 (1 + 0)	— — — —
	p-queue	t	16 (12 + 4)	2 (2 + 0)	3 (3 + 0)	16 (16 + 0)	18 (18 + 0)
$\mathbf{GF}p$	abp1	t	87 (19 + 68)	31 (31 + 0)	34 (34 + 0)	41 (41 + 0)	48 (48 + 0)
	abp4	t	87 (19 + 68)	31 (31 + 0)	34 (34 + 0)	41 (41 + 0)	48 (48 + 0)
	reactor-base-1	t	298 (271 + 27)	272 (272 + 0)	272 (272 + 0)	661 (661 + 0)	661 (661 + 0)
	reactor-base-2	t	369 (271 + 98)	381 (381 + 0)	381 (381 + 0)	661 (661 + 0)	661 (661 + 0)
	reactor-bc56-sensors-1	t	429 (390 + 39)	391 (391 + 0)	391 (391 + 0)	— — — —	— — — —
	reactor-bc56-sensors-2	t	496 (390 + 106)	592 (592 + 0)	592 (592 + 0)	— — — —	— — — —
	reactor-bc57-sensors-1	t	369 (302 + 67)	303 (303 + 0)	303 (303 + 0)	— — — —	— — — —
	reactor-bc57-sensors-2	f	5020 (302 + 4718)	103 (103 + 0)	103 (103 + 0)	103 (103 + 0)	103 (103 + 0)
	reactor-motors-stuck-1	t	456 (401 + 55)	407 (407 + 0)	407 (407 + 0)	— — — —	— — — —
	reactor-motors-stuck-2	f	589 (401 + 188)	315 (315 + 0)	315 (315 + 0)	315 (315 + 0)	315 (315 + 0)
	reactor-valves-gates-1	t	644 (616 + 28)	617 (617 + 0)	617 (617 + 0)	— — — —	— — — —
	reactor-valves-gates-2	t	726 (616 + 110)	— — — —	— — — —	— — — —	— — — —
$\mathbf{G}(p \rightarrow \mathbf{F}q)$	guidance	t	68 (41 + 27)	56 (56 + 0)	82 (82 + 0)	76 (76 + 0)	106 (106 + 0)
	prod-cons-1	f	58 (48 + 10)	24 (24 + 0)	24 (24 + 0)	24 (24 + 0)	24 (24 + 0)
	prod-cons-3	f	114 (48 + 66)	24 (24 + 0)	24 (24 + 0)	24 (24 + 0)	24 (24 + 0)
	prod-cons-4	t	132 (48 + 84)	68 (68 + 0)	69 (69 + 0)	120 (120 + 0)	— — — —
	production-cell-1	t	112 (81 + 31)	110 (110 + 0)	110 (110 + 0)	173 (173 + 0)	173 (173 + 0)
	production-cell-3	t	90 (81 + 9)	83 (83 + 0)	83 (83 + 0)	146 (146 + 0)	146 (146 + 0)

Table 4. Iterations performed to check property

			# state holding booleans			# reachable states				
property	model	truth	live	var	l2s	live	var + halt	halt	var	l2s
$\mathbf{F}p$	1394-2-2-true	t	60	96	128	1.07856e+08	1.09334e+08	5.40174e+08	1.1707e+08	1.21886e+09
	1394-2-2-false	f	60	96	128	1.07856e+08	1.10073e+08	5.91601e+08	1.14829e+08	9.86541e+08
	dme-03-true	t	54	164	—	6579	2.3233e+20	—	4.67112e+20	—
	dme-03-false	f	54	161	—	6579	1.80144e+16	—	1.80144e+16	—
	p-queue	t	39	79	86	1824	275	15510	64739	6.06062e+06
$\mathbf{GF}p$	abp1	t	17	39	50	180	11202	229326	18622	661506
	abp4	t	17	39	74	180	11202	9.81073e+09	18622	3.02828e+10
	reactor-base-1	t	65	142	144	398	2912	2912	264889	264889
	reactor-base-2	t	65	142	144	398	102293	102293	264157	264157
	reactor-bc56-sensors-1	t	69	150	152	6023	80860	80888	—	—
	reactor-bc56-sensors-2	t	69	150	152	6023	3.60834e+06	3.6133e+06	—	—
	reactor-valves-gates-1	t	77	166	168	1.80469e+06	4.03702e+07	4.08945e+07	—	—
	reactor-valves-gates-2	t	77	—	—	1.80469e+06	—	—	—	—
$\mathbf{G}(p \rightarrow \mathbf{F}q)$	guidance	t	55	113	193	3.29395e+10	5.10475e+18	2.59509e+25	5.45806e+19	3.05457e+26
	prod-cons-1	f	28	56	62	211144	3.9479e+07	2.42509e+08	1.72519e+08	1.08308e+09
	production-cell-1	t	54	111	111	81	1257	1257	6415	6415

