Model-Checking
A Tutorial Introduction

Markus Müller-Olm[1], David Schmidt[2], and Bernhard Steffen[1]

[1] Dortmund University, Department for Computer Science, FB 4, LS 5,
44221 Dortmund, Germany,
{mmo,steffen}@ls5.cs.uni-dortmund.de

[2] Kansas State University, Department of Computing and Information Sciences,
Manhattan, Kansas 66506, USA,
schmidt@cis.ksu.edu

Abstract. In the past two decades, model-checking has emerged as a promising and powerful approach to fully automatic verification of hardware systems. But model checking technology can be usefully applied to other application areas, and this article provides fundamentals that a practitioner can use to translate verification problems into model-checking questions. A taxonomy of the notions of "model," "property," and "model checking" are presented, and three standard model-checking approaches are described and applied to examples.

1 Introduction

In the last two decades model-checking [11, 34] has emerged as a promising and powerful approach to automatic verification of systems. Roughly speaking, a *model checker* is a procedure that decides whether a given structure M is a model of a logical formula ϕ, i.e. whether M satisfies ϕ, abbreviated $M \models \phi$. Intuitively, M is an (abstract) model of the system in question, typically a finite automata-like structure, and ϕ, typically drawn from a temporal or modal logic, specifies a desirable property. The model-checker then provides a push-button approach for proving that the system modeled by M enjoys this property. This full automation together with the fact that *efficient* model-checkers can be constructed for *powerful* logics, forms the attractiveness of model-checking.

The above "generic" description of model-checking leaves room for refinement. What exactly is a model to be checked? What kind of formulas are used? What is the precise interpretation of satisfaction, $\models$? We present a rough map over the various answers to these questions, and in the process, we introduce the main approaches.

The various model-checking approaches provide a cornucopia of generic decision procedures that can be applied to scenarios that go far beyond the problem domains for which the approaches were originally invented. (The work of of some of the authors on casting data flow analysis questions as model-checking problems is an example [35].) We intend to provide a practitioner with a basis she

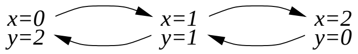


Fig. 1. Example Kripke structure

can use to translate problems into model structures and formulas that can be solved by model checking.

The rest of this article is organized as follows: In the next section we discuss the model structures underlying model-checking—Kripke structures, labeled transition systems and a structure combining both called Kripke transition systems. Section 3 surveys the spectrum of the logics used for specifying properties to be automatically checked for model structures. We then introduce three basic approaches to model-checking: the semantic or iterative approach, the automata-theoretic approach, and the tableau method. The paper finishes with a number of concluding remarks.

2 Models

Model-checking typically depends on a discrete model of a system—the system's behavior is (abstractly) represented by a graph structure, where the nodes represent the system's states and the arcs represent possible transitions between the states. It is common to abstract from the identity of the nodes. Graphs alone are too weak to provide an interesting description, so they are annotated with more specific information. Two approaches are in common use: *Kripke structures*, where the nodes are annotated with so-called *atomic propositions* and *labeled transition systems* (LTS), where the arcs are annotated with so-called *actions*. We study these two structures and introduce a third called *Kripke transition systems*, which combines Kripke structures and labeled transition systems and which is often more convenient for modeling purposes.

Kripke Structures. A *Kripke structure (KS)* over a set AP of atomic propositions is a triple (S, R, I), where S is a set of *states*, $R \subseteq S \times S$ is a *transition relation*, and $I : S \to 2^{\mathsf{AP}}$ is an *interpretation*. Intuitively the atomic propositions, which formally are just symbols, represent basic local properties of system states; I assigns to each state the properties enjoyed by it. We assume that a set of atomic propositions AP always contains the propositions true and false and that, for any state s, $\mathsf{true} \in I(s)$ and $\mathsf{false} \notin I(s)$. A Kripke structure is called *total* if R is a total relation, i.e. if for all $s \in S$ there is a $t \in S$ such that $(s, t) \in R$ otherwise it is called *partial*. For model-checking purposes S and AP are usually finite.

Figure 1 displays an example Kripke structure whose propositions take the form, $var = num$; the structure represents the states that arise while the program's components, x and y, trade two resources back and forth.

Kripke structures were first devised as a model theory for modal logic [5, 25], whose propositions use modalities that express necessity ("must") and possibility ("may"). In this use of a Kripke structure, the states correspond to different "worlds" in which different basic facts (the atomic propositions) are true; transitions represent reachability between worlds. The assertion that some fact is possibly true is interpreted to mean there is a reachable state (world) in which the fact holds; the assertion that a fact is necessarily true means that the fact holds in all reachable worlds. Kripke showed that the axioms and rules in different systems of modal logics correspond to properties that hold in different classes of Kripke structures [28, 29]. These logical settings for Kripke structures (in particular, the notion of "worlds") can provide useful guidance for expressing computing applications as Kripke structures [5, 16].

Labeled Transition Systems. A *labeled transition system (LTS)* is a triple $T = (S, \mathsf{Act}, \rightarrow)$, where S is a set of *states*, Act is a set of *actions*, and $\rightarrow \subseteq S \times \mathsf{Act} \times S$ is a *transition relation*. A transition $(s, a, s') \in \rightarrow$, for which we adopt the more intuitive notation $s \stackrel{a}{\rightarrow} s'$, states that the system can evolve from state s to state s' thereby exchanging action a with its environment. We call $s \stackrel{a}{\rightarrow} s'$ a *transition* from s to s' labeled by action a, and s' is an *a-successor* of s. In an LTS, the transitions are labeled with single actions, while in a Kripke structure, states are labeled with sets of atomic propositions. Labeled transition systems originate from concurrency theory, where they are used as an operational model of process behavior [33]. In model-checking applications S and Act are usually finite.

Fig. 3 displays two small examples of labeled transition systems that display the actions a vending machine might take.

Kripke Transition Systems. Labels on arcs appear naturally when the labeling models the dynamics of a system, whereas labels on node appear naturally when the labeling models static properties of states. There are various ways to encode arc labelings by node labelings and vice versa. (One of them is described below.) And, logical considerations usually can be translated between these two representations. For these reasons, theoretical analyses study just one form of labeling. For modeling purposes, however, it is often natural to have *both* kinds of labeling available. Therefore, we introduce a third model structure that combines labeled transition systems and Kripke structures.

A *Kripke transition system (KTS)* over a set AP of *atomic propositions* is a structure $T = (S, \mathsf{Act}, \rightarrow, I)$, where S is a set of *states*, Act is a set of *actions*, $\rightarrow \subseteq S \times \mathsf{Act} \times S$ is a *transition relation* and $I : S \rightarrow 2^{\mathsf{AP}}$ is an *interpretation*. For technical reasons we assume that AP and Act are disjoint. Kripke transition systems generalize both Kripke structures and labeled transition systems: A Kripke structure is a Kripke transition system with an empty set of actions, Act, and a labeled transition system is a Kripke transition system with a trivial interpretation, I.

Kripke transition systems work well for modeling sequential imperative programs for data flow analysis purposes, as they concisely express the implied predicate transformer scheme: nodes express the predicates or results of the considered analysis, and edges labeled with the statements express the nodes'

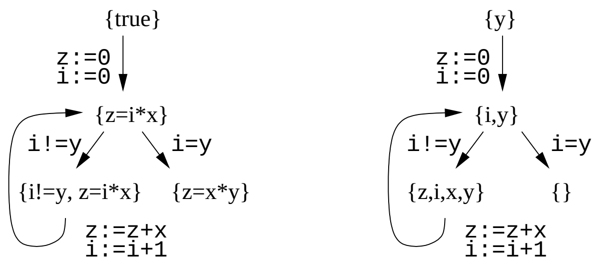


Fig. 2. Two Kripke transition systems for a program

interdependencies. If data flow analysis is performed via model-checking, Kripke transition systems thus enable to use the result of one analysis phase as input for the next one.

Figure 2 shows two Kripke transition systems for the program, `z:=0; i:=0; while i!=y do z:=z+x; i:=i+1 end`. Both systems label arcs with program phrases. The first system uses properties that are logical propositions of the form, *var* = *expr*; it portrays a predicate-transformer semantics. The second system uses propositions that are program variables; it portrays the results of a definitely-live-variable analysis.

Any Kripke transition system $T = (S, \mathsf{Act}, \rightarrow, I)$ over AP induces in a natural way a Kripke structure K_T which codes the same information. The idea is to associate the information about the action exchanged in a transition with the reached state instead of the transition itself. This is similar to the classic translation of Mealy-Automata to Moore-Automata. Formally, K_T is the Kripke structure $(S \times \mathsf{Act}, R, I')$ over $\mathsf{AP} \cup \mathsf{Act}$ with $R = \{(\langle s, a\rangle, \langle s', a'\rangle) \mid s \xrightarrow{a'} s'\}$ and $I'(\langle s, a\rangle) = I(s) \cup \{a\}$. Logical consideration about T usually can straightforwardly be translated to considerations about K_T and vice versa. Therefore, logicians usually prefer to work with the structurally more simple Kripke structures. Nevertheless, the richer framework of Kripke transition systems is often more convenient for modeling purposes.

Often we may want to designate a certain state $s_0 \in S$ in a KS, LTS, or KTS as the *initial state*. Intuitively, execution of the system starts in this state. A structure together with such a designated initial state is called a *rooted* structure.

