Electronic Notes in Theoretical Computer Science 66 No. 2 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume66.html 18 pages

Liveness Checking as Safety Checking

Armin Biere, Cyrille Artho, Viktor Schuppan

Computer Systems Institute, ETH Zentrum RZ H, CH-8092 Zürich, Switzerland

Abstract

Temporal logic is widely used for specifying hardware and software systems. Typically two types of properties are distinguished, safety and liveness properties. While safety can easily be checked by reachability analysis, and many efficient checkers for safety properties exist, more sophisticated algorithms have always been considered to be necessary for checking liveness. In this paper we describe an efficient translation of liveness checking problems into safety checking problems. A counter example is detected by saving a previously visited state in an additional state recording component and checking a loop closing condition. The approach handles fairness and thus extends to full LTL.

1 Introduction

Model Checking [12] is one of the most successful approaches for verifying temporal specifications of hardware and software systems. System properties are specified in temporal logic [13] for which various formalisms exist. Typically two types of properties are distinguished, safety and liveness [19]. In practical applications, safety properties are prevalent. Therefore very efficient algorithms and tools have been devised for checking safety properties. Still the specification of most systems contains liveness parts. We describe a generic translation procedure that takes a system with a liveness specification and translates it into a new system, for which a safety property is valid iff the liveness property in the original system holds.

The main motivation is to enable existing tools and techniques to check liveness which were originally supposed to work on safety properties only. For instance sequential ATPG (automatic test pattern generation) [22] can be used to check simple classes of temporal formulae [3], but general liveness properties have been out of reach. The same applies to STE (symbolic trajectory evaluation) [26,9], though a generalized version of STE has been published that can handle all ω-regular properties [27]. Both technologies have been in use in industry for over a decade [22,6] and efficient implementations exist.

For symbolic model checking [21] there is a vast literature on optimizations which are only applicable to safety. Frontier set simplification [7], dense

[23] and prioritized [8] reachability analysis all try to speed up BDD-based reachability calculation, but have not been adapted to handle liveness so far.

Forward model checking [17,15,2] is an attempt to improve on backward based symbolic model checking by visiting reachable states only and catching bugs as early as possible. It is motivated by the observation that checking safety properties amounts to reachability analysis. Forward model checking tries to use forward image calculations exclusively. Since we are able to translate liveness into safety we expect to have the same benefits without changing the model checking algorithms.

Kupferman and Vardi have developed an approach to simplify automaton-based model checking of safety properties by searching for finite violating prefixes [18]. With our translation we follow a similar goal by reducing liveness properties to safety properties and thus enabling the application of a much wider range of verification algorithms.

Our translation is structural. It respects the hierarchy of the system and can easily be applied, even manually, on the design entry level, eg in a Hardware Description Language. This is particularly useful if a tool does not include other model checking algorithms beside safety checking, and, as it is usually the case in a commercial setting, there is no access to the source code.

The basic idea is borrowed from explicit on-the-fly model checking [14] and bounded model checking [1]: a counter example to a liveness property in a finite system is *lasso-shaped*, it consists of a prefix that leads to a loop. As in [1] the major challenge is how to detect the loop. In our translation a loop is found by saving a previously visited state and later checking whether the current state already occurred.

For simple liveness properties, the result of the translation is not much larger than the original model checking problem. We also show how to handle more complicated liveness properties, for instance involving the until operator. By adding fairness constraints the technique can be extended to full LTL.

The next section elaborates on various examples to establish an intuitive understanding of the translation. In Sect. 3 we introduce the necessary formal background. We precisely define the translation in Sect. 4, prove its correctness and compare the complexity of the original and the resulting model checking problems. In the same section we mention how to extend our translation to handle fairness and LTL. Preliminary experiments in Sect. 5 show the feasibility of our approach, and Sect. 6 concludes.

2 Intuition

A counter example trace to a simple liveness property $\mathbf{AF}p$ is an infinite path where the body p of the liveness property holds nowhere, or equivalently $\neg p$ holds along the whole path. Since we restrict our models to be finite, such a trace can always be assumed to be lasso-shaped as depicted in Fig. 1. It consists of a prefix that leads to a loop, starting at the loop state s_l. From

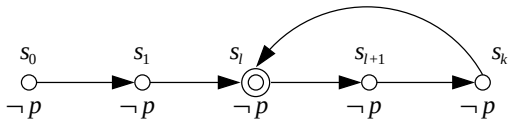

Fig. 1. A generic lasso-shaped counter example trace for $\mathbf{AF}p$.

every infinite trace in a finite model we can construct a lasso-shaped trace by closing the loop as soon as a state occurs the second time. Note that $\neg p$ still holds along the constructed path.

This observation is at the core of various model checking algorithms. Examples are explicit state algorithms for Büchi Automata [14] and unfolding liveness properties in bounded model checking [1]. With this restriction we only need to search for lasso-shaped counter examples. This is used in both translations discussed in this paper. The first translation, shown for illustration only, is called *counter based translation*. It extends the well known technique to check for *bounded liveness* only, but is not of practical value. Our main contribution is the *state recording translation.* It produces a state machine that may save at any time a previously reached state. Both translations do not only modify the property to be checked but also add additional checking components to the model while still maintaining bisimulation equivalence [12].

2.1 Counter Based Translation

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

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

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 trivial translation would just exchange $\mathbf{AF}p$ by $\mathbf{AF}^k p$ 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. To avoid this explicit expansion, our counter based translation adds a counter to the model which counts the number of states reached so far. Now it only remains to check, whenever the counter reaches the number of states of the original model, that p was found to hold in at least one state reached so far. This latter property can be checked by attaching a boolean flag to the model that remembers whether p was satisfied in the past. This last step is property dependent.

