Form Method Syst Des
DOI 10.1007/s10703-006-0019-9

Bounded model checking of infinite state systems

Tobias Schuele · Klaus Schneider

© Springer Science + Business Media, LLC 2006

Abstract Bounded model checking (BMC) is an attractive alternative to symbolic model checking, since it often allows a more efficient verification. The idea of BMC is to reduce the model checking problem to a satisfiability problem of the underlying base logic, so that sophisticated decision procedures can be utilized to check the resulting formula. We present a new approach to BMC that extends current methods in three ways: First, instead of a reduction to propositional logic which restricts BMC to finite state systems, we focus on infinite state systems and therefore consider more powerful, yet decidable base logics. Second, instead of directly unwinding temporal logic formulas, we use special translations to ω-automata that take into account the temporal logic hierarchy and maintain safety and liveness properties. Third, we employ both global and local model checking procedures to take advantage of the different types of specifications that can be handled by these techniques. Based on three-valued logic, our bounded model checking procedures may either prove or disprove a specification, or they may explicitly state that no information has been obtained due to insufficient bounds.

Keywords Bounded model checking · Infinite state systems · Temporal logic hierarchy · Global model checking · Local model checking

1. Introduction

1.1. Symbolic model checking

Model checking of *finite state systems* has evolved as a major technology for the verification of reactive systems [26, 65] and has become state-of-the-art in many hardware design flows.

T. Schuele (✉) · K. Schneider
Reactive Systems Group, Department of Computer Science, University of Kaiserslautern,
P.O. Box 3049, 67653 Kaiserslautern, Germany
e-mail: Tobias.Schuele@informatik.uni-kl.de

K. Schneider
e-mail: Klaus.Schneider@informatik.uni-kl.de

The breakthrough was achieved in the early nineties when it was observed that large but finite sets can be efficiently represented by means of binary decision diagrams (BDDs), a canonical normal form for propositional logic [13]. To this end, sets of states and the transition relation of the system to be verified are encoded by their characteristic functions in propositional logic. The development of BDDs was a cornerstone for *symbolic model checking* procedures that perform an implicit state space exploration [5, 18, 26, 65]. With sophisticated implementations and refinements of symbolic model checking, it became possible to verify very complex systems, and to detect errors that are unlikely to be found using simulation [26, 28]. On the other hand, it is well-known that for most propositional formulas, all canonical normal forms (including BDDs) suffer from an exponential blow-up [51]. Unfortunately, this blow-up does not only occur in theory, but also for many functions of practical interest such as multiplication of binary numbers [14]. As a result, BDD-based symbolic model checkers may be able to handle very large systems, but they may also fail for relatively small systems.

1.2. Infinite state systems

In recent years, the verification of *infinite state systems* has attained increasing interest. According to [34], there are at least the following five ‘sources of infinity’:

- data structures over infinite domains like integers
- control structures like unbounded call stacks or dynamic creation of processes
- unbounded queues for asynchronous process communication
- parameterization to describe generic systems
- timing constraints in real-time systems

In the sequel, we concentrate on systems with a finite state control flow, but with data structures over infinite domains. More precisely, the systems we consider contain integer variables that may have arbitrarily large values. Systems of this class appear in different forms, e.g. as extended finite state machines [43], and belong to the most powerful machine models, since they are equivalent to Turing machines. As a consequence, most verification problems are undecidable for such systems.[1]

As mentioned previously, symbolic model checkers utilize efficient data structures like BDDs to encode large sets. However, as propositional logic formulas have only finitely many models, they are naturally limited to the representation of finite sets. For the representation of infinite sets, more powerful logics are required. In principle, any decidable predicate logic can serve as the base logic, provided that there are efficient decision procedures or canonical normal forms for that logic.

Presburger arithmetic [33, 58], a decidable first-order logic that subsumes propositional logic, is a promising candidate for a base logic. As a major advantage, Presburger arithmetic can be translated to finite state automata, thus obtaining a canonical normal form for arbitrary formulas. Interestingly, BDDs can be viewed as a special kind of finite state automata, where the root is the initial state and the leaves the accepting states. Hence, Presburger arithmetic and finite state automata correspond to (quantified) propositional logic and BDDs, respectively.

[1] For restricted classes of systems and properties, e.g. pushdown automata and existential reachability, some problems are decidable [20, 78].

1.3. Global vs. local model checking

In general, there are two approaches to model checking, namely *global* and *local* model checking [65]. Global model checking procedures first compute those states of a transition system that satisfy a formula using fixpoint iteration [24, 59]. In a second step, it is then checked whether the initial states are included in this set. In contrast, local model checking procedures directly answer the question whether the initial states satisfy the formula [11, 12, 29, 74, 75]. This is accomplished by constructing a proof tree using syntax directed decomposition rules.

There are many differences between global and local model checking, in particular, the way a formula is evaluated: In global model checking, the syntax tree of a specification is traversed in a bottom-up manner, whereas in local model checking, a specification is evaluated by a top-down traversal of its syntax tree. However, the most important difference is that global model checking is based on fixpoint iteration, while local model checking follows an inductive style of reasoning.

For *finite state systems*, all verification problems are decidable, and for both approaches there are algorithms with essentially the same worst case asymptotic complexity. In practice, however, the runtimes of global and local model checking procedures may differ significantly due to the different nature of these approaches. For example, local model checking has the advantage that subformulas can be checked by need, i.e., in the spirit of lazy evaluation. Moreover, only parts of the transition relation are required during the construction of a proof tree, so that the transition relation can be computed on-the-fly. In contrast, global model checking procedures usually require the construction of the complete transition relation to perform a breadth-first traversal of the state space.

On the other hand, if the complete transition relation can be computed efficiently, global model checking may be significantly faster than local model checking. In particular, classical approaches to local model checking [12, 29, 74, 75] only consider single states and perform a depth-first traversal of the state space. However, this does not mean that local approaches cannot benefit from symbolic set representations. In fact, a local model checking procedure for the propositional μ-calculus that considers sets of states instead of single states, was already described in [11, 12]. Hence, local model checking can also benefit from symbolic set representations.

As mentioned previously, most verification problems are undecidable for *infinite state systems*, and hence, it may happen that the chosen verification procedure will not terminate. Clearly, termination of the verification procedure does not only depend on the formula and the transition system, but also on the verification procedure itself. Thus, for infinite state systems, the choice of the best verification procedure is not primarily a matter of efficiency as in the case of finite state systems, but rather a matter of termination. For instance, due to the different nature of global and local model checking algorithms, it may happen that one algorithm terminates for a given specification while the other does not, and vice versa. The differences between global and local model checking of infinite state systems were investigated in [69].

1.4. Bounded model checking

In recent years, *bounded model checking* (BMC) [6, 8] has gained wide acceptance as an alternative to symbolic model checking based on BDDs. In principle, BMC can be viewed as a restriction of global model checking, where the number of fixpoint computations is bounded. The idea of BMC is to approximate the fixpoints according to an a priori estimated

bound on the number of iterations. Technically, this is achieved by unwinding both the specification and the implementation a finite number of times. The major advantage offered by BMC is that the verification task is reduced to a *satisfiability problem of the base logic* (usually propositional logic). The obtained formulas can then be checked using sophisticated SAT solvers [37, 49, 53] so that there is no need for canonical normal forms. Experimental results show that BMC is sometimes much more efficient than symbolic model checking using BDDs [2, 22].

In general, the bound for the reduction to a SAT problem is hard to determine, since the number of iterations required to reach a fixpoint is usually not known in advance. For this reason, BMC is often regarded as an incomplete verification method that can only be used to verify (falsify) a specification by searching for witnesses (counterexamples) up to a certain length. For finite state systems, however, there always exists a maximal bound which is called the *completeness threshold*. Given a bound that is greater than or equal to the completeness threshold, BMC is complete for finite systems [7, 27]. Another approach to achieve completeness is *loop detection*, a technique to detect cycles in the transition system [6, 8, 23]. However, this increases the size of the resulting formulas significantly and slows down the verification process. For infinite state systems, the situation is even worse: A completeness threshold may not even exist, i.e., completeness cannot be achieved by choosing a sufficiently large bound, since the fixpoint iteration may not terminate, although it converges to the fixpoint. Moreover, techniques like loop detection may fail due to the existence of infinite paths that do not contain loops.

As another problem of BMC, unwinding temporal logic formulas is an intricate task, at least if one wants to share common subterms that usually appear after some unwinding steps. A better solution [27, 31] is to translate a given temporal logic formula to an equivalent ω-automaton, since automata naturally have the ability to share common parts. In [27], Clarke et al. refer to the use of automata in BMC as the *semantic approach*.

1.5. Temporal logic hierarchy

By considering a finite prefix of a given computation path, BMC is mainly used for the falsification and verification of *safety* and *liveness* properties, respectively. Intuitively, safety properties state that a property invariantly holds along a computation path, and liveness properties state that a property holds at least once on a given path. As a result, a liveness property is satisfied once a state on a path is found that fulfills the desired property. Analogously, a safety property is falsified as soon as a state is found that violates the property.

Besides safety and liveness properties, linear time temporal logic as well as ω-automata can express much more powerful properties [44, 48, 65, 82]. Manna and Pnueli [48] were the first who investigated the *hierarchy of temporal logics* in correspondence to the hierarchy of ω-automata [44, 65, 77, 82]. This hierarchy consists of six classes of temporal properties: safety, liveness, obligation, persistence, recurrence, and reactivity.

Various translations from linear time temporal logic to ω-automata have been developed [25, 36, 40, 45, 63, 65, 84]. Most of these approaches translate a given formula to an equivalent (generalized) Büchi automaton, thus ignoring the membership to the classes in the above mentioned hierarchy. The acceptance condition of a Büchi automaton requires that some set of states must be visited infinitely often, which cannot be checked directly using BMC. For this reason, it is desirable to translate temporal logic formulas to simpler types of ω-automata whose acceptance conditions are just safety and liveness properties. Though impossible for arbitrary temporal logic formulas, most specifications that appear in practice can be translated to such safety and liveness automata.

In theory, such a translation can be accomplished as follows: First, the formula is translated to a Büchi automaton. Then, it is checked whether the automaton can be converted to a safety or a liveness automaton. This can be done by the algorithms given in [42, 44, 47, 65]. However, these algorithms are very complex, thus making the translation to ω-automata hardly feasible in practice.

A more practical approach is to follow a syntactic classification of temporal logic formulas regarding the translation to safety and liveness automata. To this end, Schneider extended Manna and Pnueli's work to future time temporal logic formulas [63, 65], and defined complete temporal logics that correspond to the six classes of the automata hierarchy. The algorithms for translating these sublogics to symbolic descriptions of ω-automata run in linear time w.r.t. the length of the given formula. In addition, they yield symbolic descriptions of the automata which may also be viewed as alternating ω-automata.

The original motivation of [63] was to speed up symbolic model checking of linear time temporal formulas by eliminating unnecessary fairness constraints, or at least by replacing them with simpler liveness constraints. In [68], it was shown that these translations are also very important for bounded model checking for the following reasons:

- The classification given in [63, 65] can be used to *determine the class a formula belongs to in the temporal logic hierarchy*. This is important since the applicability of BMC depends on the kind of specification to be checked. Recall that BMC is mainly used to disprove safety properties and to prove liveness properties.
- *Unwinding the specification* is simpler and more efficient for ω-automata. In particular, this is straightforward for automata whose acceptance conditions are either safety or liveness conditions. Moreover, this allows the direct translation of the acceptance conditions to the alternation-free μ-calculus.
- *Verification of safety properties* can be done by alternative techniques such as invariant checking and temporal induction [67, 70]. The resulting proof goals can then be checked by decision procedures for the base logic.

1.6. Outline

In the following, we do not presuppose a particular base logic $\mathcal{L}$, but present the methods and algorithms in a generic way. Specifications are given in a linear time temporal logic $\mathcal{L}^{\mathsf{LTL}}$ whose atomic propositions belong to $\mathcal{L}$. We use the approach proposed in [63, 65] for translating the specifications to equivalent ω-automata such that the membership to the temporal logic hierarchy is respected. Of course, we are mainly interested in specifications that can be translated to ω-automata whose acceptance conditions are simple safety and liveness conditions.

Then, it is straightforward to formulate the final verification problem in a μ-calculus $\mathcal{L}^{\mu}$ whose atomic propositions in turn belong to the base logic $\mathcal{L}$. After this final translation, we can use bounded variants of both global and local model checking procedures for $\mathcal{L}^{\mu}$ to check the resulting verification problems. Clearly, these procedures can also be used to check problems that are not obtained via the translation from temporal logic.

There is not much other work about BMC of infinite state systems. In [31], a combination of SAT checkers with domain-specific theorem provers is described. The proposed method is based on a reduction of Boolean constraint formulas to a satisfiability problem of propositional logic. The obtained formulas are incrementally refined using a theorem prover by generating lemmas on demand. In this way, the approach can be used with various theories such as linear and bitvector arithmetic. However, sophisticated techniques are required to

efficiently prune out spurious counterexamples that are generated by the SAT solver, but discarded by the theorem prover. Even though the method presented in [31] also follows the semantic approach, it is based on the construction of Büchi automata. In contrast, our method directly exploits safety and liveness properties as described above.

The rest of this paper is organized as follows: In the next section, we describe the syntax and semantics of $\mathcal{L}^\mu$, the μ-calculus on top of our base logic $\mathcal{L}$. Then, we define the temporal logic $\mathcal{L}^{\mathsf{LTL}}$ that allows more readable specifications. In Section 3, we consider the temporal logic hierarchy and sketch translations to equivalent ω-automata. In Section 4, we then show how alternation-free model checking problems for $\mathcal{L}^\mu$ can be reduced to satisfiability problems of $\mathcal{L}$. In Section 5, we briefly describe some basics of our verification tool and present experimental results. Finally, we conclude with a summary and directions for future work.

2. Preliminaries

2.1. μ-calculus

The μ-calculus does not directly provide any means for reasoning about atomic properties of a system, it rather serves as an extension of a given base logic $\mathcal{L}$. In principle, every logic $\mathcal{L}$ can be extended to a corresponding temporal logic $\mathcal{L}^\mu$ by adding fixpoint and modal operators as follows:

Definition 1 (Syntax of the μ-calculus). Given a logic $\mathcal{L}$ and a set of Boolean variables $\mathcal{V}^{\mathbb{B}}$, the set of μ-calculus formulas $\mathcal{L}^\mu$ is defined as follows with $\alpha \in \mathcal{L}$, $\varphi, \psi \in \mathcal{L}^\mu$, and $Z \in \mathcal{V}^{\mathbb{B}}$:

$$\mathcal{L}^\mu := \alpha \mid Z \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \Diamond\varphi \mid \Box\varphi \mid \overleftarrow{\Diamond}\varphi \mid \overleftarrow{\Box}\varphi \mid \mu Z.\varphi \mid \nu Z.\varphi$$

Note that we do not allow negations in $\mathcal{L}^\mu$. Nevertheless, negated formulas may be used in the atomic propositions $\alpha \in \mathcal{L}$. Alternatively, one can allow negations in $\mathcal{L}^\mu$, but in this case it is required that all occurrences of bound variables in fixpoint formulas $\mu Z.\varphi$ and $\nu Z.\varphi$ are positive to guarantee the monotonicity of the underlying state transformers. Otherwise, it is possible to construct fixpoint formulas with no meaning, since the specified fixpoint may not necessarily exist. For the sake of simplicity, we decided to shift negation symbols into atomic propositions, which makes the algorithms more readable.

