

REDUCING COMPLEXITY OF SUPERVISOR SYNTHESIS

Roberto Ziller * Klaus Schneider **

* *University of Karlsruhe, Germany*

** *University of Kaiserslautern, Germany*

AbstractThe objective of the supervisory control problem proposed by Ramadge and Wonham is to find a supervisor that constrains a system's behaviour according to a given specification. All its known solutions are of quadratic time complexity wrt. the number of states of the product of the system to be controlled and the specification. We present a new solution that can solve a large class of practical problems in linear time. Experimental results show that the improvement is not only a theoretical issue, but enables us to handle industrial-size problems efficiently.

Keywords: Supervisor synthesis, Ramadge-Wonham, μ -calculus.

1. INTRODUCTION

Many embedded systems used in safety-critical applications consist of reactive real-time controllers, whose design requires automatic tools to improve efficiency and avoid errors made by humans. Modern verification methods (Clarke *et al.*, 1999) allow designers to check a given specification for a controller, but do not support the actual specification process except for providing a simulation trace when an error has been found. Ideally, the specification itself should be generated by a tool that takes the informal requirements of the designer and either outputs a correct specification or rejects them if they cannot be implemented. In general, the result of such a process is a representation of various possible solutions. The present work relates this representation to the notion of *supervisor* and the process of generating it to existing *supervisor synthesis* methods that use finite automata.

A supervisor is an entity that observes the sequence of events in a system and affects the next possible events in order to ensure a desired behaviour. Supervisor synthesis methods and their application vary according to the requirements they can handle. One aspect is whether a method distinguishes events that can be prevented from happening, like turning on an actuator,

from those that can not, like a stimulus coming from a sensor. This can be modelled by dividing the event set of an automaton into *controllable* and *uncontrollable* events. A second aspect is whether the requirements may include assertions about deadlocks and livelocks, collectively referred to as *blocking* situations. These can be modelled through an automaton's final states. The latter define a language, and blocking freedom means that a system never gets stuck at a proper prefix of a string in that language. If a model is not concerned with blocking, then any generated prefix will be acceptable and the associated language will be prefix-closed.

A solution that can handle both controllability and blocking is offered by the Ramadge-Wonham (RW) framework (Ramadge and Wonham, 1987; Wonham and Ramadge, 1987; Wonham, 2002; Cassandras and Lafortune, 1999). It includes a method to synthesise a supervisor from two automata, one representing the physically possible behaviour of the system to be controlled, and the other a specification for its desired behaviour. The latter is not yet the product of a formal procedure, but rather a translation of the designer's requirements into a finite automaton. Therefore, it may express the wish for a behaviour of the system that is impossible to implement (for example, preventing

a part to be loaded onto a manufacturing machine in a state where this happens automatically). It may also contain blocking situations not realised by the designer. A formal *synthesis procedure* then takes both automata above and generates a third automaton from them. The latter represents the largest implementable and non-blocking subset of the specification. It can therefore replace the original specification, as long as the resulting behaviour is still acceptable, in which case it acts as a supervisor for the system. The strength of the method lies in this formal synthesis capability.

Other methods, some of which were inspired in RW, were developed in parallel. For example, (Aziz *et al.*, 1995) consider the problem of finding a finite state machine (FSM) that, composed with a given FSM, satisfies a given specification. The work is restricted to prefix-closed languages and so applies to problems where blocking considerations are not necessary. In (Yevtushenko *et al.*, 2001), a similar problem is cast in terms of solving equations over languages. Here, prefix-closure is no longer a restriction, but there is no distinction between controllable and uncontrollable events. Failure semantics is used in (Overkamp, 1997) to introduce non-determinism. The method uses prefix-closed languages and cannot handle livelocks. It can handle deadlocks, but there is no distinction between the case where a system stops generating events because it has accomplished some task from that where it just gets stuck in the middle of a task. In (Kumar *et al.*, 1997), RW-synthesis is applied to protocol conversion and extended by a requirement that ensures progress, which also avoids deadlocks. The procedure uses prefix-closed languages and is not concerned with livelocks.

