Combining Supervisor Synthesis and Model Checking

ROBERTO ZILLER
University of Karlsruhe
and
KLAUS SCHNEIDER
University of Kaiserslautern

Model checking and supervisor synthesis have been successful in solving different design problems related to discrete systems in the last decades. In this paper, we analyze some advantages and drawbacks of these approaches and combine them for mutual improvement. We achieve this through a generalization of the supervisory control problem proposed by Ramadge and Wonham. The objective of that problem is to synthesize a supervisor which constrains a system's behavior according to a given specification, ensuring controllability and coaccessibility. By introducing a new representation of the solution using systems of μ-calculus equations, we are able to handle these two conditions separately and thus to exchange the coaccessibility requirement by any condition that could be used in model checking. Well-known results on μ-calculus model checking allow us to easily assess the computational complexity of any generalization. Moreover, the model checking approach also delivers algorithms to solve the generalized synthesis problem. We include an example in which the coaccessibility requirement is replaced by fairness constraints. The paper also contains an analysis of related work by several authors.

Categories and Subject Descriptors: D.2.10 [**Software Engineering**]: Design; D.2.4 [**Software Engineering**]: Program Verification

General Terms: Design and Verification

Additional Key Words and Phrases: Ramadge–Wonham, supervisor synthesis, model checking

1. INTRODUCTION

Many embedded systems used in safety-critical applications consist of reactive real-time controllers, whose design requires automatic tools for improving efficiency and for preventing human errors. Due to the wide application field of discrete, event-driven systems, there has been a great deal of interest in developing adequate design techniques to handle discrete space. Although this

Authors' address: R. Ziller, Institute for Computer Design and Fault Tolerance—Prof. Dr. D. Schmid, University of Karlsruhe, 76128 Karlsruhe, Germany; email: ziller@informatik.uni-karlsruhe.de; K. Schneider, Department of Computer Science, University of Kaiserslautern, Germany.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2005 ACM 1539-9087/05/0500-0331 $5.00

has attracted the attention of computer scientists and engineers with different background and motivation, many problems still challenge the scientific community as they remain very hard to solve. As opposed to the world of continuous systems, where a large majority of problems can be modeled and solved by differential equations, there is no method in the discrete world that is likely to evolve into an analogous counterpart.

This paper is about two methods that have proven successful in working with many discrete systems over the last two or three decades, namely formal verification (in the form of model checking) and supervisor synthesis. Formal verification has its origins in the work by Floyd [1967] and Hoare [1969]. Temporal logics were first used for verification by Burstall [1974] and Kröger [1977] for sequential systems and by Pnueli [1977] for concurrent systems. With these methods, it was possible to prove the correctness of computer programs. However, the proofs had to be constructed by hand. For finite-state systems, this difficulty was overcome by automatic *model-checking* techniques introduced by Emerson and Clarke [1980] and Clarke et al. [1986, 1993, 1999]. Model checking bases mainly on temporal logics and as such is the work of mathematicians and computer scientists. Supervisor synthesis, on the other hand, originated in a group of control engineers around W. M. Wonham, with the intent of improving the design process of discrete event systems [Ramadge and Wonham 1987; Wonham 2004]. Despite some interaction between the two approaches [Antoniotti and Mishra 1996; Arnold et al. 2003; Asarin et al. 1995; de Alfaro et al. 2001; Hoffmann and Wong-Toi 1992; Jiang and Kumar 2001], they evolved mostly independently. In this paper, we point out some strengths and weaknesses of both sides and introduce a generalized approach to supervisor synthesis that allows interaction with model-checking methods through a unique model for the system under construction. We also point out how both sides benefit from this approach and in fact complement each other in some important open problems.

Modern verification methods allow designers to check a given specification for a controller. In order to do this, the designer has to build a model of the controlled system (usually in the form of a finite transition system) and to write down a temporal logics expression for each property to be verified. A model-checking algorithm then either confirms that a property is satisfied or else delivers a counterexample for it. In the latter case, the designer modifies the specification and ideally reaches a correct design after some iterations. As implied by this description, verification methods are not concerned with the process of creating the specification in the first place, leaving the designer without support for a task whose degree of difficulty should not be underestimated. For example, the fact that the designer of a communication protocol knows that loss and duplication of messages should be avoided is not very useful when it comes to writing down the communication rules that make up a first attempt for the protocol, neither to guide modifications when a counterexample is found. Ideally, the specification itself should be generated by a tool that takes these kinds of high-level requirements and either outputs a correct specification or rejects them if they cannot be implemented. However, no such approach can be found in the verification literature.

A solution that takes the latter approach is offered by the Ramadge–Wonham (RW) framework. Finite automata are used to model both the physically possible behavior of a system and a specification for its desired behavior. The first automaton thus describes both wanted and unwanted features, while the second one is intended to restrict the possible actions to something useful. The framework takes into account that not every event in the system can be prevented from occurring. Technically, this is achieved by distinguishing *controllable events*, for example, commands issued by a controller to turn things on or off, or to send a message, from *uncontrollable events*, for example, timeouts, the loss of a message, and sensor or alarm signals. Because there is no limitation on what the designer may wish, it is of course possible to come up with specifications that cannot be enforced by any controller. This could happen through a rather unreasonable specification like saying that a lossy channel should never lose a message, but also in much more subtle cases, for example, when prohibiting a message from being lost happens as an unanticipated consequence of an action that is, apart from that error, well thought out. It can also happen when a specification is written at a higher level, for in that case the designer is not concerned with the details by hypothesis. For example, in the case of the communication protocol, the designer could express the wish to keep the order of the messages sent without wanting to take care about all the possible message sequences. The strength of the Ramadge–Wonham framework lies precisely in the ability to use such specifications. When a given specification is not implementable, a synthesis procedure takes both automata (possible and desired behaviors) and computes a third automaton from them, which corresponds to the largest solution that does not violate the desired behavior. The idea is that the latter can replace the original specification, provided that it has not been excessively restricted by the synthesis process. This is an important advantage over the model-checking approach, where the transition relation of the system to be verified has to be given before the formal process begins. While model checking only allows asking questions about a transition relation that has been put together by the designer, the synthesis approach generates the transition relation of the system to be built.

The synthesis process also has a drawback, namely the fact that it is not always obvious whether the resulting restricted behavior is still useful (in an extreme case, the resulting protocol in our example could simply never send a message in order to never lose one, which would be useless). In theory, the formulation of the *supervisory control problem* (see Section 2) offers a way to specify a minimally acceptable behavior that could be used to reject an incomplete solution, but then the problem is transferred to finding out what that minimal behavior should be. If this were easy to determine, it could already be regarded as a solution, which we do not have by hypothesis. In practical terms, the Ramadge–Wonham framework does not include a method to answer the question whether the behavior obtained through the synthesis process is acceptable.

In this paper, we argue that the drawbacks exposed above for model checking and supervisor synthesis can be alleviated by combining them. First, synthesis can be used to produce a specification in the cases where the description of

what is wanted can only be made in a form that is not immediately useful for implementation. One possibility is to use verification right after the synthesis process to accept or reject the result. Better still, we generalize the supervisory control problem so that the properties to be verified can already be considered during synthesis. Hence, the designer is also given the alternative to include any property which would be checked after the synthesis process into the synthesis process itself, thereby making postsynthesis verification superfluous. Moreover, while synthesis methods have been dealing mostly with safety and liveness properties, verification methods enable us to also consider fairness.

Since synthesis is based on finite automata and language theory, while model checking uses Kripke structures and temporal logics, there is a representation incompatibility to overcome. We achieve this by introducing a translation of finite automata into a Kripke structure which allows doing synthesis and verification on a single model of the system. The synthesis algorithm is translated into the μ-calculus and then generalized by substituting a fixed coaccessibility requirement by any μ-calculus condition chosen by the designer. Moreover, the μ-calculus equations are well suited for implementation with symbolic methods, which effectively reduce the state explosion problem in many practical cases [Burch et al. 1992]. The chosen approach also enables us to use well-known results on the complexity of μ-calculus model checking [Cleaveland et al. 1992; Emerson and Lei 1986b; Emerson et al. 1993; Long et al. 1994; Seidl 1996] to derive the computational complexity of any generalization. Finally, the μ-calculus description can be understood by tools originally intended for verification, which are hereby extended to also handle supervisor synthesis.

Related work [Thistle and Wonham 1994a, 1994b] uses Büchi and Rabin automata to allow fairness specifications in supervisor synthesis and to derive results on controllability analogous to those of the basic RW framework. However, the method does not consider terminating computations and therefore does not subsume the basic model. Besides, translating an informal description of a system into ω-automata is not a trivial task, and the absence of directions to support that translation leaves a gap between theoretical possibilities and application. Both the basic model and the extension by Thistle and Wonham become special cases of our generalized setting and are now treated in a uniform way.

The approach followed by Arnold et al. [2003] can handle both terminating and nonterminating computations. A similar, but less general approach is given by Jiang and Kumar [2001], and de Alfaro et al. [2001] solve the generalized synthesis problem for game graphs. Because these approaches are closer to our work, we defer analysis to Subsection 4.5, where the discussion can be based on results we present in the next sections.

The paper is organized as follows: Section 2 presents the Ramadge–Wonham framework and the supervisory control problem (SCP). Section 3 brings in μ-calculus expressions to present the known solution to SCP in a new formulation, which is used in Section 4 to obtain the generalized supervisory control problem and its solution. The conclusion summarizes the work.

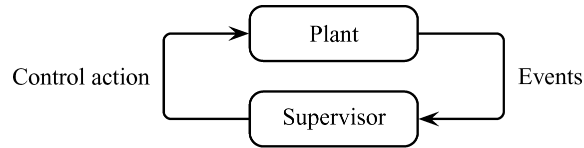


Fig. 1. The basic RW-model.

2. THE RAMADGE–WONHAM FRAMEWORK

The framework parallels continuous systems control theory, in which a system and its controller form a closed loop. There, the feedback signal from the controller influences the behavior of the system, enforcing a given specification that would not be met by the open-loop behavior. This foundation on control theory explains some of the terminology adopted within the RW framework, such as the terms *discrete event system* (to designate an event-driven, discrete-space system, in opposition to time-driven, continuous systems) and *plant* (to designate the system to be controlled). It also leads naturally to the basic assumption that the description of the plant encompasses the whole physically possible behavior of the system to be controlled (therefore including unwanted situations), and that a specification is a subset of this behavior representing the actions wanted to remain executable under control.

The plant is viewed as a system that generates events. It is also assumed that it has a control input, through which some of the events that could happen in each state can be prevented from occurring. The controller, referred to as *supervisor*, is an external agent that has the ability to observe the events generated by the plant and to influence its behavior through the control input, as illustrated in Figure 1.

Control problems are formulated using language theory and deterministic finite automata. A finite automaton is a 5-tuple $\mathcal{A} = \langle \Sigma, Q, \delta, q^0, M \rangle$, where Σ is a set of events, Q is a set of states, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, and $q^0 \in Q$ is the initial state. The states in the set $M \subseteq Q$ are chosen to mark the completion of tasks by the system and are therefore called *marker states*. Those readers familiar with the RW literature will recall that δ is traditionally defined as a partial *function*. We use a *relation* instead because this simplifies notation later on. Hence, we write $\delta(q, \sigma, q')$ to signify that $(q, \sigma, q') \in \delta$. In order to ensure the same functionality, we require the relation to be deterministic, that is, $\delta(q, \sigma, q') \wedge \delta(q, \sigma, q'') \Rightarrow q' = q''$.

It is convenient to define the set of *active events* $\mathsf{act}_{\mathcal{A}}(q)$ as the subset of Σ for which there is a transition leaving state q:

Definition 2.1 (*Active Events*). Given an automaton $\mathcal{A} = \langle \Sigma, Q, \delta, q^0, M \rangle$ and a particular state $q \in Q$, the set of active events of q is

$$\mathsf{act}_{\mathcal{A}}(q) := \{\sigma \in \Sigma \mid \exists q' \in Q \,.\, \delta(q, \sigma, q')\}.$$

When a plant and its supervisor are represented by finite automata $\mathcal{A}_{\mathcal{P}}$ and $\mathcal{A}_{\mathcal{S}}$, respectively, the control action of the latter amounts to running these automata in parallel, according to the following definition:

Definition 2.2 (*Automata Product*). Given two automata $\mathcal{A}_\mathcal{P} = \langle \Sigma, Q_\mathcal{P}, \delta_\mathcal{P}, q^0_\mathcal{P}, M_\mathcal{P} \rangle$ and $\mathcal{A}_\mathcal{S} = \langle \Sigma, Q_\mathcal{S}, \delta_\mathcal{S}, q^0_\mathcal{S}, M_\mathcal{S} \rangle$, the product $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{S}$ is the automaton $\langle \Sigma, Q_\mathcal{P} \times Q_\mathcal{S}, \delta_{\mathcal{P}\times\mathcal{S}}, (q^0_\mathcal{P}, q^0_\mathcal{S}), M_\mathcal{P} \times M_\mathcal{S} \rangle$, where

$$\delta_{\mathcal{P}\times\mathcal{S}}((p,q),\sigma,(p',q')) \Leftrightarrow \delta_\mathcal{P}(p,\sigma,p') \wedge \delta_\mathcal{S}(q,\sigma,q').$$

For examples, we refer to Subsection 3.3 in Wonham's lecture notes [Wonham 2004].