As a first example we use a modulo 4 counter with initial state `0`. In Fig. 2 an SMV program [21] and a state graph of the counter are shown. While

```
MODULE main
VAR state : -1..3;
DEFINE
  found := state = -1;
ASSIGN
  init(state) := 0;
  next(state) :=
    case
      state = 3 : 0;
      1 : state + 1;
    esac;
SPEC AF found
```

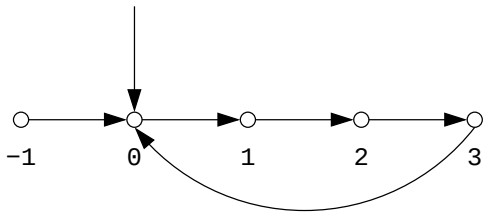

Fig. 2. A modulo 4 counter with unreachable state `-1`.

all states are reachable from `-1`, that state itself is unreachable because the counter wraps back to `0`.

The state space of the example encompasses five states. After five transitions we check whether `found` stayed false all the way from the initial state. In this case a counter example is found. Otherwise, the liveness property is valid, since every potential lasso-shaped trace of length 5 contains a state in which `found` holds.

This allows for a trivial translation of the liveness property into a safety property. The model is extended with a boolean variable `live`, which denotes whether `found` has already been true. A variable `counter` counts the number of states.

The left column of Fig. 3 shows the translated specification. The liveness property `AF found` translates into the safety property `AG (finished -> live)`. This translation is extremely inefficient, because it always requires traversing five states.

2.2 State Recording Translation

Instead of conservatively searching as long as required in the worst case, the search should terminate whenever a previously seen state s_l is traversed. Each time such a loop has been found, the liveness property p has to hold for at least one state visited before. Otherwise we have a counter example (see Fig. 1). Because state space traversal is memoryless, there is no way of explicitly expressing that property p must have been true at an earlier time as soon as we reach state s_l a second time.

The new model needs a way of "saving" a previously seen state for detecting a loop. Since we do not know beforehand whether we will see the current state again later, we use an oracle `save` that tells the model whether the current state is assumed to be the first state of a loop. To prevent overwriting the copy, another variable `saved` is used. After s_k, the last state of the loop, s_l is encountered again (see Fig. 4). At that time, the predicate `looped` becomes

```
MODULE main
VAR state   : -1..3;
    counter : 0..5;
    live    : boolean;


DEFINE
  found := state = -1;
ASSIGN
  init(state) := 0;
  next(state) :=
    case
      state = 3 : 0;
      1 : state + 1;
    esac;

  init(counter) := 0;
  next(counter) :=
    case
      counter < 5 :
        counter + 1;
      1 : counter;
    esac;
  init(live) := 0;
  next(live) :=
    live | found;
DEFINE
  finished := counter = 5;
SPEC AG (finished -> live)
```

(a) counter

```
MODULE main
VAR state : -1..3;
    loop  : -1..3;
    live  : boolean;
    save  : boolean;
    saved : boolean;
DEFINE
  found := state = -1;
ASSIGN
  init(state) := 0;
  next(state) :=
    case
      state = 3 : 0;
      1 : state + 1;
    esac;
  init(saved) := 0;
  next(saved) :=
    saved | save;
  next(loop) :=
    case
      !saved & save : state;
      1 : loop;
    esac;
  init(live) := 0;
  next(live) := live | found;
DEFINE
  looped :=
    saved & state = loop;
SPEC AG (looped -> live)
```

(b) safe

Fig. 3. A counter based translation and our new translation of the liveness property.

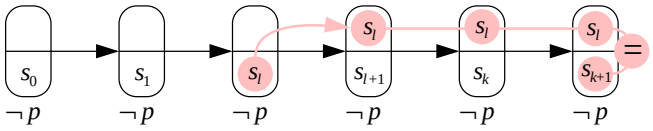

Fig. 4. Loop checking for $\mathbf{AF}p$ counter examples as reachability.

true, and property p must have been fulfilled at least once.

As visualized in Fig. 4, the loop closing condition `looped` checks, whether the current state has been visited earlier. Correspondingly Fig. 4 shows one more state than Fig. 1. Therefore `live` and `saved` should not refer to the current state. Their purpose is to remember whether `found` and respectively `save` were true in the past. In particular their initial value should be false.

2.3 Translation of fairness into safety

Fairness properties can also be translated using the same methodology. Figure 5 shows an example with two tasks t_0 and t_1 that count from 0 to 7 each. At each step, only one task is allowed to take its turn. The liveness property, stating that each task eventually arrives at state 7, can only be fulfilled if the turns are taken in a fair manner, i.e. each task eventually gets its turn.

In order to include fairness in our example, we define a new property `fair`. It records whether the fairness property is true at least once within each loop. Because `save` and `saved` are global, they are shared with the task modules.

3 Preliminaries

Our notation follows [12]. Let A be a set of *atomic propositions*. A *Kripke structure* K, or simply *model*, wrt. A is defined as $K = (S, I, T, L)$ with S a finite set of states, $I \subseteq S$ a set of initial states, $T \subseteq S \times S$ its transition relation, and $L : S \to 2^A$ a labelling function. We assume that the set of initial states is non-empty and the transition relation is total, i.e. for every state $s \in S$ there exists a state $s' \in S$ with $(s, s') \in T$. We write $T(s, s')$ whenever $(s, s') \in T$ and similarly $I(s)$.