This paper is devoted to the *supervisory control problem* (Ramadge and Wonham, 1987), which considers controllability of events as well as both types of blocking. Given a system with state set $Q_{\mathcal{P}}$ and a specification with state set $Q_{\mathcal{E}}$ that communicate over an event set Σ , the known synthesis algorithms (Wonham and Ramadge, 1987; Kumar and Garg, 1995) have time complexity $O((|Q_{\mathcal{P}}| |Q_{\mathcal{E}}|)^2 |\Sigma|)$ (Ramadge and Wonham, 1989; Cassandras and Lafortune, 1999). An attempt to solve the problem in linear time by analysing both controllability and blocking in the same pass of an algorithm was first made in (Ziller and Cury, 1994), but the solution obtained there was not guaranteed to be livelock-free. Up to now, only the special case of prefix-closed languages (i.e., without considering any type of blocking) could be solved in linear time (Kumar and Garg, 1995).

In this paper, we identify a class of problems that strictly contains the prefix-closed case and show how to solve the problems therein in time $O(|Q_{\mathcal{P}}| |Q_{\mathcal{E}}| |\Sigma|)$. *This means that many problems concerned with blocking, for which the existing algorithms require quadratic runtime, can now be solved in linear time.* We also argue that the new class contains many problems

of practical importance. For the problems outside of it, the new solution automatically falls back into quadratic runtime, but will iterate fewer times than the existing algorithms. Therefore, the result improves all known solutions to the supervisory control problem.

Algorithms depend on some programming or informal language and are thus not well-suited for mathematical manipulation. In this work, we choose a formulation based on μ -calculus, which enables us to first present a new formulation for the algorithm in (Kumar and Garg, 1995) and then modify it to derive our main result. This approach is independent of any such languages and allows us to use well-known results about model checking (Cleaveland *et al.*, 1992) to assess the complexity of the new solution. Further, the equations can be understood by tools originally intended for verification, which are hereby extended to also handle controller synthesis.

The paper is organized as follows: Section 2 presents the Ramadge-Wonham model and states the problem being solved. Section 3 introduces the theoretical tools needed and describes the existing algorithm by means of μ -calculus expressions. Section 4 builds on these expressions to derive the new solution, and Section 5 discusses implementation and shows experimental results.

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 enforces a given specification that would not be met by the open-loop behaviour. This foundation on control theory explains the use of 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 to the assumptions that the latter encompasses the whole possible behaviour of the system (including unwanted situations, like the crash of two robot arms in a manufacturing cell), and that a specification describes 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. It is assumed to have a control input, through which some of the events that could happen in each state can be prevented from occurring. The *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 shown in Figure 1.

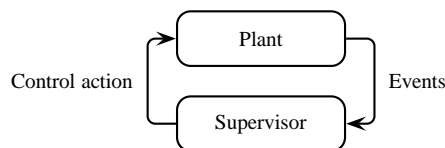


Figure 1. The basic RW-model

Control problems are formulated using language theory and 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 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 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 deterministic *function*. We use a *relation* instead because this simplifies notation later on. 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, $\delta(q, \sigma, q') \wedge \delta(q, \sigma, q'') \Rightarrow q' = q''$.

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

Definition 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:

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

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

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

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., $\text{act}_{\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{S}}}((p, q)) = \text{act}_{\mathcal{A}_{\mathcal{P}}}(p) \cap \text{act}_{\mathcal{A}_{\mathcal{S}}}(q)$. The control action of the supervisor enables only the events in $\text{act}_{\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{S}}}$. 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 $\mathcal{A}_{\mathcal{S}}$.

However, reactive systems may contain events that can not be prevented from occurring. The event set Σ is therefore partitioned into the sets of *controllable events* Σ_c (which the supervisor can disable) and *uncontrollable events* Σ_u (whose occurrence cannot be avoided). This places a condition on the existence 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 $\text{act}_{\mathcal{A}_{\mathcal{P}}}(p) \setminus \text{act}_{\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}}((p, q))$ is controllable.

Specifications that do not fulfil 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:

$$\text{act}_{\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}}((p, q)) \supseteq \text{act}_{\mathcal{A}_{\mathcal{P}}}(p) \cap \Sigma_u. \quad (1)$$

Analysing the controllability of a specification further requires some language theory: Every automaton \mathcal{A} has an associated *marked language*, denoted $L_m(\mathcal{A})$, 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^* ,

