A $\mu$-Calculus Approach to Supervisor Synthesis

Roberto M. Ziller$^1$ and Klaus Schneider$^2$

$^1$ University of Karlsruhe
Institute for Computer Design and Fault Tolerance
P.O. Box 6980, 76128 Karlsruhe, Germany
ziller@informatik.uni-karlsruhe.de

$^2$ University of Kaiserslautern
Department of Computer Science
P.O. Box 3049, 67653 Kaiserslautern, Germany
Klaus.Schneider@informatik.uni-kl.de

Abstract. We present a new formulation to the solution of the supervisory control problem proposed by Ramadge and Wonham. The objective of that problem is to synthesise a controller which constrains a system’s behaviour according to a given specification. We reformulate the solution using set of $\mu$-calculus equations. This makes it independent of any natural or programming languages and allows us to use well-known results on model checking to assess its computational complexity. The $\mu$-calculus representation further enables existing tools conceived for model-checking to handle synthesis without modification, and paves the way for different generalisations of the RW-framework.

1 Introduction

Many embedded systems used in safety-critical applications consist of reactive real-time controllers, whose design requires a seamless chain of automatic tools to avoid errors made by humans. Modern verification methods [1] allow designers to check a given specification for a controller, but do not support the actual specification process except for providing the designer with a simulation trace when an error has been found. Ideally, the specification itself should be generated by a tool that takes a description of the designer’s requirements and either outputs a correct specification or rejects them if they cannot be implemented.

A solution for controller synthesis that takes this approach is offered by the Ramadge-Wonham framework [2,3]. Initially, the physically possible behaviour of a system and a specification for its desired behaviour are modelled by finite-state machines. Such a specification may contain requirements that cannot be ensured by any controller. This can reflect a design error, but also be used on purpose, when it is too difficult to describe the wanted behaviour exactly. In either case, a formal synthesis procedure computes the largest subset of the specification for which a controller exists. The result can then be used in place of the original specification, as long as the corresponding behaviour is still acceptable.

The above is an informal description of the supervisory control problem proposed and solved by Ramadge and Wonham [2,4]. In its formulation, the solution
is required to satisfy two conditions. The first is *controllability*, meaning that the behaviour of the system under supervision must remain within the specification. The second is *coaccessibility*, meaning that the system must always be able to complete at least one task. The solution for this problem is traditionally given as an algorithm [3,5,6]. Although such a description precisely indicates the steps to be followed in the computation, it depends on a natural language (e.g. English) and is not well-suited for further mathematical manipulation. In this paper, we describe the solution for the supervisory control problem by a system of $\mu$-calculus equations. This makes it independent of any programming or natural languages and allows us to use well-known results on $\mu$-calculus model checking [7] to derive the time complexity of the computations. The new solution can also be understood by tools originally intended for verification, which are hereby extended to handle controller synthesis. Finally, the system of equations achieves a separation between the descriptions of controllability and coaccessibility that paves the way for future work about generalisations of the supervisory synthesis problem.

The paper is organized as follows: Section 2 presents the Ramadge-Wonham model and the supervisory control problem. Section 3 introduces the tools needed to present our main contribution, namely the solution to that problem in the $\mu$-calculus. This is given in Section 4 along with an example. The conclusion summarizes the work and points out directions for continuation. This section recalls some basic concepts related to the RW-framework originally presented in [2,4,8]; for a comprehensive description, the reader is referred to [3,5].

## 2 The Ramadge-Wonham Framework

The framework parallels classical (continuous) systems control theory, in which a system and its controller form a closed loop. There, the feedback signal from the controller influences the behaviour of the system, enforcing a given specification that would not be met by the open-loop behaviour. This foundation on control theory explains some of the terminology adopted within the RW-framework, like 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 behaviour of the system to be controlled (including unwanted situations, like the crash of two robot arms in a manufacturing cell), and that a *specification* is always a subset of this behaviour that corresponds to the actions wanted to remain executable under control.