Note that if a given transition is present in only one of the states p or q, it will *not* be present in state (p,q), i.e.,

$$\mathsf{act}_{\mathcal{A}_\mathcal{P}\times\mathcal{A}_\mathcal{S}}((p,q)) = \mathsf{act}_{\mathcal{A}_\mathcal{P}}(p) \cap \mathsf{act}_{\mathcal{A}_\mathcal{S}}(q).$$

The control action of the supervisor enables only the events in $\mathsf{act}_{\mathcal{A}_\mathcal{P}\times\mathcal{A}_\mathcal{S}}$. Hence, in order to forbid the occurrence of event σ when $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{S}$ is in state (p,q), it suffices to omit σ in state q of the supervisor $\mathcal{A}_\mathcal{S}$.

The control mechanism above entails two important features of the RW model: first, the control action is a passive one, which means that the supervisor is not responsible for choosing among the set of events allowed to happen next. This is left to an active controller, which is considered to be part of the plant. The supervisor just tells that controller the set of actions from which it may choose at any given time.

Second, the model in Figure 1 can only work if the plant is actually capable of preventing the forbidden events to occur. However, reactive systems may contain events for which this is not possible, such as system failures, timeouts, and sensor or alarm signals. This is captured by partitioning the event set Σ into the sets of *controllable events* Σ_c (which the supervisor can disable) and *uncontrollable events* $\Sigma_u := \Sigma \setminus \Sigma_c$ (whose occurrence cannot be avoided). Hence, there is a condition on the existence of supervisors: a specification given by an automaton $\mathcal{A}_\mathcal{E}$ can be implemented by a supervisor only if, for every state (p,q) of $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$, every event in $\mathsf{act}_{\mathcal{A}_\mathcal{P}}(p) \setminus \mathsf{act}_{\mathcal{A}_\mathcal{P}\times\mathcal{A}_\mathcal{E}}((p,q))$ is controllable.

Specifications that do not fulfill this requirement are termed *uncontrollable*, because they allow the plant to reach a state in which uncontrollable events can occur and, at the same time, try to forbid the occurrence of one or more of these events in that state. Formally, this means that the product $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ has one or more *bad states*, which are states (p,q) that fail to satisfy the following condition:

$$\mathsf{act}_{\mathcal{A}_\mathcal{P}}(p) \cap \Sigma_u \subseteq \mathsf{act}_{\mathcal{A}_\mathcal{P}\times\mathcal{A}_\mathcal{E}}((p,q)). \tag{1}$$

Analyzing the controllability of a specification further requires some language theory: every automaton $\mathcal{A}$ has an associated *marked language* $\mathsf{L_m}(\mathcal{A})$, which consists of all event sequences that end up in a marker state, hence representing the tasks the system is able to complete. When δ is extended in the usual way to process strings from Σ^*,

$$\mathsf{L_m}(\mathcal{A}) := \{s \in \Sigma^* \mid \exists q \in M.\delta(q^0, s, q)\}.$$

Given a specification automaton $\mathcal{A}_\mathcal{E}$, the language $K = \mathsf{L_m}(\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E})$ is *controllable* if and only if $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ has no bad states.

The marked language of plant $\mathcal{A}_\mathcal{P}$ under control of supervisor $\mathcal{A}_\mathcal{S}$ is given by $\mathsf{L_m}(\mathcal{A}_\mathcal{P}) \cap \mathsf{L_m}(\mathcal{A}_\mathcal{S})$ and denoted as $\mathsf{L_m}(\mathcal{A}_\mathcal{S}/\mathcal{A}_\mathcal{P})$. Ramadge and Wonham have shown

that, for any plant $\mathcal{A}_\mathcal{P}$ and any specification language $K \subseteq \mathsf{L_m}(\mathcal{A}_\mathcal{P})$, there exists the *supremal controllable sublanguage* of K, denoted as $\mathsf{supC}(K)$. This result is of practical interest: given that the specification language K is uncontrollable, it is possible to compute $\mathsf{supC}(K)$ and to construct a supervisor $\mathcal{A}_\mathcal{S}$ such that $\mathsf{L_m}(\mathcal{A}_\mathcal{S}/\mathcal{A}_\mathcal{P}) = \mathsf{supC}(K)$. This language can replace the original specification, as long as the resulting behavior under control is still acceptable.

Another aspect to consider is whether the supervisor always allows the system to make progress toward the completion of some task. This is not the case when the system can (1) reach a state in which no task is finished and no more events can occur (deadlock) or (2) be caught forever within a subset of states, none of which corresponds to a finished task (livelock). A supervisor that avoids these situations is said to be *nonblocking*. A nonblocking automaton is coaccessible, which means that there is at least one path leading from every state to a marker state. Controllability and coaccessibility come together in the following problem, introduced by Ramadge and Wonham [1987]:

Definition 2.3 (*Supervisory Control Problem, SCP*). Given a plant represented by an automaton $\mathcal{A}_\mathcal{P}$, a specification language $K \subseteq \mathsf{L_m}(\mathcal{A}_\mathcal{P})$ representing the desired behavior of $\mathcal{A}_\mathcal{P}$ under supervision, and a minimally acceptable behavior $A_{\min} \subseteq K$, find a nonblocking supervisor $\mathcal{A}_\mathcal{S}$ such that $A_{\min} \subseteq \mathsf{L_m}(\mathcal{A}_\mathcal{S}/\mathcal{A}_\mathcal{P}) \subseteq K$.

Ramadge and Wonham [1987] have shown that this problem is solvable if and only if $\mathsf{supC}(K) \supseteq A_{\min}$. Also, because the class of controllable sublanguages of a given language is closed under arbitrary unions, $\mathsf{supC}(K)$ is its least restrictive solution. A coaccessible automaton $\mathcal{A}_\mathcal{S}$ whose marked language is equal to $\mathsf{supC}(K)$ can be computed from the automata $\mathcal{A}_\mathcal{P}$ and $\mathcal{A}_\mathcal{E}$, with $\mathcal{A}_\mathcal{E}$ constructed so that $\mathsf{L_m}(\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}) = K$. Because the resulting automaton is a supervisor, this computation is referred to as *supervisor synthesis*.

The above is a summary of the most important concepts originally presented in Ramadge and Wonham [1987, 1989] and Wonham and Ramadge [1987]; for a comprehensive description the reader is also referred to Cassandras and Lafortune [1999] and Wonham [2004].

3. SUPERVISOR SYNTHESIS IN THE μ-CALCULUS

In this section, we associate Kripke structures with the automata used in the RW framework and define a μ-calculus over them. Kripke structures are used in Subsection 3.4 to present a new description of the solution for SCP and are also needed to present our main result in Section 4. Because the μ-calculus is not usual in the context of supervisor synthesis, we start with a brief review of basic concepts.

3.1 Fixpoint Calculus

A survey by Lassez et al. [1982] shows that fixpoints of monotone functions have been extensively studied. Such fixpoints are useful to describe properties of discrete mathematical entities. In particular, the Tarski–Knaster theorem in Tarski [1955] lies at the very heart of both the model-checking and supervisor

synthesis algorithms [Emerson and Lei 1986b; Wonham and Ramadge 1987]. This theorem applies to monotone operators on lattices. An operator $f : 2^X \to 2^X$ on the powerset 2^X is said to be *monotone* if, for any subsets $X_i, X_j \subseteq X$,

$$X_i \subseteq X_j \Rightarrow f(X_i) \subseteq f(X_j). \tag{2}$$

Such an operator has least and greatest fixpoints, which are defined as the solutions of

$$Y \stackrel{\mu}{=} f(Y) \quad \text{and} \quad Y \stackrel{\nu}{=} f(Y),$$

where the symbols $\stackrel{\mu}{=}$ and $\stackrel{\nu}{=}$ indicate that we seek for the least and greatest values of $Y \subseteq X$ that satisfy these equations. The solutions are denoted as $\mu Y. f(Y)$ and $\nu Y. f(Y)$, and known to satisfy

$$\mu Y. f(Y) = \cap\{Z \subseteq X : Z = f(Z)\} = \cap\{Z \subseteq X : f(Z) \subseteq Z\}$$
$$\nu Y. f(Y) = \cup\{Z \subseteq X : Z = f(Z)\} = \cup\{Z \subseteq X : Z \subseteq f(Z)\}.$$

Given that X is finite and f is monotone, the least fixpoint of f can be found by an iteration starting with $X_0 = \emptyset$ and computing $X_{i+1} = f(X_i)$ until, for some $j, X_j = X_{j-1}$. The greatest fixpoint can be obtained by the same iteration, starting with $X_0 = X$.

3.2 Kripke Structures and Automata

In this paper, the fixpoint operators will be applied to the state set of a finite Kripke structure. To this end, we assume that our systems are finite state, and that every state can be characterized by a set of Boolean properties. Examples of Boolean properties are ‘waiting for message,’ ‘sensor active,’ and so on. Such properties arise naturally when the system is being modeled. We further assume that any two states in the system differ in at least one of these properties. We represent them by a set of Boolean variables.

Definition 3.1 (*Kripke Structure*). Given a set of Boolean variables $\mathcal{V}$, a Kripke structure is a tuple $\mathcal{K} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$, where $\mathcal{S}$ is a finite set of states, $\mathcal{I} \subseteq \mathcal{S}$ is a set of initial states, $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is a transition relation and $\mathcal{L} : \mathcal{S} \to 2^{\mathcal{V}}$ is a labeling function that assigns to each state $s \in \mathcal{S}$ the set of variables $x \in \mathcal{V}$ that are true in s.

Sequences of states connected by transitions form paths, according to the following:

Definition 3.2 (*Paths*). An infinite path on the Kripke structure $\langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ is an infinite sequence of states $(s_0, s_1, \ldots)$ of $\mathcal{S}$ such that $\forall i \geq 0, (s_i, s_{i+1}) \in \mathcal{R}$.

A finite path on the same structure is a sequence of states $(s_0, \ldots, s_n)$ such that $\neg\exists s' \in \mathcal{S}, (s_n, s') \in \mathcal{R}$ and, if $n > 0$, then $\forall i \in \{0, \ldots, n-1\}, (s_i, s_{i+1}) \in \mathcal{R}$.

Note that a path is always maximal, so a finite prefix of an infinite path and a proper prefix of a finite path are *not* examples of finite paths.

The function $\mathcal{L}$ allows us to represent state sets through Boolean expressions. For example, suppose a Kripke structure $\mathcal{K}$ with state set $\mathcal{S} = \{s_0, s_1, \ldots, s_n\}$,

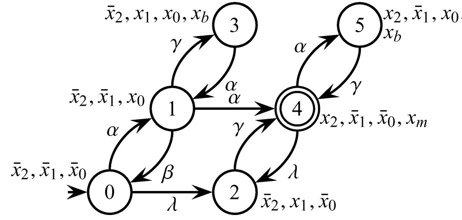


Fig. 2. An example automaton.

where every state in $Q = \{s_i \in \mathcal{S} \mid i \text{ is odd}\}$ has one of the Boolean properties $x_1, x_2 \in \mathcal{V}$ and no other states have any of these properties. Then the set Q can also be described as *all states that satisfy the Boolean expression* $x_1 \vee x_2$. A Boolean expression φ is said to *represent* the set of states whose Boolean variables satisfy φ, and this set of states is denoted by $[\![\varphi]\!]_{\mathcal{K}}$. This is called a *symbolic representation*. In the above example, $[\![x_1 \vee x_2]\!]_{\mathcal{K}} = Q$.

Conversely, we also need a notation for the Boolean expression that represents a given set of states. This is introduced as follows:

Definition 3.3 (*Boolean Expression of a State Set*). Given a set of states Q, we denote by φ_Q any Boolean expression such that $[\![\varphi_Q]\!]_{\mathcal{K}} = Q$.