Definition 2 (Kripke structure). A Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R})$ for a base logic $\mathcal{L}$ over the variables $\mathcal{V} \cup \mathcal{V}'$ is a transition system, where $\mathcal{S}$ is the possibly infinite set of states, $\mathcal{I} \subseteq \mathcal{S}$ are the initial states, and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation. Every state is associated with an assignment of the variables $\mathcal{V}$ in order to evaluate formulas of $\mathcal{L}$ in particular states. The set of initial states $\mathcal{I}$ and the transition relation $\mathcal{R}$ must be definable in the base logic $\mathcal{L}$. In particular, it is assumed that $\mathcal{I}$ and $\mathcal{R}$ can be represented as formulas of the logic $\mathcal{L}$ over the variables $\mathcal{V}$ and $\mathcal{V} \cup \mathcal{V}'$, respectively.

In the following, we denote the subset of states that satisfy a formula $\Phi \in \mathcal{L}^\mu$ by $[\![\Phi]\!]_{\mathcal{K}}$. If the index $\mathcal{K}$ is missing, the expression denotes the set of all $\mathcal{L}$-models of a formula $\Phi \in \mathcal{L}$. Moreover, we write $\mathsf{pre}_{\exists}^{\mathcal{R}}(Q)$ and $\mathsf{pre}_{\forall}^{\mathcal{R}}(Q)$ for the existential and universal predecessors of a

set of states Q under the transition relation $\mathcal{R}$. The sets of existential and universal successors of Q are denoted by $\mathsf{suc}_{\exists}^{\mathcal{R}}(Q)$ and $\mathsf{suc}_{\forall}^{\mathcal{R}}(Q)$, respectively.

Definition 3 (Semantics of $\mathcal{L}^{\mu}$). Given a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R})$ and a formula $\alpha \in \mathcal{L}$, the semantics of $\mathcal{L}^{\mu}$ is defined as follows:

- $[\![\alpha]\!]_{\mathcal{K}} :\equiv [\![\alpha]\!] \cap \mathcal{S}$
- $[\![Z]\!]_{\mathcal{K}} :\equiv \{s \in \mathcal{S} \mid s(Z)\}$
- $[\![\varphi \wedge \psi]\!]_{\mathcal{K}} :\equiv [\![\varphi]\!]_{\mathcal{K}} \cap [\![\psi]\!]_{\mathcal{K}}$
- $[\![\varphi \vee \psi]\!]_{\mathcal{K}} :\equiv [\![\varphi]\!]_{\mathcal{K}} \cup [\![\psi]\!]_{\mathcal{K}}$
- $[\![\Diamond\varphi]\!]_{\mathcal{K}} :\equiv \mathsf{pre}_{\exists}^{\mathcal{R}}([\![\varphi]\!]_{\mathcal{K}})$
- $[\![\Box\varphi]\!]_{\mathcal{K}} :\equiv \mathsf{pre}_{\forall}^{\mathcal{R}}([\![\varphi]\!]_{\mathcal{K}})$
- $[\![\overleftarrow{\Diamond}\varphi]\!]_{\mathcal{K}} :\equiv \mathsf{suc}_{\exists}^{\mathcal{R}}([\![\varphi]\!]_{\mathcal{K}})$
- $[\![\overleftarrow{\Box}\varphi]\!]_{\mathcal{K}} :\equiv \mathsf{suc}_{\forall}^{\mathcal{R}}([\![\varphi]\!]_{\mathcal{K}})$
- $[\![\mu Z.\varphi]\!]_{\mathcal{K}}$ is the least set Q where $Q = [\![\varphi]\!]_{\mathcal{K}_Z^Q}$ holds
- $[\![\nu Z.\varphi]\!]_{\mathcal{K}}$ is the greatest set Q where $Q = [\![\varphi]\!]_{\mathcal{K}_Z^Q}$ holds

  $\mathcal{K}_Z^Q$ is the Kripke structure obtained from $\mathcal{K}$ by changing the states s of $\mathcal{K}$ such that $s(Z)$ holds iff $s \in Q$ holds. A Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R})$ satisfies a formula Φ iff $\mathcal{I} \subseteq [\![\Phi]\!]_{\mathcal{K}}$ holds.

2.2. Linear time temporal logic

The μ-calculus defined in the previous subsection is very expressive and its model checking problem is well understood, although the precise complexity class is still an open problem [65]. Unfortunately, the μ-calculus is not very readable so that even simple fixpoint formulas may be difficult to understand. For this reason, more readable specification logics have been proposed that can be translated to the μ-calculus in order to benefit from existing model checking procedures. Thus, the μ-calculus is often viewed as some sort of 'assembly' language for verification [32, 65]. In this section, we consider a linear time temporal logic $\mathcal{L}^{\mathsf{LTL}}$ [57] as a comfortable specification logic.

Definition 4 (Syntax of $\mathcal{L}^{\mathsf{LTL}}$). The set of $\mathcal{L}^{\mathsf{LTL}}$ formulas is defined as follows with $\alpha \in \mathcal{L}$ and $\varphi, \psi \in \mathcal{L}^{\mathsf{LTL}}$:

$$\mathcal{L}^{\mathsf{LTL}} := \alpha \mid \neg\varphi \mid \varphi \wedge \psi \mid \mathsf{X}\varphi \mid \overleftarrow{\underline{\mathsf{X}}}\,\varphi \mid [\varphi\, \underline{\mathsf{U}}\, \psi] \mid [\varphi\, \overleftarrow{\underline{\mathsf{U}}}\, \psi]$$

The atomic formulas of $\mathcal{L}^{\mathsf{LTL}}$ are exactly the formulas of our base logic $\mathcal{L}$. Hence, $\mathcal{L}^{\mathsf{LTL}}$ is the closure of $\mathcal{L}$ under Boolean and temporal operators. $\mathcal{L}^{\mathsf{LTL}}$ has four temporal operators: X, $\overleftarrow{\underline{\mathsf{X}}}$ (next and previous) and $\underline{\mathsf{U}}$, $\overleftarrow{\underline{\mathsf{U}}}$ (until and past-until, often called 'since'). Intuitively, the formula $\mathsf{X}\varphi$ holds iff φ holds at the next point of time and $[\varphi\, \underline{\mathsf{U}}\, \psi]$ holds iff φ holds until ψ holds. The operators $\overleftarrow{\underline{\mathsf{X}}}$ and $\overleftarrow{\underline{\mathsf{U}}}$ are defined similarly but refer to the *past* instead of to the future.

To interpret a formula, we need the notion of paths. Given a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R})$, a path $\pi : \mathbb{N} \to \mathcal{S}$ is an infinite sequence π of states that are connected by transitions, i.e., we have $\forall t \in \mathbb{N}.(\pi(t), \pi(t+1)) \in \mathcal{R}$. The set of paths originating in a state $s \in \mathcal{S}$ is denoted by $\mathsf{Paths}_{\mathcal{K}}(s) := \{\pi : \mathbb{N} \to \mathcal{S} \mid \pi(0) = s\}$.

Definition 5 (Semantics of $\mathcal{L}^{\mathsf{LTL}}$). Let $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R})$ be a Kripke structure, π a path, and $t \in \mathbb{N}$ a natural number. Then, the semantics of $\mathcal{L}^{\mathsf{LTL}}$ is recursively defined as follows:

- $(\pi, t) \models \alpha$ iff $\pi(t) \in [\![\alpha]\!]$ for $\alpha \in \mathcal{L}$
- $(\pi, t) \models \neg\varphi$ iff not $(\pi, t) \models \varphi$
- $(\pi, t) \models \varphi \wedge \psi$ iff $(\pi, t) \models \varphi$ and $(\pi, t) \models \psi$
- $(\pi, t) \models \mathsf{X}\varphi$ iff $(\pi, t+1) \models \varphi$
- $(\pi, t) \models \overleftarrow{\underline{\mathsf{X}}}\,\varphi$ iff $t \neq 0$ and $(\pi, t-1) \models \varphi$
- $(\pi, t) \models [\varphi\, \underline{\mathsf{U}}\, \psi]$ iff there is a $v \in \mathbb{N}$ with $v \geq t$ and $(\pi, v) \models \psi$ such that for all $u \in \mathbb{N}$ with $t \leq u < v$, we have $(\pi, u) \models \varphi$
- $(\pi, t) \models [\varphi\, \overleftarrow{\underline{\mathsf{U}}}\, \psi]$ iff there is a $v \in \mathbb{N}$ with $v \leq t$ and $(\pi, v) \models \psi$ such that for all $u \in \mathbb{N}$ with $v < u \leq t$, we have $(\pi, u) \models \varphi$

So far, we only considered the truth of a formula with respect to a given path, but not with respect to a particular state. To this end, path quantifiers A and E are usually used to obtain state formulas from path formulas. Thus, the semantics of a path formula is a set of states where the formula holds. For a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R})$, a state $s \in \mathcal{S}$, and a formula $\varphi \in \mathcal{L}^{\mathsf{LTL}}$, we define:

$$(\mathcal{K}, s) \models \mathsf{A}\varphi :\Leftrightarrow \forall \pi \in \mathsf{Paths}_{\mathcal{K}}(s).(\pi, 0) \models \varphi$$
$$(\mathcal{K}, s) \models \mathsf{E}\varphi :\Leftrightarrow \exists \pi \in \mathsf{Paths}_{\mathcal{K}}(s).(\pi, 0) \models \varphi$$

In the above definitions, we only considered a small set of Boolean and temporal operators. These are sufficient to reach the expressiveness of the first-order monadic theory of linear orders [65]. In practice, it is nevertheless convenient to define further temporal operators as syntactic sugar:

$$\begin{array}{llll}
 & & \overleftarrow{\mathsf{X}}\,\varphi & :\equiv \neg\overleftarrow{\underline{\mathsf{X}}}\neg\varphi \\
\mathsf{F}\varphi & :\equiv [\mathsf{true}\,\underline{\mathsf{U}}\,\varphi] & \overleftarrow{\mathsf{F}}\,\varphi & :\equiv [\mathsf{true}\,\overleftarrow{\underline{\mathsf{U}}}\,\varphi] \\
\mathsf{G}\varphi & :\equiv \neg\mathsf{F}\neg\varphi & \overleftarrow{\mathsf{G}}\,\varphi & :\equiv \neg\overleftarrow{\mathsf{F}}\neg\varphi \\
[\varphi\,\mathsf{U}\,\psi] & :\equiv [\varphi\,\underline{\mathsf{U}}\,\psi] \vee \mathsf{G}\varphi & [\varphi\,\overleftarrow{\mathsf{U}}\,\psi] & :\equiv [\varphi\,\overleftarrow{\underline{\mathsf{U}}}\,\psi] \vee \overleftarrow{\mathsf{G}}\,\varphi
\end{array}$$

The formula $\mathsf{F}\varphi$ holds along a path iff φ eventually holds, and $\mathsf{G}\varphi$ holds iff φ holds on all positions of the path. $[\varphi\,\mathsf{U}\,\psi]$ holds if either $[\varphi\,\underline{\mathsf{U}}\,\psi]$ holds or φ invariantly holds. The semantics of the past operators $\overleftarrow{\mathsf{F}}$, $\overleftarrow{\mathsf{G}}$, and $\overleftarrow{\mathsf{U}}$ are defined analogously. $\overleftarrow{\mathsf{U}}$ and U are often called weak until operators, while $\overleftarrow{\underline{\mathsf{U}}}$ and $\underline{\mathsf{U}}$ are the corresponding strong until operators. The distinction between weak and strong operators is the key to defining liveness and safety properties, respectively.

3. Translating safety and liveness properties to fixpoint problems

In this section, we show how specifications given in the linear time temporal logic $\mathcal{L}^{\mathsf{LTL}}$ can be translated to equivalent ω-automata and finally to the μ-calculus $\mathcal{L}^{\mu}$. As mentioned previously, we are particularly interested in alternation-free formulas, i.e., fixpoint formulas without mutually interdependent fixpoints. Although such translations are impossible for arbitrary temporal logic formulas, this can be achieved for many interesting formulas including all safety and liveness properties. Given a particular specification, we first have to check whether such a translation is possible. If so, we have to translate them to simple classes of

ω-automata so that we finally obtain alternation-free μ-calculus model checking problems. The key to solve these problems is the work on the temporal logic hierarchy presented in [63, 65].

In the following, we briefly consider the principles of these translations. For that purpose, we make use of symbolic descriptions of finite state ω-automata which are given as formulas $\mathcal{A}_{\exists}(\mathcal{Q}, \mathcal{I}, \mathcal{R}, \mathcal{F})$, where

- $\mathcal{Q}$ is the finite set of state variables of the automaton,
- $\mathcal{I}$ is a formula that encodes the set of initial states,
- $\mathcal{R}$ is a formula with X operators that represents the transition relation, and
- $\mathcal{F}$ is the acceptance condition given as an $\mathcal{L}^{\mathsf{LTL}}$ formula.

An automaton formula holds on a path π of a Kripke structure iff there is a run through the ω-automaton that satisfies the acceptance condition $\mathcal{F}$. Such a run is obtained by extending the labels on π with an assignment of the state variables $\mathcal{Q}$ of the automaton such that $\mathcal{I}$ is initially and $\mathcal{R}$ invariantly satisfied. The acceptance condition $\mathcal{F}$ is thereby restricted to special templates for particular classes of ω-automata (see below).

The translation we will consider as the first step is essentially given in [25], and has already been sketched in [19]. In principle, this procedure is the same as the one developed for alternating ω-automata [54, 80]. The difference between symbolically represented nondeterministic ω-automata and alternating ω-automata can be neglected, and seems to be rather a matter of taste [79]. For our explanations, we take the view on symbolically represented nondeterministic ω-automata. To understand the translation, we have to consider the *elementary subformulas* of a given formula Φ, which are those subformulas of Φ that start with a temporal operator. The states of the ω-automaton to be constructed consist of the different truth values of these elementary formulas. For example, if Φ has the elementary formulas $\{\varphi_1, \ldots, \varphi_n\}$, we need n state variables $\{q_1, \ldots, q_n\}$ to encode the state set.

As the introduced state variables q_i are used to abbreviate elementary subformulas, we want for any run through the automaton that $q_i \leftrightarrow \varphi_i$ holds at every point of time. For this reason, the transition relation of the automaton must respect the semantics of the temporal operators that occur in φ_i. Since transition relations represent the values of two succeeding points of time, it is clear that we have to obey the recursion laws of the temporal operators. For example, if a state variable q shall invariantly satisfy $q \leftrightarrow [\varphi \mathsf{U} \psi]$, then we demand that $q \leftrightarrow \psi \vee \varphi \wedge \mathsf{X}q$ is implied by the transition relation, since $[\varphi \mathsf{U} \psi] \leftrightarrow \psi \vee \varphi \wedge \mathsf{X}[\varphi \mathsf{U} \psi]$ holds. However, the following theorem shows that this is not sufficient [63, 65]:

Theorem 1. *Given propositional formulas φ and ψ, then $\mathsf{G}[q \leftrightarrow \psi \vee \varphi \wedge \mathsf{X}q]$ holds iff $\mathsf{G}(q \leftrightarrow [\varphi \underline{\mathsf{U}} \psi]) \vee \mathsf{G}(q \leftrightarrow [\varphi \mathsf{U} \psi])$ holds.*

Similar equivalences hold for other temporal operators including past time temporal operators [65]. The theorem intuitively states that (1) strong and weak operators fulfill exactly the same recursion laws ($\Leftarrow$), and (2) no other formulas satisfy these laws ($\Rightarrow$). Hence, simply using the recursion equations to abbreviate elementary subformulas is not sufficient for a translation to ω-automata.