The plant is viewed as a system that generates events as these happen. 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 behaviour through the control input, as illustrated in figure 1.
The RW-framework formulates control problems using language theory and gives the conditions under which they can be solved. Although these abstract results are not restricted to finite-state systems, the algorithms given to compute the solutions usually apply only to systems that can be described by regular languages, and hence represented by finite automata. The present paper is also concerned with this class of systems. Formally, a finite automaton is a 5-tuple \( A = (\Sigma, Q, \delta, q^0, M) \), where \( \Sigma \) is a set of event labels, \( Q \) 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 set \( M \subseteq Q \) is a set of distinguished states known as final states in traditional automata theory. In the RW-framework, these 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 \( \delta \) is traditionally defined as a partial deterministic function. We use a relation instead because this simplifies notation later on. In order to ensure the same functionality, we require the relation to be deterministic, that is, \( \delta(q, \sigma, q') \land \delta(q, \sigma, q'') \Rightarrow q' = q'' \). We write \( \delta(q, \sigma, q') \) to signify that \( (q; \sigma; q') \in \delta \).

In the following, it is convenient to define the set of active events \( \text{act}_A(q) \) as the subset of events for which there is a transition leaving state \( q \):

**Definition 1 (Active Events).** Given an automaton \( A = (\Sigma, Q, \delta, q^0, M) \) and a particular state \( q \in Q \), the set of active events of \( q \) is:

\[
\text{act}_A(q) := \{ \sigma \in \Sigma \mid \exists q' \in Q. \delta(q, \sigma, q') \}.
\]

When a plant and its supervisor are represented by finite automata \( A_P \) and \( A_S \), respectively, the control mechanism depicted in figure 1 is equivalent to running these automata in parallel, according to the following definition:

**Definition 2 (Automata product).** Given two automata \( A_P = (\Sigma, Q_P, \delta_P, q^0_P, M_P) \) and \( A_S = (\Sigma, Q_S, \delta_S, q^0_S, M_S) \), the product \( A_P \times A_S \) is the automaton \( (\Sigma, Q_P \times Q_S, \delta_{P \times S}, (q^0_P, q^0_S), M_P \times M_S) \), where

\[
\delta_{P \times S}((p, q), \sigma, (p', q')) \Leftrightarrow \delta_P(p, \sigma, p') \land \delta_S(q, \sigma, q').
\]

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

\[
\text{act}_{A_P \times A_S}((p, q)) = \text{act}_{A_P}(p) \cap \text{act}_{A_S}(q).
\]
This is how the supervisor restricts the behaviour of the plant. To forbid the occurrence of an event, it suffices to omit it in the corresponding state. Therefore, any automaton $A_E$ that represents a specification for the desired behaviour of a plant $A_P$ can be used as a supervisor, provided that, for every state $(p, q)$ of the automaton $A_P \times A_E$, every event in $\text{act}_{A_P}(p) \setminus \text{act}_{A_P \times A_E}((p, q))$ can be disabled.

However, this is not true for all events. While it is reasonable to assume that some events, like the start of a process, can always be disabled by a supervisor, there are also others that cannot be prevented from occurring, like system failures, and sensor or alarm signals. In order to reflect this, the event set $\Sigma$ is partitioned into the sets of controllable events $\Sigma_c$ (which the supervisor can disable) and uncontrollable events $\Sigma_u$ (whose occurrence cannot be avoided). Because of the existence of uncontrollable events, the desired behaviour for the plant may result in a specification not suitable as a supervisor. Such specifications 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 $A_P \times A_E$ has one or more bad states, which are states $(p, q)$ that fail to satisfy the following condition:

$$\text{act}_{A_P \times A_E}((p, q)) \supseteq \text{act}_{A_P}(p) \cap \Sigma_u.$$  \hspace{1cm} (1)

Analysing the controllability of a specification requires some language theory: Every automaton $A$ has an associated marked language, denoted $L_m(A)$, that consists of all event sequences that end up in a marker state, hence representing the tasks the system is able to complete. Extending $\delta_{P \times S}$ in the usual way to process strings from $\Sigma^*$, $L_m(A)$ is given by:

$$L_m(A) = \{ s \in \Sigma^* : \delta_{P \times S} (q^0, s, q) \land q \in M \}.$$  

The marked language of plant $A_P$ under control of supervisor $A_S$ is $L_m(A_P) \cap L_m(A_S)$ and is denoted $L_m(A_S/A_P)$.