In the above example, we could write $\varphi_Q = x_1 \vee x_2$. Note that there are, in general, many different Boolean expressions that evaluate to the same set. For example, $x_0 \wedge (x_1 \vee x_2)$ is syntactically different from $x_0 \wedge x_1 \vee x_0 \wedge x_2$, but both mean the same set under $[\![\cdot]\!]$. This is why we defined φ_Q as a notation for *any* expression that represents the set Q. Moreover, we will also write the set itself as an index of φ when this is convenient. For example, given $P = \{s_0, s_2\}$, we will use the notations φ_P and $\varphi_{\{s_0,s_2\}}$ interchangeably. The notation just established will be used in the definition of the μ-calculus semantics in Subsection 3.3 and also in Subsection 3.4.

Symbolic representation can be used to encode the state set of any finite transition structure. In particular, we consider automata resulting from the product of some plant and specification, as explained in Section 2. For example, consider the automaton in Figure 2, where the composite numbers of the states have been replaced by singletons for simplicity. Given the set of Boolean variables $\mathcal{V}_{\mathcal{A}} = \{x_0, x_1, x_2\} \cup \{x_m, x_b\}$, we can define a function $\mathcal{L}_{\mathcal{A}} : Q \to 2^{\mathcal{V}_{\mathcal{A}}}$ that encodes the state set Q of the automaton by encoding the state numbers in binary form and also identifies marker states with the label x_m and bad states with x_b. These encodings are shown close to the states in the figure, where the variables from $\{x_0, x_1, x_2\}$ that are not true in each state appear in negated form. The state set $\{0, 1, 2, 3\}$, for example, would then be denoted symbolically by $[\![\bar{x}_2]\!]_{\mathcal{A}}$. In the example, state 4 is the only marker state and states 3 and 5 are supposed to be bad states.

We now define a special Kripke structure, whose elements are derived from automata like the above. This will allow a translation from a problem formulated in the Ramadge–Wonham model into a structure where model-checking techniques can be applied.

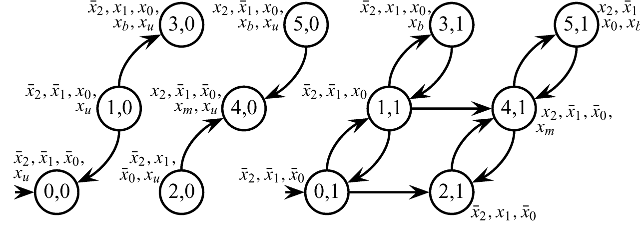


Fig. 3. The Kripke structure associated with the automaton in Figure 2.

Definition 3.4 (*Kripke Structure of an Automaton*). Let $\mathcal{A} = \langle \Sigma, Q, \delta, q^0, M \rangle$ be an automaton representing the product of a plant and a specification, whose states are encoded by a function $\mathcal{L}_\mathcal{A} : Q \to 2^{\mathcal{V}_\mathcal{A}}$ over the set of Boolean variables $\mathcal{V}_\mathcal{A}$. Then $\mathcal{K}_\mathcal{A} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ is its associated Kripke structure, defined over the Boolean variables $\mathcal{V} := \mathcal{V}_\mathcal{A} \cup \{x_u\}$ as follows:

—$\mathcal{S} := Q \times \{0, 1\}$;
—$\mathcal{I} := \{(q^0, 0), (q^0, 1)\}$;
—$\mathcal{R}((q, 0), (q', 0)) :\Leftrightarrow \exists \sigma \in \Sigma_u . \delta(q, \sigma, q')$;
—$\mathcal{R}((q, 1), (q', 1)) :\Leftrightarrow \exists \sigma \in \Sigma . \delta(q, \sigma, q')$;
—$\mathcal{L}((q, 0)) := \mathcal{L}_\mathcal{A}(q) \cup \{x_u\}$;
—$\mathcal{L}((q, 1)) := \mathcal{L}_\mathcal{A}(q)$.

Here, $\Sigma = \Sigma_c \cup \Sigma_u$ is the event set of the automaton (see Section 2), $\mathcal{S}$ is a finite set of states, $\mathcal{I}$ is the set of initial states, and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ relates states $(q, 0)$ and $(q', 0)$ exactly when an uncontrollable event leads from q to q', and states $(q, 1)$ and $(q', 1)$ exactly when there is an event (controllable or not) leading from q to q' in $\mathcal{A}$. This creates a structure with two disconnected substructures, each of which has a copy of the original states in $\mathcal{A}$.[1] The two substructures are needed for collecting relevant states during the synthesis process. Finally, the function $\mathcal{L} : \mathcal{S} \to 2^\mathcal{V}$ labels the states using labels from the originating automaton, as well as the new Boolean variable x_u, which is used to distinguish between the two substructures. The Kripke structure can be constructed from the automaton in time $O(|Q|\,|\Sigma|)$.

Continuing the above example, suppose that the event set of the automaton in Figure 2 is partitioned into $\Sigma_c = \{\alpha, \lambda\}$ and $\Sigma_u = \{\beta, \gamma\}$. The two halves of the associated Kripke structure are shown in Figure 3. The left substructure has transitions only where the automaton has uncontrollable transitions, while the right substructure reflects all transitions of the automaton. Each state (q, i) is labeled with the variables from the corresponding state q in the automaton. Additionally, the states on the left half have in common the label x_u to distinguish them from their counterparts on the right half.

[1]This partition of the state space also allows using a labeled Kripke structure with half the number of states to achieve the same result. However, this state reduction does not reflect itself in an implementation and, in our opinion, would make understanding of the computations presented later more difficult, so we preferred the representation above.

In Section 4 we will need to refer to the Kripke structure formed by the states $[\![\bar{x}_u]\!]_{\mathcal{K}_\mathcal{A}}$ only. The following definition establishes a notation for that purpose:

Definition 3.5 (*Restriction of a Kripke Structure*). Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ be a Kripke structure over the variables $\mathcal{V}$. Given a Boolean expression φ over $\mathcal{V}$, the restriction of $\mathcal{K}$ to the state set $[\![\varphi]\!]_\mathcal{K}$ is $\mathcal{K}|_\varphi := \langle \mathcal{S}|_\varphi, \mathcal{I}|_\varphi, \mathcal{R}|_\varphi, \mathcal{L}|_\varphi \rangle$, where

—$\mathcal{S}|_\varphi := [\![\varphi]\!]_\mathcal{K}$,
—$\mathcal{I}|_\varphi := \mathcal{I} \cap [\![\varphi]\!]_\mathcal{K}$,
—$\mathcal{R}|_\varphi := \mathcal{S}|_\varphi \times \mathcal{S}|_\varphi \cap \mathcal{R}$,
—$\mathcal{L}|_\varphi(s) := \begin{cases} \mathcal{L}(s) & \text{if } s \in \mathcal{S}|_\varphi \\ \text{undefined} & \text{otherwise.} \end{cases}$

For example, $\mathcal{K}_\mathcal{A}|_{\bar{x}_u}$ means the structure obtained by discarding the left substructure of $\mathcal{K}_\mathcal{A}$ in Figure 3.

Any formula φ over the variables $\mathcal{V}$ also describes a subset of the states of $\mathcal{A}$ according to the following projection, which maps states from the Kripke structure back to the originating automaton:

Definition 3.6 (*Kripke Structure State Projection*). Given an automaton $\mathcal{A}$ with state set Q, its associated Kripke structure $\mathcal{K}_\mathcal{A}$ over the variables $\mathcal{V}$, and a formula φ over $\mathcal{V}$, we define the projection of the states $[\![\varphi]\!]_{\mathcal{K}_\mathcal{A}}$ onto the state set Q as

$$\left\{q \in Q \mid (q, 0) \in [\![\varphi]\!]_{\mathcal{K}_\mathcal{A}} \vee (q, 1) \in [\![\varphi]\!]_{\mathcal{K}_\mathcal{A}}\right\}.$$

This projection simply looks at the states in $[\![\varphi]\!]_{\mathcal{K}_\mathcal{A}}$ and then translates them back into the corresponding states from the automaton $\mathcal{A}$. It can be computed symbolically by $\varphi|_{x_u=0} \vee \varphi|_{x_u=1}$, which is denoted by $[\![\exists x_u.\varphi]\!]_\mathcal{A}$ and known as the *existential quantification* of the variable x_u in φ. The above projection therefore completes the set of tools we need to convert an automaton $\mathcal{A}$ into the Kripke structure $\mathcal{K}_\mathcal{A}$, compute some subset of states (e.g., the states that form a supervisor), and translate the result back into an automaton by restricting the state set of $\mathcal{A}$ to $[\![\exists x_u.\varphi]\!]_\mathcal{A}$.

In the following, we will describe sets of states of the Kripke structure associated with an automaton, using symbolic representation. We will describe not only sets associated with Boolean expressions, but also sets defined with respect to structural properties, such as the set of states that can be reached from a given state. It turns out that this kind of property can be expressed by fixpoints of appropriate monotone operators.

3.3 The μ-Calculus

The concepts exposed so far come together in the definition of the μ-calculus, a logic that combines fixpoints and Boolean expressions. Its definition will require the following notation:

Definition 3.7 (*Variable Substitution*). Let $\mathcal{K}$ be a Kripke structure over the variables $\mathcal{V}_x = \{x_0, \ldots, x_{k-1}\}$ and $\mathcal{V}_y = \{y_0, \ldots, y_{k-1}\}$, and φ a formula over the

variables $\mathcal{V}_x \cup \mathcal{V}_y$. Then $[\varphi]_x^y$ denotes the formula obtained from φ by exchanging x_i by y_i for $i = 0, \ldots, k-1$.

The following two definitions formally introduce the μ-calculus:

Definition 3.8 (*Syntax of the μ-Calculus*). Given a set of Boolean variables $\mathcal{V}$, the set of μ-calculus formulae over $\mathcal{V}$ is defined as the least set $\mathcal{L}_\mu$ that satisfies the following rules:

—$\mathcal{V} \cup \{0, 1\} \subseteq \mathcal{L}_\mu$;
—$\neg\varphi, \varphi \vee \psi, \varphi \wedge \psi \in \mathcal{L}_\mu$, provided that $\varphi, \psi \in \mathcal{L}_\mu$;
—$\Diamond \varphi, \overleftarrow{\Diamond} \varphi \in \mathcal{L}_\mu$;
—$\kappa_\pi(\varphi) \in \mathcal{L}_\mu$, provided that $\varphi \in \mathcal{L}_\mu$;
—$\mu x.\varphi \in \mathcal{L}_\mu$, provided that $\varphi \in \mathcal{L}_\mu$.

Definition 3.8 differs from those usually found in the literature in that it includes the formula $\kappa_\pi(\varphi)$. This allows us to use any monotone state transformer function $\pi : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$ in the computations.

Definition 3.9 (*Semantics of the μ-Calculus*). Given a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ over the variables $\mathcal{V}$, we associate with each formula $\Phi \in \mathcal{L}_\mu$ a set of states $[\![\Phi]\!]_\mathcal{K} \subseteq \mathcal{S}$ by the following rules:

—$[\![x]\!]_\mathcal{K} := \{s \in \mathcal{S} \mid x \in \mathcal{L}(s)\}$ for all variables $x \in \mathcal{V}$;
—$[\![\neg\varphi]\!]_\mathcal{K} := \mathcal{S} \setminus [\![\varphi]\!]_\mathcal{K}$;
—$[\![\varphi \wedge \psi]\!]_\mathcal{K} := [\![\varphi]\!]_\mathcal{K} \cap [\![\psi]\!]_\mathcal{K}$;
—$[\![\varphi \vee \psi]\!]_\mathcal{K} := [\![\varphi]\!]_\mathcal{K} \cup [\![\psi]\!]_\mathcal{K}$;
—$[\![\kappa_\pi(\varphi)]\!]_\mathcal{K} := \pi([\![\varphi]\!]_\mathcal{K})$ for monotone functions $\pi : 2^S \to 2^S$;
—$[\![\Diamond \varphi]\!]_\mathcal{K} := \{s \in \mathcal{S} \mid \exists s' \in \mathcal{S}.\mathcal{R}(s, s') \wedge s' \in [\![\varphi]\!]_\mathcal{K}\}$;
—$[\![\overleftarrow{\Diamond} \varphi]\!]_\mathcal{K} := \{s' \in \mathcal{S} \mid \exists s \in \mathcal{S}.\mathcal{R}(s, s') \wedge s \in [\![\varphi]\!]_\mathcal{K}\}$;
—$[\![\mu x.\varphi]\!]_\mathcal{K} := \bigcap\{\mathcal{Q} \subseteq \mathcal{S} \mid f_\varphi(\mathcal{Q}) \subseteq \mathcal{Q}\}$, with $f_\varphi(\mathcal{Q}) := \left[\!\!\left[\, [\varphi]_x^{\varphi_\mathcal{Q}} \,\right]\!\!\right]_\mathcal{K}$.