3 Logics

The interpretation, I, in a Kripke transition system defines local properties of states. Often we are also interested in global properties connected to the transitional behavior. For example, we might be interested in *reachability* properties, like, "Can we reach from the initial state a state where the atomic proposition P holds?" Temporal logics [17, 36] are logical formalisms designed for expressing such properties.

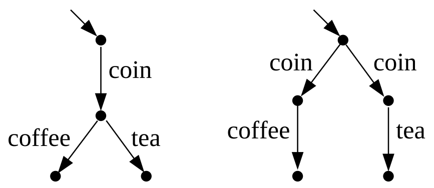


Fig. 3. Two vending machines

Temporal logics come in two variants, *linear-time* and *branching-time*. Linear-time logics are concerned with properties of paths. A state in a transition system is said to satisfy a linear-time property if all paths emanating from this state satisfy the property. In a labeled transition system, for example, two states that generate the same language satisfy the same linear-time properties. Branching-time logics, on the other hand, describe properties that depend on the branching structure of the model. Two states that generate the same language but by using different branching structures can often be distinguished by a branching-time formula.

As an example, consider the two rooted, labeled transition systems in Fig. 3, which model two different vending machines offering tea and coffee. Both machines serve coffee or tea after a coin has been inserted, but from the customer's point of view the right machine is to be avoided, because it decides internally whether to serve coffee or tea. The left machine, in contrast, leaves this decision to the customer. Both machines have the same set of computations (maximal paths): $\{\langle \text{coin}, \text{coffee} \rangle, \langle \text{coin}, \text{tea} \rangle\}$. Thus, a linear-time logic will be unable to distinguish the two machines. In a branching-time logic, however, the property, "a coffee action is possible after any coin action" can be expressed, which differentiates the two machines.

The choice of using a linear-time or a branching-time logic depends on the properties to be analyzed. Due to their greater selectivity, branching-time logics are often better for analyzing reactive systems. Linear-time logics are preferred when only path properties are of interest, as when analyzing data-flow properties of graphs of imperative programs.

3.1 Linear-Time Logics

Propositional linear-time logic (PLTL) is the basic prototypical linear-time logic. It is often presented in a form to be interpreted over Kripke structures. Its formulae are constructed as follows, where p ranges over a set AP of atomic propositions:

$$\phi ::= p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \mathsf{X}(\phi) \mid \mathsf{U}(\phi, \psi) \mid \mathsf{F}(\phi) \mid \mathsf{G}(\phi)$$

$$
\begin{array}{rll}
\pi \models p & \text{iff} & p \in I(\pi_0) \\
\pi \models \neg\phi & \text{iff} & \pi \not\models \phi \\
\pi \models \phi_1 \vee \phi_2 & \text{iff} & \pi \models \phi_1 \text{ or } \pi \models \phi_2 \\
\pi \models \mathsf{X}(\phi) & \text{iff} & |\pi| > 1 \text{ and } \pi^1 \models \phi \\
\pi \models \mathsf{U}(\phi, \psi) & \text{iff} & \text{there is } k,\ 0 \leq k < |\pi|, \text{ with } \pi^k \models \psi \text{ and for all } i,\ 0 \leq i < k,\ \pi^i \models \phi \\
\pi \models \mathsf{F}(\phi) & \text{iff} & \text{there is } k,\ 0 \leq k < |\pi|, \text{ with } \pi^k \models \psi \\
\pi \models \mathsf{G}(\phi) & \text{iff} & \text{for all } k \text{ with } 0 \leq k < |\pi|,\ \pi^k \models \psi
\end{array}
$$

Fig. 4. Semantics of PLTL

PLTL formulas are interpreted over *paths* in a Kripke structure $K = (S, R, I)$ over AP. A *finite path* is a finite, non-empty sequence $\pi = \langle \pi_0, \ldots, \pi_{n-1} \rangle$ of states $\pi_0, \ldots, \pi_{n-1} \in S$ such that $(\pi_i, \pi_{i+1}) \in R$ for all $0 \leq i < n - 1$. n is called the *length* of path, denoted by $|\pi|$. An *infinite path* is an infinite sequence $\pi = \langle \pi_0, \pi_1, \pi_2 \ldots \rangle$ of states in S such that $(\pi_i, \pi_{i+1}) \in R$ for all $i \geq 0$. The length of an infinite path is ∞. For $0 \leq i < |\pi|$, π_i denotes the i-th state in path π, and π^i is $\langle \pi_i, \pi_i + 1, \ldots \rangle$, the tail of the path starting at π_i. In particular, $\pi^0 = \pi$. A path in a Kripke structure is called *maximal* if it cannot be extended. In particular, every infinite path is maximal.

In Fig. 4, we present an inductive definition of when a path, π, in a Kripke structure $K = (S, R, I)$ *satisfies* a PLTL formula, ϕ. Intuitively, π satisfies an atomic proposition, p, if its first state does; atomic propositions represent local properties. $\neg$ and $\vee$ are interpreted in the obvious way; further Boolean connectives may be introduced as abbreviations in the usual way, e.g., $\phi_1 \wedge \phi_2$ can be introduced as $\neg(\neg\phi_1 \wedge \neg\phi_2)$.

The modality $\mathsf{X}(\phi)$ ("next ϕ") requires the property ϕ for the next situation in the path; formally, $\mathsf{X}(\phi)$ holds if ϕ holds for the path obtained by removing the first state. $\mathsf{G}(\phi)$ ("generally ϕ" or "always ϕ") requires ϕ to hold for all situations; $\mathsf{F}(\phi)$ ("finally ϕ") for some (later) situation. Thus G and F provide a kind of universal (resp., existential) quantification over the later situations in a path. $\mathsf{U}(\phi, \psi)$ ("ϕ until ψ") requires ψ to become true at some later situation and ϕ to be true at all situations visited before. This operator sometimes is called "strong until" because it requires ψ to become true finally. This is different for a variant of the until modality, called "weak until," because the formula holds true when ϕ is true forever. Strong- and weak-until can be defined from each other using F and G:

$$\mathsf{U}(\phi, \psi) = \mathsf{WU}(\phi, \psi) \wedge \mathsf{F}(\psi) \quad \text{and} \quad \mathsf{WU}(\phi, \psi) = \mathsf{U}(\phi, \psi) \vee \mathsf{G}(\phi) \ .$$

They are also (approximate) duals:

$$\neg\mathsf{U}(\phi, \psi) = \mathsf{WU}(\neg\psi, \neg\phi \wedge \neg\psi) \quad \text{and} \quad \neg\mathsf{WU}(\phi, \psi) = \mathsf{U}(\neg\psi, \neg\phi \wedge \neg\psi) \ .$$

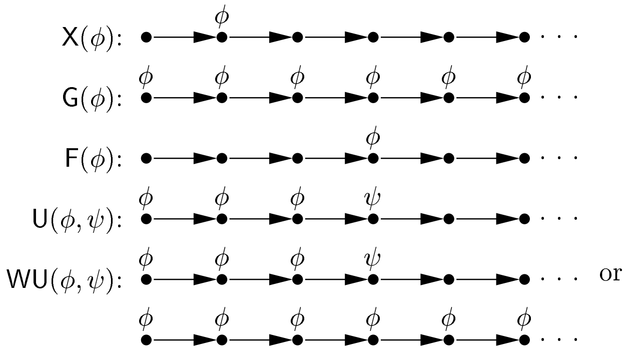


Fig. 5. Illustration of linear-time modalities

Moreover, F can easily be defined in terms of U, and G in terms of WU

$$\mathsf{F}(\phi) = \mathsf{U}(\mathsf{true}, \phi) \quad \text{and} \quad \mathsf{G}(\phi) = \mathsf{WU}(\phi, \mathsf{false}) \ ,$$

and F and G are duals:

$$\mathsf{F}(\phi) = \neg\mathsf{G}(\neg\phi) \quad \text{and} \quad \mathsf{G}(\phi) = \neg\mathsf{F}(\neg\phi) \ .$$

The meaning of the modalities is illustrated in Fig. 5.

While the basic structures of a linear-time logic are paths, the model-checking question usually is presented for a given Kripke structure. The question is then to determine, for each state, whether all paths emanating from the state satisfy the formula. Sometimes, one restricts the question to certain kinds of paths, e.g., just infinite paths, or maximal finite paths only, or all (finite or infinite) maximal paths. Perhaps the most common case is to consider infinite paths in total Kripke structures.

Variants of PLTL may also be defined for speaking about the actions in a KTS or LTS. A (finite) path in a KTS or LTS is then a non-empty alternating sequence $\pi = \langle s_0, a_1, s_1, \ldots, s_{n-1} \rangle$ of states and actions that begins and ends with a state and satisfies $s_i \stackrel{a_{i+1}}{\to} s_{i+1}$ for $i = 0, \ldots, n-1$. Again we call $|\pi| = n$ the length of path π, π_i stands for s_i, and π^i denotes the path $\langle s_i, a_{i+1}, s_{i+1}, \ldots, s_{n-1} \rangle$. Infinite paths are defined similarly. With these conventions, PLTL can immediately be interpreted on such extended paths with the definition in Fig. 4. We may now also extend the syntax of PLTL by allowing formulas of the form (a), where a is an action in Act. These formulas are interpreted as follows:

$$\pi \models (a) \ \text{ iff } \ |\pi| > 1 \text{ and } a_1 = a \ ,$$

where a_1 is the first action in π.