As temporal logic we use a subset of CTL* with the next time operator **X**, and the eventuality operator **F**. We do not treat the until operator **U** or further operators in detail, since our translation works for full LTL which includes these operators. We will only consider the universal path quantifier **A**. The propositional operators are conjunction ($\wedge$) and negation ($\neg$). We also add the propositional constants $\{0, 1\}$.

The set of CTL* formulae is made of two types of formulae, *path formulae* Φ and *state formulae* Ψ. All atomic propositions $p \in A$ are state formulae, which can always be coerced to path formulae. Negation maintains the type of the argument. The same applies to conjunction if the types of the arguments match. Otherwise their conjunction is a path formula. Temporal operators are applied to state formulae. A path formula may be prefixed by a path quantifier to obtain a state formula.

Semantics for path formulae are defined wrt. *paths*, where the set Π of paths of K is the union of all finite and infinite sequences $\pi = (s_i)$ with $s_i \in S$ and $T(s_i, s_{i+1})$ for $0 \leq i < |\pi|$. The length $|\pi|$ of π is defined as the number of transitions. We only consider non-empty paths. We write $\pi(i)$ for s_i and π^n for the sequence (s_{i+n}), which is the same as the original path with its first n states removed. Let $p \in A$, $\phi, \phi_1, \phi_2 \in \Phi$ and $\psi, \psi_1, \psi_2 \in \Psi$. The validity of

```
MODULE task(id,turn)
VAR
  state : 0..7;


ASSIGN
  init(state) := 0;
  next(state) :=
    case
      turn = id &
      state < 7
        : state + 1;
      1 : state;
    esac;
DEFINE
  found := state = 7;

FAIRNESS
  turn = id




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



DEFINE
  found :=
    t0.found &
    t1.found;




SPEC
  AF found
```

(a) live

```
MODULE task(id,turn,save,saved)
VAR
  state : 0..7;
  loop  : 0..7;
  fair  : boolean;
ASSIGN
  init(state) := 0;
  next(state) :=
    case
      turn = id &
      state < 7
        : state + 1;
      1 : state;
    esac;
DEFINE
  found := state = 7;
  looped := saved & state = loop;
ASSIGN
  init(fair) := 0;
  next(fair) := fair |
    id = turn & (save | saved);
  next(loop) :=
    case
      save & !saved : state;
      1 : loop;
    esac;

MODULE main
VAR
  turn  : 0..1;
  t0    : task(0,turn,save,saved);
  t1    : task(1,turn,save,saved);
  save  : boolean;
  saved : boolean;
  live  : boolean;
DEFINE
  found  := t0.found  & t1.found;
  looped := t0.looped & t1.looped;
  fair   := t0.fair   & t1.fair;
ASSIGN
  init(saved) := 0;
  next(saved) := saved | save;
  init(live)  := 0;
  next(live)  := live | found;
SPEC
  AG (looped & fair -> live)
```

(b) safe

Fig. 5. Hierarchical translation of liveness with fairness into pure safety.

state and path formulae for $s \in S$ and infinite $\pi \in \Pi$ are defined as follows:

$s \models \mathbf{A}\ \phi$	iff $\pi \models \phi$ for all $\pi \in \Pi$ with $\pi(0) = s$	$s \models p$	iff $p \in L(s)$
$s \models \psi_1 \wedge \psi_2$	iff $s \models \psi_1$ and $s \models \psi_2$	$s \models \neg\psi$	iff $s \not\models \psi$
$\pi \models \phi_1 \wedge \phi_2$	iff $\pi \models \phi_1$ and $\pi \models \phi_2$	$\pi \models \neg\phi$	iff $\pi \not\models \phi$
$\pi \models \mathbf{F}\ \phi$	iff there exists $i \geq 0$ with $\pi^i \models \phi$	$\pi \models \mathbf{X}\ \phi$	iff $\pi^1 \models \phi$
$\pi \models \psi$	iff $\pi(0) \models \psi$		

We will also use other boolean operators, such as disjunction ($\vee$) and implication ($\rightarrow$). The temporal operator globally $\mathbf{G}$ is defined as $\mathbf{G}\ \phi \equiv \neg\mathbf{F}\neg\phi$ and the existential path quantifier as $\mathbf{E}\ \phi \equiv \neg\mathbf{A}\neg\phi$.

A path π is *initialized* wrt. a given model $K = (S, I, T, L)$ iff $\pi(0) \in I$. Then a CTL* formula f is valid for K iff $\pi \models f$ for all initialized infinite paths π. Model checking determines the validity of f for K. Two model checking problems $P = (K, f)$ and $P' = (K', f')$ are equivalent iff $K \models f \ \Leftrightarrow K' \models f'$.

The first two steps of our translation in Sect. 4 produce equivalent model checking problems, proved by bisimulation equivalence. Two models $K = (S, I, T, L)$ and $K' = (S', I', T', L')$ over the same set of atomic propositions are bisimulation equivalent iff there exists a relation $\sim\ \subseteq S \times S'$ with the following properties: Let $s \in S$ and $s' \in S'$ with $s \sim s'$. First the labelling has to match, that is $L(s) = L'(s')$. Second for all $t \in S$ with $T(s, t)$ there has to exist $t' \in S'$ with $T'(s', t')$ and $t \sim t'$. Finally, for all initial states $s \in I$ there has to be an initial state $s' \in I'$ with $s \sim s'$. The dual properties have to hold as well.