Given a formula φ and a state $s \in [\![\varphi]\!]_\mathcal{K}$, we say that φ *holds* in s, or equivalently that s *satisfies* φ, which is written as $s \models \varphi$. We also say that a Kripke structure $\mathcal{K}$ satisfies φ, written as $\mathcal{K} \models \varphi$, when $\forall s \in \mathcal{I}, s \models \varphi$.

The next paragraphs explain Definition 3.9 in some detail. From now on, we drop the index $\mathcal{K}$ from $[\![\varphi]\!]_\mathcal{K}$ whenever the Kripke structure is clear from the context. The first four expressions in the definition implement symbolic state representation through Boolean expressions, as explained in Subsection 3.2. For Kripke structures stemming from automata according to Definition 3.4, some important sets are given below (note that we use the notation $\bar{x}$ for $\neg x$ when convenient):

$$\begin{aligned}
[\![x_u]\!]_{\mathcal{K}_\mathcal{A}} &:= \{(q, 0) \mid q \in \boldsymbol{Q}\} \\
[\![\bar{x}_u]\!]_{\mathcal{K}_\mathcal{A}} &:= \{(q, 1)\} \mid q \in \boldsymbol{Q}\} \\
[\![x_b x_u]\!]_{\mathcal{K}_\mathcal{A}} &:= \{(q, 0) \mid q \text{ violates condition } 1\} \\
[\![x_m \bar{x}_u]\!]_{\mathcal{K}_\mathcal{A}} &:= \{(q, 1) \mid q \in \boldsymbol{M}\}.
\end{aligned}$$

In what follows, we will need to switch between the two halves of the above Kripke structure. For this purpose, we define $\pi(Q) := \{(q, \neg i) \mid (q, i) \in Q\}$. This function is easily verified to be monotone (see condition (2)). The function κ_π then toggles the variable x_u that identifies the substructure to which a given state pertains. As there will be no other functions of this type, we write just κ for κ_π from now on.

$[\![\Diamond\,\varphi]\!]$ is the set of states from which there is a transition to a state in $[\![\varphi]\!]$. The states in $[\![\Diamond\,\varphi]\!]$ are called *existential predecessors* of $[\![\varphi]\!]$. Analogously, the states in $[\![\Diamond\,\varphi]\!]$ are called *existential successors* of $[\![\varphi]\!]$.[2]

It is usual to define the following macro operators, which can be reduced to those presented in Definition 3.9:

—$\Box\,\varphi := \neg\Diamond\,\neg\varphi$,
—$\boxdot\,\varphi := \neg\diamonddot\,\neg\varphi$.

$\Box\,\varphi$ represents the set of states from which all transitions lead into a state in $[\![\varphi]\!]$. The states in $[\![\Box\,\varphi]\!]$ are called *universal predecessors* of φ. This condition is trivially satisfied by states with no successors. Analogously, $\boxdot\,\varphi$ represents the *universal successors* of $[\![\varphi]\!]$. This condition is trivially satisfied by states with no predecessors. The operators $\Diamond$, $\diamonddot$, $\Box$, and $\boxdot$ are called *modal operators*. Given a symbolic representation of the transition relation, they can also be evaluated symbolically.

In $[\![\mu x.\varphi]\!]$, φ is a function of the *free variable* $x \in \mathcal{V}$. This means that x is not used to encode any Boolean property of the states, but rather acts as a placeholder to allow other Boolean expressions to be substituted into φ. The notation $[\varphi]_x^{\varphi_Q}$ in the definition of f_φ means the expression φ with x substituted by φ_Q (see Definitions 3.3 and 3.7). According to Subsection 3.1, $[\![\mu x.\varphi]\!]$ is the least fixpoint of f_φ. This fixpoint can be evaluated symbolically by an iteration starting with $\varphi^0 := 0$, and computing $\varphi^{i+1} := [\varphi]_x^{\varphi^i}$ until, for some j, $\varphi^j = \varphi^{j-1}$. It is usual to define

$$\nu x.\varphi(x) := \neg\mu x.\neg\varphi(\neg x),$$

which can be shown as the greatest fixpoint of φ. The greatest fixpoint can be obtained by the same iteration above, starting with $\varphi^0 := 1$.

Writing a least fixpoint in the form of a greatest fixpoint by negating its argument (and vice versa) can lead to confusion when one wants to tell how many fixpoints of each type are contained in an expression. In order to avoid this we require, without loss of generality, that all free variables of φ occur under an even number of negation symbols. It can be shown that this also ensures monotonicity of f_φ.

Fixpoints allow us to describe sets of states whose evaluation depends on the following transitions on the Kripke structure. For example, take only the right

[2]The operators $\Diamond$ and $\diamonddot$ refer to successors and predecessors, regardless of whether they are found on finite or infinite paths. This is the usual approach in the μ-calculus literature and also what we need in this paper. We point out that this constitutes a subtle but important difference with respect to temporal logics when applied to model-checking, where these operators are usually denoted as EX and EY and apply only to states on infinite paths.

side of the Kripke structure in Figure 3, and suppose that we want to compute the set of states from which it is possible to reach the initial state. The set of states from which a state in a given set Q can be reached is given by the least fixpoint of the expression $\varphi := \Diamond x \vee \varphi_Q$. In our case, $\varphi_Q := \bar{x}_2\bar{x}_1\bar{x}_0$ and $\varphi^0 := 0$. The iteration steps are as follows:

$$\begin{aligned}
\varphi^1 &= [\varphi]_x^{\varphi^0} = \Diamond 0 \vee \bar{x}_2\bar{x}_1\bar{x}_0 = 0 \vee \bar{x}_2\bar{x}_1\bar{x}_0 = \bar{x}_2\bar{x}_1\bar{x}_0 \\
\varphi^2 &= [\varphi]_x^{\varphi^1} = \Diamond(\bar{x}_2\bar{x}_1\bar{x}_0) \vee \bar{x}_2\bar{x}_1\bar{x}_0 = \bar{x}_2\bar{x}_1 x_0 \vee \bar{x}_2\bar{x}_1\bar{x}_0 = \bar{x}_2\bar{x}_1 \\
\varphi^3 &= [\varphi]_x^{\varphi^2} = \Diamond(\bar{x}_2\bar{x}_1) \vee \bar{x}_2\bar{x}_1\bar{x}_0 = \bar{x}_2(\bar{x}_1 \vee x_1 x_0) \vee \bar{x}_2\bar{x}_1\bar{x}_0 = \bar{x}_2(\bar{x}_1 \vee x_0) \\
\varphi^4 &= [\varphi]_x^{\varphi^3} = \Diamond(\bar{x}_2(\bar{x}_1 \vee x_0)) \vee \bar{x}_2\bar{x}_1\bar{x}_0 = \bar{x}_2(\bar{x}_1 \vee x_0) = \varphi^3.
\end{aligned}$$

The least fixpoint is $\mu x, \varphi = \bar{x}_2(\bar{x}_1 \vee x_0)$ and, as expected, $[\![\mu x.\varphi]\!] = \{0, 1, 3\}$.

There is also another version of the modal μ-calculus, originally defined by Arnold and Crubille [1988] and put in the form we use here by Cleaveland et al. [1992] and Schneider [2003]. They show how equations with multiple fixpoints can be translated into systems of equations with single fixpoints of the form

$$\begin{cases} x_1 \stackrel{\sigma_1}{=} \varphi_1 \\ \vdots \\ x_n \stackrel{\sigma_n}{=} \varphi_n \end{cases}$$

where $\sigma_i \in \{\mu, \nu\}$ for $i \in \{1, \ldots, n\}$. For the sake of completeness, we note that besides solving the above μ-calculus equation system, there are at least two other alternatives to find the set of states that satisfies it: Emerson and Jutla [1988] have shown that the model-checking problem is equivalent to the word problem in alternating tree automata, and Jurdziński [1998] showed that these are both equivalent to finding winning strategies in parity games. We shall use the equation system form to present our results throughout the paper.

Equations with plain fixpoints can be translated into the form above. For example, given a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ and a subset $Q \subseteq \mathcal{S}$, the accessible states of Q (i.e., the states of Q that can be reached from the initial states going only through states in Q) are given by $[\![x_{Ac}]\!]$, with

$$x_{Ac} \stackrel{\mu}{=} \varphi_Q \wedge (\Diamond x_{Ac} \vee \varphi_{\mathcal{I}}), \tag{3}$$

where φ_Q represents the states of Q. Similarly, the coaccessible states of Q (i.e., the states of Q from which a marker state can be reached going only through states in Q) are given by $[\![x_{Co}]\!]$, with

$$x_{Co} \stackrel{\mu}{=} \varphi_Q \wedge (\Diamond x_{Co} \vee x_m). \tag{4}$$

When evaluating an expression with multiple μ and ν operators, each group of successive operators of the same type (μ or ν) can be computed within a single loop. However, when fixpoints of opposed types are nested in a form that causes mutual dependence, the iterations have also to be nested, and complexity increases. A notion that captures this degree of dependency is the *alternation depth* of a fixpoint expression or equation system. It was introduced by Emerson and Lei [1986a], and further refined by Niwiński [1986]. The formal definitions

are quite involved, but for our purposes it suffices to explain them intuitively. The alternation depth of an expression with nested μ and ν operators is given by the number of mutually dependent fixpoints, except that successive fixpoints of the same type count only once. When an expression is translated into the equation system form, each fixpoint yields one equation. The order in which the equations appear in the system is therefore dictated by the nesting in the expression, and hence has also to be followed during evaluation. The alternation depth is then equal to the number of mutually dependent equations, except that successive equations of the same type count only once. Expressions and equation systems with a single type of fixpoint, or where it suffices to compute different fixpoints only once because there is no mutual dependence, have alternation depth 1 and are also called *alternation free*.

Emerson and Lei [1986a] showed that the computational complexity is also affected by the *length* of an expression, given by the number of symbols it contains. The length of an equation system is obtained by adding the lengths of the right-hand sides of all the equations.

The alternation depth and the length of an equation system come together in the following result from Cleaveland et al. [1992] (see also the description by Schneider [2003]), which will be used to assess the computational complexity of our solution:

THEOREM 3.10 (COMPLEXITY OF μ-CALCULUS MODEL CHECKING). *Given a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ and an equation system E of alternation depth l and length $|E|$ over $\mathcal{K}$, there is an algorithm to compute the solution of the equation system in time*[3]

$$O\left(\left(\frac{|E|\,|\mathcal{S}|}{l}\right)^{l-1} |\mathcal{R}|\,|E|\right).$$

COROLLARY 3.11. *A system of equations of constant length and alternation depth l written for a Kripke structure associated with an automaton can be solved in time*

$$O\left(|Q|^{l}\,|\Sigma|\right).$$

PROOF. By construction, the Kripke structure from Definition 3.4 has $|\mathcal{S}| = 2\,|Q|$ and $|\mathcal{R}| \leq 2\,|Q|\,|\Sigma|$. For constant $|E|$, the result follows immediately. □

There is also another result about computational complexity due to Long et al. [1994] and Browne et al. [1997], which has also been explained by Seidl [1996] and confirmed in the work of Jurdziński about the word problem [Jurdziński 1998; Kirsten 2002]. It comes into play when $l \geq 3$ and, combined with Corollary 3.11, leads to the following:

COROLLARY 3.12. *A system of equations of constant length and alternation depth l written for a Kripke structure associated with an automaton can be*

[3] The result holds provided that the function κ_π can be computed in time $O(\mathcal{R})$. The function κ_π we use in this paper meets this requirement.

solved in time

$$\begin{cases} O(|Q|^{l}\,|\Sigma|) & (l \leq 2) \\ O\big(\,|Q|^{\lceil l/2 \rceil}\,|\Sigma|\,\big) & (l \geq 3). \end{cases}$$

Since the equation systems in this paper have an alternation depth of at most 3, all of them will be computable in at most quadratic time with respect to the state space of the Kripke structure.

3.4 Solving SCP

In this subsection, we present the solution for SCP in a new form. We consider the original algorithm from Wonham and Ramadge [1987], Cassandras and Lafortune [1999], and Wonham [2004] and its improvement by Kumar and Garg [1995]. The textual description of the latter is replaced by a system of μ-calculus equations. To this end, we use the Kripke structure associated with the automaton $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$, with $\mathcal{A}_\mathcal{E}$ constructed so that $\mathsf{L_m}(\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}) = K$. The solution obtained is amenable to further mathematical manipulation, leading naturally to the generalization we propose.

The approach from Wonham and Ramadge compares the automata $\mathcal{A}_\mathcal{P}$ and $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ to find all states that are initially bad and then removes them from $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ along with their transitions. Next, the resulting automaton is made trim, that is, accessible and coaccessible. Because removing bad states can destroy coaccessibility and removing noncoaccessible states can expose new bad states, the algorithm is restarted with the trimmed automaton replacing the initial automaton $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$, until a fixpoint is reached. Therefore, the search for initial bad states has to be repeated at each iteration. Since this requires information from $\mathcal{A}_\mathcal{P}$ which is not present in our Kripke structure, this approach is not well suited as a base for our new formulation.