For future time operators the distinction between the weak and the corresponding strong operators can be done by using an additional fairness constraint, i.e., a formula of the form $\mathsf{GF}\varphi$ which states that φ eventually occurs at every point of time. For example, for an abbreviation $q \leftrightarrow [\varphi \underline{\mathsf{U}} \psi]$, we have to guarantee that ψ will eventually hold whenever q holds, since the strong until operator requires that ψ must eventually hold. The required fairness constraints that complete the translation are listed in the following theorem, where

we write $\Phi\langle\varphi\rangle_x$ for the formula obtained by replacing all occurrences of the variable x in Φ with the formula φ:

Theorem 2. *Given a formula Φ, a variable x, and propositional formulas φ and ψ, the following equations are valid:*

$$\begin{aligned}
\Phi\langle \overleftarrow{\mathsf{X}}\, \varphi\rangle_x &\Leftrightarrow \mathcal{A}_\exists(\{q\},\ q,\ \mathsf{X}q \leftrightarrow \varphi,\ \Phi\langle q\rangle_x)\\
\Phi\langle \underline{\overleftarrow{\mathsf{X}}}\, \varphi\rangle_x &\Leftrightarrow \mathcal{A}_\exists(\{q\},\ \neg q,\ \mathsf{X}q \leftrightarrow \varphi,\ \Phi\langle q\rangle_x)\\
\Phi\langle[\varphi\, \overleftarrow{\mathsf{U}}\, \psi]\rangle_x &\Leftrightarrow \mathcal{A}_\exists(\{q\},\ q,\ \mathsf{X}q \leftrightarrow \psi \vee \varphi \wedge q,\ \Phi\langle \psi \vee \varphi \wedge q\rangle_x)\\
\Phi\langle[\varphi\, \underline{\overleftarrow{\mathsf{U}}}\, \psi]\rangle_x &\Leftrightarrow \mathcal{A}_\exists(\{q\},\ \neg q,\ \mathsf{X}q \leftrightarrow \psi \vee \varphi \wedge q,\ \Phi\langle \psi \vee \varphi \wedge q\rangle_x)\\
\Phi\langle \mathsf{X}\varphi\rangle_x &\Leftrightarrow \mathcal{A}_\exists(\{q\},\ \mathsf{true},\ q \leftrightarrow \mathsf{X}\varphi,\ \Phi\langle q\rangle_x)\\
\Phi\langle[\varphi\, \mathsf{U}\, \psi]\rangle_x &\Leftrightarrow \mathcal{A}_\exists(\{q\},\ \mathsf{true},\ q \leftrightarrow \psi \vee \varphi \wedge \mathsf{X}q, \Phi\langle q\rangle_x \wedge \mathsf{GF}[\varphi \rightarrow q])\\
\Phi\langle[\varphi\, \underline{\mathsf{U}}\, \psi]\rangle_x &\Leftrightarrow \mathcal{A}_\exists(\{q\},\ \mathsf{true},\ q \leftrightarrow \psi \vee \varphi \wedge \mathsf{X}q, \Phi\langle q\rangle_x \wedge \mathsf{GF}[q \rightarrow \psi])
\end{aligned}$$

The symbol $\varphi \Leftrightarrow \psi$ means that φ and ψ hold on the same paths (of all Kripke structures). Hence, we can use the above equivalences to iteratively replace the elementary subformulas of a given $\mathcal{L}^{\mathsf{LTL}}$ formula with the corresponding state formulas of the intended ω-automaton.

As an example, consider the formula $[(a \vee [b\, \mathsf{U}\, c])\, \underline{\mathsf{U}}\, d] \vee e$. We first replace $[b\, \mathsf{U}\, c]$ with a new state variable p by adding the recursion equation $p \leftrightarrow c \vee b \wedge \mathsf{X}p$ and the required fairness constraint $\mathsf{GF}[b \rightarrow p]$:

$$\mathcal{A}_\exists(\{p\},\ \mathsf{true},\ p \leftrightarrow c \vee b \wedge \mathsf{X}p, ([(a \vee p)\, \underline{\mathsf{U}}\, d] \vee e) \wedge \mathsf{GF}[b \rightarrow p])$$

In the next step, we replace $[(a \vee p)\, \underline{\mathsf{U}}\, d]$ with a new state variable q by adding the recursion equation $q \leftrightarrow d \vee (a \vee p) \wedge \mathsf{X}q$ and the required fairness constraint $\mathsf{GF}[q \rightarrow d]$:

$$\mathcal{A}_\exists\left(\begin{array}{l}\{p,q\}, \mathsf{true},\\ (p \leftrightarrow c \vee b \wedge \mathsf{X}p) \wedge\\ (q \leftrightarrow d \vee (a \vee p) \wedge \mathsf{X}q),\\ (q \vee e) \wedge \mathsf{GF}[b \rightarrow p] \wedge \mathsf{GF}[q \rightarrow d]\end{array}\right)$$

The acceptance condition requires that $q \vee e$ must initially hold and that infinitely often both $b \rightarrow p$ and $q \rightarrow d$ hold. For this reason, we can alternatively use $q \vee e$ as initial condition and obtain the following generalized Büchi automaton:

$$\mathcal{A}_\exists\left(\begin{array}{l}\{p,q\}, q \vee e,\\ (p \leftrightarrow c \vee b \wedge \mathsf{X}p) \wedge\\ (q \leftrightarrow d \vee (a \vee p) \wedge \mathsf{X}q),\\ \mathsf{GF}[b \rightarrow p] \wedge \mathsf{GF}[q \rightarrow d]\end{array}\right)$$

Further (and different) explanations of this translation can be found in [19, 25, 40, 54, 63, 65, 80]. Note that the translation takes only linear time to compute the linear sized symbolic representation of the resulting ω-automaton.

For bounded model checking, the fairness constraints in the acceptance condition of a Büchi automaton impose a major problem: They require to reason about an infinite behavior which is not directly possible using bounded model checking. It is therefore mandatory to

be able to eliminate these constraints or at least to replace them with simpler constraints that can be handled by bounded model checking.

A first step towards such an improvement is the observation that all temporal operators are monotonic and that the strong operators imply the weak ones. Therefore, positive occurrences of weak operators and negative occurrences of strong operators do not require the additional fairness constraint (see [63, 65] for details).

Theorem 3 *(Translation to ω-automata w.r.t. positive/negative occurrences). Given a formula Φ, a variable x, and propositional formulas φ and ψ, the following equation is valid, provided that all occurrences of x in Φ are positive:*

$$\Phi\langle[\varphi \mathsf{U} \psi]\rangle_x \Leftrightarrow \mathcal{A}_\exists(\{q\}, \mathsf{true}, q \leftrightarrow \psi \vee \varphi \wedge \mathsf{X}q, \Phi\langle q\rangle_x)$$

If all occurrences of x in Φ are negative, then the following holds:

$$\Phi\langle[\varphi \underline{\mathsf{U}} \psi]\rangle_x \Leftrightarrow \mathcal{A}_\exists(\{q\}, \mathsf{true}, q \leftrightarrow \psi \vee \varphi \wedge \mathsf{X}q, \Phi\langle q\rangle_x)$$

In the above example, we can therefore eliminate $\mathsf{GF}[b \rightarrow p]$ and thus simply use the acceptance condition $\mathsf{GF}[q \rightarrow d]$.

A second improvement is obtained by checking whether a positive occurrence of a subformula φ starting with a strong operator is in the scope of another temporal operator that requires to check φ infinitely often. If this is not the case, as in our example, we can replace the fairness constraint with simpler reachability constraints.[2] Hence, we can simplify the acceptance condition once more and use $\mathsf{F}[q \rightarrow d]$. As this formula is now a simple liveness property, it can be proved by bounded model checking. This is not possible without our improvements, i.e., with the classic translation introduced at the beginning of this section.

The second improvement allows us to *replace some of the remaining fairness constraints with simpler reachability constraints*. This improvement can be used as long as only strong temporal future operators are nested into each other, followed only by nestings of weak temporal future operators. Based on these improvements, the logics TL_κ given in Fig. 1 have been defined in [63] and the following theorem has been proved:

Theorem 4 *(Temporal logic hierarchy). We define the logics TL_κ for $\kappa \in \{\mathsf{G}, \mathsf{F}, \mathsf{Prefix}, \mathsf{FG}, \mathsf{GF}, \mathsf{Streett}\}$ by the grammar rules given in Fig. 1, where TL_κ is the set of formulas that can be derived from the nonterminal P_κ. Then, all formulas in TL_κ can be translated to equivalent ω-automata whose acceptance conditions are as follows (with formulas $\Phi_i, \Psi_i \in \mathcal{L}$):*

$$\kappa = \mathsf{G}\text{: } \mathsf{G}\Phi_0 \qquad \kappa = \mathsf{F}\text{: } \mathsf{F}\Phi_0 \qquad \kappa = \mathsf{Prefix}\text{: } \bigwedge_{j=0}^{f} \mathsf{G}\Phi_j \vee \mathsf{F}\Psi_j$$

$$\kappa = \mathsf{FG}\text{: } \mathsf{FG}\Phi_0 \qquad \kappa = \mathsf{GF}\text{: } \mathsf{GF}\Phi_0 \qquad \kappa = \mathsf{Streett}\text{: } \bigwedge_{j=0}^{f} \mathsf{FG}\Phi_j \vee \mathsf{GF}\Psi_j$$

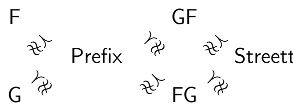
[2] There are some involved details that have to be considered for such a simplification. We neglect these details here and refer to [65] instead.

$P_{\mathsf{G}} ::= \mathcal{L} \mid \neg P_{\mathsf{F}} \mid P_{\mathsf{G}} \wedge P_{\mathsf{G}} \mid P_{\mathsf{G}} \vee P_{\mathsf{G}}$ $\mid \overleftarrow{\mathsf{X}} P_{\mathsf{G}} \mid [P_{\mathsf{G}}\ \overleftarrow{\mathsf{U}}\ P_{\mathsf{G}}]$ $\mid \overleftarrow{\underline{\mathsf{X}}} P_{\mathsf{G}} \mid [P_{\mathsf{G}}\ \overleftarrow{\underline{\mathsf{U}}}\ P_{\mathsf{G}}]$ $\mid \mathsf{X} P_{\mathsf{G}} \mid [P_{\mathsf{G}}\ \mathsf{U}\ P_{\mathsf{G}}]$	$P_{\mathsf{F}} ::= \mathcal{L} \mid \neg P_{\mathsf{G}} \mid P_{\mathsf{F}} \wedge P_{\mathsf{F}} \mid P_{\mathsf{F}} \vee P_{\mathsf{F}}$ $\mid \overleftarrow{\mathsf{X}} P_{\mathsf{F}} \mid [P_{\mathsf{F}}\ \overleftarrow{\mathsf{U}}\ P_{\mathsf{F}}]$ $\mid \overleftarrow{\underline{\mathsf{X}}} P_{\mathsf{F}} \mid [P_{\mathsf{F}}\ \overleftarrow{\underline{\mathsf{U}}}\ P_{\mathsf{F}}]$ $\mid \mathsf{X} P_{\mathsf{F}} \mid [P_{\mathsf{F}}\ \underline{\mathsf{U}}\ P_{\mathsf{F}}]$
$P_{\mathsf{Prefix}} ::= P_{\mathsf{G}} \mid P_{\mathsf{F}} \mid \neg P_{\mathsf{Prefix}} \mid P_{\mathsf{Prefix}} \wedge P_{\mathsf{Prefix}} \mid P_{\mathsf{Prefix}} \vee P_{\mathsf{Prefix}}$	
$P_{\mathsf{GF}} ::= P_{\mathsf{Prefix}}$ $\mid \neg P_{\mathsf{FG}} \mid P_{\mathsf{GF}} \wedge P_{\mathsf{GF}} \mid P_{\mathsf{GF}} \vee P_{\mathsf{GF}}$ $\mid \overleftarrow{\mathsf{X}} P_{\mathsf{GF}} \mid \overleftarrow{\underline{\mathsf{X}}} P_{\mathsf{GF}} \mid \mathsf{X} P_{\mathsf{GF}}$ $\mid [P_{\mathsf{GF}}\ \overleftarrow{\mathsf{U}}\ P_{\mathsf{GF}}] \mid [P_{\mathsf{GF}}\ \overleftarrow{\underline{\mathsf{U}}}\ P_{\mathsf{GF}}]$ $\mid [P_{\mathsf{GF}}\ \mathsf{U}\ P_{\mathsf{GF}}] \mid [P_{\mathsf{GF}}\ \underline{\mathsf{U}}\ P_{\mathsf{F}}]$	$P_{\mathsf{FG}} ::= P_{\mathsf{Prefix}}$ $\mid \neg P_{\mathsf{GF}} \mid P_{\mathsf{FG}} \wedge P_{\mathsf{FG}} \mid P_{\mathsf{FG}} \vee P_{\mathsf{FG}}$ $\mid \overleftarrow{\mathsf{X}} P_{\mathsf{FG}} \mid \mathsf{X} P_{\mathsf{FG}} \mid \overleftarrow{\underline{\mathsf{X}}} P_{\mathsf{FG}}$ $\mid [P_{\mathsf{FG}}\ \overleftarrow{\mathsf{U}}\ P_{\mathsf{FG}}] \mid [P_{\mathsf{FG}}\ \overleftarrow{\underline{\mathsf{U}}}\ P_{\mathsf{FG}}]$ $\mid [P_{\mathsf{FG}}\ \underline{\mathsf{U}}\ P_{\mathsf{FG}}] \mid [P_{\mathsf{G}}\ \mathsf{U}\ P_{\mathsf{FG}}]$
$P_{\mathsf{Streett}} ::= P_{\mathsf{GF}} \mid P_{\mathsf{FG}} \mid \neg P_{\mathsf{Streett}} \mid P_{\mathsf{Streett}} \wedge P_{\mathsf{Streett}} \mid P_{\mathsf{Streett}} \vee P_{\mathsf{Streett}}$	

Fig. 1 Syntactic characterization of the six classes of the temporal logic hierarchy

TL_{G} is the set of formulas where each occurrence of a weak/strong temporal future operator is positive/negative, and similarly, each occurrence of a weak/strong temporal future operator in TL_{F} is negative/positive. Hence, both logics are dual to each other, which means that one contains the negations of the other. $\mathsf{TL}_{\mathsf{Prefix}}$ is the Boolean closure of TL_{G} (and TL_{F}). The logics $\mathsf{TL}_{\mathsf{GF}}$ and $\mathsf{TL}_{\mathsf{FG}}$ are constructed in the same way as TL_{G} and TL_{F} (with minor differences explained in [63, 65]).

In [63, 65], translation procedures are given that translate formulas of TL_κ to ω-automata whose acceptance conditions are of the corresponding type κ. It is well-known that these acceptance conditions yield six classes of ω-automata, where each class can be represented by deterministic automata of type κ [44, 48, 65, 77, 82]. The different expressiveness is indicated with $\precnapprox$ below:



For bounded model checking, we are mainly interested in ω-automata with acceptance conditions $\mathsf{G}\varphi$ and $\mathsf{F}\varphi$. As we can disprove $\mathsf{G}\varphi$ and prove $\mathsf{F}\varphi$ by only considering a finite prefix of a path of a Kripke structure, these formulas can be checked using bounded model checking.