Ramadge and Wonham have shown that, for any language $K$ representing a specification for a given plant, there exists the supremal controllable sublanguage of $K$, denoted $\sup C(K)$. This result is of practical interest: given an uncontrollable specification, it is possible to compute its largest controllable subset and to use it as a supervisor, as long as the corresponding behaviour under control is still acceptable.

Another aspect to consider is whether the supervisor always allows the system to make progress towards 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 non-blocking. A non-blocking automaton is coaccessible, which means that there is at least one path leading from every state to a marker state. Controllability and absence of blocking come together in the following problem:
Definition 3 (Supervisory Control Problem (SCP) [2]). Given a plant $A_P$, a specification $A_S$ represented by the language $K = L_m(A_S)$, and a minimally acceptable behaviour under control $A_{\text{min}} \subseteq K$, find a non-blocking supervisor $A_S$ such that $A_{\text{min}} \subseteq L_m(A_S/A_P) \subseteq K$.

SCP is solvable iff $\sup C(K) \supseteq A_{\text{min}}$, and $\sup C(K)$ is its least restrictive solution [2]. A coaccessible automaton whose marked language is equal to $\sup C(K)$ can be computed from the automata $A_P$ and $A_S$. Because the resulting automaton is a supervisor, this computation is often referred to as supervisor synthesis.

3 μ-Calculus for the RW-Framework

In this section we associate mathematical structures with the automata used in the RW-framework and define a μ-calculus over them. We will use these structures in section 4 to present our new formulation for solution of SCP. Because the μ-calculus is not usual in this context, we start with a brief review of basic concepts.

3.1 Fixpoint Calculus

Notations for extremal fixpoints of monotone operators have been introduced by different authors [9]. In particular, Tarski’s work [10] has been frequently used in verification and synthesis literature [11,12]. The following is an adaptation of the results found in these sources to suit our needs.

An operator $f : 2^X \rightarrow 2^X$ 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 the solutions of:

$$x : f(x)$$

and

$$x : f(x).$$

where the symbols $\mu$ and $\nu$ indicate that we seek for the least and greatest value of $x$ that satisfy these equations. The solutions are denoted $\mu x. f(x)$ and $\nu x. f(x)$, and known to satisfy:

$$\mu x. f(x) = \cap \{x \subseteq X : x = f(x)\} \quad \text{and} \quad \nu x. f(x) = \cup \{x \subseteq X : x = f(x)\}.$$  

Given that $X$ is finite and $f$ is monotone, the least fixpoint can be found by an iteration starting with $x_0 = \emptyset$ and computing $x_{j+1} = f(x_j)$ until, for some $j$, $x_j = x_{j-1}$ holds. The greatest fixpoint can be obtained by the same iteration starting with $x_0 = X$. The following expressions will also be useful:

$$x \mu f(x) \cup y_0 \tag{3}$$

and

$$x \nu f(x) \cap y_0 \tag{4}$$

In expression 3, $y_0 \subseteq X$ acts as a lower limit for the fixpoint, because the result of the first iteration step is $f(\emptyset) \cup y_0$, which contains $y_0$. Since $f$ is monotone, the fixpoint must also contain $y_0$. Dually, $y_0$ is an upper limit for expression 4 because $y_0 \supseteq f(X) \cap y_0$. 


3.2 Automata and Kripke Structures

Definition 4 (Kripke Structure of an Automaton). Given an automaton $A = \langle \Sigma, Q, \delta, q_0, M \rangle$ representing the product of a plant and a specification, we define its associated Kripke structure $K_A = (S, I, R, L)$ over the variables $V_A := \{x_q \mid q \in Q\} \cup \{x_b, x_m, x_u\}$ as follows:

- $S := Q \times \{0, 1\}$
- $I := \{(q_0, 0), (q_0, 1)\}$
- $R((q, 0), (q', 0)) \iff \exists \sigma \in \Sigma_u \delta(q, \sigma, q')$
- $R((q, 1), (q', 1)) \iff \exists \sigma \in \Sigma \delta(q, \sigma, q')$
- $L((q, 0)) := \{x_q, x_u\} \cup \begin{cases} \{x_b\} & \text{if } q \text{ violates condition 1} \\ \{\} & \text{else} \end{cases}$
- $L((q, 1)) := \{x_q\} \cup \begin{cases} \{x_m\} & \text{if } q \in M \\ \{\} & \text{if } q \notin M. \end{cases}$

Here, $\Sigma = \Sigma_c \cup \Sigma_u$ (see section 2), $S$ is a set of states, $I$ is the set of initial states, and $R$ 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 (regardless whether it is controllable or not) leading from $q$ to $q'$ in $A$. This creates a structure with two disconnected substructures, each of which has a copy of the original states in $A$. Finally, $L$ assigns a set of labels from $V_A$ to each state, thereby enabling us to address sets of states through Boolean expressions.