On the other hand, the algorithm from Kumar and Garg does not throw away bad or non-coaccessible states from $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ at each iteration, but collects them and delays elimination until a fixpoint is reached. A state is considered bad if it has an uncontrollable transition leading to a state already classified as bad or non-coaccessible. Noncoaccessible states are computed as if the already collected states were no longer present. Both operations alternate in a loop and eventually reach a fixpoint. After eliminating the collected states, it suffices to take the accessible component of the resulting automaton to obtain a trim supervisor. This way, the computation of the bad states by comparison between $\mathcal{A}_\mathcal{P}$ and $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ has to be done only once at the beginning of the solution process. Since this is exactly what we do when computing the set $[\![x_b]\!]$ of the Kripke structure associated with $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$, we base our solution on the latter approach.

We have derived μ-calculus expressions for the bad and for the non-coaccessible states in Ziller and Schneider [2003]. However, the generalizations we aim at now are best described in terms of the states to be preserved, instead of those to be eliminated. We shall therefore collect the coaccessible states (instead of the noncoaccessible ones) into a set represented by x_C and the complements of the bad states, which we call *good states*, into a set represented by x_G.

When collecting states, it is important to choose the appropriate half of the Kripke structure according to the transitions that matter in each case. For the good states, the computation has to be carried out on the substructure identified by x_u, since only the uncontrollable transitions matter. For the coaccessible states, all transitions are relevant, and hence they have to be computed on the substructure identified by $\bar{x}_u$. When it comes to consider the good states in the computation of the coaccessible states and vice versa, we switch from one substructure to the other using the function κ. Hence, the expression for x_C can be derived by setting $\varphi_Q = \kappa(x_G)$ in Eq. (4) to keep only coaccessible states that are good and restricting the result to $\bar{x}_u$:

$$x_C \stackrel{\mu}{=} \kappa(x_G) \wedge (\Diamond\, x_C \vee x_m \bar{x}_u). \tag{5}$$

An expression for x_G is difficult to obtain directly, so we start with an expression for the bad states and complement it later. We collect the bad states into a set denoted as x_B, adding a new state to this set when it has an uncontrollable transition leading into a state that was already found to be bad or noncoaccessible. The initial value for x_B is $x_b x_u$, and the noncoaccessible states are given by $\neg\kappa(x_C)$, which maps them onto the corresponding states of the substructure identified by x_u. The expression for x_B is thus

$$x_B \stackrel{\mu}{=} \Diamond\,(x_B \vee \neg\kappa(x_C)) \vee x_b x_u. \tag{6}$$

The expression for x_G is obtained by complementing Eq. (6) and substituting x_G for $\bar{x}_B$. Because the complement can bring in unwanted states from $[\![\bar{x}_u]\!]$ (this happens when there are states with no successors, which are captured by the operator $\Box$), we explicitly restrict the result to x_u in Eq. (8). Note also that the complement makes x_G a *greatest* fixpoint. We then have the following system of equations:

$$\begin{cases} x_C \stackrel{\mu}{=} \kappa(x_G) \wedge (\Diamond\, x_C \vee x_m \bar{x}_u) & (7) \\ x_G \stackrel{\nu}{=} \Box\,(x_G \wedge \kappa(x_C)) \wedge \bar{x}_b x_u. & (8) \end{cases}$$

Note that $[\![x_C]\!] \subseteq [\![\kappa(x_G)]\!]$. Therefore, x_C contains the states that are both good and coaccessible. Hence, the solution for SCP is the automaton derived from the accessible states of x_C according to the projection in Definition 3.6. The set of accessible and coaccessible (trim) states can be computed by setting $\varphi_Q = x_C$ and $\varphi_{\mathcal{I}} = \varphi_{\{(0,1)\}}$ in Eq. (3):

$$x_{Ac} \stackrel{\mu}{=} x_C \wedge \left(\Diamond x_{Ac} \vee \varphi_{\{(0,1)\}}\right). \tag{9}$$

The discussion above is a proof of the following:

PROPOSITION 1 (SOLUTION OF SCP). *The solution of SCP is given by restricting the automaton $\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}$ to the states $[\![\exists x_u . x_{Ac}]\!]_{\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}}$, with x_{Ac} given by Eq.* (9).

The complexity of the overall computation is given by Corollary 3.12. Since the system of Eqs. (7) and (8) has $l = 2$, we get $O(|Q|^2\,|\Sigma|)$, as expected [Ramadge and Wonham 1989; Cassandras and Lafortune 1999].

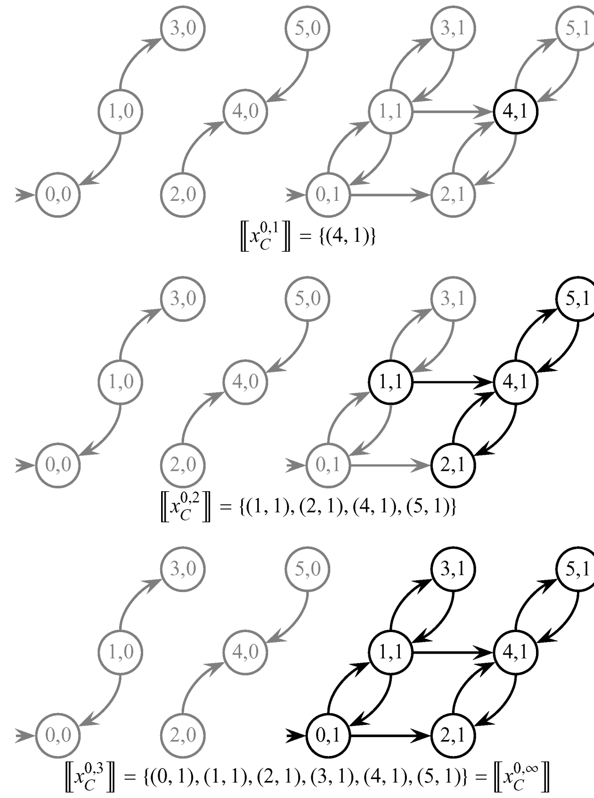


Fig. 4. The computation of $x_C^{0,\infty}$.

3.5 SCP Example

To illustrate the computation of the fixpoints, we solve the problem posed by the automaton in Figure 2. Recall that the automaton is supposed to represent the product of a plant and a specification, with $\Sigma_c = \{\alpha, \lambda\}$ and $\Sigma_u = \{\beta, \gamma\}$. States 3 and 5 are bad states and state 4 is the only marker state. The associated Kripke structure is given in Figure 3.

The iteration steps for the solution of Eqs. (7) and (8) are given below. Each iteration step is presented by a simplified version of Figure 3 (Boolean variables omitted) in which the states that pertain to the current value of x_C (right Kripke structure half) or x_G (left half) are drawn with black lines, while the other ones appear in gray. Each figure is labeled by the corresponding iteration step. To improve readability, we use the state sets in the labeling (e.g., $\{(0,1),(1,1)\}$) instead of the μ-calculus expression (in this case, $\bar{x}_u\bar{x}_2\bar{x}_1$). ${x_C}^{i,j}$ denotes the result of the jth iteration for the ith computation of x_C. Similarly, ${x_G}^i$ denotes the ith step for the computation of x_G.

The initial values are $[\![x_C^{i,0}]\!] = \{\}$ and $[\![x_G^0]\!] = \mathcal{S}$. Also needed are the constant sets $[\![x_m\bar{x}_u]\!] = \{(4,1)\}$ and $[\![\bar{x}_b x_u]\!] = \{(0,0),(1,0),(2,0),(4,0)\}$. The first fixpoint to be computed is $x_C^{0,\infty}$. The iteration steps (except for the initial empty set) are illustrated in Figure 4. The fixpoint is reached at $x_C^{0,\infty} = x_C^{0,3}$.

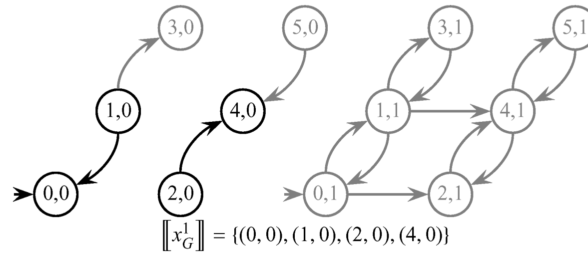
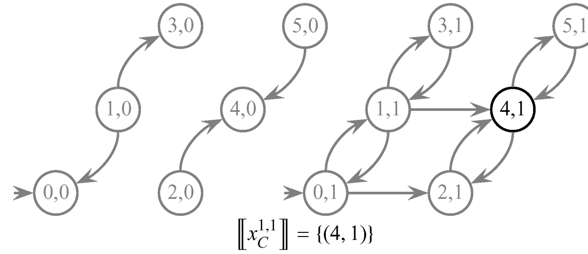
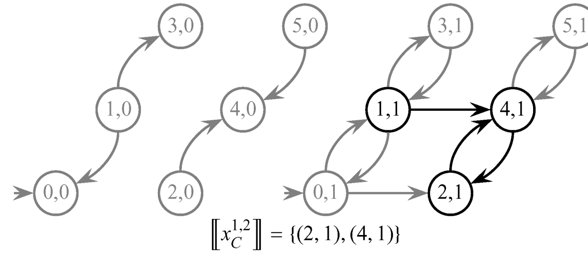


Fig. 5. The computation of x_G^1.





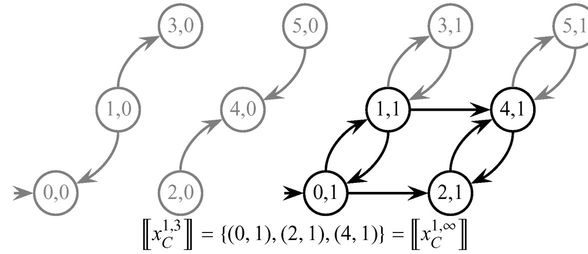


Fig. 6. The computation of $x_C^{1,\infty}$.

$x_C^{0,\infty}$ is then used to compute x_G^1. A state is part of the result when all of its outgoing transitions lead into states considered good and coaccessible up to the current iteration. The set $[\![x_G^1]\!]$ is shown in Figure 5.

Next, x_G^1 is used for the computation of $x_C^{1,\infty}$. The search for coaccessible states is limited by the states found to be good so far. The iteration stops at $x_C^{1,\infty} = x_C^{1,3}$, after the steps given in Figure 6.

$x_C^{1,\infty}$ is then used to compute x_G^2. Because state $(3,1)$ is not part of $[\![x_C^{1,\infty}]\!]$, and $(3,0)$ can be reached by a transition coming from state $(1,0)$, the latter is excluded from the set of good states. The set $[\![x_G^2]\!]$ is shown in Figure 7.

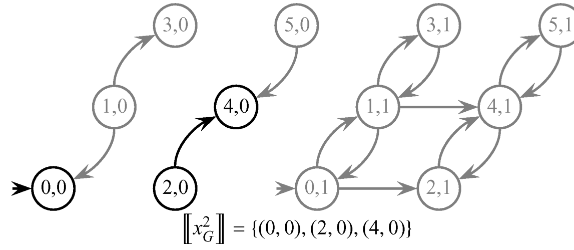
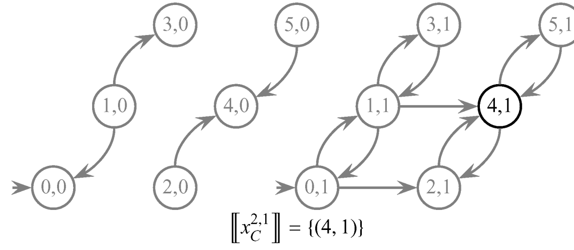
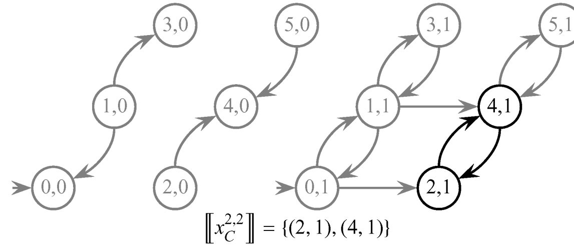


Fig. 7. The computation of x_G^2.