The complexity of the original model checking algorithm [11] for simple properties, such as $\mathbf{AF}p$ and $\mathbf{AG}p$, is linear in the size of the model K. Particularly it is linear in the number of states $|S|$ and the number of transitions $|T|$. In the case of on-the-fly model checking [14] the complexity can further be restricted to be linear in the number of reachable states $|R|$ with $R = \{\pi(i) \mid i \geq 0, \pi \in \Pi, \pi \text{ initialized}\}$. For symbolic model checking with BDDs [21] the number of (reachable) states is less important than the number of fixpoint iterations. This number is bounded by the *diameter* d which is defined as the maximal *distance* $\delta(s, t)$ between two states $s, t \in S$ with

$$\delta(s, t) = \min \{\ k\ \mid\ \pi \in \Pi,\ \ |\pi| = k,\ \ \pi(0) = s,\ \ \pi(k) = t\}$$

In BFS reachability analysis the number of iterations can further be restricted to the maximal distance r, called *radius*, of all reachable states to some possibly varying initial state. In backward fixpoint computations, which are the traditional way of checking liveness properties, we can introduce a similar notion of a *backward radius* which is the number of backward iterations after which the fixpoint is reached. The backward radius depends not only on the model but also on the property. Note that backward and forward radius are

not related. For instance, an inductive invariant p has a backward radius of one when checking $\mathbf{AG}p$, independent of the size of the model. In practice pure backward model checking is usually outperformed by forward model checking [17] or a restricted version of backward model checking in which the approximations in the fixpoint computation are restricted to the pre-computed set of reachable states.

4 Translation

In this section we precisely describe our state recording translation on an abstract level and prove its correctness. The application to a concrete model description language such as the SMV input language used for the experiments is left to the reader. We also do not treat the counter based translation formally. In the second part of the section we discuss the efficiency of our translation by comparing size and diameter of the original and the translated model. In the last part we describe the extention to fairness and LTL.

4.1 Correctness

Let $K = (S, I, T, L)$ be a Kripke structure and $\mathbf{AF}p$ be the liveness property we want to check. As a first step we construct $K_\perp = (S_\perp, I_\perp, T_\perp, L_\perp)$, with $S_\perp = S \times (S \cup \{\perp\})$, and $I_\perp = I \times \{\perp\}$. The new transition relation is defined as

$$T_\perp((s,t),(s',t')) \quad \Leftrightarrow \quad T(s,s') \ \wedge \ (t' = t \ \vee \ (t = \perp \ \wedge \ t' = s)) \tag{2}$$

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 (see also Fig. 4). Therefore $T_\perp$ is monotonic in the second state component for the order $\leq_\perp \subseteq (S \cup \{\perp\})^2$ with $s \leq_\perp t$ iff $s = t$ or $s = \perp$. The new labelling is obtained as $L_\perp = L \circ \rho$ using the projection function ρ operating on pairs with $\rho((s,t)) = s$.

We further assume that $\perp$ is a new state that does not already occur in S. In essence our translation simulates the original behavior of K without introducing dead ends, maintaining the labelling of the states. Therefore we can prove that K and $K_\perp$ are bisimulation equivalent under the bisimulation $\sim \subseteq S \times S_\perp$, with $s \sim s_\perp \Leftrightarrow \rho(s_\perp) = s$. To prove that $\sim$ is a bisimulation we use $\lambda_\perp \colon S \to S_\perp$ defined as $\lambda_\perp(s) = (s, \perp)$ and extend both $\lambda_\perp$ and the projection function ρ to paths in the natural way. Then we can easily check that $\pi \sim \lambda_\perp(\pi)$ and $\rho(\pi_\perp) \sim \pi_\perp$ for all paths. These functions provide the necessary witnesses for the existential quantifiers in the requirements for $\sim$ being a bisimulation.

Lemma 4.1 *K and $K_\perp$ are bisimulation equivalent.*

The next step adds a flag that remembers whether p has ever been valid on the path so far. The result is $K_p = (S_p, I_p, T_p, L_p)$ with $S_p = S_\perp \times \{0, 1\}$,

$I_p = I_\perp \times \{0\}$, and $T_p((s,x),(s',x'))$ iff

$$T_\perp(s,s') \ \wedge \ (p \in L_\perp(s) \ \rightarrow \ x' = 1) \ \wedge \ (p \notin L_\perp(s) \ \rightarrow \ x' = x)$$

The rest is defined as in the first step. Again T_p is monotonic in the second state component, in this case for the order of natural numbers restricted to $\{0,1\}$. Note, that K_p depends on the property being checked. Similar reasoning as before with a slightly more complex $\lambda_p : S_\perp \rightarrow S_p$ and a transitivity argument gives the following Lemma.

Lemma 4.2 *K and K_p are bisimulation equivalent.*

Since validity of CTL* formulae is preserved under bisimulation equivalence [4,12], we obtain the equivalence of $(K, \mathbf{AF}p)$ and $(K_p, \mathbf{AF}p)$. The final step in our translation consists of adding a new atomic proposition q with

$$q \in L_p(\ ((s,t),x)\ ) \quad \Leftrightarrow \quad s = t \ \rightarrow \ x = 1 \tag{3}$$

This definition shows the correctness of our translation.

Theorem 4.3 *$(K, \mathbf{AF}p)$ and $(K_p, \mathbf{AG}\ q)$ are equivalent.*

Proof. What remains to be shown is the equivalence of $\mathbf{EG}\neg p$ and $\mathbf{EF}\ \neg q$ in K_p. First assume $K_p \models \mathbf{EG}\neg p$. Then there exists an infinite initialized path $\pi \in \Pi_p$ with $p \notin L_p(\pi(i))$ for all $i \geq 0$. Since the number of states of S_p is finite, there have to exist indices $k \geq l \geq 0$ with $\pi(k+1) = \pi(l)$. Let $\pi(i) = (\ (s_i,t_i),x_i\ )$ for $i \geq 0$ and define $\pi'(i) = (\ (s_i,t'_i),x_i\ )$ with $t'_i = \perp$ for $0 \leq i \leq l$ and $t'_i = s_l$ for $l < i \leq k+1$.