Table 5. Size of state space

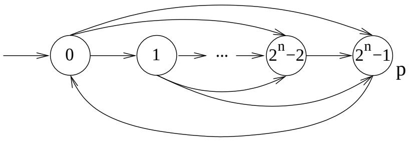

Fig. 7. Forward jumping counter

9 A Forward Jumping Counter

Our translation may lead to a model that can be verified exponentially faster. Consider the *n*-bit counter shown in Fig. 7. It can jump forward from state i to an arbitrary state $j > i$. Only in the last state p is true. For the correct version $\mathbf{F}p$ holds, self-loops are added to generate an erroneous version. A standard algorithm for symbolic model checking [10] needs $\mathbf{O}(2^n)$ backward iterations to verify the correct counter. If the state-recording translation is applied, a constant number of forward iterations suffices as $r, r_{\neg p}, d, d_{\neg p} \leq 2$.

We used the model checker of the *VIS* system (v. 1.4) [15] to verify the forward jumping counter. Apart from backward (standard) model checking *VIS* also provides an implementation of the forward model checking algorithm by Iwashita et al. [21]. The experiments were performed on an Intel PC running at 800 MHz with 1.5 Gbytes RAM, a wall clock limit was set at $1h$.

The results confirm that standard and forward model checking require exponentially many iterations while the translated version is verified with a constant number of iterations in the correct case. All algorithms can find a counterexample with a constant number of iterations.

Fig. 8 shows that both classical and forward model checking need time exponential in n. The translated variant can be checked in linear time. The standard algorithm is more than 25 % faster than forward model checking. A counterexample is found in the erroneous version in linear time by all algorithms. Standard and forward model checking give similar results for the translated variant.

10 Conclusion

We have extended our translation of liveness checking problems into safety checking problems for finite state systems. To improve applicability of our method in practice, we have provided translations for more complex formulae and optimizations to speed up verification. The feasibility of our approach is underlined by a series of experiments. In one example an exponential speedup is observed. Using the new notions of predicated radius and completeness bound we have derived revised bounds for BDD based model checking.

The current optimizations ensure that only variables are removed from the translation that don't influence the truth of the formula. While removing further variables may produce spurious counter examples, considerable speed-ups can be achieved with these reduced models. We have very promising initial results on an incremental procedure that starts with only few variables copied and compared in the translation and adds further variables until the formula is either proved true or all variables have been added.

Our tight bounds on the minimal completeness bound for liveness properties may potentially lead to faster algorithms for liveness checking in general. The performance of the counter based translation should be evaluated for low completeness bounds. Future research could evaluate how our translation can be applied to other formalisms such as process algebras. Another direction for research is to look into structural algorithms to determine bounds on the '$\neg p$'-predicated diameter, similar to the algorithms for plain diameters in [1].

Finally we would like to thank Ofer Strichman for sharing some of his insights on the completeness threshold with us.