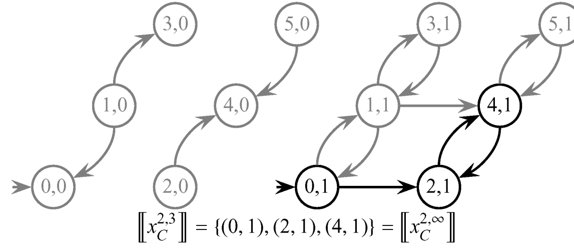


Fig. 8. The computation of $x_C^{2,\infty}$.

Next, x_G^2 is used to compute $x_C^{2,\infty}$, as shown in Figure 8.

Finally, the computation of x_G^3 yields the same result as x_G^2 in Figure 7, so the iteration stops at the fixpoints $[\![x_G^\infty]\!] = \{(0,0),(2,0),(4,0)\}$ and $[\![x_C^{\infty,\infty}]\!] = \{(0,1),(2,1),(4,1)\}$.

Equation (9) can now be solved for x_{Ac} using $x_C^{\infty,\infty}$. Since x_{Ac} is computed as a least fixpoint, the iteration starts with $[\![x_{Ac}^0]\!] := \{\}$. The iteration steps yield the sequence depicted in Figure 9.

According to Proposition 1, the supervisor is obtained by restricting the original automaton in Figure 2 to the states given by the projection in Definition 3.6, namely $[\![\exists x_u . x_{Ac}^\infty]\!]_{\mathcal{A}} = \{0,2,4\}$. The resulting automaton is shown in Figure 10.

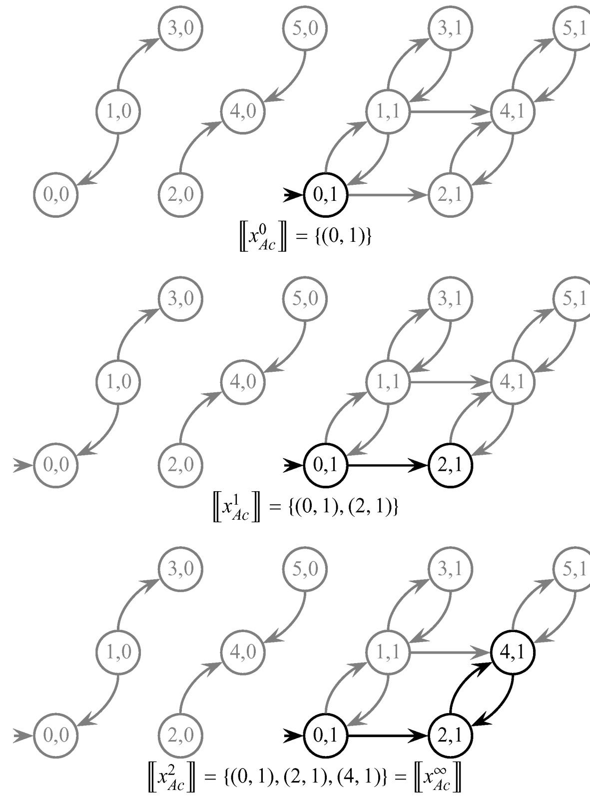


Fig. 9. The computation of x_{Ac}^{∞}.

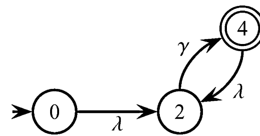


Fig. 10. The supervisor for the SCP example.

We can now proceed to our main result, which allows using other equations to describe the states to be collected, thereby generalizing the class of problems that can be solved.

4. GENERALIZING SUPERVISOR SYNTHESIS

The solution we obtained for SCP in Subsection 3.4 contains a fixed μ-calculus specification requiring that all paths starting at the initial state of $\mathcal{K}_{\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}}$ consist only of states that are both controllable and coaccessible. We achieve a generalization of the problem by allowing the coaccessibility requirement to be replaced by any specification that can be expressed in the μ-calculus. Our approach thus subsumes the temporal logics like CTL and CTL*, as there are well-known algorithms to translate formulae from these logics into the μ-calculus.

The next subsection lists some candidate specifications, ranging from the special case represented by SCP to fairness conditions. Such expressions are used in formal verification to solve the model-checking problem, which consists in solving the corresponding equation system and *checking* whether its solution contains the initial states of the Kripke structure. In contrast, we shall use the solution of the equation system to *restrict* the Kripke structure $\mathcal{K}_{\mathcal{A}_\mathcal{P}\times\mathcal{A}_\mathcal{E}}$ to the states satisfying the desired property. The generalized problem is given in Subsection 4.2, and its solution in Subsection 4.3.

4.1 Fixpoint Specifications

The formulae presented here contain the existential path quantifier E and the universal path quantifier A. A formula containing such a quantifier is true in a state s of a Kripke structure when the property following the quantifier in the formula holds on at least one path (E) or on all paths (A) starting at s. The path quantifiers appear in conjunction with the temporal operators F and G to signify that the property following them must hold on some state (F) or all states (G) of the paths.

The set of states that satisfy a temporal logics expression can be computed by translating it into a μ-calculus expression using well-known methods [Dam 1994; Emerson and Lei 1986a], or into a system of equations [Schneider 2003]. In any case, the μ-calculus formulae describe the desired set of states. The complexity of the translation is linear in the length of the formula for CTL and single exponential for CTL*. Because this is higher than the polynomial complexity of μ-calculus modelchecking (see Theorem 3.10), one might argue that the complexity of our approach is also exponential. However, because the formulae are usually short, the above translation is the easy part of the problem. The polynomial part is much harder, because it depends on the size of the transition relation, which can be very large. Besides, the translation has to be done only once and is independent of the particular system for which the problem is being solved. Hence, we consider that our problem consists in solving the equation system.

In the following, we present some often used CTL and CTL* expressions along with their equivalent equation systems. The list is not exhaustive, since the generalization we propose is open to any expression written according to a specific need:

—$\mathsf{EG}\varphi$ gives the set of states from which there is at least one path (either finite or infinite) along whose states φ always holds. This property can be expressed in the μ-calculus as

$$\{\, z \overset{\nu}{=} \varphi \wedge (\Box 0 \vee \Diamond z).$$

—$\mathsf{EG}|_\infty\, \varphi$ is the restriction of the above condition to infinite paths. The μ-calculus expression is

$$\{\, z \overset{\nu}{=} \varphi \wedge \Diamond z.$$

—$\mathsf{EF}\varphi$ gives the set of states from which there is at least one path (either finite or infinite) that reaches a state where φ holds. We say that φ *eventually* holds

on the referred paths. The set can be computed by

$$\{\, z \overset{\mu}{=} \varphi \vee \Diamond z. \tag{10}$$

—$\mathsf{EF}|_\infty\varphi$ is the restriction of the above condition to infinite paths. The μ-calculus equation system is

$$\begin{cases} x \overset{\nu}{=} \Diamond x \\ z \overset{\mu}{=} \varphi \wedge x \vee \Diamond z. \end{cases}$$

Here, x represents the set of states that have an infinite path, and z will therefore be restricted to the states belonging to infinite paths.

—$\mathsf{AF}\varphi$ gives the set of states with the property that all paths (either finite or infinite) starting at them contain at least one state where φ holds. We say that for all paths φ eventually holds. It is computed by

$$\{\, z \overset{\mu}{=} \varphi \vee \Diamond 1 \wedge \Box z.$$

—$\mathsf{AF}|_\infty\varphi$ is the restriction of the above condition to infinite paths. The μ-calculus equation system is

$$\begin{cases} x \overset{\nu}{=} \Diamond x \\ z \overset{\mu}{=} (\varphi \vee \Box z) \wedge x. \end{cases}$$

—$\mathsf{E}[\varphi\ \mathsf{U}\ \psi]$ holds in every state of a structure that has a path (either finite or infinite) that reaches a state satisfying the property ψ, and up to (but not necessarily including) this state, will only run through states that satisfy the property φ. The property trivially holds in every state satisfying ψ, regardless of whether φ holds or not. In any case, we say that φ holds *until* ψ holds. An equivalent interpretation is "φ while not ψ." The μ-calculus definition is

$$\{\, z \overset{\mu}{=} \psi \vee \varphi \wedge \Diamond z.$$

—$\mathsf{E}[\varphi\ \mathsf{U}|_\infty\psi]$ is the version of $\mathsf{E}[\varphi\ \mathsf{U}\ \psi]$ that considers only infinite paths

$$\begin{cases} x \overset{\nu}{=} \Diamond x \\ z \overset{\mu}{=} \psi \wedge x \vee \varphi \wedge \Diamond z. \end{cases}$$

Here x represents the set of states that have an infinite path, and z will therefore be restricted to those states, while only states satisfying φ may be traversed.

—$\mathsf{EFG}\varphi$ holds in states that have at least one path (either finite or infinite) where after some point of time φ always holds. This translates to

$$\begin{cases} x \overset{\nu}{=} \varphi \wedge (\Box 0 \vee \Diamond x) \\ z \overset{\mu}{=} x \vee \Diamond z. \end{cases}$$

As in our first example, x computes the set of states that have a path where φ always holds. z computes the set of states that can reach the set x.

—$\mathsf{EFG}|_\infty\,\varphi$ is the restriction of the above condition to infinite paths:

$$\begin{cases} x \overset{\nu}{=} \varphi \wedge \Diamond x \\ z \overset{\mu}{=} x \vee \Diamond z. \end{cases}$$

Here, the equation for x has been replaced with that used in $\mathsf{EG}|_\infty\,\varphi$, which considers only infinite paths.

—To compute the set of states that can reach a state with some property ψ at least twice, while only states satisfying φ may be traversed, the equation from $\mathsf{E}[\varphi \mathsf{U} \psi]$ is used twice:

$$\begin{cases} z_1 \overset{\mu}{=} \psi \vee \varphi \wedge \Diamond z_1 \\ y \overset{\mu}{=} \psi \wedge \varphi \wedge \Diamond z_1 \\ z_2 \overset{\mu}{=} y \vee \varphi \wedge \Diamond z_2. \end{cases}$$

The first equation computes $\mathsf{E}[\varphi \mathsf{U} \psi]$. The second equation is not an iterative expression because its right-hand side does not depend on y, and just gives the predecessors from states in z_1 that satisfy $\psi \wedge \varphi$. So, every state in y satisfies φ and ψ and also can reach states in z_1, all of which have a path running only through states that satisfy φ until it reaches a state satisfying ψ. Finally, the third equation computes $\mathsf{E}[\varphi \mathsf{U} y]$, that is, every state in z_2 pertains to a path that reaches a state in y (where both φ and ψ hold) running only through states satisfying φ. From there, it is always possible to reach a state in z_1, where ψ will be satisfied a second time, always running only through states where φ holds.

—As in $\mathsf{E}[\varphi \mathsf{U}|_\infty \psi]$, the restriction of the above equation system to infinite paths can be considered by adding the equation for x:

$$\begin{cases} x \overset{\nu}{=} \Diamond x \\ z_1 \overset{\mu}{=} \psi \wedge x \vee \varphi \wedge \Diamond z_1 \\ y \overset{\mu}{=} \psi \wedge \varphi \wedge \Diamond z_1 \\ z_2 \overset{\mu}{=} y \vee \varphi \wedge \Diamond z_2. \end{cases}$$

—$\mathsf{EGF}|_\infty \varphi$ holds in states that have at least one path where states satisfying φ are traversed infinitely often. This is computed as follows:

$$\begin{cases} x \overset{\mu}{=} z \wedge \varphi \vee \Diamond x \\ z \overset{\nu}{=} \Diamond x. \end{cases}$$

—$\mathsf{E}\big[\mathsf{G}|_\infty \beta \wedge \mathsf{GF}|_\infty \varphi\big]$ extends the previous condition in that the path may only run through states satisfying β. This is given by

$$\begin{cases} x \overset{\mu}{=} z \wedge \varphi \vee \beta \wedge \Diamond x \\ z \overset{\nu}{=} \beta \wedge \Diamond x. \end{cases}$$

—We extend the previous condition once more considering different sets of states φ_i that should be reached infinitely often. The property to be computed is

$$\mathsf{E}\left[\mathsf{G}|_\infty \beta \wedge \bigwedge_{i=1}^{n} \mathsf{GF}|_\infty \varphi_i\right], \tag{11}$$

which can be expressed in the μ-calculus as follows:

$$\begin{cases} x_1 \overset{\mu}{=} z \wedge \varphi_1 \vee \beta \wedge \Diamond x_1 \\ \vdots \\ x_n \overset{\mu}{=} z \wedge \varphi_n \vee \beta \wedge \Diamond x_n \\ z \overset{\nu}{=} \beta \wedge \bigwedge_{i=1}^{n} \Diamond x_i . \end{cases}$$

—Finally, we consider the property $\mathsf{E} \bigwedge_{j=1}^{n} \mathsf{GF}|_{\infty} \varphi_j \vee \mathsf{FG}|_{\infty} \psi_j$, which is known as the acceptance condition of Rabin automata [Emerson and Jutla 1999; Emerson and Lei 1986b; Schneider 2003] used in Thistle and Wonham [1994a, 1994b]. It is given by

$$\begin{cases} x_1 \overset{\mu}{=} y \wedge \varphi_1 \vee y \wedge \Diamond x_1 \\ \vdots \\ x_n \overset{\mu}{=} y \wedge \varphi_n \vee y \wedge \Diamond x_n \\ y \overset{\nu}{=} \bigwedge_{j=1}^{n} \Diamond x_j \vee \psi_j \wedge \Diamond y \\ z \overset{\mu}{=} y \vee \Diamond z . \end{cases}$$

4.2 Generalized Supervisory Control Problem

The solution for SCP presented in Subsection 3.4 generates an automaton $\mathcal{A}_{\mathcal{S}}$ such that $\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{S}}$ is controllable and nonblocking, that is, coaccessible.