3.2 Branching-Time Logics

Hennessy-Milner Logic. Hennessy-Milner logic (HML) is a simple modal logic introduced by Hennessy and Milner in [24, 33]. As far as model-checking is

Fig. 6. Illustration of branching-time modalities

concerned, Hennessy-Milner logic is limited because it can express properties of only bounded depth. Nevertheless, it is of interest because it forms the core of the modal μ-calculus, which appears in the next section.

HML is defined over a given set, Act, of actions, ranged over by a. Formulas are constructed according to the grammar,

$$\phi ::= \mathsf{true} \mid \mathsf{false} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\phi \mid \langle a\rangle\phi$$

The logic is interpreted over labeled transition systems. Given an LTS $T = (S, \mathsf{Act}, \rightarrow)$, we define inductively when state $s \in S$ satisfies HML formula ϕ:

$$\begin{array}{rll} s \models \mathsf{true} & & s \not\models \mathsf{false} \\ s \models \phi_1 \wedge \phi_2 & \text{iff} & s \models \phi_1 \text{ and } s \models \phi_2 \\ s \models \phi_1 \vee \phi_2 & \text{iff} & s \models \phi_1 \text{ or } s \models \phi_2 \\ s \models [a]\phi & \text{iff} & \text{for all } t \text{ with } s \xrightarrow{a} t,\ t \models \phi \\ s \models \langle a\rangle\phi & \text{iff} & \text{there is } t \text{ with } s \xrightarrow{a} t \text{ and } t \models \phi \end{array}$$

All states satisfy true and no state satisfies false. A state satisfies $\phi_1 \wedge \phi_2$ if it satisfies both ϕ_1 and ϕ_2; it satisfies $\phi_1 \vee \phi_2$ if it satisfies either ϕ_1 or ϕ_2 (or both). The most interesting operators of HML are the *branching time modalities* $[a]$ and $\langle a\rangle$. They relate a state to its a-successors. While $[a]\phi$ holds for a state if *all* its a-successors satisfy formula ϕ, $\langle a\rangle$ holds if an a-successor satisfying formula ϕ *exists*. This is illustrated in Fig. 6. The two modalities provide a kind of universal and existential quantification over the a-successors of a state.

As introduced above, HML has as its atomic propositions only true and false. If HML is interpreted on a KTS $T = (S, \mathsf{Act}, \rightarrow, I)$ over a certain set of atomic propositions, AP, we may add atomic formulae p for each $p \in \mathsf{AP}$. These formulas are interpreted as follows:

$$s \models p \ \text{ iff } \ p \in I(s)$$

Moreover, it is sometimes useful in practice to use modalities $[A]$ and $\langle A\rangle$ that range over set of actions $A \subseteq \mathsf{Act}$ instead of single actions. They can be introduced as derived operators:

$$[A]\phi \stackrel{\text{def}}{=} \bigwedge_{a\in A} [a]\phi \qquad\qquad \langle A\rangle\phi \stackrel{\text{def}}{=} \bigvee_{a\in A} \langle a\rangle\phi \ .$$

We also write $[\,]$ for $[\mathsf{Act}]$ and $\langle\,\rangle$ for $\langle\mathsf{Act}\rangle$. A version of HML suitable for Kripke structures would provide just the modalities $[\,]$ and $\langle\,\rangle$.

Modal μ-Calculus. The modal mu-calculus [27] is a small, yet expressive branching-time temporal logic that extends Hennessy-Milner logic by fixpoint operators. Again it is defined over a set, Act, of actions. We also assume a given infinite set Var of variables. Modal μ-calculus formulas are constructed according to the following grammar:

$$\phi ::= \mathsf{true} \mid \mathsf{false} \mid [a]\phi \mid \langle a\rangle\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid X \mid \mu X \,.\, \phi \mid \nu X \,.\, \phi$$

Here, X ranges over Var and a over Act. The two *fixpoint operators*, μX and νX, bind free occurrences of variable X. We will apply the usual terminology of *free* and *bound variables* in a formula, *closed* and *open formulas*, etc. Given a least (resp., greatest) fixpoint formula, $\mu X \,.\, \phi$ ($\nu X \,.\, \phi$), we say that the μ (ν) is the formula's *parity.*

The above grammar does not permit negations in formulas. Of course, negation is convenient for specification purposes, but negation-free formulas, known as formulas in *positive form*, are more easily handled by model-checkers. In most logics, formulas with negation can easily be transformed into equivalent formulas in positive form by driving negations inwards to the atomic propositions with duality laws like

$$\neg\langle a\rangle\phi = [a]\neg\phi \quad , \quad \neg(\phi_1 \wedge \phi_2) = \neg\phi_1 \vee \neg\phi_2 \quad , \text{and} \quad \neg(\mu X \,.\, \phi) = \nu X \,.\, \neg\phi[\neg X/X]$$

But there is a small complication: We might end up with a subformula of the form $\neg X$ from which the negation cannot be eliminated. We avoid this problem if we pose this restriction on fixpoint formulas: in every fixpoint formula, $\mu X \,.\, \phi$ or $\nu X \,.\, \phi$, every free occurrence of X in ϕ must appear under an *even* number of negations. This condition ensures that the meaning of ϕ depends monotonically on X, which ensures the well-definedness of the semantics of fixpoint formulas.

Modal mu-calculus formulae are interpreted over labeled transition systems. Given an LTS $T = (S, \mathsf{Act}, \rightarrow)$, we interpret a closed formula, ϕ, as that subset of S whose states make ϕ true. To explain the meaning of open formulas, we employ *environments*, partial mappings $\rho : \mathsf{Var} \stackrel{\text{part.}}{\rightarrow} 2^S$, which interpret the free variables of ϕ by subsets of S. $\rho(X)$ represents an assumption about the set of states satisfying the formula X. The inductive definition of $\mathcal{M}_T(\phi)(\rho)$, the set of states of T satisfying the mu-calculus formula ϕ w.r.t. environment ρ, is given in Fig. 7. The meaning of a closed formula does not depend on the environment. We write, for a closed formula ϕ and a state $s \in S$, $s \models^T \phi$ if $s \in \mathcal{M}_T(\phi)(\rho)$ for one (and therefore for all) environments.

Intuitively, true and false hold for all, resp., no states, and $\wedge$ and $\vee$ are interpreted by conjunction and disjunction. As in HML, $\langle a\rangle\phi$ holds for a state s if there is an a-successor of s which satisfies ϕ, and $[a]\phi$ holds for s if all its a-successors, satisfy ϕ. The interpretation of a variable X is as prescribed by the environment. The least fixpoint formula, $\mu X \,.\, \phi$, is interpreted by the smallest subset x of S that recurs when ϕ is interpreted with the substitution of x for X. Similarly, the greatest fixpoint formula, $\nu X \,.\, \phi$, is interpreted by the largest such set. These sets can be characterized as the least and greatest fixpoints, $\mathrm{fix}_\mu F_{\phi,\rho}$

$$
\begin{aligned}
\mathcal{M}_T(\mathsf{true})(\rho) &= S \\
\mathcal{M}_T(\mathsf{false})(\rho) &= \emptyset \\
\mathcal{M}_T([a]\phi)(\rho) &= \{s \mid \forall s' : s \xrightarrow{a} s' \;\Rightarrow\; s' \in \mathcal{M}_T(\phi)(\rho)\} \\
\mathcal{M}_T(\langle a\rangle\phi)(\rho) &= \{s \mid \exists s' : s \xrightarrow{a} s' \;\wedge\; s' \in \mathcal{M}_T(\phi)(\rho)\} \\
\mathcal{M}_T(\phi_1 \wedge \phi_2)(\rho) &= \mathcal{M}_T(\phi_1)(\rho) \cap \mathcal{M}_T(\phi_2)(\rho) \\
\mathcal{M}_T(\phi_1 \vee \phi_2)(\rho) &= \mathcal{M}_T(\phi_1)(\rho) \cup \mathcal{M}_T(\phi_2)(\rho) \\
\mathcal{M}_T(X)(\rho) &= \rho(X) \\
\mathcal{M}_T(\mu X \,.\, \phi)(\rho) &= \mathrm{fix}_\mu F_{\phi,\rho} \\
\mathcal{M}_T(\nu X \,.\, \phi)(\rho) &= \mathrm{fix}_\nu F_{\phi,\rho}
\end{aligned}
$$

Fig. 7. Semantics of modal mu-calculus

and $\mathrm{fix}_\nu F_{\phi,\rho}$, of the functional $F_{\phi,\rho} : 2^S \to 2^S$ defined by

$$F_{\phi,\rho}(x) = \mathcal{M}_T(\phi)(\rho[X \mapsto x])$$

Here, $\rho[X \mapsto x]$ denotes, for a set $x \subseteq S$ and a variable $X \in \mathsf{Var}$, the environment that maps X to x and that coincides on the other variables with ρ. We now must review the basic theory of fixpoints.

3.3 Fixpoints in Complete Lattices

A convenient structure accommodating fixpoint construction is the complete lattice, i.e., a non-empty, partially ordered set in which arbitrary meets and joins exist. We assume the reader is familiar with the basic facts of complete lattices (for a thorough introduction see the classic books of Birkhoff and Grätzer [3, 21]).