Let us now complete the picture regarding the translation of a given model checking problem for $\mathcal{L}^{\mathsf{LTL}}$ to an equivalent model checking problem for $\mathcal{L}^{\mu}$. For that purpose, assume we are given the model checking problem $\mathcal{K}, s \models \mathsf{A}\varphi$ with $\varphi \in \mathcal{L}^{\mathsf{LTL}}$. First, we translate $\neg\varphi$ to an ω-automaton $\mathcal{A}_{\neg\varphi} \equiv \mathcal{A}_{\exists}(\mathcal{Q}, \mathcal{I}, \mathcal{R}, \mathcal{F})$ by the methods presented above, so that we obtain the model checking problem $\mathcal{K}, s \models \neg\mathsf{E}\mathcal{A}_{\exists}(\mathcal{Q}, \mathcal{I}, \mathcal{R}, \mathcal{F})$. The translation of $\mathsf{E}\mathcal{A}_{\exists}(\mathcal{Q}, \mathcal{I}, \mathcal{R}, \mathcal{F})$ to the μ-calculus can be circumvented by considering the product structure $\mathcal{K} \times \mathcal{A}_{\neg\varphi}$. We have to check whether there is a state q of the automaton $\mathcal{A}_{\neg\varphi}$ such that $\mathcal{K} \times \mathcal{A}_{\neg\varphi}, (s, q) \models \mathcal{I} \wedge \neg\mathsf{E}\mathcal{F}$ holds. As $\mathcal{I}$ is in $\mathcal{L}$, and thus already an $\mathcal{L}^{\mu}$-formula, it remains to translate $\mathsf{E}\mathcal{F}$ to the μ-calculus. For the most interesting cases, this is accomplished as follows:

- $\mathsf{EG}\varphi = \nu Y.\varphi \wedge \Diamond Y$
- $\mathsf{EF}\varphi = \mu Y.(\varphi \wedge \nu X.\Diamond X) \vee \Diamond Y$
- $\mathsf{EFG}\varphi = \mathsf{EFEG}\varphi = \mu Y.(\nu X.\varphi \wedge \Diamond X) \vee \Diamond Y$
- $\mathsf{EGF}\varphi = \nu Y.\Diamond \mu X.(Y \wedge \varphi) \vee \Diamond X$

As can be seen, we obtain alternation-free μ-calculus formulas in the first three cases. Moreover, $\mathsf{EG}\varphi$ yields a simple greatest fixpoint, and $\mathsf{EF}\varphi$ essentially a least fixpoint. The included greatest fixpoint $\nu X.\Diamond X$ is used to guarantee that the considered states have an infinite path. If this is guaranteed by other means, e.g. if the transition relation is total, we can alternatively use the simpler reduction $\mathsf{EF}\varphi = \mu Y.\varphi \vee \Diamond Y$.

4. Verification of fixpoint problems

In the previous section, we considered the translation of temporal logic specifications to equivalent ω-automata. This reduces the model checking problem to a simple fixed point problem. In principle, it would now be sufficient to consider the fixpoints obtained by the acceptance conditions of the corresponding ω-automata, i.e., the formulas $\mathsf{EG}\varphi$, $\mathsf{EF}\varphi$, $\mathsf{EGF}\varphi$, and $\mathsf{EFG}\varphi$.

In this section, we consider the more general model checking problem for $\mathcal{L}^\mu$. In this way, we are able to deal with other temporal logics such as CTL and even to check a large class of specifications that go beyond temporal logic. As an example, consider the formula $\nu Z.\varphi \wedge \Box\Box Z$ which states that φ holds at every other point of time. This property cannot be expressed in LTL [50, 83], and hence, it cannot be checked using traditional approaches for bounded model checking. Other examples include the computation of winning strategies for controller synthesis.

To check such specifications, we present bounded model checking procedures for both global and local model checking. For that purpose, we employ three-valued logic to explicitly encode information about the membership of a state in a set which may be either definitely true, definitely false, or unknown due to insufficient bounds. We start with a description of the formal background of ternary fixpoints in the following subsection.

4.1. Three-valued logic

Three-valued logic has been applied in many areas of computer science [1, 4, 15, 46, 60, 66, 71]. Besides the Boolean values 0 and 1, three-valued logic [15, 41] provides an additional truth value denoted as $\perp$ that is used to express unknown or uncertain information. In our case, we will use $\perp$ to express that bounded model checking was neither able to prove nor to disprove that a state satisfies the considered property. To this end, we replace the two-valued characteristic function of a set S with its three-valued generalization:

$$\chi_S(x) := \begin{cases} 0 & : \text{if } x \notin S \text{ is known} \\ 1 & : \text{if } x \in S \text{ is known} \\ \perp & : \text{if neither } x \in S \text{ nor } x \notin S \text{ is known} \end{cases}$$

In analogy to the two-valued case, all set operations can be performed using negation, conjunction, and disjunction that are defined as follows in the three-valued setting:

x	$\neg x$
0	1
$\perp$	$\perp$
1	0

$\wedge$	0	$\perp$	1
0	0	0	0
$\perp$	0	$\perp$	$\perp$
1	0	$\perp$	1

$\vee$	0	$\perp$	1
0	0	$\perp$	1
$\perp$	$\perp$	$\perp$	1
1	1	1	1

We use the partial order[3] $0 \sqsubseteq \bot \sqsubseteq 1$ and extend it to arbitrary functions $f, g\colon \mathcal{S} \rightarrow \{0, \bot, 1\}$ by pointwise comparison: $f \sqsubseteq g :\Leftrightarrow \forall s \in \mathcal{S}. f(s) \sqsubseteq g(s)$. Given an arbitrary set $\mathcal{S}$, it is easily seen that the set of all functions $f\colon \mathcal{S} \rightarrow \{0, \bot, 1\}$ is a complete lattice with this partial order. The minimal element is $f_{\min}(s) := 0$ (the set that is definitely empty) and the maximal element is $f_{\max}(s) := 1$ (the set that definitely contains all states).

By the Tarski-Knaster fixpoint theorem [65, 76], every continuous function $\gamma\colon(\mathcal{S} \rightarrow \{0, \bot, 1\}) \rightarrow (\mathcal{S} \rightarrow \{0, \bot, 1\})$ has fixpoints, and the uniquely defined least and greatest fixpoints are $\lim_{k\rightarrow\infty} \gamma^k(f_{\min})$ and $\lim_{k\rightarrow\infty} \gamma^k(f_{\max})$, respectively. This is in complete analogy to the two-valued case, where fixpoints are computed over state sets, i.e., fixpoints of a continuous function[4] $\gamma\colon (\mathcal{S} \rightarrow \{0, 1\}) \rightarrow (\mathcal{S} \rightarrow \{0, 1\})$.

Note, however, that the three-valued negation is not monotonic, and thus, it is not a continuous function. Fortunately, the μ-calculus requires that all occurrences of fixpoint variables are positive, so that the functions we have to consider are continuous, and therefore monotonic.

In our implementation, we encode the three truth values by pairs of two-valued ones using dual-rail encoding:

x	0	$\bot$	1
$\epsilon(x)$	(0, 1)	(0, 0)	(1, 0)

The three-valued extension of a formula φ is consequently denoted by $\epsilon(\varphi)$. It is easily seen that for this encoding, the three-valued operations can be implemented as follows:

- $\dot{\neg}(\varphi_1, \varphi_2) := (\varphi_2, \varphi_1)$
- $(\varphi_1, \varphi_2)\dot{\wedge}(\psi_1, \psi_2) := (\varphi_1 \wedge \psi_1, \varphi_2 \vee \psi_2)$
- $(\varphi_1, \varphi_2)\dot{\vee}(\psi_1, \psi_2) := (\varphi_1 \vee \psi_1, \varphi_2 \wedge \psi_2)$
- $(\varphi_1, \varphi_2)\dot{\rightarrow}(\psi_1, \psi_2) := (\varphi_2 \vee \psi_1, \varphi_1 \wedge \psi_2)$
- $\dot{\exists} x.(\varphi_1, \varphi_2) := (\exists x.\varphi_1, \forall x.\varphi_2)$
- $\dot{\forall} x.(\varphi_1, \varphi_2) := (\forall x.\varphi_1, \exists x.\varphi_2)$

Dual-rail encoding is an elegant means to reason about three-valued logic using data structures and decision procedures for the base logic. Given that $\chi_S : \mathcal{S} \rightarrow \{0, \bot, 1\}$ is represented by (φ_1, φ_0) with $\varphi_i : \mathcal{S} \rightarrow \{0, 1\}$, dual-rail encoding has the following meaning regarding the representation of three-valued characteristic functions:

- $\varphi_1(s) = 1$ implies $\chi_S(s) = 1$, and therefore $\varphi_1(s)$ holds iff s definitely belongs to the represented set.
- $\varphi_0(s) = 1$ implies $\chi_S(s) = 0$, and therefore $\varphi_0(s)$ holds iff s does definitely not belong to the represented set.
- $\varphi_0(s) = \varphi_1(s) = 0$ implies $\chi_S(s) = \bot$, which means that we have no information about the membership of s.

In the following, we will show how to employ dual-rail encoding for the implementation of efficient bounded model checking procedures.

[3] The truth tables are the same as in other applications of three-valued logic like causality analysis [66]. However, they may use other partial orders, where $\bot$ is the least element.

[4] For fixpoint formulas $\mu Z.\varphi$ or $\nu Z.\varphi$, this function is defined as $\gamma(Q) := [\![\varphi]\!]_{\mathcal{K}_Z^Q}$.

4.2. Bounded global model checking

Global model checking of μ-calculus properties is essentially performed by fixpoint iteration as described in the previous subsection. To make use of the Tarski-Knaster theorem in bounded model checking, let us consider the following definition of a syntactical representation of fixpoint iteration:

Definition 6 (Fixpoint approximation). Let $[\varphi]_Z^{\psi}$ be the formula obtained by replacing all occurrences of Z in φ with ψ. Given a fixpoint formula $\sigma Z.\varphi$ with $\sigma \in \{\mu, \nu\}$, its k-th approximation $\mathsf{apx}_k(\sigma Z.\varphi)$ is recursively defined as follows:

$$\begin{array}{ll} \mathsf{apx}_0(\mu Z.\varphi) := \mathsf{false} & \mathsf{apx}_{k+1}(\mu Z.\varphi) := [\varphi]_Z^{\mathsf{apx}_k(\mu Z.\varphi)} \\ \mathsf{apx}_0(\nu Z.\varphi) := \mathsf{true} & \mathsf{apx}_{k+1}(\nu Z.\varphi) := [\varphi]_Z^{\mathsf{apx}_k(\nu Z.\varphi)} \end{array}$$

In global model checking (GMC), the set of satisfying states of a formula $\sigma Z.\varphi$ is obtained by computing a sequence of fixpoint approximations $[\![\mathsf{apx}_i(\sigma Z.\varphi)]\!]_{\mathcal{K}}$ until $[\![\mathsf{apx}_i(\sigma Z.\varphi)]\!]_{\mathcal{K}} = [\![\mathsf{apx}_{i+1}(\sigma Z.\varphi)]\!]_{\mathcal{K}}$ holds. Since by definition $\mathcal{K} \models \varphi$ iff $\mathcal{I} \subseteq [\![\varphi]\!]_{\mathcal{K}}$ holds, and every sequence of fixpoint approximations is increasing for least fixpoints and decreasing for greatest fixpoints, the following implications hold for all $k > 0$:

$$\mathcal{I} \subseteq [\![\mathsf{apx}_k(\mu Z.\varphi)]\!]_{\mathcal{K}} \Rightarrow \mathcal{K} \models \mu Z.\varphi \qquad \mathcal{I} \not\subseteq [\![\mathsf{apx}_k(\nu Z.\varphi)]\!]_{\mathcal{K}} \Rightarrow \mathcal{K} \not\models \nu Z.\varphi$$

Hence, estimating a bound k and checking the above conditions is sufficient to prove least fixpoint formulas and to disprove greatest fixpoint formulas. Though necessary, this is *not* sufficient to disprove least fixpoint formulas and to prove greatest fixpoint formulas. This is only sufficient if we know that increasing the bound does not further increase or decrease the set of computed states, i.e., if a fixpoint has been reached. Due to the monotonicity of the fixpoint approximations, checking whether a fixpoint has been reached can be reduced to checking whether $[\![\mathsf{apx}_k(\nu Z.\varphi)]\!]_{\mathcal{K}} \subseteq [\![\mathsf{apx}_{k+1}(\nu Z.\varphi)]\!]_{\mathcal{K}}$ and $[\![\mathsf{apx}_k(\mu Z.\varphi)]\!]_{\mathcal{K}} \supseteq [\![\mathsf{apx}_{k+1}(\mu Z.\varphi)]\!]_{\mathcal{K}}$ holds for least and greatest fixpoints, respectively.

Instead of computing each fixpoint approximation one after the other as in GMC, one can also syntactically unwind a fixpoint formula a bounded number of times to obtain a formula without fixpoint operators. Having symbolic representations of the initial states $\mathcal{I}$ and the transition relation $\mathcal{R}$, the algorithm[5] shown in Fig. 2 translates a model checking problem $\mathcal{K} \models \varphi$ with $\varphi \in \mathcal{L}^{\mu}$ to a formula (φ_1, φ_0) in dual-rail encoding representing those states where the specification holds. According to the meaning of three-valued characteristic functions we obtain the following propositions:

- if $s \models \varphi_1$ holds, then $\mathcal{K}, s \models \Phi$ holds
- if $s \models \varphi_0$ holds, then $\mathcal{K}, s \not\models \Phi$ holds
- if $s \not\models \varphi_1$ and $s \not\models \varphi_0$ holds, then we neither know $\mathcal{K}, s \models \Phi$ nor $\mathcal{K}, s \not\models \Phi$

[5] We assume that the substitution operation used in our algorithms renames bound variables such that no free occurrence of a variable is turned into a bound occurrence. Renaming of bound variables can be accomplished with only two sets of variables [38, 39, 52, 65, 81]. However, if the formula has to be brought to (cleansed) prenex normal form for checking satisfiabilty, additional variables have to be introduced to resolve multiple bindings.

```
function BGMC(k, Φ, φ)
    return ∀̇x. ϵ(Φ)→̇Unwind(k, φ);
end;

function Unwind(k, φ)
    case φ of
        L :         return ϵ(φ);
        ψ1 ∧ ψ2 :   return Unwind(k, ψ1)∧̇Unwind(k, ψ2);
        ψ1 ∨ ψ2 :   return Unwind(k, ψ1)∨̇Unwind(k, ψ2);
        □ψ :        return ∀̇x'.ϵ(R)→̇[Unwind(k, ψ)]_x^x';
        ◇ψ :        return ∃̇x'.ϵ(R)∧̇[Unwind(k, ψ)]_x^x';
        ←□ψ :       return [∀̇x.ϵ(R)→̇Unwind(k, ψ)]_x'^x;
        ←◇ψ :       return [∃̇x.ϵ(R)∧̇Unwind(k, ψ)]_x'^x;
        μZ.ψ :      Ψ := Unwind(k, ψ);
                    Φ_new := ϵ(0);
                    for i := 0 to k do
                        Φ_old := Φ_new;
                        Φ_new := [Ψ]_Z^Φ_old;
                    end;
                    Γ_fix := ∀̇x.Φ_new→̇Φ_old;
                    return (Γ_fix∨̇ϵ(⊥))→̇Φ_new;
        νZ.ψ :      Ψ := Unwind(k, ψ);
                    Φ_new := ϵ(1);
                    for i := 0 to k do
                        Φ_old := Φ_new;
                        Φ_new := [Ψ]_Z^Φ_old;
                    end;
                    Γ_fix := ∀̇x.Φ_old→̇Φ_new;
                    return (Γ_fix∨̇ϵ(⊥))∧̇Φ_new;
        Z :         return ϵ(Z);
    end;
end;
```

Fig. 2 Bounded global model checking

In other words, the algorithm computes an underapproximation φ_1 and an overapproximation $\neg\varphi_0$ of $[\![\varphi]\!]_{\mathcal{K}}$. We may call φ_1 the 'must' set and $\neg\varphi_0$ the 'cannot' set, in accordance to other three-valued analyses.

The correctness of the algorithm is proved as follows: For $k \to \infty$, the algorithm follows global model checking procedures for the μ-calculus, i.e., modal operators are evaluated by computing predecessor or successor states, and fixpoint operators are evaluated by Tarski-Knaster fixpoint iteration. The only difference is that these computations are performed on pairs of formulas by means of dual-rail encoding. For fixpoint-free formulas, the returned pairs (φ_1, φ_0) are always complementary, i.e., we have $\varphi_0 \leftrightarrow \neg\varphi_1$. Hence, these pairs represent precisely known state sets.