Clearly π' is an initialized legal path of K_p. By definition we have $s_{k+1} = t'_{k+1} = s_l$ and $x_i = 0$ for $0 \leq i \leq k+1$, since $p \notin L_p(\pi'(j)) = L(s_j) = L_p(\pi(j))$ for $0 \leq j \leq k$. From (3) we get $q \notin L_p(\pi'(k+1))$ and π' proves to be a witness for $\mathbf{EF}\neg q$, assuming π' is extended to an infinite path in the obvious way. Note that T_p is total since our translation does not introduce dead ends.

For the reverse direction assume $\mathbf{EF}\neg q$ holds. Without loss of generality we find an initialized path $\pi \in \Pi_p$ with $|\pi| = k+1$ and $\pi(k+1) \models \neg q$. With $\pi(i) = (\ (s_i,t_i),x_i\ )$ we deduce from (3) that $s_{k+1} = t_{k+1}$ and $x_{k+1} = 0$. From the monotonicity of $T_\perp$ in its second state component, we obtain an l with $0 < l \leq k$, such that $\perp = t_0 = \ldots = t_l$ and $s_l = t_{l+1} = \ldots = t_{k+1}$. Now we construct an infinite path π' with $\pi'(i) = (\ (s'_i,t'_i),x_i\ )$ as follows: for $0 \leq i \leq k$ we simply set $\pi'(i) = \pi(i)$. If $i > k$ we define $t'_i = t_{k+1}$, $x'_i = x_{k+1}$ and $s'_i = s_{l+c}$ with $c = (i-l) \bmod (k+1-l)$. From the monotonicity of T_p in its second state component, we have $x_{k+1} = \ldots = x_0 = 0$, which implies $s_i \models \neg p$ for $0 \leq i \leq k$. Since these original states determine the non-validity of p for every $\pi'(i)$, and π' is a legal initialized infinite path, it serves as witness for $\mathbf{EG}\neg p$.□

4.2 Complexity

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. 5, the size of a non-canonical symbolic description in program code, increases only by a small constant factor. The counter based translation will produce very large counter examples. Therefore we restrict the discussion to the state recording translation.

In global (explicit) model checking [11] the complexity is governed by the number of states, which increases quadratically:

$$|S_p| = 2 \cdot |S_\perp| = 2 \cdot |S| \cdot (|S| + 1) = O(|S|^2)$$

In the case of on-the-fly (explicit) model checking [14] only the size of the reachable state space R_p is of interest. A reachable state $(s,t) \in R_\perp$ of $K_\perp$ either contains $\perp$ as second component t, or t is reachable in K since only reachable states are recorded. Therefore $R_\perp$ is bounded by $|R| \cdot (|R| + 1)$. This bound is tight: a modulo n counter, like the model in Fig. 2 for $n = 4$, has $|R_\perp| = n \cdot (n + 1)$ reachable states. If $n = 4$ then every combination of $\{0, \ldots, 3\} \times \{\perp, 0, \ldots 3\}$ can be reached. Further introducing the p-recording flag at most doubles the number:

$$|R_p| \leq 2 \cdot |R_\perp| \leq 2 \cdot |R| \cdot (|R| + 1) = O(|R|^2)$$

Regarding symbolic model checking with BDDs [21] we have two results. First we relate the size of reduced ordered BDDs for the transition relation of K, $K_\perp$ and K_p. Assuming S is encoded with $n = \lceil \log_2 |S| \rceil$ state bits, we can encode $S_\perp$ with $2n + 1$ boolean variables. It is important to interleave the boolean variables for the first and second component. Otherwise the size of the BDD for the term $(t' = t \vee (t = \perp \wedge t' = s))$ in (2) may explode. With an interleaved order it is linear in n with a factor of approx. 11. The factor has been determined empirically for large state spaces. Thus the size of the BDD for $T_\perp$ can be bounded by $11 \cdot n$ the size of the BDD for T by using the fact from [5] that computing any boolean binary operation on BDDs will produce a BDD of size that is linear with factor 1 in the size of the argument BDDs. Finally, the size of the BDD for T_p compared to the size of the BDD for $T_\perp$ 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_p 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. To measure the *dynamic* complexity we determine bounds on the diameter and radius, which also serve as bounds on the maximal number of fixpoint iterations. Note that the counter based translation has a radius at least as large as the number of states in the original system, which makes traditional symbolic reachability

analysis impractical even for medium sized problems. One important observation is that the state recording translation produces a much smaller diameter d_p and radius r_p:

Theorem 4.4 $d_p \leq 4 \cdot d + 3$ *and* $r_p \leq r + 3 \cdot d + 3$

Proof. Let $\pi \in \Pi_\perp$ be a finite path with $\pi(i) = (s_i, t_i)$ and $|\pi| = k$. Since $T_\perp$ is monotonic in the second component we have to distinguish two cases. If first $t_0 = \ldots = t_k$, then $\delta_\perp(\pi(0), \pi(k)) = \delta(s_0, s_k) \leq d$, since all paths in K can be extended to legal paths in $K_\perp$ by adding a fixed non changing second state component. In the second case there exists an l with $0 \leq l < k$ with $t_0 = \ldots = t_l = \perp$ and $t_{l+1} = \ldots = t_k = s_l$ (cf Fig. 4). Now we have two sub-paths with constant second state component as in the first case and obtain

$$\delta_\perp(\pi(0), \pi(k)) \;\leq\; \delta(s_0, s_l) + 1 + \delta(s_{l+1}, s_k) \;\leq\; 2 \cdot d + 1$$

which also subsumes the bound of the first case and thus $d_\perp \leq 2 \cdot d + 1$. To determine the bound for the radius we additionally assume that π is initialized. Then $\delta(s_0, s_l) \leq r$ and we obtain $r_\perp \leq r+d+1$. With the same reasoning, since T_p is monotonic in the second state component as well, we derive $d_p \leq 2 \cdot d_\perp + 1$ and $r_p \leq r_\perp + d_\perp + 1$. By substitution we derive the desired inequalities. □