We now recall some and results directly related to fixpoint theory. Let $(A, \leq_A)$ and $(B, \leq_B)$ be complete lattices and C a subset of A. C is a *chain* if it is non-empty and any two elements of C are comparable with respect to $\leq_A$. A mapping $f \in (A \to B)$ is *monotonic* if $a \leq_A a'$ implies $f(a) \leq f(a')$ for all $a, a' \in A$. The mapping is $\vee$-*continuous* if it distributes over chains, i.e., for all chains $C \subseteq A$, $f(\vee C) = \vee\{f(c) \mid c \in C\}$. The notion of $\wedge$-continuity is defined dually. Both $\vee$- and $\wedge$-continuity of a function imply monotonicity. $\perp$ and $\top$ denote the smallest and largest elements of a complete lattice. Finally, a point $a \in A$ is called a *fixpoint* of a function $f \in (A \to A)$ if $f(a) = a$. It is a *pre-fixpoint* of f if $f(a) \leq a$ and a *post-fixpoint* if $a \leq f(a)$.

Suppose that $f : A \to A$ is a monotonic mapping on a complete lattice $(A, \leq)$. The central result of fixpoint theory is the following [40, 31]:

Theorem 1 (Knaster-Tarski fixpoint theorem). *If $f : A \to A$ is a monotonic mapping on a complete lattice $(A, \leq)$, then f has a least fixpoint* $\mathrm{fix}_\mu f$

as well as a greatest fixpoint $\mathrm{fix}_\nu f$ *which can be characterized as the smallest pre-fixpoint and largest post-fixpoint respectively:*

$$\mathrm{fix}_\mu f = \bigwedge\{a \mid f(a) \leq a\} \quad \textit{and} \quad \mathrm{fix}_\nu f = \bigvee\{a \mid a \leq f(a)\} \ .$$

For continuous functions there is a "constructive" characterization that constructs the least and greatest fixpoints by iterated application of the function to the smallest (greatest) element of the lattice [26]. The iterated application of f is inductively defined by the two equations $f^0(a) = a$ and $f^{i+1}(a) = f(f^i(a))$.

Theorem 2 (Kleene fixpoint theorem). *For complete lattice* $(A, \leq)$*, if* $f : A \to A$ *is* $\vee$*-continuous, then its least fixpoint is the join of this chain:*

$$\mathrm{fix}_\mu f = \bigvee\{f^i(\bot) \mid i \geq 0\} \ .$$

Dually, if f *is* $\wedge$*-continuous, its greatest fixpoint is the meet of this chain:*

$$\mathrm{fix}_\nu f = \bigwedge\{f^i(\top) \mid i \geq 0\} \ .$$

In the above characterization we have $f^0(\bot) \leq f^1(\bot) \leq f^2(\bot) \leq \cdots$ and, dually, $f^0(\top) \geq f^1(\top) \geq f^2(\top) \geq \cdots$. As any monotonic function on a finite lattice is both $\vee$-continuous and $\wedge$-continuous, this lets us effectively calculate least and greatest fixpoints for arbitrary monotonic functions on *finite* complete lattices: The least fixpoint is found with the smallest value of i such that $f^i(\bot) = f^{i+1}(\bot)$; the greatest fixed point is calculated similarly. This observation underlies the semantic approach to model-checking described in Sect. 4.3.

The following variant of Kleene's fixpoint theorem shows that the iteration can be started with any value below the least or above the greatest fixpoint; it is not necessary to take the extremal values $\bot$ and $\top$. This observation can be exploited to speed up the fixpoint calculation if a safe approximation is already known. In particular, it can be used when model-checking nested fixpoint formulas of the same parity (see Sect. 4.3.

Theorem 3 (Variant of Kleene's fixpoint theorem). *Suppose that* $f : A \to A$ *is a* $\vee$*-continuous function on complete lattice* $(A, \leq)$*, and* $a \in A$*. If* $a \leq \mathrm{fix}_\mu f$*, then* $\mathrm{fix}_\mu f = \bigvee\{f^i(a) \mid i \geq 0\}$.

Dually, if f *is* $\wedge$*-continuous and* $\mathrm{fix}_\nu f \leq a$*, then* $\mathrm{fix}_\nu f = \bigwedge\{f^i(a) \mid i \geq 0\}$.

In the context of the modal mu-calculus, the fixpoint theorems are applied to the complete lattice, $(2^S, \subseteq)$, of the subsets of S ordered by set-inclusion. With the expected ordering on environments ($\rho \sqsubseteq \rho'$ iff $\mathrm{dom}\rho = \mathrm{dom}\rho'$ and $\rho(X) \subseteq \rho'(X)$ for all $X \in \mathrm{dom}\rho$), we can prove that $F_{\phi,\rho}$ must be monotonic. Thus, the Knaster-Tarski fixpoint theorem ensures the existence of $\mathrm{fix}_\mu F_{\phi,\rho}$ and $\mathrm{fix}_\nu F_{\phi,\rho}$ and gives us the following two equations that are often used for defining semantics of fixpoint formulas:

$$\mathcal{M}_T(\mu X \,.\, \phi)(\rho) = \bigcap\{x \subseteq S \mid \mathcal{M}_T(\phi)(\rho[X \mapsto x]) \subseteq x\}$$
$$\mathcal{M}_T(\nu X \,.\, \phi)(\rho) = \bigcup\{x \subseteq S \mid \mathcal{M}_T(\phi)(\rho[X \mapsto x]) \supseteq x\}$$

For finite-state transition systems, we have the Kleene fixpoint theorem:

$$\mathcal{M}_T(\mu X \,.\, \phi)(\rho) = \bigcup\{F^i_{\phi,\rho}(\emptyset) \mid i \geq 0\}$$
$$\mathcal{M}_T(\nu X \,.\, \phi)(\rho) = \bigcap\{F^i_{\phi,\rho}(S) \mid i \geq 0\}$$

These characterizations are central for the semantic approach to model-checking described in Sect. 4.3.

It is simple to extend the modal mu-calculus to work on Kripke transition systems instead of labeled transition systems: We allow the underlying atomic propositions $p \in \mathsf{AP}$ as atomic formulas p. The semantic clause for these formulas looks as follows:

$$\mathcal{M}_T(p)(\rho) \;=\; \{s \in S \mid p \in I(s)\} \;\; .$$

If we replace the modalities $[a]$ and $\langle a \rangle$ by $[\,]$ and $\langle\,\rangle$ we obtain a version of the modal mu-calculus that fits to pure Kripke structures as model structures.

Computational Tree Logic. Computational Tree Logic (CTL) was the first temporal logic for which an efficient model-checking procedure was proposed [11]. Its syntax looks as follows:

$$\phi ::= p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \mathsf{AU}(\phi,\psi) \mid \mathsf{EU}(\phi,\psi) \mid \mathsf{AF}(\phi) \mid \mathsf{EF}(\phi) \mid \mathsf{AG}(\phi) \mid \mathsf{EG}(\phi)$$

CTL has the six modalities $\mathsf{AU}, \mathsf{EU}, \mathsf{AF}, \mathsf{EF}, \mathsf{AG}, \mathsf{AF}$. Each takes the form QL, where Q is one of the *path quantifiers* A and E, and L is one of the *linear-time modalities* U, F, and G. The path quantifier provides a universal (A) or existential (E) quantification over the paths emanating from a state, and on these paths the corresponding linear-time property must hold. For example, the formula $\mathsf{EF}(\phi)$ is true for a state, s, if there is a path, π, starting in s on which ϕ becomes true at some later situation; i.e., the path π has to satisfy $\pi \models \mathsf{F}(\phi)$ in the sense of PLTL. In contrast, $\mathsf{AF}(\phi)$ holds if on all paths starting in s ϕ becomes true finally.

The meaning of the CTL modalities can be expressed by means of fixpoint formulas. In this sense, CTL provides useful abbreviations for frequently used formulas of the modal μ-calculus. Here are the fixpoint definitions of the U modalities:

$$\mathsf{AU}(\phi,\psi) \;\stackrel{\text{def}}{=}\; \mu X \,.\, (\psi \vee (\phi \wedge [\,]X \wedge \langle\,\rangle\mathsf{true}))$$
$$\mathsf{EU}(\phi,\psi) \;\stackrel{\text{def}}{=}\; \mu X \,.\, (\psi \vee (\phi \wedge \langle\,\rangle X)) \;\; .$$

The F modalities can easily be expressed by the U modalities

$$\mathsf{AF}(\phi) \;\stackrel{\text{def}}{=}\; \mathsf{AU}(\mathsf{true},\phi) \qquad\qquad \mathsf{EF}(\phi) \;\stackrel{\text{def}}{=}\; \mathsf{EU}(\mathsf{true},\phi) \;\; .$$

and the G modalities are easily defined as the duals of the F modalities:

$$\mathsf{AG}(\phi) \;\stackrel{\text{def}}{=}\; \neg\mathsf{EF}(\neg\phi) \qquad\qquad \mathsf{EG}(\phi) \;\stackrel{\text{def}}{=}\; \neg\mathsf{AG}(\neg\phi) \;\; .$$

By unfolding this definitions, direct fixpoint characterizations of the F and G modalities can easily be obtained.