The third truth value $\bot$ is introduced after unrolling fixpoint formulas. At this stage, we first check whether the fixpoint has already been reached by the k-th approximation. This is accomplished by checking the validity of the formula Γ_{fix}. Since this formula is closed, it is equivalent to one of the constant values $\mathsf{0}$, $\bot$, and $\mathsf{1}$. As a consequence, $\Gamma_{\mathsf{fix}} \dot{\vee} \epsilon(\bot)$ is

equivalent to one of the constant values $\mathsf{1}$ or $\bot$, depending on whether the fixpoint has been reached or not.

Now consider the formula returned by unwinding a least fixpoint formula $\mu Z.\psi$. In case the fixpoint has been reached, the formula $(\Gamma_{\mathsf{fix}} \dot{\vee} \epsilon(\bot)) \dot{\rightarrow} \Phi_{\mathsf{new}}$ reduces to Φ_{new}, otherwise to $\epsilon(\bot) \dot{\rightarrow} \Phi_{\mathsf{new}}$. This is explained as follows: For least fixpoints, the Tarski–Knaster iteration starts with the empty set encoded by $\epsilon(\mathsf{0})$ and iteratively adds states to the current approximation. If the fixpoint has been reached, we know that exactly those states contained in the final approximation belong to the fixpoint. However, if the fixpoint has not been reached, we only know that the states of the last approximation belong to the fixpoint, while we have no information about the remaining states. In this case, we modify the characteristic function by replacing $\mathsf{0}$ with $\bot$. An analogous argumentation holds for greatest fixpoint formulas.

To verify that the returned formulas implement the required modifications, consider the following reduced truth table:

Γ_{fix}	Φ_{new}	$(\Gamma_{\mathsf{fix}} \dot{\vee} \epsilon(\bot)) \dot{\rightarrow} \Phi_{\mathsf{new}}$	$(\Gamma_{\mathsf{fix}} \dot{\vee} \epsilon(\bot)) \dot{\wedge} \Phi_{\mathsf{new}}$
$\mathsf{1}$	$*$	Φ_{new}	Φ_{new}
$\bot, \mathsf{0}$	$\mathsf{0}$	$\bot$	$\mathsf{0}$
$\bot, \mathsf{0}$	$\bot$	$\bot$	$\bot$
$\bot, \mathsf{0}$	$\mathsf{1}$	$\mathsf{1}$	$\bot$

We return the full information Φ_{new} in case a fixpoint has been reached. Otherwise, we return $\mathsf{1}$ if the approximation of a least fixpoint returns $\mathsf{1}$, and $\bot$ in all other cases. Due to the monotonicity laws, we know that once a state is included in the k-th approximation of a least fixpoint, it will belong to all n-th approximations with $n \geq k$. The converse, however, does not hold. For greatest fixpoints, we return $\mathsf{0}$ if its approximation returns $\mathsf{0}$, and $\bot$ in all other cases. Analogously, once a state is excluded in the k-th approximation, it will be excluded from all n-th approximations with $n \geq k$.

Note that the formula Γ_{fix} appears in only one of the two rails. Given that $\Gamma_{\mathsf{fix}} \equiv (\Gamma_{\mathsf{fix}}^{(1)}, \Gamma_{\mathsf{fix}}^{(0)})$ and $\Phi_{\mathsf{new}} \equiv (\Phi_{\mathsf{new}}^{(1)}, \Phi_{\mathsf{new}}^{(0)})$ holds, expanding the dual-rail formulas yields:

- $(\Gamma_{\mathsf{fix}} \dot{\vee} \epsilon(\bot)) \dot{\rightarrow} \Phi_{\mathsf{new}} \equiv (\Phi_{\mathsf{new}}^{(1)}, \Gamma_{\mathsf{fix}}^{(1)} \wedge \Phi_{\mathsf{new}}^{(0)})$
- $(\Gamma_{\mathsf{fix}} \dot{\vee} \epsilon(\bot)) \dot{\wedge} \Phi_{\mathsf{new}} \equiv (\Gamma_{\mathsf{fix}}^{(1)} \wedge \Phi_{\mathsf{new}}^{(1)}, \Phi_{\mathsf{new}}^{(0)})$

Having computed an approximation $\mathsf{Unwind}(k, \varphi) \equiv (\varphi_1, \varphi_0)$, which contains an underapproximation φ_1 and an overapproximation $\neg\varphi_0$ of the desired state set, it is finally checked whether the initial states imply the unwound formula. Thus, the call $\mathsf{BGMC}(k, \Phi, \varphi)$ returns the pair $((\forall \mathbf{x}.\Phi \rightarrow \varphi_1), (\exists \mathbf{x}.\ \Phi \wedge \varphi_0))$. If all initial states belong to φ_1, we know that the property holds, and therefore the result is $\mathsf{1}$. If one of the initial states belongs to φ_0, we know that the property is definitely false, and therefore the result is $\mathsf{0}$. Otherwise, we cannot say anything about the truth, and therefore, we return $\bot$. This leads us to the following theorem:

Theorem 5 *(Bounded global model checking). Let* $\mathbf{x} = (x_1, \ldots, x_n)$ *and* $\mathbf{x}' = (x_1', \ldots, x_n')$ *be vectors of variables with* $x_i \in \mathcal{V}$ *and* $x_i' \in \mathcal{V}'$. *Moreover, let* $\mathcal{I} \in \mathcal{L}$ *be a formula over the variables* $\mathbf{x}$ *and* $\mathcal{R} \in \mathcal{L}$ *a formula over the variables* $\mathbf{x}$ *and* $\mathbf{x}'$ *such that* $\mathcal{I}$ *represents the initial states and* $\mathcal{R}$ *the transition relation of a Kripke structure* $\mathcal{K}$. *Then, for all formulas* $\varphi \in \mathcal{L}^{\mu}$ *and all* $k \in \mathbb{N}$ *with* $(\varphi_1, \varphi_0) = \mathsf{BGMC}(k, \mathcal{I}, \varphi)$, *we have:*

- *if* φ_1 *is true, then* φ_0 *is false and* $\mathcal{K} \models \varphi$
- *if* φ_0 *is true, then* φ_1 *is false and* $\mathcal{K} \not\models \varphi$
- *if neither* φ_1 *nor* φ_0 *is true, then we know nothing about the truth of* $\mathcal{K} \models \varphi$

$$(1)\ \frac{\Phi \vdash \varphi \wedge \psi}{\Phi \vdash \varphi \wedge \Phi \vdash \psi} \qquad (2)\ \frac{\Phi \vdash \varphi \vee \psi}{\exists \Psi_1, \Psi_2.\ \Psi_1 \vdash \varphi \wedge \Psi_2 \vdash \psi \wedge \Phi \subseteq \Psi_1 \cup \Psi_2}$$

$$(3)\ \frac{\Phi \vdash \Box \varphi}{\mathsf{suc}_{\exists}^{\mathcal{R}}(\Phi) \vdash \varphi} \qquad (4)\ \frac{\Phi \vdash \Diamond \varphi}{\exists \Psi.\ \Psi \vdash \varphi \wedge \Phi \subseteq \mathsf{pre}_{\exists}^{\mathcal{R}}(\Psi)}$$

$$(5)\ \frac{\Phi \vdash \overleftarrow{\Box} \varphi}{\mathsf{pre}_{\exists}^{\mathcal{R}}(\Phi) \vdash \varphi} \qquad (6)\ \frac{\Phi \vdash \overleftarrow{\Diamond} \varphi}{\exists \Psi.\ \Psi \vdash \varphi \wedge \Phi \subseteq \mathsf{suc}_{\exists}^{\mathcal{R}}(\Psi)}$$

$$(7)\ \frac{\Phi \vdash Z}{\Phi \vdash \Delta(Z)} \qquad (8)\ \frac{\Phi \vdash \sigma Z.\varphi}{\Phi \vdash [\varphi]_Z^{Z'}} \quad \Delta := \Delta \cup \{(Z', \sigma Z.\varphi)\}$$

$$(9)\ \frac{\Phi \vdash \varphi}{\Phi \subseteq [\![\varphi]\!]_{\mathcal{K}}} \quad \varphi \in \mathcal{L} \qquad (10)\ \frac{\Phi \vdash \varphi}{\exists \Psi.\ \Psi \vdash \varphi \wedge \Phi \subseteq \Psi}$$

Fig. 3 Decomposition rules for local model checking

As an example, let $\mathcal{I} :\equiv x = 0$ and $\mathcal{R} :\equiv x' = x + 1 \wedge x \geq 0$ represent the set of initial states and the transition relation of a Kripke structure $\mathcal{K}$, respectively. For the function call $\mathsf{BGMC}(1, \mathcal{I}, \varphi)$ with $\varphi :\equiv \nu Z.\, x \geq 0 \wedge \Box Z$ we obtain the following formulas after unwinding φ, i.e., after the loop:

$$\begin{aligned} \Phi_{\mathsf{old}} &= \epsilon(x \geq 0)\, \dot{\wedge}\, \epsilon(\mathsf{1}) \\ \Phi_{\mathsf{new}} &= \epsilon(x \geq 0)\, \dot{\wedge}\, (\dot{\forall} x'.\epsilon(x' = x + 1 \wedge x \geq 0) \dot{\rightarrow} \epsilon(x' \geq 0) \dot{\wedge} \epsilon(\mathsf{1})) \end{aligned}$$

To keep the formulas small, we assume that $\Gamma_{\mathsf{fix}} \equiv (\dot{\forall} x.\Phi_{\mathsf{old}} \dot{\rightarrow} \Phi_{\mathsf{new}}) \dot{\vee} \epsilon(\bot))$ has already been evaluated. Since Γ_{fix} is valid in our example, we finally obtain the formula

$$\dot{\forall} x.\epsilon(x = 0) \dot{\rightarrow} (\epsilon(x \geq 0)\, \dot{\wedge}\, (\dot{\forall} x'.\epsilon(x' = x + 1 \wedge x \geq 0) \dot{\rightarrow} \epsilon(x' \geq 0) \dot{\wedge} \epsilon(\mathsf{1}))).$$

Since this formula is true, i.e., equivalent to $(\mathsf{1}, \mathsf{0})$, we conclude that the specification holds.

4.3. Bounded local model checking

In this subsection, we present a *bounded local model checking* (BLMC) procedure. However, before we go into detail, we explain the foundations of unbounded LMC. In contrast to GMC, LMC [11, 12, 29, 74, 75] aims at directly answering the question whether $\mathcal{I} \subseteq [\![\varphi]\!]_{\mathcal{K}}$ holds. To this end, proof goals of the form $\Phi \vdash \varphi$ are considered where Φ is a set of states and φ is a $\mathcal{L}^{\mu}$ formula. The meaning of a goal $\Phi \vdash \varphi$ is that it has to be proved that $\Phi \subseteq [\![\varphi]\!]_{\mathcal{K}}$ holds. In the following, we use formulas $\Phi \in \mathcal{L}$ over the free variables $\mathbf{x} = (x_1, \ldots, x_n)$ as symbolic representations of state sets. Proofs of such goals are obtained by syntax-directed decomposition into subgoals using the rules shown in Fig. 3. The propositions above and below a line are equivalent for all rules.

Rule (1) simply splits a conjunction into two subgoals. When considering rules (3) and (5) one might be surprised that the universal modal operators are reduced to computing the existential successor or predecessor states. This is due to the fact that after rewriting the expanded formulas, the universal quantifiers turn into existential ones. For example, rule (3)

is proved as follows:

$$\begin{aligned}
[\![\Phi]\!]_{\mathcal{K}} \subseteq [\![\Box\varphi]\!]_{\mathcal{K}} &\Leftrightarrow \forall s.\ s \in [\![\Phi]\!]_{\mathcal{K}} \Rightarrow s \in [\![\Box\varphi]\!]_{\mathcal{K}} \\
&\Leftrightarrow \forall s.\ s \in [\![\Phi]\!]_{\mathcal{K}} \Rightarrow (\forall s'.(s, s') \in \mathcal{R} \Rightarrow s' \in [\![\varphi]\!]_{\mathcal{K}}) \\
&\Leftrightarrow \forall s, s'.\ s \in [\![\Phi]\!]_{\mathcal{K}} \wedge (s, s') \in \mathcal{R} \Rightarrow s' \in [\![\varphi]\!]_{\mathcal{K}} \\
&\Leftrightarrow \forall s'.\ (\exists s.s \in [\![\Phi]\!]_{\mathcal{K}} \wedge (s, s') \in \mathcal{R}) \Rightarrow s' \in [\![\varphi]\!]_{\mathcal{K}} \\
&\Leftrightarrow \forall s'.\ s' \in \mathsf{suc}_{\exists}^{\mathcal{R}}(\Phi) \Rightarrow s' \in [\![\varphi]\!]_{\mathcal{K}} \\
&\Leftrightarrow \mathsf{suc}_{\exists}^{\mathcal{R}}(\Phi) \subseteq [\![\varphi]\!]_{\mathcal{K}}
\end{aligned}$$

Rule (2) is explained as follows: If $\Phi \vdash \varphi \vee \psi$ holds, then the set of states encoded by Φ can be partitioned into the set of states Ψ_1 that satisfy φ and the set of states Ψ_2 that satisfy ψ. The application of this rule requires that the user guesses suitable sets Ψ_1, Ψ_2 such that the goal $\Phi \vdash \varphi \vee \psi$ can be decomposed into provable subgoals. In a similar way, rules (4) and (6) that implement the semantics of existential modal operators, require to have a suitable set of states Ψ. Rules that reduce a proposition to an existentially quantified one are often called *choice* rules. Their application requires that the user provides suitable witnesses, so that the remaining proof succeeds.

Before we continue with the explanation of the rules, let us briefly discuss the meaning of choice rules and their impact on local model checking algorithms. At first view, the existence of these rules seems to be a severe drawback of local model checking, since they require user interaction as opposed to global model checking. In fact, these rules are inevitable due to the fact that local model checking is complete [11], but undecidable for the considered transition systems. In contrast, global model checking is not complete without additional techniques such as fixpoint induction, which in turn require user interaction. Thus, the choice rules should not be viewed as a burden, but as a feature for semi-automatic model checking procedures. Nevertheless, we will later present improved rules for local model checking that do not require user interaction for many specifications occurring in practice.

Rules (7) and (8) are used for unwinding fixpoint formulas. To this end, the procedure maintains a set Δ that consists of pairs $(Z, \sigma Z.\varphi)$ that associate a bound variable Z with the subformula where it is bound.[6] Provided that $(Z', \sigma Z.\varphi) \in \Delta$ holds, we write $\Delta(Z')$ for the second component of this pair. For example, let $\Delta = \emptyset$ and $\Phi \vdash \mu Z.x = 0 \vee \Box Z$ be a goal to be decomposed. Then, by rule (8) we have $\Delta = \{(Z', \mu Z.x = 0 \vee \Box Z)\}$ and proceed with the goal $\Phi \vdash x = 0 \vee \Box Z'$. Once the decomposition process has reached the variable Z', the application of rule (7) regenerates the original formula and yields a goal $\Phi' \vdash \mu Z.x = 0 \vee \Box Z$ with a new set of states Φ'.

To avoid infinite recursion, it has to be checked whether the regeneration of a fixpoint formula leads to a previously created goal with the same query. In this case, the construction of the proof is successful for greatest fixpoints. For least fixpoint formulas, it has to be checked whether the corresponding path in the proof tree is well-founded [11, 12]. The reason for this is a deeply rooted property of the μ-calculus that intuitively states that least fixpoints are related to finite recursion, while greatest fixpoints additionally allow infinite recursion.