References

1. J. Baumgartner, A. Kühlmann, and J. Abraham. Property checking via structural analysis. In E. Brinksma and K. Larsen, editors, *Computer Aided Verification, Proceedings of the 14th International Conference, CAV 2002, Copenhagen, Denmark, July 27–31, 2002*, volume 2404 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2002.
2. A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In R. Cleaveland and H. Garavel, editors, *Formal Methods for Industrial Critical Systems, Proceedings of the 7th International ERCIM Workshop, FMICS'02, Málaga, Spain, July 12–13, 2002*, Electronic Notes in Theoretical Computer Science, 66(2). Elsevier, 2002.
3. A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation, Proceedings of the 36th ACM/IEEE Conference, DAC'99, New Orleans, Louisiana, United States, June 21–25, 1999*, pages 317–320. ACM Press, 1999.
4. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, Proceedings of the 5th International Conference, TACAS '99, Amsterdam, The Netherlands, March 22–28, 1999*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.
5. A. Biere, E. Clarke, and Y. Zhu. Multiple state and single state tableaux for combining local and global model checking. In E.-R. Olderog and B. Steffen, editors, *Correct System Design, Recent Insight and Advances*, volume 1710 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 1999.
6. M. Browne, E. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional logic. *Theoretical Computer Science*, 59(1-2):115–131, 1988.
7. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
8. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model verifier. In Halbwachs and Peled [17], pages 495–499.
9. E. Clarke and A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logic of Programs, Workshop, Yorktown Heights, New York, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1982.

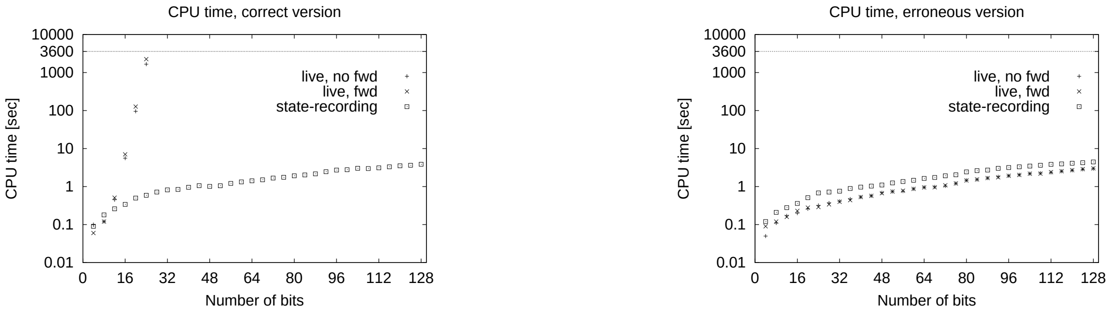

Fig. 8. Forward jumping counter - CPU Time [s]

10. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
11. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.
12. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B, Formal Methods and Semantics*, pages 995–1072. North-Holland Pub. Co., 1990.
13. R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1996.
14. D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Automated Software Engineering, Proceedings of the 16th IEEE International Conference, ASE 2001, 26–29 November 2001, Coronado Island, San Diego, CA, USA*, pages 412–416. IEEE Computer Society, 2001.
15. The VIS Group. VIS: A system for verification and synthesis. In R. Alur and T. Henzinger, editors, *Computer Aided Verification, Proceedings of the 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer, 1996.
16. N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology (AMAST '93), Proceedings of the Third International Conference on Methodology and Software Technology, University of Twente, Enschede, The Netherlands, 21–25 June, 1993*, Workshops in Computing, pages 83–96. Springer, 1994.
17. N. Halbwachs and D. Peled, editors. *Computer Aided Verification, Proceedings of the 11th International Conference, CAV '99, Trento, Italy, July 6–10, 1999*, volume 1633 of *Lecture Notes in Computer Science*. Springer, 1999.
18. K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In K. Havelund and G. Roşu, editors, *Runtime Verification, Proceedings of the First International Workshop, RV'01, Paris, France, 23 July, 2001*, Electronic Notes in Theoretical Computer Science, 55(2). Elsevier, 2001.
19. K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of the 8th International Conference, TACAS 2002, Grenoble, France, April 8–12, 2002*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.
20. T. Henzinger, O. Kupferman, and S. Qadeer. From pre-historic to post-modern symbolic model checking. In A. Hu and M. Vardi, editors, *Computer Aided Verification, Proceedings of the 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 – July 2, 1998*, volume 1427 of *Lecture Notes in Computer Science*, pages 195–206. Springer, 1998.
21. H. Iwashita, T. Nakata, and F. Hirose. CTL model checking based on forward state traversal. In *Computer-Aided Design, Proceedings of the 1996 IEEE/ACM International Conference, San Jose, CA, USA, November 10–14, 1996*, pages 82–87. IEEE Computer Society, 1996.
22. L. Jagadeesan, C. Puchol, and J. Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunications software. In P. Wolper, editor, *Computer Aided Verification, Proceedings of the 7th International Conference, Liège, Belgium, July, 3–5, 1995*, volume 939 of *Lecture Notes in Computer Science*, pages 127–140. Springer, 1995.
23. D. Kröning and O. Strichman. Efficient computation of recurrence diameters. In L. Zuck, P. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Verification, Model Checking, and Abstract Interpretation, Proceedings of the 4th International Conference, VMCAI 2003, New York, NY, USA, January 9-11, 2002*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2003.
24. O. Kupferman and M. Vardi. Model checking of safety properties. In Halbwachs and Peled [17], pages 172–183.
25. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.
26. K. McMillan. Cadence SMV. `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv`.
27. K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
28. T. Niermann and J. Patel. HITEC: A test generation package for sequential circuits. In *European Design Automation, Conference, Amsterdam, The Netherlands, February 25–28, 1991*, pages 214–218. IEEE Computer Society Press, 1991.
29. D. Peled. *Software Reliability Methods*. Springer, 2001.
30. V. Schuppan and A. Biere. Verifying the IEEE 1394 FireWire Tree Identify Protocol with SMV. *Formal Aspects of Computing*, 14(3):267–280, 2003.
31. C.-J. Seger and R. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.

32. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In W. Hunt Jr. and S. Johnson, editors, *Formal Methods in Computer-Aided Design, Proceedings of the Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1–3, 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.
33. F. Somenzi. Wring 1.1.0. `ftp://vlsi.colorado.edu/pub/Wring-1.1.0.tar.gz`.
34. F. Somenzi and R. Bloem. Efficient büchi automata from ltl formulae. In A. Emerson and P. Sistla, editors, *Computer Aided Verification, Proceedings of the 12th International Conference, CAV 2000, Chicago, IL, USA, July 15–19, 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2000.
35. B. Yang. SMV models. `http://www.cs.cmu.edu/~bwolen/software/smv-models/`.

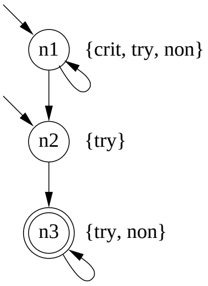

Fig. 10. Büchi automaton for $\mathbf{G}((s = try) \rightarrow (\mathbf{F}(s = crit)))$

A Example Translations

A.1 Fairness and Hierarchy

Fig. 9 shows an example of our translation that includes fairness and hierarchy. Two tasks are trying to enter a critical section. If both are in their *try*-state a non-deterministic choice decides which task is allowed to proceed. Fairness ensures that each task eventually gets its turn.

A.2 Using a Büchi Automaton

The example in Fig. 11 shows the translation of the mutex model with a specification given as a Büchi automaton. The original specification $\mathbf{G}(t0.s = try) \rightarrow (\mathbf{F}(t0.s = crit))$ states that if task 0 is trying to enter its critical section, it will eventually be able to do so. The negated specification was translated into a generalized Büchi automaton with Wring v1.1.0 (available from [33]). The resulting automaton is shown in Fig. 10.

```
MODULE task(id, turn)


VAR
  s: {non, try, crit};
ASSIGN
  init(s) := non;
  next(s) := case
    s = non: try;
    s = try & (id = turn): crit;
    s = try & !(id = turn): try;
    s = crit: non;
  esac;


FAIRNESS
  turn = id


SPEC
  AF s = crit


MODULE main


VAR
  turn: 0..1;
  t0: task(0, turn);
  t1: task(1, turn);
```