The above described version of CTL operates on pure Kripke structures. For LTSes and KTSes it is less useful, as it does not specifying anything about the labels on arcs. We might extend CTL's modalities by relativizing them with respect to sets of actions $A \subseteq \mathsf{Act}$—the the path quantifiers consider only those paths whose actions come from A; all other paths are disregarded. In the following system, e.g.,

$$\begin{tikzcd} s \arrow[r, "a"] & t \arrow[r, "b"] & u \arrow[ll, bend left, "c"] \\ & & P \end{tikzcd}$$

the state s satisfies $\mathsf{AF}_{\{a,b\}}(P)$, as the path $\langle s, a, t, b, u\rangle$ is taken into account. But s does not satisfy $\mathsf{AF}_{\{a,c\}}(P)$ as here only the path $\langle s, a, t\rangle$ is considered. Again these modalities can be defined by fixpoint formulas, for example:

$$\mathsf{AG}_A(\phi) \stackrel{\text{def}}{=} \nu X \,.\, \phi \wedge [A]X \quad \text{and} \quad \mathsf{EF}_A(\phi) \stackrel{\text{def}}{=} \mu Y \,.\, \phi \vee \langle A\rangle Y \ .$$

4 Model Checking

4.1 Global vs. Local Model-Checking

There are two ways in which the model-checking problem can be specified:

Global model-checking problem: Given a finite model structure, M, and a formula, ϕ, determine the set of states in M that satisfy ϕ.

Local model-checking problem: Given a finite model structure, M, a formula, ϕ, *and a state s in M*, determine whether s satisfies ϕ.

While the local model-checking problem must determine modelhood of a single state the global problem must decide modelhood for all the states in the structure. Obviously, solution of the global model-checking problem comprises solution of the local problem, and solving the local model-checking problem for each state in the structure solves the global model-checking problem. Thus, the two problems are closely related, but global and local model-checkers have different applications.

For example, a classic application of model-checking is the verification of properties of models of hardware systems, where the hardware system contains many parallel components whose interaction is modeled by interleaving. The system's model structure grows exponentially with the number of parallel components, a problem known as the *state-explosion problem.* (A similar problem arises when software systems, with variables ranging over a finite domain, are analyzed—the state space grows exponentially with the number of variables.)

In such an application, local model-checking is usually preferred, because the property of interest is often expressed with respect to a specific initial state—a local model checker might inspect only a small part of the structure to decide

	Branching-time	Linear-time	Global	Local
Semantic methods	X		X	
Automata-theoretic methods		X	X	X
Tableau methods	X	X		X

Fig. 8. Classification of model-checking approaches

the problem, and the part of the structure that is not inspected need not even be constructed. Thus, local model-checking is one means for fighting the state-explosion problem.

For other applications, like the use of model-checking for data-flow analysis, one is really interested in solving the global question, as the very purpose of the model-checking activity is to gain knowledge about all the states of a structure. (For example, in Figure 2, the structure is the flow graph of the program to be analyzed, and the property specified by a formula might be whether a certain variable is "definitely live.") Such applications use structures that are rather small in comparison to those arising in verification activities, and the state-explosion problem holds less importance. Global methods are preferred in such situations.

4.2 Model-Checking Approaches

Model checking can be implemented by several different approaches; prominent examples are the *semantic approach*, the *automata-theoretic approach*, and the *tableau approach*.

The idea behind the *semantic (or iterative) approach* is to inductively compute the semantics of the formula in question on the given finite model. This generates a global model-checker and works well for branching-time logics like the modal mu-calculus and CTL. Modalities are reduced to their fixpoint definitions, and fixpoints are computed by applying the Kleene fixpoint theorem to the finite domain of the state powerset.

The *automata-theoretic approach* is mainly used for linear-time logics; it reduces the model-checking problem to an inclusion problem between automata. An automaton, A_ϕ, is constructed from formula, ϕ; A_ϕ accepts the paths satisfying ϕ. Another automaton, A_M, is constructed from the model, M, and accepts the paths exhibited by the model. M satisfies ϕ iff $L(A_M) \subseteq L(A_\phi)$. This problem can in turn be reduced to the problem of deciding non-emptiness of a product automaton which is possible by a reachability analysis.

The *tableau method* solves the local model-checking problem by subgoaling. Essentially, one tries to construct a proof tree that witnesses that the given state has the given property. If no proof tree can be found, this provides a disproof of the property for the given state. Since the tableau method intends to inspect only a small fraction of the state space, it combines well with incremental construction of the state space, which is a prominent approach for fighting state explosion.

Figure 8 presents typical profiles of the three approaches along the axes of branching- vs. linear-time and global vs. local model-checking. The classification is, of course, to be understood *cum grano salis*. Applications of the methods for other scenarios are also possible but less common. In the remainder of this section, we describe each of these approaches in more detail.

4.3 Semantic Approach

Based on the iterative characterization of fixpoints, the semantics of modal mu-calculus formulae can be effectively evaluated on finite-state Kripke transition systems. But in general, this is quite difficult, due to the potential interference between least and greatest fixpoints. As we see later in this section, the *alternated nesting* of least and greatest fixpoints forces us to introduce *backtracking* into the the fixpoint iteration procedure, which causes an exponential worst-case time complexity of iterative model checking for the full mu-calculus. Whether this is a tight bound for the model-checking problem as such is a challenging open problem.

Before we explain the subtleties of alternation, we illustrate iterative model checking for *hierarchical formulas*, which are formulas whose fixpoint subformulas never contain free variables of the other parity. In this case, the iteration process can be organized in a hierarchical fashion, giving a decision procedure whose worst-case time complexity is proportional to the size of the underlying finite-state system and the size of the formula:

1. Associate all variables belonging to greatest fixpoints to the full set of states S and all variables belonging to least fixpoints with $\emptyset$.
2. Choose a subformula, $\phi = \mu X.\phi'$ (or $\nu X.\phi'$), where ϕ' is fixpoint free, determine its semantics, and replace it by an atomic proposition A_ϕ, whose valuation is defined by its semantics.
3. Repeat the second step until the whole formula is processed.

We illustrate this procedure for $\phi \stackrel{\mathrm{def}}{=} \mathsf{AF}_{\{b\}}(\mathsf{EF}_{\{b\}}(\langle a\rangle\mathsf{true})$ and the following transition system T:

$$\begin{tikzcd} s & t \arrow[l, "b"'] \arrow[r, "b"] & u \arrow[loop above, "b"] \arrow[r, shift left, "a"] & v \arrow[l, shift left, "b"] \end{tikzcd}$$

Intuitively ϕ describes the property that from all states reachable via b-transition, an a-transition is finitely reachable along a path of b-transitions. Intuitively, u and v enjoy this property while s and t do not.

Unfolding the CTL-like operators in ϕ using the corresponding fixpoint definitions, we obtain

$$\phi \;=\; \nu X\,.\,(\mu Y\,.\,\langle a\rangle\mathsf{true} \vee \langle b\rangle Y) \wedge [b]X \;\;.$$

A hierarchical model-checker first evaluates $(\mu Y\,.\,\langle a\rangle\mathsf{true} \vee \langle b\rangle Y)$, the inner fixpoint formula: Letting ϕ_Y denote the formula $\langle a\rangle\mathsf{true} \vee \langle b\rangle Y)$, we have, for any environment ρ,

$$\mathcal{M}_T(\phi)(\rho) \;=\; \mathrm{fix}_\mu F_{\phi_Y,\rho} \;\;.$$

Now, by the Kleene fixpoint theorem this fixpoint formula can be calculated by iterated application of $F_{\phi_Y,\rho}$ to the smallest element in 2^S, $\emptyset$. Here are the resulting approximations:

$$\begin{aligned}
F^0_{\phi_Y,\rho}(\emptyset) &= \emptyset \\
F^1_{\phi_Y,\rho}(\emptyset) &= \mathcal{M}(\langle a\rangle\mathsf{true})(\rho[Y \mapsto \emptyset]) \cup \mathcal{M}(\langle b\rangle Y)(\rho[Y \mapsto \emptyset]) \\
&= \{u\} \cup \emptyset = \{u\} \\
F^2_{\phi_Y,\rho}(\emptyset) &= F_{\phi_Y,\rho}(\{u\}) \\
&= \mathcal{M}(\langle a\rangle\mathsf{true})(\rho[Y \mapsto \{u\}]) \cup \mathcal{M}(\langle b\rangle Y)(\rho[Y \mapsto \{u\}]) \\
&= \{u\} \cup \{t,u,v\} = \{t,u,v\} \\
F^3_{\phi_Y,\rho}(\emptyset) &= F_{\phi_Y,\rho}(\{t,u,v\}) \\
&= \mathcal{M}(\langle a\rangle\mathsf{true})(\rho[Y \mapsto \{t,u,v\}]) \cup \mathcal{M}(\langle b\rangle Y)(\rho[Y \mapsto \{t,u,v\}]) \\
&= \{u\} \cup \{t,u,v\} = \{t,u,v\} \ .
\end{aligned}$$

Thus, $\{t,u,v\}$ is the meaning of $\mu X \,.\, \phi_X$ in any environment. Next, the hierarchical model-checker evaluates the formula $\phi' \stackrel{\text{def}}{=} \nu X \,.\, p_Y \wedge [b]X$, where p_Y is a new atomic proposition that holds true for the states t, u, and v. Again, this is done by iteration that starts this time with $S = \{s,t,u,v\}$, as we are confronted with a greatest fixpoint. Let ϕ_X denote the formula $p_Y \wedge [b]X$. The iteration's results look as follows:

$$\begin{aligned}
F^0_{\phi_X,\rho}(S) &= S \\
F^1_{\phi_X,\rho}(S) &= \mathcal{M}(p_Y)(\rho[X \mapsto S]) \cap \mathcal{M}([b]X)(\rho[X \mapsto S]) \\
&= \{t,u,v\} \cap S = \{t,u,v\} \\
F^2_{\phi_Y,\rho}(S) &= F_{\phi_Y,\rho}(\{t,u,v\}) \\
&= \mathcal{M}(p_Y)(\rho[X \mapsto \{t,u,v\}]) \cap \mathcal{M}([b]X)(\rho[X \mapsto \{t,u,v\}]) \\
&= \{t,u,v\} \cap \{s,u,v\} = \{u,v\} \\
F^3_{\phi_Y,\rho}(S) &= F_{\phi_Y,\rho}(\{u,v\}) \\
&= \mathcal{M}(p_Y)(\rho[X \mapsto \{u,v\}]) \cap \mathcal{M}([b]X)(\rho[X \mapsto \{u,v\}]) \\
&= \{t,u,v\} \cap \{s,u,v\} = \{u,v\} \ .
\end{aligned}$$

The model-checking confirms our expectation that just states u and v have property ϕ.