Rule (9) is applied to formulas of the base logic and terminates the decomposition process. Finally, rule (10) can be used to strengthen a goal $\Phi \vdash \varphi$ by finding an appropriate superset of Φ. Again, this rule requires user interaction to set up a suitable lemma $\Psi \vdash \varphi$.

[6] It is assumed that every variable is bound only once and has no additional free occurrences. Moreover, the formulas are given in guarded normal form [65], i.e., every occurrence of a bound variable Z in $\sigma Z.\varphi$ is guarded by a modal operator.

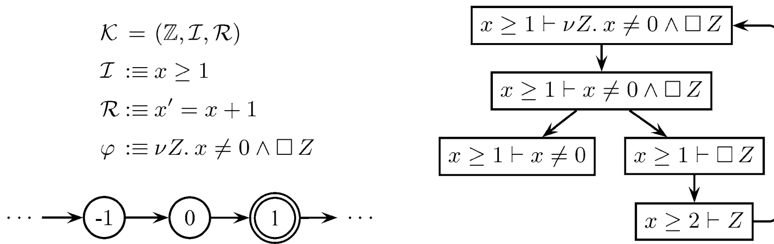


Fig. 4 Example for local model checking

Figure 4 gives an example for local model checking. The right-hand side shows a proof tree for the Kripke structure and the specification shown on the left-hand side. Interestingly, the given specification cannot be proved using GMC, since the fixpoint iteration does not terminate (the differences between local and global model checking regarding infinite state systems were considered in [69]).

As mentioned previously, the decomposition rules shown in Fig. 3 are sufficient to prove every valid specification using LMC [11, 12]. Unfortunately, this often requires user interaction which is due to the disjunctive nature of the choice rules. Additionally, the user needs to know when to apply rule (10) which can be really challenging. Note that this rule allows LMC to switch to a different proof goal that requires to guess the right *induction lemma*. However, in practice one is mainly interested in automatic proof procedures. For this reason, we restrict ourselves to the universal fragment of the μ-calculus that does not require the application of choice rules.[7] Although this may seem to be a hard restriction, the remaining fragment is still powerful enough to automatically prove properties that cannot be proved by GMC [69]. Additionally, we use the following techniques to automate the decomposition process:

(A) Regarding rule (2): A goal $\Phi \vdash \varphi \lor \psi$ with $\varphi \in \mathcal{L}$ is reduced to the goal $\Phi \land \neg\varphi \vdash \psi$ which is the least remaining requirement. In this way, we are able to check disjunctions automatically, provided that one of the disjuncts belongs to the base logic.[8] For example, this allows us to check the property $\mathsf{AF}x = 0$ which is translated to the μ-calculus formula $\mu Z.x = 0 \lor \Box Z$.

(B) Regarding rule (10): Checking repetition is not only successful if the same goal $\Phi \vdash \varphi$ appears, but also if a goal $\Psi \vdash \varphi$ appears such that $\Psi \subseteq \Phi$ holds. Recall that rule (10) allows us to replace the set Ψ with a superset in order to match a previous goal. In the following, this is referred to as *loop test*. The loop test is always performed before unwinding a greatest fixpoint formula.

(C) Least fixpoint formulas are always unwound until the bound is reached. In this way, many least fixpoint formulas can be verified without having to check well-foundedness of the proof tree, provided that the formula Φ will eventually be unsatisfiable. This is

[7] Existential properties can be proved by disproving the negated specification, provided that there are only finitely many initial states.

[8] The disjunct may also contain modal operators, but we restrict ourselves to the base logic for the sake of simplicity.

due to the fact that, by definition, a goal $\emptyset \vdash \varphi$ holds for all formulas φ. However, this requires that the specification can be decomposed in a finite number of steps.

Figures 5 and 6 show the algorithm[9] for BLMC. According to the rules of Fig. 3, it recursively decomposes a μ-calculus formula φ into subformulas and returns a pair of formulas using dual-rail encoding. The algorithm first checks whether the formula φ belongs to the base logic $\mathcal{L}$. If this is the case, rule (9) is applied and the algorithm returns a formula that represents the inclusion $\Phi \subseteq [\![\varphi]\!]_{\mathcal{K}}$. The translation of conjunctions follows directly from rule (1). By technique (A), disjunctions are decomposed if at least one of the subformulas belongs to the base logic. If this is not the case, the algorithm returns the unknown value $\perp$. Modal operators are decomposed according to rules (3) and (5) and the definition of predecessor/successor state sets.

Recall that rule (8) substitutes the variable bound by the fixpoint operator with a new variable to distinguish between different incarnations. The introduction of new variables can be avoided if we explicitly take into account scopes of variables. For that purpose, we use a stack[10] to keep track of unwound formulas. In addition, the stack is used to store the current state set and the current bound. An element of the stack is therefore a triple (Φ, Z, k), where Φ represents the state set, Z the fixpoint variable, and k the bound. In principle, one could also use the stack to store complete fixpoint formulas instead of only the associated fixpoint variables. However, while this eliminates the need for the set Δ, this comes at the cost of redundantly storing one and the same formula multiple times.

Let us now continue with the description of the decomposition process. If the algorithm encounters a goal $\Phi \vdash \sigma Z.\psi$, it first updates the set Δ. Then, it successively examines the stack from the top using function GetBound to determine the bound k' of the last incarnation of Z. After that, the algorithm pushes the triple $(\Phi, Z, k' - 1)$ on the stack and continues with the goal $\Phi \vdash \psi$. After termination of all recursive calls, the top-level element is removed from the stack and the resulting formula is returned.

If the algorithm encounters a fixpoint variable, the corresponding formula is unwound once more, provided that the bound has not yet been exceeded. Otherwise, the algorithm returns $\perp$ which means that the truth of the specification cannot be decided due to insufficient bounds. Additionally, a loop test is performed for greatest fixpoint formulas according to technique (B). This is accomplished by the function LoopTest which, similar to the function GetBound, recursively examines the stack to find a matching element. There are three cases to be distinguished: First, if the stack is empty, the function returns $\epsilon(0)$ which indicates that the loop test has failed. Second, if an element is found where the current fixpoint variable matches the one on top of the stack, the formula $\epsilon(\forall \mathbf{x}.\Phi \rightarrow \Phi')$ is returned. Recall that repetition is successful if $\Phi \subseteq \Phi'$, where Φ represents the current state set and Φ' the one of the previously generated goal. Third, it may happen that the top-level element of the stack refers to a different fixpoint formula. In this case, the scope has been left and the function returns $\epsilon(0)$. For least fixpoint formulas it is finally checked whether the set Φ is empty according to technique (C).

Theorem 6 *(Bounded local model checking). Let* $\mathbf{x} = (x_1, \ldots, x_n)$ *and* $\mathbf{x}' = (x'_1, \ldots, x'_n)$ *be vectors of variables with* $x_i \in \mathcal{V}$ *and* $x'_i \in \mathcal{V}'$*. Moreover, let* $\mathcal{I} \in \mathcal{L}$ *be a formula over the*

[9] Underscores denote anonymous variables for pattern matching.

[10] We assume the operations **push**, **pop**, and **top** to push an element on the top of the stack (**push**), to remove the top element of the stack (**pop**), and to read the topmost element of the stack (**top**). Moreover, empty denotes the empty stack.

```
function BLMC(k, Φ, φ)
   case φ of
      L :         return ε(∀x. Φ → φ);
      ψ1 ∧ ψ2 :   return BLMC(k, Φ, ψ1) ∧̇ BLMC(k, Φ, ψ2);
      ψ1 ∨ ψ2 :   if ψ1 ∈ L then
                     return BLMC(k, Φ ∧ ¬ψ1, ψ2);
                  end;
                  if ψ2 ∈ L then
                     return BLMC(k, Φ ∧ ¬ψ2, ψ1);
                  end;
                  return ε(⊥);
      □ ψ :       return BLMC(k, [∃x.R ∧ Φ]^x_x', ψ);
      □⃖ ψ :       return BLMC(k, ∃x'.R ∧ [Φ]^x'_x, ψ);
      σZ.ψ :      Δ := Δ ∪ (Z, φ);
                  k' := GetBound(k, Z, Stack);
                  Stack := push((Φ, Z, k' − 1), Stack);
                  φ' := BLMC(k, Φ, ψ);
                  Stack := pop(Stack);
                  return φ';
      Z :         if GetBound(k, Z, Stack) > 0 then
                     φ' := BLMC(k, Φ, Δ(Z));
                  else
                     φ' := ε(⊥);
                  end;
                  if Δ(Z) = νZ._ then
                     φ' := φ' ∨̇ LoopTest(Φ, Z, Stack);
                  else
                     φ' := φ' ∨̇ ε(∀x.¬Φ);
                  end;
                  return φ';
   end;
end;
```

Fig. 5 Bounded local model checking

```
function GetBound(k, Z, S)
   if S = empty then
      return k;
   end;
   (_, Z', k') := top(S);
   if Z = Z' then
      return k';
   end;
   if Z' ∈ FreeVariables(Δ(Z)) then
      return k;
   end;
   return GetBound(k, Z, pop(S));
end;
```

```
function LoopTest(Φ, Z, S)
   if S = empty then
      return ε(0);
   end;
   (Φ', Z', _) := top(S);
   if Z = Z' then
      return ε(∀x.Φ → Φ');
   end;
   if Z' ∈ FreeVariables(Δ(Z)) then
      return ε(0);
   end;
   return LoopTest(Φ, Z, pop(S));
end;
```

Fig. 6 Bounded local model checking (cont.)

variables $\mathbf{x}$ *and* $\mathcal{R} \in \mathcal{L}$ *a formula over the variables* $\mathbf{x}$ *and* $\mathbf{x}'$ *such that* $\mathcal{I}$ *represents the initial states and* $\mathcal{R}$ *the transition relation of a Kripke structure* $\mathcal{K}$*. Then, for all formulas* $\varphi \in \mathcal{L}^{\mu}$ *and all* $k \in \mathbb{N}$ *with* $(\varphi_1, \varphi_0) = \mathsf{BLMC}(k, \mathcal{I}, \varphi)$*, we have:*

- *if* φ_1 *is true, then* φ_0 *is false and* $\mathcal{K} \models \varphi$
- *if* φ_0 *is true, then* φ_1 *is false and* $\mathcal{K} \not\models \varphi$
- *if neither* φ_1 *nor* φ_0 *is true, then we know nothing about the truth of* $\mathcal{K} \models \varphi$

Proof: Algorithm BLMC implements local model checking for the universal fragment of $\mathcal{L}^{\mu}$ when $k \to \infty$. Since the construction of the proof tree does not require interactive rules, a goal $\mathcal{I} \vdash \varphi$ is true iff all leaf vertices of the proof tree are true. Due to the correctness of the proof tree construction of local model checking, it follows that the goal of the root node is valid if all goals in the leaves are valid. The converse may not hold due to the restriction of the bound k. However, if a leaf vertex cannot be extended even if k would be increased, the non-validity of a leaf vertex implies that the root goal $\mathcal{I} \vdash \varphi$ is false, since there is no other way to construct a proof tree due to the absence of choice rules. Note that the result of such a leaf vertex definitely returns a Boolean value, and that $\perp$ is only returned in case a fixpoint formula could not be decided for the bound k. $\square$

As an example, reconsider Fig. 4 where we used the formulas $\mathcal{I} :\equiv x \geq 1$ and $\mathcal{R} :\equiv x' = x + 1$ to symbolically represent the initial states and the transition relation of the Kripke structure $\mathcal{K}$. Figure 7 shows a trace (entry and exit points) for the call $\mathsf{BLMC}(1, \mathcal{I}, \varphi)$ with $\varphi :\equiv \nu Z.\, x \neq 0 \wedge \Box Z$. After the first call we have $\Delta = \{(Z, \nu Z.x \neq 0 \wedge \Box Z)\}$ and $\mathbf{top}(Stack) = (x \geq 1, Z, 0)$. Expanding the resulting dual-rail formula yields a pair (φ_1, φ_0) with

$$\begin{aligned}
\varphi_1 &= (\forall x.x \geq 1 \rightarrow x \neq 0) \wedge (0 \vee \forall x.[\exists x.\mathcal{R} \wedge x \geq 1]^{x}_{x'} \rightarrow x \geq 1) \\
&\equiv (\forall x.x \geq 1 \rightarrow x \neq 0) \wedge (\forall x.(\exists x'.x = x' + 1 \wedge x' \geq 1) \rightarrow x \geq 1) \\
&\equiv \forall x.\forall x'.(x \geq 1 \rightarrow x \neq 0) \wedge (x \neq x' + 1 \vee x' < 1 \vee x \geq 1)
\end{aligned}$$

and

$$\begin{aligned}
\varphi_0 &= \neg(\forall x.x \geq 1 \rightarrow x \neq 0) \vee (0 \wedge \neg\forall x.[\exists x.\mathcal{R} \wedge x \geq 1]^{x}_{x'} \rightarrow x \geq 1) \\
&\equiv \neg(\forall x.x \geq 1 \rightarrow x \neq 0) \\
&\equiv \exists x.x \geq 1 \wedge x = 0.
\end{aligned}$$

As φ_1 is valid, it follows that $\mathcal{I} \vdash \varphi$ is true and $\mathcal{K} \models \varphi$ holds.

Having reduced the model checking problem to a satisfiability problem, we can employ the same backend tools to solve the final problem as for BGMC. Needless to say that these tools should first perform some precomputations. For example, checking the validity of a formula $\varphi \wedge \psi$ can be done by checking whether φ is valid and ψ is valid. The validity of a formula $\varphi \vee \psi$ can be checked analogously, provided that φ and ψ do not share free variables. Moreover, conjunctions can be checked lazily, i.e., evaluating the first subformula may already yield the resulting truth value, so that the second subformula needs not to be evaluated.

$\mathsf{BLMC}(1, x \geq 1, \nu Z.x \neq 0 \wedge \Box Z)$
$\quad\mathsf{BLMC}(1, x \geq 1, x \neq 0 \wedge \Box Z)$
$\quad\quad\mathsf{BLMC}(1, x \geq 1, x \neq 0)$
$\quad\quad$**return** $\epsilon(\forall x.x \geq 1 \rightarrow x \neq 0)$
$\quad\quad\mathsf{BLMC}(1, x \geq 1, \Box Z)$
$\quad\quad\quad\mathsf{BLMC}(1, [\exists x.\mathcal{R} \wedge x \geq 1]^x_{x'}, Z)$
$\quad\quad\quad$**return** $\epsilon(\bot) \vee \epsilon(\forall x.[\exists x.\mathcal{R} \wedge x \geq 1]^x_{x'} \rightarrow x \geq 1)$
$\quad\quad$**return** $\epsilon(\bot) \vee \epsilon(\forall x.[\exists x.\mathcal{R} \wedge x \geq 1]^x_{x'} \rightarrow x \geq 1)$
$\quad$**return** $\epsilon(\forall x.x \geq 1 \rightarrow x \neq 0) \wedge (\epsilon(\bot) \vee \epsilon(\forall x.[\exists x.\mathcal{R} \wedge x \geq 1]^x_{x'} \rightarrow x \geq 1))$
return $\epsilon(\forall x.x \geq 1 \rightarrow x \neq 0) \wedge (\epsilon(\bot) \vee \epsilon(\forall x.[\exists x.\mathcal{R} \wedge x \geq 1]^x_{x'} \rightarrow x \geq 1))$

Fig. 7 Example for bounded local model checking

5. Experimental results

The algorithms presented in this paper have been implemented in Averest,[11] a framework for the design and verification of reactive systems. Averest consists of a compiler for our synchronous language Quartz [61, 62, 64], a symbolic model checker, and a tool for hardware/software synthesis. The benchmarks have been implemented in Quartz and were compiled to symbolically encoded transition systems. The resulting transition systems were then verified using our symbolic model checker Beryl.