Unfortunately, there are examples where r is much smaller than d and for reachability analysis in K_p we still have to perform more than d fix point iterations. A modulo n counter as in Fig. 2 without the `-1` state becomes such an example if we allow all states to be initial states. Then we have $d = n - 1$, $r = 0$, but $d_\perp = 2 \cdot n - 1$ and $r_\perp = n$, which is already larger than d. The number of backward iterations necessary to check a liveness property in the original model could also be very large.

4.3 Fairness and LTL

Our translation is able to incorporate fairness. A fairness constraint is simply 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 model, 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 models with fairness constraints is defined as in the unfair case, except that all paths are required to be fair. Bisimulation with fairness is defined by expanding the transition based definition stated above to whole fair paths as in [12]: the additional requirement is that for all fair paths $\pi \in \Pi$ there exists a fair path $\pi' \in \Pi'$ with $\pi \sim \pi'$, where $\pi \sim \pi'$ iff $\pi(i) \sim \pi'(i)$ for all $i \geq 0$. To handle a fair Kripke structure $K(S, I, T, L, F)$ we construct $K_p(S_p, I_p, T_p, L_p, F_p)$ where S_p, I_p, T_p, and L_p are defined as above and F is extended to

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

We define $K_p^F = (S_p^F, I_p^F, T_p^F, L_p^F)$ with $S_p^F = S_p \times \{0,1\}^m$ and $I_p^F = I_p \times \{(0,\ldots,0)\}$ 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_p^F be the natural extension of L_p as before. Let (s,t,x,v), $(s',t',x',v') \in S_p^F$ with $s,s' \in S$, $t,t' \in S \cup \{\bot\}$, $x,x' \in \{0,1\}$ and $v,v' \in \{0,1\}^m$. The transition relation T_p^F is satisfied for (s,t,x,v) and (s',t',x',v') as current and next state iff

$$T_p(\ ((s,t),x)\ ,\ ((s',t'),x')\ ) \wedge$$

$$\textstyle\bigwedge_{i=1}^{m}\ (v'(i) = v(i)\ \vee\ (t' \neq \bot\ \wedge\ s \in F^i\ \wedge\ v'(i) = 1))$$

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

$$q_F \in L_p^F(\ ((s,t,x,v)\ ) \Leftrightarrow\ (v(1) = \ldots = v(m) = 1)\ \rightarrow\ q \in L_p((s,t),x)$$

where q is defined as for K_p. We can prove a correctness result like before, now including fairness.

Theorem 4.5 *$(K, \mathbf{AF}p)$ and $(K_p^F, \mathbf{AG}\ q_F)$ are equivalent.*

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_p^F 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 $\mathbf{AG}\ q_F$ will always find shortest counter examples.

An alternative approach counts the number of fairness constraints satisfied sofar, 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 logarithmic additional state bits. However it fails to generate counter example 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.

Since generalized Büchi automata and thus LTL [14] can be translated into fair Kripke structures, our translation also applies to LTL model checking in general. Additionally it is possible to derive special translation rules for other standard LTL operators. For example to handle $p_1 \mathbf{U}\ p_2$ we use

$$p_1 \mathbf{U}\ p_2\ \equiv\ (p_1\ \mathbf{U}_{\text{weak}}\ p_2)\ \wedge\ \mathbf{F}p_2$$

where the *weak until* operator $p_1\ \mathbf{U}_{\text{weak}}\ p_2$ is defined to be valid for a path iff p_1 does not stop to hold before p_2 holds or p_1 holds along the whole path. By adding a state bit that remembers whether p_2 was fulfilled already, the weak until can easily be transformed into a simple safety property. Then the

	check true			check false			counterexample false		
n	live	count	safe	live	count	safe	live	count	safe
4	8 (4+ 4)	9 (9+0)	8 (8+0)	5 (4+1)	5 (5+0)	4 (4+0)	7 (3+ 4)	5 (0+5)	4 (0+4)
8	16 (8+ 8)	17(17+0)	16(16+0)	9 (8+1)	9 (9+0)	8 (8+0)	15 (7+ 8)	9 (0+9)	8 (0+8)
12	24(12+12)	25(25+0)	24(24+0)	13(12+1)	13(13+0)	12(12+0)	23(11+12)	13(0+13)	12(0+12)
16	32(16+16)	33(33+0)	32(32+0)	17(16+1)	17(17+0)	16(16+0)	31(15+16)	17(0+17)	16(0+16)

Table 1
Counters

eventuality $\mathbf{F}p_2$ is translated into a safety property as well, with our original translation. Finally, we check both safety properties simultaneously.

5 Experiments

In this section, we show the results of our translation applied to various examples, both theoretical and "real world" ones. Each table is divided into three main parts: the left part, with the iterations needed for the correct model, the middle part, where the model is incorrect, and the right part, which shows the iterations needed to compute a counter example for the incorrect model. The three main parts are further split up into one column for each different approach: *live* for the conventional liveness approach, *count* for the counter based approach (not used in the FireWire example), and *safe* for our state recording translation. For each version, the number of overall, forward, and reverse iterations is shown.