In the above example, the inner fixpoint formula, $(\mu Y \,.\, \langle a\rangle\mathsf{true} \vee \langle b\rangle Y)$, does not use the variable introduced by the outer fixpoint operator, X. Therefore, its value does not depend on the environment, ρ. This is always the case for CTL-like formulas and enables the hierarchical approach to work correctly. If, however, the inner fixpoint depends on the variable introduced further outwards, we must—at least in principle—evaluate the inner fixpoint formula again and again in each iteration of the outer formula. Fortunately, if the fixpoint formulas have the same parity, i.e., they are either both least fixpoint formulas or both greatest fixpoint formulas, we can avoid the problem and correctly compute the values of the inner and outer formulas simultaneously, because the value of a fixpoint formula depends monotonically on the value of its free variables and the iterations of both formulas proceed in the same direction.

In the case of mutual dependencies between least and greatest fixpoints, however, the iterations proceed in opposite directions, which excludes a simple monotonic iteration process. Such formulas are called *alternating fixpoint formulas*. They require backtracking (or resets) in the iteration process. The following is a minimal illustrative example.

Example 1. Consider the formula $\psi \stackrel{\text{def}}{=} \nu Z \,.\, \mu Y \,.\, (\langle b \rangle Z \vee \langle a \rangle Y)$, which intuitively specifies that there is a path consisting of a- and b-steps with infinitely many b-steps. We would like to check it for the following LTS:

$$\begin{tikzcd} s & t \arrow[l, "b"'] \arrow[loop right, "a"] \end{tikzcd}$$

Here are the results of the iterations for the outer fixpoint variable Z with the nested iterations for Y:

Iteration for Z	1			2		3	
Assumption for Z	$\{s,t\}$			$\{t\}$		$\emptyset$	
Iterations for Y	$\emptyset$	$\{t\}$	$\{t\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Thus, we correctly calculate that neither s nor t satisfies ψ.

If, however we do not reset the iterations of Y to $\emptyset$ in each iteration of Z but simply start Y's iterations with the old approximations, we produce the wrong result $\{t\}$ that "stabilizes" itself:

Iteration for Z	1			2	
Assumption for Z	$\{s,t\}$			$\{t\}$	
Iterations for Y	$\emptyset$	$\{t\}$	$\{t\}$	$\{t\}$	$\{t\}$

Due to the nested iteration, the complexity of model-checking formulas with alternation is high. A careful implementation of the iterative approach leads to an asymptotic worst-case running time of $O((|T| \cdot |\phi|)^{ad})$ [13], where $|T|$ is the number of states and transitions in the LTS (or KS or KTS) T and $|\phi|$ is the size of the formula (measured, say, by the number of operators). *ad* refers to the *alternation depth* of the formula, which essentially is the number of non-trivial alternations between least and greatest fixpoints. (The alternation depth of an alternation-free formula is taken to be 1.) While the precise definition of alternation depth is not uniform in the literature, all definitions share the intuitive idea and the above stated model-checking complexity. By exploiting monotonicity, a better time-complexity can be achieved but at the cost of a tremendous storage requirement [32]. It is a challenging open problem to determine the precise complexity of μ-calculus model-checking; it might well turn out to be polynomial (it is known to be in the intersection of the classes NP and co-NP)!

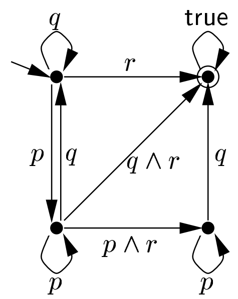


Fig. 9. An automaton corresponding to $\phi = \mathsf{U}(\mathsf{U}(P,Q),R)$

Alternation-free formulas can be checked efficiently in time $O((s+t)\cdot|\phi|)$. This holds in particular for all CTL-like formulas as they unfold to alternation-free fixpoint formulas.

4.4 Automata-Theoretic Approach

For easy accessibility, we illustrate the automata-theoretic approach for checking PLTL formulas on maximal finite paths in Kripke structures: Given a Kripke structure (S,R,I), a state s in S, and a PLTL formula ϕ, we say that s_0 *satisfies* ϕ if any maximal finite path $\pi = \langle s_0, s_1, \ldots, s_n\rangle$ satisfies ϕ.

A path π as above can be identified with the finite word $w_\pi = \langle I(s_0), I(s_1), \ldots, I(s_n)\rangle$ over the alphabet 2^{AP}. Note that the letters of the alphabet are subsets of the set of atomic propositions. In a straightforward way, validity of PLTL formulae can also be defined for such words. Now, a PLTL formula, ϕ, induces a language of words over 2^{AP} containing just those words that satisfy ϕ. For any PLTL formula, the resulting language is regular, and there are systematic methods for constructing a finite automaton, A_ϕ, that accepts just this language. (This automaton can also be used to check satisfiability of ϕ; we merely check whether the language accepted by the automaton is non-empty, which amounts to checking whether a final state of the automaton is reachable from the initial state.) In general, the size of A_ϕ grows exponentially with the size of ϕ but is often small in practice.

Example 2. Consider the formula $\phi = \mathsf{U}(\mathsf{U}(P,Q),R)$. The corresponding automaton A_ϕ is shown in Fig. 9. We adopt the convention that an arrow marked with a lower case letter represents transitions for all sets containing the proposition denoted by the corresponding upper case letter. An arrow marked with p, for example, represents transitions for the sets $\{P\},\{P,Q\},\{P,R\}$ and, $\{P,Q,R\}$. Similarly a conjunction of lower case letters represents transitions containing both corresponding upper case propositions. For example, an arrow marked with $p\wedge q$ represents transitions marked with $\{P,Q\}$ and $\{P,Q,R\}$.

It is easy to construct from the Kripke structure, K, a finite automaton, A_K, accepting just the words w_π corresponding to the maximal finite paths starting

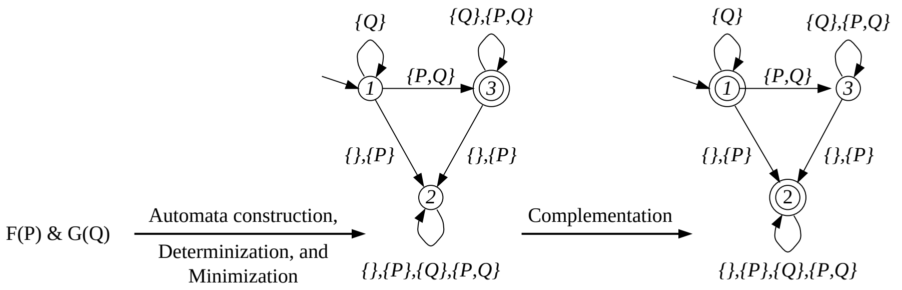


Fig. 10. A formula and the corresponding automata

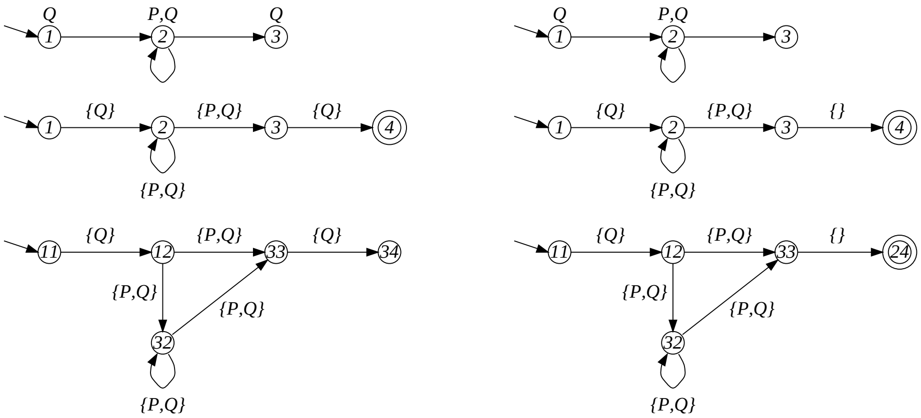


Fig. 11. Two Kripke structures, the corresponding automata, and their products with the formula automaton from Fig. 10. Model-checking succeeds for the left Kripke structure and fails for the right one.

in s: It is given by $A_K = (S \cup \{s_f\}, 2^{\mathsf{AP}}, \delta, s, \{s_f\})$, where s_f is a new state and is the only final state in the automaton, and

$$\delta = \{(s, I(s), t) \mid (s,t) \in R\} \cup \{(s, I(s), s_f) \mid s \text{ is final in } K\}$$

Here, a state s is called *final* in K if it has no successor, i.e., there is no t with $(s,t) \in R$.