As an example, suppose the automaton from figure 2 represents the product of some plant and specification. The composite numbers of the states have been replaced by singletons for simplicity. Further, states 3 and 5 are assumed to be bad, and the event set is partitioned into $c = \{\alpha\}$ and $u = \{\beta\}$.

As an example, suppose the automaton from figure 2 represents the product of some plant and specification. The composite numbers of the states have been replaced by singletons for simplicity. Further, states 3 and 5 are assumed to be bad, and the event set is partitioned into $c = \{\alpha\}$ and $u = \{\beta\}$.

Fig. 2. Example automaton

The associated Kripke structure is shown in figure 3. Each state $(q, i)$ is labelled with the variable $x_q$. Further, the left substructure has transitions only where the automaton has uncontrollable transitions, and its states have in common the label $x_u$. Additionally, the states that correspond to bad states in the automaton have the label $x_b$. The right substructure reflects all transitions from the automaton, with the labels $x_m$ identifying the states that correspond to
marker states. Note that \( x_u \) distinguishes the states from the two substructures, and that the left side does not know anything about the marker states, while the right side does not know about the bad states.

**Fig. 3.** Associated Kripke structure

Syntax and semantics of \( \mu \)-calculus formulas are given in the following definitions:

**Definition 5 (Syntax of \( \mu \)-Calculus).** Given a set of variables \( \mathcal{V} \), the set of \( \mu \)-calculus formulas over \( \mathcal{V} \) is defined as the least set \( \mathcal{L}_\mu \) that satisfies the following rules:

\[ \begin{align*}
& \forall \in \mathcal{L}_\mu \\
& 0, 1 \in \mathcal{L}_\mu \\
& \neg \varphi, \varphi \lor \psi, \varphi \land \psi \in \mathcal{L}_\mu, \text{ provided that } \varphi, \psi \in \mathcal{L}_\mu \\
& \text{EX} \varphi, \text{EY} \varphi \in \mathcal{L}_\mu \\
& \mu \varphi \in \mathcal{L}_\mu, \text{ provided that } \varphi \in \mathcal{L}_\mu.
\end{align*} \]

Definition 5 differs from those usually found in the literature in that it includes the formula \( \mu \varphi \). This allows us to use any monotone state transformer function \( \pi : 2^S \to 2^S \) in the computations. In particular, we will define a function \( \pi \) to map states of one of the above substructures to the other.

**Definition 6 (Semantics of \( \mu \)-Calculus).** Given a Kripke structure \( \mathcal{K} = \langle S, I, R, \mathcal{L} \rangle \) over the variables \( \mathcal{V} \), we associate with each formula \( \varphi \in \mathcal{L}_\mu \) a set of states \( [\varphi]_{\mathcal{K}} \subseteq S \) by the following rules:

\[ \begin{align*}
& [0]_{\mathcal{K}} := \{\} \text{ and } [1]_{\mathcal{K}} := S \\
& [x]_{\mathcal{K}} := \{s \in S \mid x \in \mathcal{L}(s)\} \text{ for all variables } x \in \mathcal{V} \\
& [-\varphi]_{\mathcal{K}} := S \setminus [\varphi]_{\mathcal{K}} \\
& [[\varphi \land \psi]_{\mathcal{K}} := [\varphi]_{\mathcal{K}} \cap [\psi]_{\mathcal{K}} \\
& [[\varphi \lor \psi]_{\mathcal{K}} := [\varphi]_{\mathcal{K}} \cup [\psi]_{\mathcal{K}} \\
& [[\mu \varphi]_{\mathcal{K}} := \pi([[\varphi]_{\mathcal{K}}]) \text{ for the monotone function } \pi : 2^S \to 2^S \\
& [\text{EX} \varphi]_{\mathcal{K}} := \{s \in S \mid \exists s' \in S. R(s, s') \land s' \in [\varphi]_{\mathcal{K}}\} \\
& [\text{EY} \varphi]_{\mathcal{K}} := \{s' \in S \mid \exists s \in S. R(s, s') \land s \in [\varphi]_{\mathcal{K}}\} \\
& [\mu x. \varphi]_{\mathcal{K}} := \{Q \subseteq S \mid [\varphi]_{\mathcal{K}}^Q \subseteq Q\}.
\end{align*} \]
The last expression gives the least set of states \( Q \subseteq S \) such that \( Q = \{ \| \varphi \|_{K^Q} \} \) holds, where \( K^Q \) is the Kripke structure where exactly the states \( Q \) are labelled with the variable \( x \). If \( \varphi \) is a monotonic function of \( x \), then \( \mu x. \varphi \) is its least fixpoint [10]. We can also define some further macro operators like \( \text{AX} \varphi := \neg \text{EX} \neg \varphi \), \( \text{AY} \varphi := \neg \text{EY} \neg \varphi \), and \( \mu x. \varphi(x) := \neg \mu x. \neg \varphi(\neg x) \). The latter can be shown to be the greatest fixpoint of \( \varphi \).

In order to apply the above definitions to Kripke structures stemming from automata according to definition 4, we define \( \{ (q,0) \} \) for all \( q \) that violate condition 1

\[ [x_0]_{K_A} := \{ (q,0) \} \text{ for all } q \text{ that violate condition 1} \]

\[ [x_m]_{K_A} := \{ (q,1) \} \text{ for all } q \in M \]

\[ [x_q]_{K_A} := \{ q \} \times \{ 0, 1 \} \text{ for all } q \in Q \]

\[ [x_u]_{K_A} := \{ (q,0) \} \text{ for all } q \in Q \]

\[ [\neg x_u]_{K_A} := \{ (q,1) \} \text{ for all } q \in Q. \]

Any formula \( \Phi \) over the variables \( V_A \) also describes a subset of the states of \( A \) according to the following projection, which maps states from the Kripke structure back to the originating automaton:

**Definition 7 (Kripke Structure State Projection).** Given an automaton \( A \), its associated Kripke structure \( K_A \) and a \( \mu \)-calculus formula \( \Phi \) over the variables \( V_A \), we define the projection of \( [\Phi]_{K_A} \) onto the state set \( Q \) of \( A \) as:

\[ [\Phi]_A = \{ q \in Q \mid (q,0) \in [\Phi]_{K_A} \lor (q,1) \in [\Phi]_{K_A} \} . \] (5)

With this projection, we can construct an automaton from \( \Phi \) by restricting the state set of the original automaton \( A \) to \( [\Phi]_A \). This completes the toolset we need to convert an automaton into a Kripke structure, compute a subset of states, and translate the result back into an automaton. For example, the set of states of \( \tau \subseteq \delta \) that are accessible only through states lying in \( \tau \) is given by \( [x_{Ac}]_A \), with:

\[ x_{Ac} \equiv \Phi_\tau \land (\text{EY} x_{Ac} \lor x_{\varphi_0}), \] (6)

where \( \Phi_\tau \) represents the states of \( \tau \). Similarly, the set of states of \( \tau \subseteq \delta \) that are coaccessible only through states lying in \( \tau \) is given by \( [x_{Co}]_A \), with:

\[ x_{Co} \equiv \Phi_\tau \land (\text{EX} x_{Co} \lor x_{m}). \] (7)

Of special interest is the _alternation depth_ of a fixpoint expression [11,13]. Roughly speaking, this is the nesting depth of alternating \( \mu \) and \( \nu \)-operators which depend on each other to compute. Expressions with a single operator have alternation depth 1 and are also called _alternation-free_. We also consider systems of \( \mu \)-calculus equations [14,7]. In order to define a semantics for them, it suffices to say that any such system of equations can be reduced to a single expression
which can be interpreted according to definition 6. The alternation depth of the resulting expression is equal to the largest number of blocks of formulas seeking for a least or greatest fixpoint in the equation system that depend on each other to compute. The following result will be useful to assess the computational complexity of our solution:

**Theorem 1 (Complexity of \(\mu\)-Calculus Model Checking [14]).** For every equation system \(E\) of alternation depth \(l\) and every Kripke structure \(K = \langle S, I, R, L \rangle\), there is an algorithm to compute its solution in time

\[
O \left( \left( \frac{|E|}{l} \right)^{l-1} |R||E| \right),
\]

where \(|E|\) is the length of \(E\), computed by adding the lengths of the right sides of all equations in the system.