As the base logic $\mathcal{L}$, we use Presburger arithmetic which is a decidable first-order predicate logic over the integers with addition as the basic operation [33, 55, 58]. The use of Presburger arithmetic in symbolic model checking was first proposed by Bultan, Gerber, and Pugh [16, 17]. Around the same time, Kukula, Shiple, and Aziz presented a technique for reachability analysis of extended finite state machines using Presburger arithmetic [43]. A comparison of Presburger decision procedures can be found in [72] and [35].

As mentioned in the introduction, an important aspect concerning the implementation of efficient decision procedures is that every Presburger formula can be translated to a finite automaton that encodes its models [9, 10, 21, 85]. Since there exists for every finite automaton an equivalent minimal one, automata can serve as a canonical representation for Presburger formulas. This is analogous to the use of binary decision diagrams as a canonical normal form for propositional logic [13]. Hence, automata can be viewed as generalizations of BDDs for representing infinite sets.

Our implementation can translate Presburger formulas to deterministic finite automata (DFAs) and alternating finite automata (AFAs). For both types, we use a semi-symbolic encoding where the states are represented explicitly, and the transitions implicitly by means of propositional logic. To this end, we use the CUDD BDD package [73] for DFAs and the SAT solver zChaff [53] for AFAs. The strengths and weaknesses of DFAs and AFAs resemble those of BDDs and SAT solvers, respectively. DFAs are thus best suited for unbounded model checking (checking equivalence of two DFAs is easy). In contrast, AFAs are best suited for bounded model checking where satisfiability (emptiness) has to checked only once. Consequently, we use DFAs for GMC/LMC and AFAs for BGMC/BLMC .

For BGMC and BLMC, we restricted ourselves to formulas without quantifier alternations since quantification is a hard operation on AFAs. Again, the situation is similar to finite

[11] `http://www.averest.org`

state bounded model checking that is usually based on the construction of quantifier-free propositional formulas. Unfortunately, this restriction does not allow us to perform the loop test in BLMC. For this reason, the following results were obtained without this test.

The results are shown in Tables 1 and 2. All experiments were performed on a Xeon processor with 3 GHz an 1 GB RAM. A dash indicates that a benchmark could not be checked within the limit of 1000 seconds. The first column gives the name of the benchmark and the second one its size (all benchmarks are scalable by the size of the used data structures or the number of processes). The third column shows the time required for the construction of the transition relation. This is only of interest for unbounded model checking, since the results for BGMC and BLMC include the times required for the construction of the formulas to be checked.

The first two benchmarks implement standard algorithms for searching an element in array [30]. The benchmark *ParallelSearch* is also a search algorithm that uses two parallel processes [56]. The next three benchmarks implement different sorting algorithms and a sorting network [30]. The benchmarks *MinMax* and *FastMax* are efficient algorithms for computing the minimum (maximum) element in an array [30]. The latter is particularly interesting, since it has constant runtime and consists of quadratically many processes w.r.t. the size of the array. *Partition* rearranges the elements of an array according to a given pivot element [30], and *ParallelPrefixSum* computes the sums of all prefixes in an array using multiple processes [3]. Finally, the *Bakery* protocol implements a mutual exclusion algorithm, and *Barber* rendezvous-like synchronization [3].

For most benchmarks we proved a liveness property that states termination, and we disproved a safety property that specifies the functional correctness (the benchmarks were slightly modified by inserting typical errors). The runtimes for GMC and LMC for both types of specifications are shown in columns 4–7. The remaining columns give the results for BGMC and BLMC, where k denotes the minimal bound required to decide the specification, *Size* the size of the resulting formula measured in the number of function symbols, and *Time* the time required to check the formula obtained for bound k. The minimal bounds were obtained by starting with bound 1 and successively increasing the current bound until the specifications can be decided. Finally, *Total* gives the runtime required for all bounds less than or equal to k.

Let us first consider the results shown in Table 1. For the first benchmark (*LinearSearch*), the liveness property is most efficiently checked using LMC. For the safety property, however, BGMC and BLMC are much faster. In particular, both methods scale better as compared to GMC and LMC. For *BinarySearch*, even the liveness property is most efficiently checked using BGMC or BLMC. For *ParallelSearch*, GMC and LMC additionally suffer from significantly increasing runtimes required for the construction of the transition relation. Note that for all three benchmarks the minimal bounds depend linearly or logarithmically on the size.

The situation is converse for the sorting algorithms. Neither BGMC nor BLMC were able to compete with the unbounded variants. Even worse, their runtimes exceeded the limit except for the smallest instances. This is primarily due to the fact that the minimal bounds required to check a specification are rather large. Note that for these benchmarks the bounds depend quadratically on the size. The next benchmark implements a sorting network with a sub-linear number of comparator stages. As can be seen, BGMC and BLMC outperform GMC and LMC.

Similar effects are observed for the benchmarks given in Table 2. For the sake of brevity, we only like to mention that the best results are achieved for *FastMax* and *Barber*. This is not surprising since the specifications could be decided for rather small bounds. To sum up, bounded model checking of infinite state systems is clearly superior if a counterexample

Table 1 Benchmark results

Name	Size	Trans	GMC		LMC		BGMC								BLMC							
							Liveness				Safety				Liveness				Safety			
			Live	Safe	Live	Safe	*k*	Size	Time	Total	*k*	Size	Time	Total	*k*	Size	Time	Total	*k*	Size	Time	Total
LinearSearch	4	0.1	0.0	0.2	0.0	0.1	3	629	0.3	0.4	3	659	0.1	0.3	2	629	0.3	0.3	2	1788	0.1	0.4
	5	0.1	0.1	5.4	0.1	0.3	4	1167	0.7	1.0	4	1211	0.3	1.3	3	1167	0.7	1.0	3	3387	0.4	2.1
	6	0.1	0.1	86.9	0.1	1.8	5	1472	1.3	1.9	5	1537	0.5	2.6	4	1472	1.2	1.8	4	4399	0.8	4.3
	7	0.1	0.3	898.4	0.2	11.1	6	1813	1.6	2.6	6	1899	0.7	5.0	5	1813	1.6	2.6	5	5523	0.9	7.3
	8	0.2	1.1	–	0.4	107.7	7	2276	2.2	4.2	7	2373	1.1	7.7	6	2276	2.5	4.3	6	6963	1.8	13.6
BinarySearch	4	0.3	0.1	1.4	0.1	0.1	4	1462	0.8	1.2	3	1249	0.2	0.5	3	1461	0.7	1.1	2	2982	0.3	0.6
	5	0.5	0.1	49.0	0.4	0.2	4	1952	1.5	2.2	3	1777	0.4	0.9	3	1951	1.4	1.9	2	4053	0.4	1.0
	6	0.5	0.5	186.2	0.5	1.2	4	2010	1.5	2.3	3	1853	0.4	0.9	3	2009	1.4	1.9	2	4245	0.4	1.2
	7	0.6	4.9	–	3.2	5.9	4	2060	1.6	2.3	4	2507	0.7	2.8	3	2059	1.5	2.1	3	5762	0.6	2.8
	8	1.8	66.4	–	6.7	44.1	5	2902	2.6	4.4	4	2803	0.7	2.8	4	2901	2.8	4.2	3	6526	1.2	4.0
ParallelSearch	4	3.2	0.1	5.8	0.0	0.1	2	784	0.3	0.3	2	758	0.1	0.2	1	784	0.3	0.3	1	1806	0.2	0.2
	5	3.3	14.3	1.7	0.0	0.1	3	1435	0.6	0.9	3	1527	0.6	1.1	2	1435	0.6	0.8	2	3547	0.9	1.6
	6	6.4	13.2	24.8	0.1	3.8	3	1450	0.8	1.1	3	1559	0.7	1.1	2	1450	0.7	0.9	2	3629	0.9	1.8
	7	159.7	317.1	540.5	0.2	1.9	4	2208	1.5	2.4	4	2371	1.3	2.9	3	2208	1.3	1.9	3	5626	2.2	5.5
	8	170.7	197.8	–	0.2	3.5	4	2236	1.2	2.0	4	2410	1.6	3.4	3	2236	1.3	2.1	3	5750	1.9	5.6
BubbleSort	4	0.4	0.0	0.6	0.1	0.1	11	11498	15.8	52.2	11	10982	10.1	76.0	10	11498	13.6	42.2	10	29280	17.4	120.1
	5	2.2	0.2	29.8	0.3	0.2	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	6	2.0	0.4	69.3	2.4	0.9	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	7	2.5	0.5	117.1	5.2	6.0	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	8	3.6	0.8	823.8	57.9	70.0	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
InsertionSort	4	0.4	3.6	4.3	0.1	0.3	10	12373	21.1	84.9	–	–	–	–	9	12373	15.7	60.3	–	–	–	–
	5	1.7	448.0	214.0	0.2	0.3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	6	1.3	–	–	1.3	3.8	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	7	1.6	–	–	5.7	5.3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	8	2.8	–	–	17.0	79.4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
SelectionSort	4	0.5	0.6	3.5	0.1	0.1	10	11080	9.3	26.6	10	10483	4.8	36.3	9	11080	7.1	19.9	9	27613	6.2	56.7
	5	1.6	0.5	659.8	1.9	0.4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	6	2.2	0.7	–	12.1	2.4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	7	2.2	3.2	–	35.8	14.5	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	8	4.9	1.9	–	156.6	132.9	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
SortingNetwork	4	0.1	0.0	0.1	0.0	0.0	4	1366	0.7	1.0	4	1242	0.2	0.8	3	1365	0.7	0.9	3	3610	0.4	1.3
	8	–	–	–	–	–	7	9338	8.2	15.8	7	8776	3.0	21.6	6	9337	7.4	13.6	6	25067	7.5	36.7

Table 2 Benchmark results (cont.)

Name	Size	Trans	GMC		LMC		BGMC								BLMC							
							Liveness				Safety				Liveness				Safety			
			Live	Safe	Live	Safe	*k*	Size	Time	Total	*k*	Size	Time	Total	*k*	Size	Time	Total	*k*	Size	Time	Total
MinMax	4	0.4	0.1	3.9	0.2	0.9	3	2678	1.8	2.4	3	2182	0.5	1.2	2	2678	1.7	2.1	2	6009	0.8	1.8
	5	0.9	0.2	69.5	1.0	37.0	3	4468	4.3	6.4	3	3477	1.4	2.9	2	4468	4.1	5.3	2	9633	2.1	4.5
	6	1.0	0.3	–	5.4	–	4	6342	6.7	11.6	4	5497	3.4	9.0	3	6342	6.9	10.9	3	14821	3.6	12.6
	7	1.3	0.4	–	27.5	–	4	6514	7.0	12.1	4	5736	3.6	9.1	3	6514	6.8	10.9	3	15275	3.7	13.0
	8	1.5	0.5	–	126.8	–	5	8266	10.9	20.8	5	7797	6.4	19.7	4	8266	10.1	18.6	4	20371	6.0	28.4
FastMax	4	0.2	0.0	0.3	0.1	0.2	2	829	0.3	0.4	2	1016	0.1	0.1	1	828	0.3	0.3	1	2090	0.1	0.1
	5	0.9	0.1	2.1	0.5	1.7	2	1285	0.6	0.6	2	1574	0.2	0.2	1	1284	0.6	0.6	1	3221	0.1	0.1
	6	5.0	0.6	21.6	1.6	24.0	2	1685	0.8	0.9	2	2108	0.2	0.2	1	1684	0.8	0.8	1	4198	0.2	0.2
	7	30.5	4.0	298.0	12.3	321.6	2	2153	1.1	1.2	2	2734	0.3	0.3	1	2152	1.2	1.3	1	5337	0.2	0.2
	8	144.2	29.8	–	227.7	–	2	2689	1.3	1.5	2	3452	0.4	0.5	1	2688	1.5	1.5	1	6638	0.3	0.3
Partition	4	1.1	2.8	17.0	0.1	0.1	5	3236	3.0	4.8	5	3115	4.5	9.4	4	3231	2.9	4.6	4	8171	6.5	12.9
	5	13.9	784.9	–	0.3	0.3	6	8324	15.7	27.9	6	7836	13.6	38.6	5	8318	12.6	22.7	5	22495	24.5	67.4
	6	7.8	588.5	–	10.5	1.0	7	9732	17.4	35.9	7	9345	36.8	81.0	6	9725	15.8	28.6	6	26845	79.4	157.0
	7	10.4	815.5	–	15.4	4.0	8	11230	22.6	48.7	8	10935	64.3	159.4	7	11222	32.4	51.4	7	31399	135.5	296.0
	8	36.7	–	–	74.6	25.8	9	13135	38.7	71.8	9	12958	69.4	215.2	8	13126	27.8	55.9	8	36912	202.6	508.6
ParallelPrefixSum	4	3.1	0.0	3.8	0.0	0.3	3	2248	1.7	2.1	3	2030	0.6	1.6	2	2248	1.6	1.9	2	5638	0.7	1.7
	5	111.7	0.2	102.1	0.1	4.3	4	6862	8.0	11.5	4	7392	4.9	16.9	3	6862	7.5	9.9	3	19189	5.3	18.7
	6	775.7	0.5	–	0.4	5.4	4	7610	10.1	13.8	4	7923	5.0	20.2	3	7610	8.0	10.6	3	21122	4.1	19.1
	7	–	–	–	–	–	4	8470	11.2	15.3	4	8547	3.7	19.3	3	8470	11.9	15.1	3	23379	4.9	24.6
	8	–	–	–	–	–	4	9154	14.4	19.0	4	8956	4.0	21.7	3	9154	11.1	14.7	3	25061	5.8	24.6
Bakery	4	1.6		1.7		0.1					6	1909	0.5	2.7					5	5149	0.6	4.0
	5	38.5		34.8		0.8					7	3143	1.1	8.6					6	8514	1.4	10.9
	6	–		–		–					8	4513	1.8	17.3					7	11917	3.4	24.7
	7	–		–		–					9	6241	2.4	32.3					8	16118	3.5	52.6
	8	–		–		–					10	8490	7.6	101.1					9	21468	8.7	129.3
Barber	10	20.4		108.7		0.0					1	260	0.0	0.0					1	2712	0.1	0.1
	11	36.4		123.7		0.0					1	276	0.0	0.0					1	2926	0.2	0.2
	12	41.4		213.5		0.0					1	292	0.0	0.0					1	3146	0.2	0.2
	13	48.4		264.3		0.0					1	308	0.0	0.0					1	3372	0.2	0.2
	14	74.7		444.3		0.0					1	324	0.0	0.0					1	3604	0.2	0.2
	15	80.4		778.1		0.0					1	340	0.0	0.0					1	3842	0.2	0.2
	16	136.6		721.7		0.1					1	356	0.0	0.0					1	4144	0.2	0.2

(witness) is found after a few number of unwinding steps. Moreover, we observe that the differences between BGMC and BLMC are rather small in contrast to GMC and LMC.

6. Summary and conclusions

In BMC, a specification is checked by reducing it to a satisfiability problem of the base logic. Traditionally, this is done by syntactically unwinding the specification a bounded number of times. Recently, more sophisticated approaches have been proposed that are based on the translation of temporal logic formulas to ω-automata. However, these approaches suffer from the fact that even for restricted classes of properties, most translations yield a very general type of ω-automata that cannot be used directly for BMC. To solve this problem, we presented a technique for the translation of temporal logic formulas to the corresponding classes of the automata hierarchy. In this way, many properties can be translated to ω-automata whose acceptance conditions are simple safety and liveness properties.