To answer the model-checking question amounts to checking whether $L(A_K) \subseteq L(A_\phi)$. This is equivalent to $L(A_K) \cap L(A_\phi)^c = \emptyset$. The latter property can effectively be checked: Finite automata can effectively be complemented, the product automaton of two automata describes the intersection of the languages of the two automata, and the resulting automaton can effectively be checked for emptiness.

Example 3. Let us illustrate this approach for the formula $\phi \stackrel{\text{def}}{=} \mathsf{F}(P) \wedge \mathsf{G}(Q)$ over $\mathsf{AP} = \{P, Q\}$. Figure 10 shows the automata generated from ϕ. The first automaton in the figure is A_ϕ, the automaton that accepts the language over $2^{\{P,Q\}}$ corresponding to ϕ. Complementation of a finite automaton is particularly simple if the automaton is deterministic and fully defined, because the automaton has for any state exactly one transition for any input symbol. An automaton can be made deterministic by the well-known power-set construction; full definedness can be obtained by adding a (non-final) state that "catches" all undefined transitions. The automaton A_ϕ shown in Fig. 10 has been made deterministic and is fully defined; to keep it small, it has also been minimized. Such transformations preserve the language accepted by the automaton and are admissible and commonly applied. The second automaton in Fig. 10, to which we refer by A_ϕ^c in the following, accepts the complement of the language corresponding to ϕ. As A_ϕ has been made deterministic and is fully defined, it can easily be complemented by exchanging final and non-final states.

In Fig. 11, we show two rooted Kripke structures K, the corresponding Automata A_K, and the product automata $A_\phi^c \times A_K$.[1] In order to allow easy comparison, the states in $A_\phi^c \times A_K$ have been named by pairs ij; i indicates the corresponding state in A_ϕ^c and j the corresponding state in A_K.

It is intuitively clear that the left Kripke structure satisfies ϕ. An automata-theoretic model-checker would analyze whether the language of $A_\phi^c \times A_K$ is empty. Here, it would check whether a final state is reachable from the initial state. It is easy to see that no final state is reachable; this means that K indeed satisfies ϕ.

Now, consider the rooted right Kripke structure in Fig. 11. As its final state does not satisfy the atomic proposition, Q, the Kripke structure does not satisfy ϕ—a final state is reachable from the initial state in the product automaton.

A similar approach to model-checking can be used in the more common case of checking satisfiability of infinite paths [42]. In this case, automata accepting languages of infinite words, like Büchi or Muller automata [41], are used instead of automata on finite words. Generation of the automata A_ϕ and A_K as well as the automata-theoretic constructions (product construction, non-emptiness check) are more involved, but nevertheless, the basic approach remains the same. PLTL model-checking is in general PSPACE-complete [17]; the exponential blow-up in the construction of A_ϕ is unavoidable.

The main applications of the automata-theoretic approach are linear-time logics as languages consisting of words. The approach can also be applied, however, to branching-time logics by using various forms of tree automata.

4.5 Tableau Approach

The *tableau approach* addresses the local-model checking problem: For a model, $\mathcal{M}$, and property, ϕ, we wish to learn whether $s \models^{\mathcal{M}} \phi$ holds for just the one state, s—global information is unneeded. We might attack the problem by a

[1] Strictly speaking, only that part of the state space is shown that is reachable from the initial state, and not the full product automaton.

search of the state space accessible from s, driving the search by decomposing ϕ. We write our query as $s \vdash_\Delta \phi$ (taking the $\mathcal{M}$ as implicit) and use subgoaling rules, like the following, to generate a proof search, a *tableau*. For the moment, the rules operate on just the Hennessy-Milner calculus; the subscript, Δ, will be explained momentarily:

$$\frac{s \vdash_\Delta \phi_1 \wedge \phi_2}{s \vdash_\Delta \phi_1 \quad s \vdash_\Delta \phi_2} \qquad \frac{s \vdash_\Delta \phi_1 \vee \phi_2}{s \vdash_\Delta \phi_1} \qquad \frac{s \vdash_\Delta \phi_1 \vee \phi_2}{s \vdash_\Delta \phi_2}$$

$$\frac{s \vdash_\Delta [a]\phi}{s_1 \vdash_\Delta \phi \cdots s_n \vdash_\Delta \phi} \text{ if } \{s_1, \ldots, s_n\} = \{s' \mid s \xrightarrow{a} s'\} \qquad \frac{s \vdash_\Delta \langle a \rangle \phi}{s' \vdash_\Delta \phi} \text{ if } s \xrightarrow{a} s'$$

A tableau for a Hennessy-Milner formula must be finite. We say that the tableau *succeeds* if all its leaves are *successful*, that is, they have form *(i)* $s \vdash_\Delta \mathsf{true}$, or *(ii)* $s \vdash_\Delta [a]\phi$ (which implies there are no a-transitions from s).

It is easy to prove that a succesful tableau for $s \vdash_\Delta \phi$ implies $s \models^{\mathcal{M}} \phi$; conversely, if there exists no successful tableau, we correctly conclude that $s \not\models^{\mathcal{M}} \phi$. (Of course, a proof search that implements and/or subgoaling builds just one "meta-tableau" to decide the query.)

When formulas in the modal-mu calculus are analyzed by tableau, there is the danger of infinite search. But for finite-state models, we can limit search due to the semantics of the fixed-point operators: Say that $s \vdash_\Delta \mu X.\phi_X$ subgoals to $s \vdash_\Delta X$. We conclude that the latter subgoal is unsuccessful, because

$$s \models^{\mathcal{M}} \mu X.\phi_X \text{ iff } s \models^{\mathcal{M}} \bigvee_{i \geq 0} X_i, \text{ where } \begin{array}{l} X_0 = \mathsf{false} \\ X_{i+1} = \phi_{X_i} \end{array}$$

That is, the path in the tableau from $s \vdash_\Delta \mu X.\phi_X$ to $s \vdash_\Delta X$ can be unfolded an arbitrary number of times, generating the X_i formulas, all of which subgoal to X_0, which fails.

Dually, a path from $s \vdash_\Delta \nu X.\phi_X$ to $s \vdash_\Delta X$ succeeds, because

$$s \models^{\mathcal{M}} \nu X.\phi_X \text{ iff } s \models^{\mathcal{M}} \bigwedge_{i \geq 0} X_i, \text{ where } \begin{array}{l} X_0 = \mathsf{true} \\ X_{i+1} = \phi_{X_i} \end{array}$$

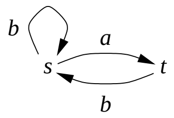
As suggested by Stirling and Walker [37], we analyze the fixed-point operators with unfolding rules, and we terminate a search path when the same state, fixed-point formula pair repeat. Of course, we must preserve the scopes of nested fixed points, so each time a fixed-point formula is encountered in the search, we introduce a unique label, $\mathcal{U}$, to denote it. The labels and the formulas they denote are saved in an environment, Δ.

Here the rules for μ; the rules for ν work in the same way:

$$\frac{s \vdash_\Delta \mu X.\phi_X}{s \vdash_{\Delta'} \mathcal{U}} \quad \text{where } \Delta' = \Delta + [\mathcal{U} \mapsto \mu X.\phi_X] \text{ and } \mathcal{U} \text{ is fresh for } \Delta$$

$$\frac{s \vdash_\Delta \mathcal{U}}{s \vdash_\Delta \phi_{\mathcal{U}}} \quad \text{where } \Delta(\mathcal{U}) = \mu X.\phi_X. \text{ Important: See note below.}$$

Transition system:



Let :

$\phi = \nu X.\phi'_X \wedge [b]X$

$\phi'_X = \mu Y.\langle a\rangle \mathsf{true} \vee \langle b\rangle Y$

$\Delta_1 = [\mathcal{U}_1 \mapsto \phi]$

$\Delta_2 = \Delta_1 + [\mathcal{U}_2 \mapsto \phi'_{\mathcal{U}_1}]$

Tableau for $s \vdash_\emptyset \phi$:

$$
\begin{array}{ccc}
 & s \vdash_\emptyset \phi & \\
 & s \vdash_{\Delta_1} \mathcal{U}_1 & \\
 & s \vdash_{\Delta_1} \phi'_{\mathcal{U}_1} \wedge [b]\mathcal{U}_1 & \\
s \vdash_{\Delta_1} \phi'_{\mathcal{U}_1} & & s \vdash_{\Delta_1} [b]\mathcal{U}_1 \\
s \vdash_{\Delta_2} \mathcal{U}_2 & & s \vdash_{\Delta_1} \mathcal{U}_1 \\
s \vdash_{\Delta_2} \langle a\rangle \mathsf{true} \vee \langle b\rangle \mathcal{U}_2 & & \\
s \vdash_{\Delta_2} \langle a\rangle \mathsf{true} & & \\
t \vdash_{\Delta_2} \mathsf{true} & &
\end{array}
$$

Fig. 12. Transition system and model check by tableau

Note: The second rule can be applied only when $s \vdash_{\Delta'} \mathcal{U}$ has *not* already appeared as an ancestor goal. This is how proof search is terminated.