We assume that controllability as given by Eq. (8) is a requirement that must always be fulfilled, while the coaccessibility condition represented by Eq. (7) can be exchanged by other requirements. For example, suppose the system to be controlled is a manufacturing cell designed to produce a number of different parts, and that we want to restrict its behavior through some specification. Suppose further that we want the resulting supervisor to allow the system to always be able to produce any of those parts. The last condition is a fairness constraint that cannot be expressed within SCP: if we model the production of each part as a finished task using marker states and apply the standard synthesis algorithm, then every nonempty supervisor will allow the system to reach at least one marker state. However, there is no guarantee that all marker states can be reached, and even if they can, it is still another problem to find out if they continue to be reachable all the time.

This leads naturally to the question whether specifications like the above can be included in the synthesis process by modifying the requirement for coaccessibility, while controllability continues to be computed as before. The states fulfilling the new requirement would be computed on the same substructure used to compute the coaccessible states, and the new requirement must be fulfilled on the right side of the Kripke structure $\mathcal{K}_{\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{S}}}$, much like the coaccessibility requirement in SCP.

Note that in SCP the desired behavior is specified by the languages $K \subseteq \mathsf{L_m}(\mathcal{A}_{\mathcal{P}})$ and $A_{\min} \subseteq K$, as well as by requiring a nonblocking supervisor. We generalize the supervisory control problem by keeping the language requirement

K and by replacing $A_{\min}$ and the nonblocking condition by a μ-calculus equation system. Our generalized problem is formulated as follows:

Definition 4.1 (*Generalized Supervisory Control Problem, GSCP*). Given a plant represented by an automaton $\mathcal{A}_{\mathcal{P}}$, a specification for the desired behavior consisting of a language $K \subseteq \mathsf{L}_{\mathsf{m}}(\mathcal{A}_{\mathcal{P}})$, and a μ-calculus equation system Ω, find a supervisor $\mathcal{A}_{\mathcal{S}}$ for $\mathcal{A}_{\mathcal{P}}$ such that $\mathsf{L}_{\mathsf{m}}(\mathcal{A}_{\mathcal{S}}/\mathcal{A}_{\mathcal{P}}) \subseteq K$ and $\mathcal{K}_{\mathcal{A}_{\mathcal{P}}\times\mathcal{A}_{\mathcal{S}}}\big|_{\bar{x}_u} \models \Omega$.

4.3 Solving GSCP

We can now present our main result, which consists in combining the solution for SCP presented in Subsection 3.4 with the conditions represented by equation systems like those in Subsection 4.1. Our generalization is governed by the following principle, which we assume to have an axiomatic character: *while coaccessibility can be exchanged by any other condition, controllability must always be respected*.

Formally, we mean that Eq. (7) can be replaced by any set of equations needed to specify some desired property, while Eq. (8) has to be modified so that the states having the new property take the place of x_C. The systems of equations presented in Subsection 4.1 have the general form:

$$\begin{cases} x_1 \overset{\sigma_1}{=} \Phi_1 \\ \vdots \\ x_n \overset{\sigma_n}{=} \Phi_n \\ z \overset{\sigma_{n+1}}{=} \Psi. \end{cases}$$

Here, z is the set of states satisfying condition Ω. According to our argumentation in Subsection 4.2, this condition should be applied only to the substructure of $\mathcal{K}_{\mathcal{A}_{\mathcal{P}}\times\mathcal{A}_{\mathcal{E}}}$ originally used to compute the coaccessible states. This can be achieved by restricting the expressions for $\Phi_1,\ldots,\Phi_n$ and Ψ to $\bar{x}_u$. Further, looking for a supervisor requires that all paths in the solution contain only good states, which implies restricting the above expressions to $\kappa(x_G)$. Hence, the generalized equation system has the following pattern:

$$\begin{cases} x_1 \overset{\sigma_1}{=} \kappa(x_G) \wedge \bar{x}_u \wedge \Phi_1 \\ \vdots \\ x_n \overset{\sigma_n}{=} \kappa(x_G) \wedge \bar{x}_u \wedge \Phi_n \\ z \overset{\sigma_{n+1}}{=} \kappa(x_G) \wedge \bar{x}_u \wedge \Psi \\ x_G \overset{\nu}{=} \Box(x_G \wedge \kappa(z)) \wedge \bar{x}_b x_u. \end{cases}$$

As in Subsection 3.4, the accessible states from z can be computed by making $\varphi_Q = z$ and $\varphi_{\mathcal{I}} = \varphi_{\{0,1\}}$ in Eq. (3):

$$x_{Ac} \overset{\mu}{=} z \wedge (\Diamond x_{Ac} \vee \varphi_{\{0,1\}}). \tag{12}$$

The above discussion is a proof of the following:

PROPOSITION 4.2 (SOLUTION OF GSCP). *The solution of GSCP is given by restricting the automaton $\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}$ to the states $[\![\exists x_u.x_{Ac}]\!]_{\mathcal{A}_{\mathcal{P}}\times\mathcal{A}_{\mathcal{E}}}$, with x_{Ac} given by Eq.* (12).

4.4 Generalization Examples

As a first example, let us derive the solution for SCP from the generalized problem. SCP requires that every state has a path (no matter whether finite or infinite) leading to a marker state. This can be expressed by the temporal logics condition $\mathsf{EF}x_m$. Substituting $\varphi = x_m$ in expression (10) and $z = x_C$ in the generalized solution pattern for GSCP we get

$$\begin{cases} x_C \overset{\mu}{=} \kappa(x_G) \wedge \bar{x}_u \wedge (x_m \vee \Diamond x_C) \\ x_G \overset{\nu}{=} \Box(x_G \wedge \kappa(x_C)) \wedge \bar{x}_b x_u. \end{cases}$$

The first equation can be simplified if we note that the computation of x_C starts with the empty set (because it is a least fixpoint) and that $\Diamond 0 = 0$. Since the first iteration step can collect only states from the right substructure due to the conjunction with $\bar{x}_u$, the term $\Diamond x_C$ will also collect only such states and the conjunction of the latter with $\bar{x}_u$ can be dropped. This transforms the first of the equations above in Eq. (7). The second equation is already Eq. (8).

As a second example, we solve the fairness problem described in Subsection 4.2. Let the states that should remain reachable infinitely often be represented by $\varphi_1, \ldots, \varphi_n$. The temporal logics expression that formalizes the problem is $\mathsf{E}\big[\bigwedge_{i=1}^{n} \mathsf{GF}\varphi_i\big]$, which is expression (11) with $\beta = 1$. According to Subsection 4.3, the generalized equation system is

$$\begin{cases} x_1 \overset{\mu}{=} \kappa(x_G) \wedge \bar{x}_u \wedge (z \wedge \varphi_1 \vee \Diamond x_1) \\ \vdots \\ x_n \overset{\mu}{=} \kappa(x_G) \wedge \bar{x}_u \wedge (z \wedge \varphi_n \vee \Diamond x_n) \\ z \overset{\nu}{=} \kappa(x_G) \wedge \bar{x}_u \wedge \bigwedge_{i=1}^{n} \Diamond x_i \\ x_G \overset{\nu}{=} \Box(x_G \wedge \kappa(z)) \wedge \bar{x}_b x_u. \end{cases}$$

The supervisor can again be constructed by restricting the automaton $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ to the set of states $[\![z]\!]_{\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}}$. The computational complexity of the solution can also be easily assessed: since the system of equations above has alternation depth 2, this problem has the same computational complexity as SCP, namely $O(|Q|^2 |\Sigma|)$.

4.5 Related Work

This subsection compares our work to controller or supervisor synthesis approaches by Jiang and Kumar [2001], Arnold et al. [2003], and de Alfaro et al. [2001]. At an abstract level, all approaches deal with the same problem, namely synthesizing a controller to ensure a specification given in some temporal logic. The details, however, reveal important differences. We classify the approaches according to whether they reduce the synthesis problem to the *satisfiability problem* or to the *model-checking problem* of the μ-calculus. Before entering discussion, we give a suitable explanation of what these problems aim at.

The satisfiability problem is illustrated in Figure 11(a). Given a μ-calculus formula φ (or the equivalent equation system), the question to be answered is whether there is a Kripke structure that satisfies φ. To solve the problem, the formula is translated into a tree automaton A_φ, which accepts exactly the

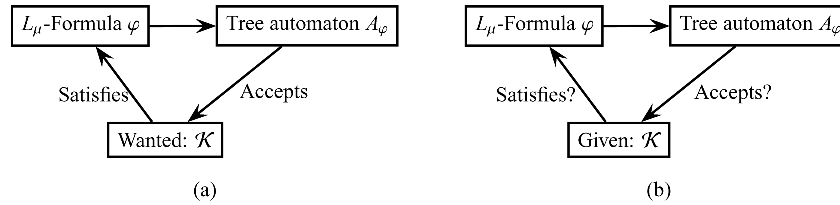


Fig. 11. (a) The μ-calculus satisfiability problem. (b) The μ-calculus model-checking problem.

Kripke structures that satisfy φ. Hence, satisfiability is equivalent to the emptiness problem of the corresponding tree automaton. There are two difficult parts in the solution of this problem. The first is the translation of the formula into the tree automaton. The complexity of the translation (which is also the upper bound for the size of the automaton) increases exponentially with the length of the μ-calculus formula (see the paper by Wilke [2001] for details about the translation). In principle, one could argue that μ-calculus formulae are in general not long enough to make a translation impossible, as we did for CTL* formulae in Subsection 4.1. However, μ-calculus formulae tend to be less compact than their temporal logics counterparts, because translating CTL* into the μ-calculus is already EXPTIME-complete. This results in a double exponential complexity for translating a CTL* formula into a tree automaton. The second difficulty lies in checking the emptiness of the automaton. The complexity of the best known algorithm, given by Emerson and Jutla [1988], increases exponentially with the length of the given μ-calculus formula and is double exponential in the length of the corresponding CTL* formula. The whole scenario is aggravated by the fact that tree automata and the above-mentioned algorithms are not well suited for symbolic representation.

The model-checking problem is illustrated in Figure 11(b). Given a μ-calculus formula φ and a Kripke structure $\mathcal{K}$, the question is whether $\mathcal{K}$ satisfies φ. The good news is that the translation into the tree automaton is not necessary in this case. Because the Kripke structure is known, it is possible to use a model-checking algorithm to find out which states of $\mathcal{K}$ satisfy the formula and then check whether the initial states of $\mathcal{K}$ are contained in this set. The complexity results mentioned in Subsection 3.3 established that this problem has a polynomial complexity in the length of the formula and the size of the transition relation. Further, model-checking algorithms are well suited for symbolic implementation, for example, with binary decision diagrams [Burch et al. 1992], and have thus been able to handle many problems with large state spaces.

Moreover, the satisfiability approach requires the user to define a criterion to choose among the possible solutions. Because these are not guaranteed to be closed under arbitrary unions, there will, not be in general, a supremal solution, and choosing an appropriate one may not be trivial. In contrast, the result of the model-checking problem is a single set of states corresponding to a least or a greatest fixpoint. In view of these facts, it is intuitive that model checking is, in general, easier than satisfiability. We therefore expect approaches based on model-checking to be easier to implement and use than those based on satisfiability.

The solution for GSCP presented in this paper differs from the model-checking problem only in the way we use the resulting set of states. While in traditional model-checking the question is whether the initial states of the Kripke structure are contained in the set of states returned by the model-checking algorithm, we use this set of states to derive our supervisor. Hence, our approach is a pure model-checking approach. In contrast, the related publications analyzed in this subsection either rely on the satisfiability problem or use translation procedures of similar degree of complexity.