Moreover, we presented two approaches to check the resulting specifications, namely bounded global and bounded local model checking. The former is based on fixpoint approximation and can be viewed as a variant of traditional BMC. In contrast, the latter is based on the construction of proof trees using syntax directed decomposition rules. For the reduction to a satisfiability problem, we employ three-valued logic in order to explicitly forward uncertain information in the case a proof cannot be established due to insufficient bounds. As for finite state systems, our experimental results show that both approaches are significantly more efficient than their unbounded counterparts, provided that a witness (counterexample) is found for small or medium-sized bounds.

Acknowledgments We thank the reviewers for their detailed and valuable comments that helped us to substantially improve this paper.

References

1. Alt M, Ferdinand C, Martin F, Wilhelm R (1996) Cache behavior prediction by abstract interpretation. In: Static analysis symposium (SAS). LNCS, vol 1145. Springer, Aachen, Germany, pp 52–66
2. Amla N, Kurshan R, McMillan K, Medel R (2003) Experimental analysis of different techniques for bounded model checking. In: Garavel H, Hatcliff J (eds) Conference on tools and algorithms for the construction and analysis of systems (TACAS). LNCS, vol 2619. Springer, Warsaw, Poland, pp 34–48
3. Andrews G (1991) Concurrent programming—Principles and practice. The Benjamin/Cummings Publishing Company, Redwood City, California
4. Berry G (1999) The constructive semantics of pure Esterel. http://www-sop.inria.fr/esterel.org
5. Berthet C, Coudert O, Madre J (1990) New ideas on symbolic manipulations of finite state machines. In: International conference on computer design (ICCD). IEEE, pp 224–227
6. Biere A, Cimatti A, Clarke E, Fujita M, Zhu Y (1999) Symbolic model checking using SAT procedures instead of BDDs. In: International design automation conference (DAC). ACM, New Orleans, Louisiana, USA, pp 317–320
7. Biere A, Cimatti A, Clarke E, Strichman O, Zhu Y (2003) Bounded model checking. Adv Comput 58
8. Biere A, Cimatti A, Clarke E, Zhu Y (1999) Symbolic model checking without BDDs. In: Cleaveland R (ed) Conference on tools and algorithms for the construction and analysis of systems (TACAS). LNCS, vol 1579. Springer, Amsterdam, The Netherlands, pp 193–207
9. Boigelot B, Wolper P (2002) Representing arithmetic constraints with finite automata: An overview. In: International conference on logic programming (ICPL). LNCS, vol 2401. Springer, Copenhagen, Denmark, pp 1–19
10. Boudet A, Comon H (1996) Diophantine equations, Presburger arithmetic and finite automata. In: Kirchner H (ed) Colloquium on trees in algebra and programming (CAAP). LNCS, vol 1059. Springer, Linköping, Sweden, pp 30–43

11. Bradfield J (1992) Verifying temporal properties of systems. Progress in theoretical computer science. Birkhäuser, Boston, Basel, Berlin
12. Bradfield J, Stirling C (1991) Local model checking for infinite state spaces. In: Larsen K, Skou A (eds) Workshop on computer aided verification (CAV)
13. Bryant R (1986) Graph-based algorithms for Boolean function manipulation. IEEE Trans Comput C-35(8):677–691
14. Bryant R (1991) On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. IEEE Trans Comput 40(2):205–213
15. Brzozowski J, Seger C-J (1995) Asynchronous circuits. Springer
16. Bultan T, Gerber R, Pugh W (1997) Symbolic model checking of infinite state systems using Presburger arithmetic. In: Grumberg O (ed) Conference on computer aided verification (CAV). LNCS, vol 1254. Springer, Haifa, Israel, pp 400–411
17. Bultan T, Gerber R, Pugh W (1999) Model-checking concurrent systems with unbounded integer variables. ACM Trans Progr Lang Syst (TOPLAS) 21(4):747–789
18. Burch J, Clarke E, McMillan K, Dill D, Hwang L (1990) Symbolic model checking: 10^{20} states and beyond. In: Symposium on logic in computer science (LICS). IEEE Computer Society, Washington, DC, pp 1–33
19. Burch J, Clarke E, McMillan K, Dill D, Hwang L (1992) Symbolic model checking: 10^{20} states and beyond. Inf Comput 98(2):142–170
20. Burkart O, Caucal D, Moller F, Steffen B (2001) Verification of infinite structures. In: Handbook of process algebra. Elsevier Science, pp 545–623
21. Büchi J (1960) On a decision method in restricted second order arithmetic. In: Nagel E (ed) International Congress on logic, methodology and philosophy of science. Stanford University Press, Stanford, CA, pp 1–12
22. Cabodi G, Camurati P, Quer S (2002) Can BDDs compete with SAT solvers on bounded model checking? In: International design automation conference (DAC). ACM, New Orleans, Louisiana, USA, pp 117–122
23. Clarke E, Biere A, Raimi R, Zhu Y (2001) Bounded model checking using satisfiability solving. Form Meth Syst Desi 19(1):7–34
24. Clarke E, Emerson E (1981) Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen D (ed) Workshop on logics of programs. LNCS, vol 131. Springer, Yorktown Heights, New York, pp 52–71
25. Clarke E, Grumberg O, Hamaguchi K (1994) Another look at LTL model checking. In: Dill D (ed) Conference on computer aided verification (CAV). LNCS, vol 818. Springer, Stanford, California, USA, pp 415–427
26. Clarke E, Grumberg O, Peled D (1999) Model checking. MIT, London, England
27. Clarke E, Kroening D, Ouaknine J, Strichman O (2004) Completeness and complexity of bounded model checking. In: Steffen B, Levi G (eds) Verification, model checking, and abstract interpretation (VMCAI). LNCS, vol 2937. Springer, Venice, Italy, pp 85–96
28. Clarke E, Wing J (1996) Formal methods: state of the art and future directions. ACM Comput Surv, 28(4):626–643
29. Cleaveland R (1989) Tableaux-based model checking in the propositional μ-calculus. Acta Informatica 27(8):725–747
30. Cormen T, Leiserson C, Rivest R, Stein C (2001) Introduction to algorithms. The MIT Press
31. de Moura L, Rueß H, Sorea M (2002) Lazy theorem proving for bounded model checking over infinite domains. In: Conference on automated deduction (CADE). LNCS, vol 2392. Springer Verlag, Copenhagen, Denmark, pp 438–455
32. Emerson E (1997) Model checking and the μ-calculus. In: Immerman N, Kolaitis P (eds) Symposium on descriptive complexity and finite models. American Mathematical Society (AMS), pp 185–214
33. Enderton H (1972) A mathematical introduction to logic. Academic, New York, NY
34. Esparza J (2003) An automata-theoretic approach to software verification. In: Developments in language theory. LNCS, vol 2710. Springer, pp 21
35. Ganesh V, Berezin S, Dill D (2002) Deciding Presburger arithmetic by model checking and comparisons with other methods. In: Aagaard M, O'Leary J (eds) Conference on formal methods in computer aided design (FMCAD). LNCS, vol 2517. Springer, Portland, USA, pp 171–186
36. Gerth R, Peled D, Vardi M, Wolper P (1995) Simple on-the-fly automatic verification of linear temporal logic. In: Symposium on protocol specification, testing, and verification (PSTV). Warsaw, North Holland
37. Goldberg E, Novikov Y (2002) BerkMin: a fast and robust SAT-solver. In: Design, automation and test in Europe (DATE). IEEE Computer Society, Paris, France, pp 143–149
38. Grädel E, Kolaitis P, Vardi M (1997) On the decision problem for two-variable first-order logic. The Bulletin of Symbolic Logic 3(1):53–69

39. Grädel E, Rosen E (1999) Preservation theorems for two-variable logic. Mathe Log Quart 45:315–325
40. Kesten Y, Pnueli A, Raviv L (1998) Algorithmic verification of linear temporal logic specifications. In: Colloquium on automata, languages and programming (ICALP). LNCS, vol 1443. Springer, Aalborg, Denmark, pp 1–16
41. Kleene S (1952) Introduction to metamathematics. North Holland
42. Krishnan S, Puri A, Brayton R (1994) Deterministic ω-automata vis-a-vis deterministic Büchi automata. In: Symposium on algorithms and computation (ISAAC). LNCS, vol 834. Springer, Beijing, China, pp 378–386
43. Kukula J, Shiple T, Aziz A (1998) Techniques for implicit state enumeration of EFSMs. In: Gopalakrishnan G, Windley P (eds) Conference on formal methods in computer aided design (FMCAD). LNCS, vol 1522. Springer, Palo Alto, California, USA, pp 469–482
44. Landweber L (1969) Decision problems for ω-automata. Mathe Syst Theory 3(4):376–384
45. Lichtenstein O, Pnueli A (1985) Checking that finite state concurrent programs satisfy their linear specification. In: Symposium on principles of programming languages (POPL). ACM, New York, pp 97–107
46. Malik S (1993) Analysis of cyclic combinational circuits. In: Conference on computer aided design (ICCAD). IEEE Computer Society, Santa Clara, California, pp 618–625
47. Manna Z, Pnueli A (1988) The anchored version of the temporal framework. In: Linear time, branching time and partial order in logics and models for concurrency. LNCS, vol 354. Springer, Noordwigherhout, Netherland, pp 428–437
48. Manna Z, Pnueli A (1990) A hierarchy of temporal properties. In: Symposium on principles of distributed computing (PODC), pp 377–408
49. Marques Silva J, Sakallah K (1999) Grasp: a search algorithm for propositional satisfiability. IEEE Trans Comput 48(5):506–521
50. McNaughton R, Papert S (1971) Counter-free automata. MIT
51. Meinel C, Theobald T (1998) Algorithms and data structures in VLSI design: OBDD—foundations and applications. Springer
52. Mortimer M (1975) On languages with two variables. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik 21:135–140
53. Moskewicz M, Madigan C, Zhao Y, Zhang L, Malik S (2001) Chaff: Engineering an efficient SAT solver. In: International design automation conference (DAC). ACM, Las Vegas, Nevada, USA, pp 530–535
54. Muller D, Saoudi A, Schupp P (1988) Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In: Symposium on logic in computer science (LICS), pp 422–427
55. Oppen D (1978) A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. Comput Syst Sci 16:323–332
56. Owicki S, Gries D (1976) An axiomatic proof technique for parallel programs *I*. Acta Informatica 6(4):319–340
57. Pnueli A (1977) The temporal logic of programs. In: Symposium on foundations of computer science (FOCS), vol 18. IEEE Computer Society, New York, pp 46–57
58. Presburger M (1930) Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: Leja F (ed) Sprawozdanie z I Kongresu Matematyków Krajów Słowiańskich, Warszawa 1929 (Comptes–rendus du I Congrès des Mathématiciens des Pays Slaves, Varsovie 1929). Warszawa, pp 92–101 (supplement on p 395)
59. Quielle J, Sifakis J (1981) Specification and verification of concurrent systems in CESAR. In: Symposium on programming
60. Reps T, Sagiv M, Wilhelm R (2004) Static program analysis via 3-valued logic. In: Alur R, Peled D (eds) Conference on computer aided verification (CAV). LNCS, vol 3114. Springer, Boston, MA, USA, pp 15–30
61. Schneider K (2000) A verified hardware synthesis for Esterel. In: Rammig F (ed) Workshop on distributed and parallel embedded systems (DIPES). Kluwer, Schloß Ehringerfeld, Germany, pp 205–214
62. Schneider K (2001) Embedding imperative synchronous languages in interactive theorem provers. In: Conference on application of concurrency to system design (ACSD). IEEE Computer Society, Newcastle upon Tyne, UK, pp 143–156
63. Schneider K (2001) Improving automata generation for linear temporal logic by considering the automata hierarchy. In: International conference on logic for programming, artificial intelligence, and reasoning (LPAR). LNAI, vol 2250. Springer, Havanna, Cuba, pp 39–54
64. Schneider K (2002) Proving the equivalence of microstep and macrostep semantics. In: Carreño V, Muñoz C, Tahar S (eds) Higher order logic theorem proving and its applications (TPHOL). LNCS, vol 2410. Springer, Hampton, VA, USA, pp 314–331

65. Schneider K (2003) Verification of reactive systems—formal methods and algorithms. Texts in theoretical computer science (EATCS Series), Springer
66. Schneider K, Brandt J, Schuele T, Tuerk T (2005) Maximal causality analysis. In: Conference on application of concurrency to system design (ACSD). IEEE Computer Society, St. Malo, France, pp 106–115
67. Schneider K, Kumar R, Kropf T (1993) Alternative proof procedures for finite-state machines in higher-order logic. In: Joyce J, Seger C-J (eds) Higher order logic theorem proving and its applications (TPHOL). LNCS, vol 780. Springer, Vancouver, Canada, pp 213–226
68. Schuele T, Schneider K (2004) Bounded model checking of infinite state systems: exploiting the automata hierarchy. In: Formal methods and models for codesign (MEMOCODE). IEEE, San Diego, CA, pp 17–26
69. Schuele T, Schneider K (2004) Global vs. local model checking: a comparison of verification techniques for infinite state systems. In: International conference on software engineering and formal methods (SEFM). IEEE, Beijing, China, pp 67–76
70. Sheeran M, Singh S, Stålmarck G (2000) Checking safety properties using induction and a SAT-solver. In: Hunt W, Johnson S (eds) Conference on formal methods in computer aided design (FMCAD). LNCS, vol 1954. Springer, Austin, Texas, USA, pp 108–125
71. Shiple T (1996) Formal analysis of synchronous circuits. PhD thesis, University of California at Berkeley
72. Shiple T, Kukula J, Ranjan R (1998) A comparison of Presburger engines for EFSM reachability. In: Hu A, Vardi M (eds) Conference on computer aided verification (CAV). LNCS, vol 1427. Springer, Vancouver, BC, Canada, pp 280–292
73. Somenzi F (2001) Efficient manipulation of decision diagrams. Softw Tools Technol Trans (STTT) 3(2):171–181
74. Stirling C, Walker D (1989) Local model checking in the modal μ-calculus. In: Diaz J, Orejas F (eds) Theory and practice of software development (TAPSOFT). LNCS, vol 351. Springer, pp 369–383
75. Stirling C, Walker D (1991) Local model checking in the modal μ-calculus. Theor Comput Sci 89(1):161–177
76. Tarski A (1955) A lattice-theoretical fixpoint theorem and its applications. Pacific J Math 5:285–309
77. Thomas W (1990) Automata on infinite objects, vol B, chapter automata on infinite objects, Elsevier, pp 133–191
78. Thomas W (2001) A short introduction to infinite automata. In: Conference on developments in language theory. LNCS, vol 2295. Springer, pp 130–144
79. Tuerk T, Schneider K (2005) Relationship between alternating omega-automata and symbolically represented nondeterministic omega-automata. Internal Report 340, Department of Computer Science, University of Kaiserslautern, http://kluedo.ub.uni-kl.de
80. Vardi M (1994) Nontraditional applications of automata theory. In: Symposium on theoretical aspects of computer science (STACS). LNCS, vol 789. Springer, Sendai, Japan, pp 575–597
81. Vardi M (1996) Why is modal logic so robustly decidable? In: Descriptive complexity and finite models, no 31 in DIMACS workshop. American Mathematical Society (AMS), pp 149–184
82. Wagner K (1979) On ω-regular sets. Inform Contr 43:123–177
83. Wolper P (1983) Temporal logic can be more expressive. Inform Contr 56(1–2):72–99
84. Wolper P (2001) Constructing automata from temporal logic formulas: A tutorial. In: Summer school on formal methods in performance analysis. LNCS, vol 2090. Springer, pp 261–277
85. Wolper P, Boigelot B (2000) On the construction of automata from linear arithmetic constraints. In: Graf S, Schwartzbach M (eds) Conference on tools and algorithms for the construction and analysis of systems (TACAS). LNCS, vol 1785. Springer, Berlin, Germany, pp 1–19