(a) original

```
MODULE task(id, turn, save, saved, on_loop)

-- unmodified part
VAR
  s: {non, try, crit};
ASSIGN
  init(s) := non;
  next(s) := case
    s = non: try;
    s = try & (id = turn): crit;
    s = try & !(id = turn): try;
    s = crit: non;
  esac;

-- loop detection part
VAR
  l2s_s: {non, try, crit};
ASSIGN
  next(l2s_s) := case
    save & !saved: s;
    1: l2s_s;
  esac;
DEFINE
  looped := saved & s = l2s_s;

-- property observing part
VAR
  fair: boolean;
ASSIGN
  init(fair) := 0;
  next(fair) := fair | on_loop & turn = id;
VAR
  live: boolean;
ASSIGN
  init(live) := 0;
  next(live) := live | s = crit;


MODULE main

-- declaration part with signal forwarding
VAR
  turn: 0..1;
  t0: task(0, turn, save, saved, on_loop);
  t1: task(1, turn, save, saved, on_loop);

-- loop detection part
VAR
  save: boolean;
  saved: boolean;
ASSIGN
  init(saved) := 0;
  next(saved) := on_loop;
DEFINE
  on_loop := save | saved;
  looped := t0.looped & t1.looped;

-- property observing part
DEFINE
  fair := t0.fair & t1.fair;

-- transformed specifications
SPEC
  AG ((looped & fair) -> t0.live)
SPEC
  AG ((looped & fair) -> t1.live)
```

(b) state-recording

Fig. 9. Original and transformed SMV code of mutex

```
MODULE task(id, turn)

-- unmodified part
VAR
  s: {non, try, crit};
ASSIGN
  init(s) := non;
  next(s) := case
    s = non: try;
    s = try & (id = turn): crit;
    s = try & !(id = turn): try;
    s = crit: non;
  esac;

MODULE main

VAR
  turn: 0..1;
  t0: task(0, turn);
  t1: task(1, turn);

-- buechi automaton
VAR
  b: {n1, n2, n3, sink};
ASSIGN
  init(b) := {n2, n3};
  next(b) := case
    b = n1 & (t0.s = non | t0.s = crit): {n1};
    b = n1 & t0.s = try: {n2, n1};
    (b=n2 | b=n3) & (t0.s=non | t0.s=try): {n3};
    1: sink;
  esac;

FAIRNESS
  turn = 0

-- buechi specification
FAIRNESS
  b = n3
SPEC
  AF 0
```

(a) original

```
MODULE task(id, turn, save, saved, on_loop)

-- unmodified part
VAR
  s: {non, try, crit};
ASSIGN
  init(s) := non;
  next(s) := case
    s = non: try;
    s = try & (id = turn): crit;
    s = try & !(id = turn): try;
    s = crit: non;
  esac;

-- loop detection part
VAR
  l2s_s: {non, try, crit};
ASSIGN
  next(l2s_s) := case
    save & !saved: s;
    1: l2s_s;
  esac;
DEFINE
  looped := saved & s = l2s_s;

MODULE main

-- declaration part with signal forwarding
VAR
  turn: 0..1;
  t0: task(0, turn, save, saved, on_loop);
  t1: task(1, turn, save, saved, on_loop);

-- buechi automaton
VAR
  b: {n1, n2, n3, sink};
ASSIGN
  init(b) := {n2, n3};
  next(b) := case
    b = n1 & (t0.s = non | t0.s = crit): {n1};
    b = n1 & t0.s = try: {n2, n1};
    (b=n2 | b=n3) & (t0.s=non | t0.s=try): {n3};
    1: sink;
  esac;

-- loop detection part
VAR
  save: boolean;
  saved: boolean;
  l2s_b: {n1, n2, n3, sink};
ASSIGN
  init(saved) := 0;
  next(saved) := on_loop;
  next(l2s_b) := case
    save & !saved: b;
    1: l2s_b;
  esac;
DEFINE
  on_loop := save | saved;
  looped := saved & b = l2s_b & t0.looped & t1.looped;

-- property observing part
VAR
  fair: boolean;
  b_fair: boolean;
ASSIGN
  init(fair) := 0;
  next(fair) := fair | on_loop & turn = 0;
  init(b_fair) := 0;
  next(b_fair) := b_fair | on_loop & (b = n3);

-- transformed buechi specification
SPEC
  AG ((looped & fair & b_fair) -> 0)
```

(b) state-recording

Fig. 11. Original and transformed SMV code of mutex with specification given as Büchi automaton