**Corollary 1.** Any system of equations of constant length and alternation depth \(l\) formulated for a Kripke structure associated to an automaton can be solved in time

\[
O \left( |Q|^l |\Sigma| \right).
\]

**Proof.** By construction, the Kripke structure from definition 4 has \(|S| = 2|Q|\) and \(|R| \leq 2|Q||\Sigma|\). For constant \(|E|\), the above result follows immediately from theorem 1.

4 Implementing Ramadge-Wonham in \(\mu\)-Calculus

In this section we present a solution for the supervisory control problem using the Kripke structure associated to the automaton \(A_P \times A_E\). This enables us to express our solution for SCP through a system of \(\mu\)-calculus equations and to apply corollary 1 to assess its computational complexity. An example illustrates the computations on the new structure.

4.1 Solving SCP

There are two slightly different approaches in the literature to solve SCP, namely the original algorithm [4,5,3] and the one given in [6]. The first approach compares the automata \(A_P\) and \(A_P \times A_E\) to find all states that are initially bad and then removes them from \(A_P \times A_E\) along with their transitions. Next, the resulting automaton is made trim, i.e., accessible and coaccessible. Because removing bad states can destroy coaccessibility and removing non-coaccessible states can expose new bad states, the algorithm is restarted with the trimmed automaton \(A_P \times A_E\), until a fixpoint is reached. Therefore, the search for initial bad states has to be repeated at each iteration. Since this requires information from \(A_P\) and \(A_P \times A_E\) 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 [6] does not eliminate bad or non-coaccessible states immediately, but rather 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, and the trimming operation is substituted by just identifying non-coaccessible states. The initial bad states have to be computed only once at the beginning of the solution process. This corresponds to the computation of the set \([x_b]_{KA}\), which is part of the construction of the Kripke structure associated to \(A_P \times A_E\). We therefore use the latter approach.

Bad and non-coaccessible states are collected into sets denoted \(x_B\) and \(x_N\), respectively. In the Kripke structure, we have to take into account on which of the two substructures the computation is happening. Because we are interested only in predecessors of collected states connected to them through uncontrollable transitions, the substructure identified by \(x_u\) is the correct choice to collect bad states. On the other hand, coaccessibility requires considering all transitions, and therefore the substructure identified by \(\neg x_u\) must be used to compute \(x_N\). When it comes to using the non-coaccessible states in the computation of \(x_B\), we switch from one substructure to the other, using the set \(x_N\). The above description resembles equation 3 and translates naturally into expression 11 below.

To derive an expression for \(x_N\), we first set \(\Phi_r = \kappa (x_B)\) in equation 7 to keep only coaccessible states that are not bad:

\[
x_C \triangleq \kappa (x_B) \land (EX x_C \lor x_m).
\]

Next, we complement expression 8 to obtain the non-coaccessible states. Note this makes the expression a greatest fixpoint:

\[
\neg x_C \triangleq \kappa (x_B) \lor AX \neg x_C \land \neg x_m.
\]

While states in \(\kappa (x_B)\) are all on the substructure identified by \(\neg x_u\), the term \(\neg x_m\) introduces unwanted states from \(x_u\). We therefore restrict \(\neg x_m\) and its predecessors to \(\neg x_u\) in equation 10, where \(\neg x_C\) is also substituted by \(x_N\). This completes our equation system to describe the solution of SCP:

\[
\begin{align*}
  x_N & \triangleq \kappa (x_B) \lor \neg x_u \land AX x_N \land \neg x_m \quad (10) \\
  x_B & \triangleq EX (x_B \lor \kappa (x_N)) \lor x_p. \quad (11)
\end{align*}
\]