5.1 Simple Counters

In the case of a simple counter in Table 1 all approaches perform linearly in the number of iterations wrt. the model size. Computing the counter example, however, requires nearly twice as many iterations with the live version as opposed to our method.

For the counters used in Table 2 the desired state `n` can be reached from any state in one step. There are only two iterations needed to complete a loop, and n backward iterations to reach all possible predecessors. With the counter based approach, $n + 1$ iterations are required to enumerate enough states, and another iteration to reach state `n`. Our approach requires a constant number of five iterations for a correct model: One iteration to reach all possible successor states; from those states, a second iteration to reach state `n`. The third iteration reaches the initial state `0` again, from which two more iterations are required to prove the liveness within the loop.

The false example requires two iterations for the loop, and with the live version, another backward iteration for the initial state as a predecessor. The counter based approach is very inefficient. The counter example analysis shows a similar behavior.

	check true			check false			counterexample false		
n	live	count	safe	live	count	safe	live	count	safe
4	6 (2+ 4)	6 (6+0)	5 (5+0)	3 (2+1)	5 (5+0)	2 (2+0)	3 (1+2)	5 (0+5)	2 (0+2)
8	10 (2+ 8)	10 (10+0)	5 (5+0)	3 (2+1)	9 (9+0)	2 (2+0)	3 (1+2)	9 (0+9)	2 (0+2)
12	14 (2+12)	14 (14+0)	5 (5+0)	3 (2+1)	13 (13+0)	2 (2+0)	3 (1+2)	13 (0+13)	2 (0+2)
16	18 (2+16)	18 (18+0)	5 (5+0)	3 (2+1)	17 (17+0)	2 (2+0)	3 (1+2)	17 (0+17)	2 (0+2)

Table 2
Skipping Counters

5.2 IEEE 1394 FireWire – Tree Identify Protocol

IEEE 1394 (FireWire) [16] is a protocol for a serial high-speed bus widely used to interconnect multimedia devices and PCs. To ensure correct functioning of the protocol the nodes connected to an IEEE 1394 bus are required to form a tree. The Tree Identify Protocol is executed each time the bus configuration changes to verify this condition and to elect a unique leader who has extended responsibilities in later phases of the protocol. In previous work [25,24] we have verified several properties of the Tree Identify Protocol with SMV.

The single most important property to be verified in the tree identify phase is the designation of a leader before the next phase of the protocol is reached. This property was checked in our experiments for both the original (correct) version of the model from [24] and a version with a bug preventing the successful completion of the protocol. In the SMV input language it is formulated for 2 nodes as follows

```
AF (node[0].root | node[1].root | timeout | known_problems)
```

where `root`, `timeout` and `known_problems` are state properties. Separate safety properties are used to ensure that neither `timeout` nor `known_problems` have occurred. Once verified these conditions could be removed from the model and are not included in the performance figures given here.

During the run of the protocol two nodes might be left competing to become root. In this case a sub-protocol is invoked to resolve this situation, called root contention. Both contending nodes non-deterministically choose to wait for either a short or a long time before continuing. If the nodes chose differently one of them will become root. Otherwise the sub-protocol is repeated. A fairness condition ensures that the two nodes will make a different choice at some point.

Most of the steps in the translation process described in Sect. 4 have been automated. For the translation a flat model is generated with NuSMV [10]. Additional variables are introduced to record the saved state, to represent the oracle, and to keep track whether each fairness condition has been true on the loop. Simple liveness properties of the form `AF p` are also translated automatically. More complicated properties need to be reformulated by the user either by using the automata based approach or by simple transformations

		check true		check false		cex false	
n	p	live	safe	live	safe	live	safe
2	2	74 (19+55)	24 (24+0)	34 (19+15)	13 (13+0)	132 (13+119)	13 (0+13)
2	3	74 (19+55)	24 (24+0)	35 (19+16)	13 (13+0)	132 (13+119)	13 (0+13)
2	4	78 (19+59)	24 (24+0)	36 (19+17)	13 (13+0)	132 (13+119)	13 (0+13)
3	2	76 (21+55)	23 (23+0)	36 (21+15)	11 (11+0)	67 (10+ 57)	11 (0+11)
3	3	77 (21+56)	23 (23+0)	37 (21+16)	11 (11+0)	67 (10+ 57)	11 (0+11)
3	4	77 (21+56)	23 (23+0)	37 (21+16)	11 (11+0)	67 (10+ 57)	11 (0+11)
4	2	129 (31+98)	36 (36+0)	52 (31+21)	19 (19+0)	215 (19+196)	19 (0+19)

Table 3
Leader election in the Tree Identify Protocol - iterations

		check true				check + cex false			
		live		safe		live		safe	
n	p	time	memory	time	memory	time	memory	time	memory
2	2	0.85	66941	4.19	397030	1.12	103299	2.64	282859
2	3	1.93	201680	11.07	782574	2.65	215169	6.82	595756
2	4	4.71	443947	28.22	1296088	5.45	402535	16.00	944482
3	2	11.33	699222	39.45	1946866	7.59	718910	12.09	772508
3	3	76.05	3777278	283.07	9578242	53.60	3678676	86.82	4217925
3	4	450.72	29220542	1567.67	31759998	259.51	19588279	554.39	14364650
4	2	357.30	14001693	1376.18	35547502	204.82	12500473	644.18	24864717

Table 4
Leader election in the Tree Identify Protocol

similar to the one we presented for the until operator in Sect. 4.3. Finally, an improved variable order is generated. To allow for a fair comparison the live model was also flattened before checking.