Jiang and Kumar [2001] use the same finite automata we do for the representation of the system, as is usual in the Ramadge–Wonham framework. The specification is given in the temporal logic CTL*. Their approach differs from ours in that the specification formula is translated into a Rabin tree automaton, thereby reducing the synthesis problem to the satisfiability problem. The size of the tree automaton can be made linear in the number of states of the plant, but is still double exponential in the length of the specification formula. Since CTL* is subsumed by the μ-calculus, our approach also subsumes this one.

In the approach by de Alfaro et al. [2001], the system to be controlled is represented by a game graph instead of an automaton. The specification is an LTL formula, and the answer to the synthesis problem consists in finding the set of states of the game graph that have a path on which the formula is satisfied. Thus the final answer is obtained by a model-checking technique similar to that we used to solve GSCP. As usual, the LTL formula has to be translated into the μ-calculus before model-checking can begin. An important difference with respect to our approach lies in the translation of the formula. While we can use the classical translation methods and the well-known corresponding formulae given in Subsection 4.1, the method in de Alfaro et al. [2001] needs a special translation of the formulae that is based on Safra's determinization of ω-automata [Safra 1988]. The resulting algorithm has a double exponential complexity in the size of the LTL formula and cannot profit from the benefits of symbolic representation.

A more powerful approach that also uses satisfiability is that given by Arnold et al. [2003], where the specification automaton is additionally restricted by μ-calculus formulae much like we do in GSCP. The formulae are translated into alternating tree automata, and the satisfiability problem is then cast as a search for winning strategies in parity games (see Emerson et al. [1993] and Kirsten [2002] for details). The approach also contains an extension to handle partial observability, which is not in the scope of our present work.

Even under full observability, it is presently unclear whether our approach and that in Arnold et al. [2003] solve exactly the same class of problems. We already know that both approaches can differ in the way a specification is interpreted, as demonstrated by the following example: suppose that the plant is the automaton given in Figure 12, and that event α is controllable. Suppose further that the specification automaton is identical to the plant. Let the temporal logics condition be *there shall be no infinite paths*, which can be written as $\mu x.\Box x$. The interpretation given to this condition in Arnold et al. [2003] should lead to a solution given by the automaton in Figure 12 with the selfloop in state 1 removed. In order to achieve this, the synthesis procedure must be able

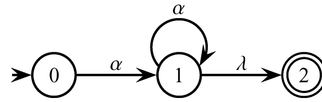


Fig. 12. A plant automaton.

to cut off transitions from the transition relation without necessarily removing any states. This is not what happens in our case. Our generalization of the Ramadge–Wonham procedure removes unwanted *states* from the automaton $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$, and the attached transitions are removed only as a consequence thereof. Hence, in GSCP, the interpretation for $\Omega = \mu x.\Box x$ would be *there shall be no states with infinite paths*, and thus the result would amount to cut states 1 and 2 (the latter due to inaccessibility) from the automaton. Note, however, that the difference between the two approaches can be overcome by supplying an automaton for the specification of GSCP in which the selfloop in state 1 is not present. So it appears that, with the appropriate specification, we are also able to solve problems according to the interpretation in Arnold et al. [2003].

Currently, algorithms for translating a given problem between our approach and those by Arnold et al. [2003] and de Alfaro et al. [2001] are still not well understood. Therefore, the example just given may also apply to the latter one, depending on how the finite automata used in GSCP are translated into game graphs and whether this representation allows handling transitions individually during the synthesis process.

In summary, we can say that the class of problems that can be solved with our approach is at least as large as that in Jiang and Kumar [2001], and at least nearly the same as those in de Alfaro et al. [2001] and Arnold et al. [2003]. We also believe that our approach is best suited for implementation. As opposed to the others, our approach is well suited for symbolic state representation, which has proven the only effective way to cope with the state explosion problem.

5. CONCLUSION

The paper presents the Ramadge–Wonham supervisory control problem in a new formulation using a system of μ-calculus equations. In addition to providing a formal description of its solution, this approach naturally separates the representation of the two requirements of the problem, namely controllability and coaccessibility. This allows us to exchange the latter condition by any temporal logics expression, and thereby to extend the advantages of supervisor synthesis to the whole class of μ-calculus model-checking problems. The computational complexity of each generalization can be easily assessed from the alternation depth of the system of equations representing the solution. An analysis of other approaches to the same problem gives reason to believe that the solution presented in this paper leads to more straightforward and efficient implementations.

ACKNOWLEDGMENTS

We thank professor André Arnold from the LaBRI in Bordeaux for the help in assessing the relation between our work and his approach in Arnold et al. [2003], and also for the example presented in Subsection 4.5.

We thank professor Murray Wonham from the Department of Electrical and Computer Engineering of the University of Toronto, for his constant disposition to discuss questions concerning our research.

We thank our anonymous referees for their careful reviews and for bringing some related work to our attention which we had not considered in the initial version of the paper. Including them in the section about related work largely improved the quality of our publication.

REFERENCES

ANTONIOTTI, M. AND MISHRA, B. 1996. NP-completeness of the supervisor synthesis problem for unrestricted CTL specifications. In *Workshop on Discrete Event Systems*. Edinburgh, Scotland, UK.

ARNOLD, A. AND CRUBILLE, P. 1988. A linear algorithm to solve fixed-point equations on transition systems. *Information Processing Letters 29*, 2, 57–66.

ARNOLD, A., VINCENT, A., AND WALUKIEWICZ, I. 2003. Games for synthesis of controllers with partial observation. *Theor. Comput. Sci. 1*, 303 (Jun.), 7–34.

ASARIN, E., MALER, O., AND PNUELI, A. 1995. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, vol. 999. LNCS, Springer, Berlin.

BROWNE, A., CLARKE, E., JHA, S., LONG, D., AND MARRERO, W. 1997. An improved algorithm for the evaluation of fixpoint expressions. *Theore. Comput. Sci. 178*, 1–2, 237–255.

BURCH, J., CLARKE, E., MCMILLAN, K., DILL, D., AND HWANG, L. 1992. Symbolic model checking: 10^{20} states and beyond. *Information and Computation 98*, 2 (June), 142–170.

BURSTALL, R. 1974. Program proving considered as hand simulation plus induction. In *Congress on Information Processing*, 308–312.

CASSANDRAS, C. G. AND LAFORTUNE, S. 1999. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, Boston, MA.

CLARKE, E. M., EMERSON, E., AND SISTLA, A. 1986. Automatic verification of finitestate concurrent systems using temporal logic specifications. *ACM Trans. Progr. Lang. Syst. 8*, 2, 244–263.

CLARKE, E., GRUMBERG, O., AND LONG, D. 1993. Verification tools for finite state concurrent systems. In *A Decade of Concurrency—Reflections and Perspectives*, J. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds. Vol. 803. Springer-Verlag, Noordwijkerhout, Netherlands, 124–175.

CLARKE, JR, E. M., GRUMBERG, O., AND PELED, D. A. 1999. *Model Checking*. The MIT Press, London.

CLEAVELAND, R., KLEIN, M., AND STEFFEN, B. 1992. Faster model checking for the modal μ-calculus. In *Computer Aided Verification (CAV'92)*, G. Bochmann and D. Probst, Eds. LNCS, Vol. 663. Springer-Verlag, Heidelberg, Germany, 410–422.

DAM, M. 1994. CTL* and ECTL* as fragments of the modal μ-calculus. *Theore. Comput. Sci. 126*, 1, 77–96.

DE ALFARO, L., HENZINGER, T., AND MAJUMDAR, R. 2001. From verification to control: dynamic programs for omega-regular objectives. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, Washington, D.C., 279–290.

EMERSON, E. AND CLARKE, E. 1980. Characterizing correctness properties of parallel programs as fixpoints. In *Colloquium on Automata, Languages and Programming (ICALP)*, LNCS, Vol. 85. Springer, Berlin, 169–181.

EMERSON, E. AND JUTLA, C. 1988. The complexity of tree automata and logics of programs. In *Symposium on Foundations of Computer Science (FOCS)*. White Plains, New York, 328–337.

EMERSON, E. AND JUTLA, C. 1999. The complexity of tree automata and logics of programs. *SIAM Journal on Computing 29*, 1, 132–158.

EMERSON, E. AND LEI, C.-L. 1986a. Efficient model checking in fragments of the propositional mu-calculus. In *IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press, Washington, D.C., 267–278.

EMERSON, E. AND LEI, C.-L. 1986b. Temporal reasoning under generalized fairness constraints. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, B. Monien and G. Vidal-Naquet, Eds. LNCS, Vol. 210. Springer, Orsay, France, 21–36.

EMERSON, E. A., JUTLA, C. S., AND SISTLA, P. 1993. On model-checking for fragments of μ-calculus. In *Computer Aided Verification*, C. Courcoubetis, Ed. LNCS, Vol. 697. Springer, Elounda, Greece, 385–396.

FLOYD, R. 1967. Mathematical aspects of computer science. In *Proceedings of Symposia in Applied Mathematics*, 19–32.

HOARE, C. 1969. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (Oct.), 576–583.

HOFFMANN, G. AND WONG-TOI, H. 1992. Symbolic synthesis of supervisory controllers. In *Proceedings of the American Control Conference*, Chicago, IL.

JIANG, S. AND KUMAR, R. 2001. Supervisory control of discrete event systems with CTL* temporal logic specifications. In *Proceedings of the of the 40th IEEE Conference on Decision and Control*, Orlando, FL.

JURDZIŃSKI, M. 1998. Deciding the winner in parity games is in UP ∩ co-UP. *Information Processing Letters 68*, 3, 119–124.

KIRSTEN, D. 2002. Alternating tree automata and parity games. In *Automata, Logics, and Infinite Games—A Guide to Current Research*, E. Grädel, W. Thomas, and T. Wilke, Eds. LNCS 2500. Springer, Heidelberg, 153–167.

KRÖGER, F. 1977. Lar: a logic of algorithmic reasoning. *Acta Informatica 8*, 243–266.

KUMAR, R. AND GARG, V. 1995. *Modeling and Control of Logical Discrete Event Systems*. Kluwer Academic Publishers, Dordrecht.

LASSEZ, J.-L., NGUYEN, V., AND SONENBERG, E. 1982. Fixed point theorems and semantics: A folk tale. *Information Processing Letters 14*, 3 (May), 112–116.

LONG, D., BROWNE, A., CLARKE, E., JHA, S., AND MARRERO, W. 1994. An improved algorithm for the evaluation of fixpoint expressions. In *Conference on Computer Aided Verification (CAV)*, D. Dill, Ed. LNCS, Vol. 818. Springer, Stanford, CA, USA, 338–350.

NIWIŃSKI, D. 1986. On fixed point clones. In *International Colloquium on Automata, Languages and Programming (ICALP)*, L. Kott, Ed. LNCS of Vol 226, Springer-Verlag, Berlin, 464–473.

PNUELI, A. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*. IEEE, Providence, RI, 46–57.

RAMADGE, P. J. AND WONHAM, W. M. 1987. Supervisory control of a class of discrete event processes. *SIAM Journal of Control and Optimization 25*, 1, 206–230.

RAMADGE, P. J. AND WONHAM, W. M. 1989. The control of discrete event systems. *Proceedings of the IEEE 77*, 1, 81–98.

SAFRA, S. 1988. On the complexity of ω-automata. In *Symposium on Foundations of Computer Science (FOCS)*, 319–327.

SCHNEIDER, K. 2003. *Verification of Reactive Systems—Formal Methods and Algorithms*, Texts in Theoretical Computer Science (EATCS Series). Springer, Berlin.

SEIDL, H. 1996. Fast and simple nested fixpoints. *Information Processing Letters 59*, 6, 303–308.

TARSKI, A. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math. 5*, 2, 285–309.

THISTLE, J. G. AND WONHAM, W. M. 1994a. Control of infinite behavior of finite automata. *SIAM Journal of Control and Optimization 32*, 4, 1075–1097.

THISTLE, J. G. AND WONHAM, W. M. 1994b. Supervision of infinite behavior of discrete–event systems. *SIAM Journal of Control and Optimization 32*, 4, 1098–1113.

WILKE, T. 2001. Alternating Tree Automata, Parity Games, and Modal μ-Calculus. *Bull. Soc. Math. Belg. 8*, 2 (May).

WONHAM, W. M. 2004. Supervisory control of discrete-event systems. Tech. rep., Dept. of Electrical and Computer Engineering, University of Toronto. ECE 1636F/1637S 2004-05, http://www.control.utoronto.ca/DES.

WONHAM, W. M. AND RAMADGE, P. 1987. On the supremal controllable sulanguange of a given language. *SIAM J. Control and Optimization 25*, 3 (May), 637–659.

ZILLER, R. AND SCHNEIDER, K. 2003. A μ-calculus approach to supervisor synthesis. In *GI/ITG/GMM—Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*. 132–143.

Received September 2003; revised March 2004; accepted June 2004