A leaf of form, $s \vdash_\Delta \mathcal{U}$, is successful iff $\Delta(\mathcal{U}) = \nu X.\phi_X$. Figure 12 shows a small transition system and proof by tableau that its state, s, has the property, $\nu X.(\mu Y.\langle a\rangle \mathsf{true} \vee \langle b\rangle Y) \wedge [b]X$, that is, an a-transition is always finitely reachable along a path of b-transitions. The tableau uses no property of state, t.

The tableau technique is pleasant because it is elegant, immune to the troubles of alternating fixpoints, and applicable to both branching-time and linear-time logics.

5 Conclusion

One of the major problems in the application of model checking techniques to practical verification problems is the so-called *state explosion problem*: models typically grow exponentially in the number of parallel components or data elements of an argument system. This observation has led to a number of techniques for tackling this problem [39, 14].

Most rigorous are *compositional* methods [2, 10, 23], which try to avoid the state explosion problem in a divide an conquer fashion. *Partial order methods* limit the size of the models representation by suppressing unnecessary interleavings, which typically arise as a result of the serialization during the model construction of concurrent systems [19, 43, 20]. *Binary Decision Diagram*-based codings, todays industrially most successful technique, allow a polynomial system representation, but may explode in the course of the model checking process [4, 6, 18]. All these techniques have their own very specific profiles . Exploring these profiles is one of the current major research topics.

All these techniques can be accompanied by *abstraction*: depending on the particular property under investigation, systems may be dramatically reduced by suppressing details that are irrelevant for verification, see, e.g., [15, 9, 22, 30].

In this article we have focused on finite model structures, but recent research shows that effective model-checking is possible also for certain classes of finitely presented infinite structures. Work in this direction falls in two categories: First, continuous variables have been added to finite structures. This work was motivated by considerations on verified design of embedded controllers. Timed systems have found much attention (Alur and Dill's work on timed automata [1] is a prominent example) but also more general classes of hybrid systems have been considered. A study of this work could start with [38] where besides a general introduction three implemented systems, HyTech, Kronos, and Uppaal, are described. Second, certain classes of discrete infinite systems have been studied that are generated by various types of grammars in various ways. The interested reader is pointed to the surveys [8, 7] that also contain numerous references.

References

1. R. Alur and D. L. Dill, A theory of timed automata. *Theoretical Computer Science* 126 (1994) 183–235.
2. H. Andersen, C. Stirling, and G. Winskel, A compositional proof system for the modal mu-calculus. In *Proc. 9th LICS*. IEEE Computer Society Press, 1994.
3. G. Birkhoff, *Lattice Theory, 3d edition*. Amer. Math. Soc., 1967.
4. R. Bryant, Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computation*, 8(35), 1986.
5. R. Bull and K. Segerberg, Basic Modal Logic. In *Handbook of Philosophical Logic, Vol. 2*, D. Gabbay and F. Guenther, eds., Kluwer, Dortdrecht, 1994, pp. 1-88.
6. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th LICS*. IEEE Computer Society Press, 1990.
7. O. Burkart, D. Caucal, F. Moller, and B. Steffen, Verification on infinite structures. In *Handbook of Process algebra*, Jan Bergstra, Alban Ponse, and Scott Smolka, eds., Elsevier, to appear.
8. O. Burkart and J. Esparza, More infinite results. *Electronic Notes in Theoretical Computer Science* 6 (1997).
   URL: `http://www.elsevier.nl/locate/entcs/volume6.html`.
9. K. Čerāns, J.C. Godesken, and K.G. Larsen, Timed modal specification – theory and tools. In *Computer Aided Verification (CAV'93)*, C. Courcoubetis, ed., Lecture Notes in Computer Science 697, Springer, 1993, pp. 253–267.
10. E. Clarke, D. Long, and K. McMillan, Compositional model checking. In *Proc. 4th LICS*. IEEE Computer Society Press, 1989.
11. E. M. Clarke, E. A. Emerson, and A. P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* 8 (1996) 244–263.
12. E. M. Clarke, O. Grumberg, and D. Long, Verification tools for finite-state concurrent systems. In *A Decade of Concurrency: Reflections and Perspectives*, J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, eds., Lecture Notes in Computer Science 803, Springer, 1993, pp. 124-175.
13. R. Cleaveland, M. Klein, and B. Steffen, Faster model checking for the modal mu-calculus. In *Computer Aided Verification (CAV'92)*, G. v. Bochmann and D. K. Probst, eds., Lecture Notes in Computer Science 663, 1992, pp. 410–422.
14. R. Cleaveland, Pragmatics of Model Checking. *Software Tools for Technology Transfer* 2(3), 1999.

15. P. Cousot and R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings 4th POPL*, Los Angeles, California, January 1977.
16. D. van Dalen, *Logic and Structure, 3d edition.* Springer, Berlin, 1994.
17. E. A. Emerson, Temporal and modal logic. In *Handbook of Theoretical Computer Science, Vol B.* J. van Leeuwen, ed., Elsevier Science Publishers B.V., 1990, pp. 995–1072.
18. R. Enders, T. Filkorn, and D. Taubner, Generating BDDs for symbolic model checking in CCS. In *Computer Aided Verification (CAV'91)*, K. G. Larsen and A. Skou, eds., Lecture Notes in Computer Science 575, Springer, pp. 203–213.
19. P. Godefroid and P. Wolper, Using partial orders for the efficient verification of deadlock freedom and safety properties. n *Computer Aided Verification (CAV'91)*, K. G. Larsen and A. Skou, eds., Lecture Notes in Computer Science 575, Springer, pp. 332–342.
20. P. Godefroid and D. Pirottin, Refining dependencies improves partial-order verification methods. In *Computer Aided Verification (CAV'93)*, C. Courcoubetis, ed., Lecture Notes in Computer Science 697, Springer pp. 438–449.
21. G. Grätzer, *General Lattice Theory.* Birkhäuser Verlag, 1978.
22. S. Graf and C. Loiseaux, Program Verification using Compositional Abstraction. In *Proceedings FASE/TAPSOFT*, 1993.
23. S. Graf, B. Steffen, and G. Lüttgen, Compositional minimization of finite state systems using interface specifications. *Formal Aspects of Computing*, 8:607–616, 1996.
24. M. C. B. Hennessy and R. Milner, Algebraic laws for nondeterminism and concurrency. *Journal of the ACM* 32 (1985) 137–161.
25. G. Hughes and M. Cresswell. *An Introduction to Modal Logic.* Methuen, London, 1972.
26. S. Kleene, *Introduction to Metamathematics.* D. van Nostrand, Princeton, 1952.
27. D. Kozen, Results on the propositional mu-calculus, *Theoretical Computer Science*, 27 (1983) 333–354.
28. Kripke, S. A completeness theorem in modal logic. *J. Symbolic Logic* 24 (1959) 1–14.
29. Kripke, S. Semantical considerations on modal logic. *Acta Philosophica Fennica* 16 (1953) 83–94.
30. K. G. Larsen, B. Steffen, and C. Weise. A constraint oriented proof methodology based on modal transition systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, E. Brinksma, W. R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen, eds, Lecture Notes of Computer Science 1019, Springer, pp. 17–40.
31. J.-L. Lassez, V. L. Nguyen, and E. A. Sonenberg, Fixed point theorems and semantics: A folk tale. *Information Processing Letters* 14 (1982) 112–116.
32. D. E. Long, A. Browne, E. M. Clarke, S. Jha, and W. R. Marrero, An improved algorithm for the evaluation of fixpoint expressions. In *Computer Aided Verification (CAV'94)*, David L. Dill, ed., Lecture Notes in Computer Science 818, Springer pp. 338–349.
33. Robin Milner, *Communication and Concurrency.* Prentice Hall, 1989.
34. J. P. Queille and J. Sifakis, Specification and verification of concurrent systems in CESAR. In *Proc. 5th Internat. Symp. on Programming*, M. Dezani-Ciancaglini and U. Montanari, eds., Lecture Notes in Computer Science 137, Springer, 1982.

35. D. Schmidt and B. Steffen, Program analysis as model checking of abstract interpretations. In *Static Analysis (SAS'98)*, Giorgio Levi, ed., Lecture Notes in Computer Science 1503, Springer, 1998, 351–380.
36. C. Stirling, Modal and temporal logics. In *Handbook of Logic in Computer Science* S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, eds., Clarendon Press, 1992, pp 477–563.
37. C. Stirling and D. Walker, Local model checking in the modal mu-calculus, *Proc. TAPSOFT '89*, J. Diaz and F. Orejas, eds., Lecture Notes in Computer Science 351, Springer, 1989, pp. 369–383.
38. Special section on timed and hybrid systems, *Software Tools for Technology Transfer* 1 (1997) 64–153.
39. Special section on model checking, *Software Tools for Technology Transfer* 2/3 (1999).
40. A. Tarski, A lattice-theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics* 5 (1955) 285–309.
41. W. Thomas, Automata on infinite objects. In *Handbook of Theoretical Computer Science, Vol B.* J. van Leeuwen, ed., Elsevier Science Publishers B.V., 1990, pp. 133–191.
42. M. Y. Vardi and P. Wolper, Reasoning about infinite computations. *Information and Computation* 115 (1994) 1–37.
43. A. Valmari, On-the-fly verification with stubborn sets. In *Computer Aided Verification (CAV'93)*, C. Courcoubetis, ed., Lecture Notes in Computer Science 697, Springer, pp. 397–408.