We used Cadence SMV [20] on a Pentium III-800 running Linux 2.2.19. Execution time and memory usage were limited to 1 hour and 1 GB respectively. Since an optimized variable order was provided explicitly, dynamic reordering had been disabled. In separate runs we checked that dynamic reordering produces comparable orders. Note that, enabling dynamic reordering would have increased runtimes dramatically.

Configurations with 2 – 4 nodes and 2 – 4 ports were checked. Table 3 shows the number of iterations. Table 4 lists execution time in seconds and memory usage in peak number of BDD nodes. Combinations of nodes and ports not shown could not be handled within the given time and memory bounds.

In each case, the safe version requires much fewer overall iterations than the live version. Only for the correct model the safe version needs more forward iterations than the live version. While run time and memory usage for the safe version of the correct model is up to 6 times higher than for the live version, the relation improves in the buggy case.

6 Conclusion

In this paper we presented a translation that allows to check liveness properties by checking safety properties. Our main contributions can be summarized as follows:

(i) For commercial or proprietary safety checking tools it may not be feasible for the user to change the algorithms. Our technique allows to apply such tools to liveness, which were supposed to check safety properties only.

(ii) The experiments indicate that our technique is comparable with specialized algorithms. Additionally we are able to find counter example traces of minimal length.

(iii) With our translation theoretical results on safety checking can be lifted to liveness checking. Therefore special treatment of liveness properties can only be justified by experiments or additional complexity results.

The main open question is how the number of state bits introduced by our translation can further be reduced. We also want to apply the method to liveness checking with sequential ATPG and STE.

References

[1] Biere, A., A. Cimatti, E. Clarke and Y. Zhu, *Symbolic model checking without BDDs*, in: *TACAS*, 99.

[2] Biere, A., E. Clarke and Y. Zhu, *Multiple state and single state tableaux for combining local and global model checking*, in: *Correct System Design (Recent Insights and Advances)*, number 1710 in LNCS, 2000.

[3] Boppana, V., S. Rajan, K. Takayama and M. Fujita, *Model checking based on sequential ATPG*, in: *CAV*, 99.

[4] Browne, M., E. Clarke and O. Grumberg, *Characterizing finite Kripke structures in propositional logic*, Theoretical Computer Science **59** (1988).

[5] Bryant, R., *Graph-based algorithms for boolean function manipulation*, IEEE Transactions on Computers **35** (1986).

[6] Bryant, R. and C.-J. Seger, *Formal verification of digital circuits using symbolic ternary system models*, in: *CAV*, 1990.

[7] Burch, J., E. Clarke, D. Long, K. McMillan and D. Dill, *Symbolic model checking for sequential circuit verification*, IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems **13** (1994).

[8] Cabodi, G., P. Camurati and S. Quer, *Improved reachability analysis of large finite state machines*, in: *ICCAD*, 1996.

[9] Chou, C.-T., *The mathematical foundation of symbolic trajectory evaluation*, in: *CAV*, 1999.

[10] Cimatti, A., E. Clarke, F. Giunchiglia and M. Roveri, *NuSMV: a new symbolic model verifier*, in: *CAV*, 99.

[11] Clarke, E. and A. Emerson, *Design and synthesis of synchronization skeletons using branching time temporal logic*, in: *IBM Workshop on Logics of Programs*, 1981.

[12] Clarke, E., O. Grumberg and D. Peled, "Model Checking," MIT Press, 1999.

[13] Emerson, A., *Temporal and modal logic*, in: *Handbook Theoretical Computer Science: Volume B, Formal Methods and Semantics* (1995).

[14] Gerth, R., D. Peled, M. Vardi and P. Wolper, *Simple on-the-fly automatic verification of linear temporal logic*, in: *15th Workshop on Protocol Specification, Testing, and Verification* (1995).

[15] Henzinger, T., O. Kupferman and S. Qadeer, *From pre-historic to post-modern symbolic model checking*, in: *CAV*, 1998.

[16] IEEE, "IEEE Standard for a High Performance Serial Bus. Std 1394-1995, and Supplement 1394a-2000," (1995, 2000).

[17] Iwashita, H. and T. Nakata, *CTL model checking based on forward state traversal*, in: *ICCAD*, 1996.

[18] Kupferman, O. and M. Vardi, *Model checking of safety properties*, in: *CAV*, 1999.

[19] Lamport, L., *Proving the correctness of multiprocess programs*, IEEE Transactions on Software Engineering **3** (1977).

[20] McMillan, K., *Cadence SMV*, `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv`.

[21] McMillan, K., "Symbolic Model Checking: An Approach to the State Explosion Problem," Kluwer Academic Publishers, 1993.

[22] Niermann, T. and J. Patel, *Hitec: A test generation package for sequential circuits*, in: *EURODAC*, 1991.

[23] Ravi, K. and F. Somenzi, *High density reachability analysis*, in: *ICCAD*, 1995.

[24] Schuppan, V. and A. Biere, *Verifying the IEEE 1394 FireWire Tree Identify Protocol with SMV* Submitted.

[25] Schuppan, V. and A. Biere, *A simple verification of the Tree Identify Protocol with SMV*, in: *IEEE 1394 (FireWire) Workshop*, 2001.

[26] Seger, C.-J. and R. Bryant, *Formal verification by symbolic evaluation of partially-ordered trajectories*, Formal Methods in System Design **6** (1995).

[27] Yang, J. and C.-J. Seger, *Introduction to generalized symbolic trajectory evaluation*, in: *ICCD*, 2001.