The solution for SCP is the accessible component of the transition relation that results when the bad states and the non-coaccessible states are eliminated. Since accessibility involves not only the uncontrollable transitions, the set of accessible states must be computed on the substructure identified by \(\neg x_u\). This can be done by setting \(\Phi_r = \gamma (\kappa (x_B) \lor x_N) \land \neg x_u\) in equation 6:

\[
x_{Ac} \triangleq \gamma (\kappa (x_B) \lor x_N) \land \neg x_u \land (EY x_{Ac} \lor x_p),
\]

so the solution for SCP is given by restricting the automaton \(A_P \times A_E\) to the states \(\{x_{Ac}\}_A\). The complexity of the overall computation is given by corollary 1. Since our solution has \(l = 2\), we get \(O(|Q|^2 |\Sigma|)\), which is the known complexity of both original algorithms [5,8,6].
4.2 Example

To illustrate the computations done by the new algorithm, we solve the problem posed by the automaton from 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, with \( [x_B]_{K_A} = \{(3,0),(5,0)\} \) and \([x_m]_{K_A} = \{(4,1)\} \). The solution of equations 10 and 11 starts with the initial values \( x_B^0 = 0 \) (least fixpoint) and \( x_N^0 = 1 \) (greatest fixpoint). The iteration steps are given below. To improve readability, we display the states represented by each expression (e.g. \( \{(0,1),(1,1)\} \)) instead of the expression itself (in this case, \( (x_0 \lor x_1) \land \neg x_u \)). \([x_N]^{1,j} \) denotes the result of the \( j \)-th iteration for the \( i \)-th computation of \( x_N \), and similarly for \( x_B \). The index \( K_A \) has been omitted for simplicity.

\[
\begin{align*}
[x_N]^{1,0} &= \{(0,1),(1,1),(2,1),(3,1),(5,1)\} \\
[x_N]^{1,1} &= \{(0,1),(3,1)\} \\
[x_N]^{1,2} &= \{\} \\
[x_N]^{1,3} &= \{\}
\end{align*}
\]

\[
\begin{align*}
[x_B]^{1,0} &= \{(3,0),(5,0)\} \\
[x_B]^{1,1} &= \{(1,0),(3,0),(5,0)\} \\
[x_B]^{1,2} &= \{(1,0),(3,0),(5,0)\}
\end{align*}
\]

\[
\begin{align*}
[x_N]^{2,0} &= \{(0,1),(1,1),(2,1),(3,1),(5,1)\} \\
[x_N]^{2,1} &= \{(0,1),(1,1),(3,1),(5,1)\} \\
[x_N]^{2,2} &= \{(1,1),(3,1),(5,1)\} \\
[x_N]^{2,3} &= \{(1,1),(3,1),(5,1)\}
\end{align*}
\]

\[
\begin{align*}
[x_B]^{2,0} &= \{(1,0),(3,0),(5,0)\} \\
[x_B]^{2,1} &= \{(1,0),(3,0),(5,0)\}
\end{align*}
\]

The fixpoints are \([x_N]^{2,3} \) and \([x_B]^{2,1} \). Equation 12 can now be solved with \( [\neg (\kappa [x_B] \lor x_N) \land \neg x_u] = \{(0,1),(2,1),(4,1)\} \) and \([x_q^0] = \{(0,0),(0,1)\} \), yielding the following steps:

\[
\begin{align*}
[x_Ac]^{1,0} &= \{(0,1)\} \\
[x_Ac]^{1,1} &= \{(0,1),(2,1)\} \\
[x_Ac]^{1,2} &= \{(0,1),(2,1),(4,1)\} \\
[x_Ac]^{1,3} &= \{(0,1),(2,1),(4,1)\}
\end{align*}
\]

The solution is given by restricting the original automaton to the states given by equation 5, namely:

\[
[x_Ac]_{K_A} = \{0,2,4\}.
\]
5 Conclusion

The new formulation of the solution for SCP provides a representation which is independent of any programming or natural languages, using equations that can be handled by software tools not necessarily conceived for supervisor synthesis. The solution is also amenable to different extensions: the coaccessibility condition represented by equation 10 can be exchanged or extended by supplementary equations to reflect requirements not expressible in the conventional RW-framework, like fairness or other system characteristics often present in formal verification. Further, if, for a subclass of problems, the same equation can be replaced by a least fixpoint, then, in view of corollary 1, these problems become solvable in linear time. Both possibilities are currently under investigation.

References