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

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

The marked language of plant $\mathcal{A}_{\mathcal{P}}$ under control of supervisor $\mathcal{A}_{\mathcal{S}}$ is equal to $L_m(\mathcal{A}_{\mathcal{P}}) \cap L_m(\mathcal{A}_{\mathcal{S}})$, and denoted $L_m(\mathcal{A}_{\mathcal{S}}/\mathcal{A}_{\mathcal{P}})$. Ramadge and Wonham have shown that, for any plant $\mathcal{A}_{\mathcal{P}}$ and any specification language $K \subseteq L_m(\mathcal{A}_{\mathcal{P}})$, there exists the *supremal controllable sublanguage* of K , denoted $\text{supC}(K)$. This result is of practical interest: Given that the specification language K is uncontrollable, it is possible to compute $\text{supC}(K)$ and to construct a supervisor $\mathcal{A}_{\mathcal{S}}$ such that $L_m(\mathcal{A}_{\mathcal{S}}/\mathcal{A}_{\mathcal{P}}) = \text{supC}(K)$. This language can replace the original specification, as long as the resulting 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 problem that is central to the present work:

Definition 3. (Supervisory Control Problem (SCP)).

Given a plant $\mathcal{A}_{\mathcal{P}}$, a specification language $K \subseteq L_m(\mathcal{A}_{\mathcal{P}})$ representing the desired behaviour of $\mathcal{A}_{\mathcal{P}}$ under supervision, and a minimally acceptable behaviour $A_{\min} \subseteq K$, find a non-blocking supervisor $\mathcal{A}_{\mathcal{S}}$ such that $A_{\min} \subseteq L_m(\mathcal{A}_{\mathcal{S}}/\mathcal{A}_{\mathcal{P}}) \subseteq K$.

SCP is solvable if and only if $\text{supC}(K) \supseteq A_{\min}$, and $\text{supC}(K)$ is its least restrictive solution (Ramadge and Wonham, 1987). A coaccessible automaton $\mathcal{A}_{\mathcal{S}}$ whose marked language is equal to $\text{supC}(K)$ can be computed from the automata $\mathcal{A}_{\mathcal{P}}$ and $\mathcal{A}_{\mathcal{E}}$, with $\mathcal{A}_{\mathcal{E}}$ constructed so that $L_m(\mathcal{A}_{\mathcal{E}}) = K$. Because the resulting automaton is a supervisor, this computation is often referred to as *supervisor synthesis*.

The reader is referred to (Wonham, 2002; Cassandras and Lafortune, 1999) for further details about the RW-framework.

3. CLASSICAL SUPERVISOR SYNTHESIS

In this section we associate Kripke structures with the automata used in the RW-framework and define a μ -calculus over them. These structures are used in Subsection 3.3 to describe the classical solution in the sense of (Wonham and Ramadge, 1987; Kumar and Garg, 1995). 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 (Lassez *et al.*, 1982). In particular, Tarski's work (Tarski, 1955) has been frequently used in verification and synthesis literature (Emerson and Lei, 1986; Thistle and Wonham, 1994). 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 $x_i, x_j \subseteq X$,

$$x_i \subseteq x_j \Rightarrow f(x_i) \subseteq f(x_j). \quad (2)$$

Such an operator has least and greatest fixpoints, which are the solutions of $x \stackrel{\mu}{=} f(x)$ and $x \stackrel{\nu}{=} f(x)$, where the symbols $\stackrel{\mu}{=}$ and $\stackrel{\nu}{=}$ indicate the seek for the least and greatest values of x that satisfy these equations. The solutions are denoted $\mu x.f(x)$ and $\nu x.f(x)$, and known to satisfy:

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

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 $x_{i+1} = f(x_i)$ until, for some j , $x_j = x_{j-1}$ holds. The greatest fixpoint can be obtained by the same iteration starting with $x_0 = X$. In this paper, the fixpoint operators will be applied to the state set of a Kripke structure.

We shall use a version of the modal μ -calculus, defined on equation systems (Cleaveland *et al.*, 1992; Schneider, 2003) 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 $i \in \{1, \dots, n\}$. Such a system can be translated into a single fixpoint expression that uses the operators μ and ν , and vice versa. The formulas φ_i may consist of the propositional operators \neg , \wedge , and \vee , and the modal operators EX, AX, EY, and AY. Here, E and A are path quantifiers that specify

that the property that follows them in the expression must hold on at least one path (E) or on all paths (A) from some state in a Kripke structure. X and Y are temporal operators that limit the length of the path to the immediate successor or the immediate ancestor states, respectively (see (Clarke *et al.*, 1999; Schneider, 2003) for background on modal operators and temporal logics). As usual, we require that the occurrences of all x_j in every φ_i must occur under an even number of negation symbols.

3.2 Automata and Kripke Structures

A Kripke structure is a finite transition system. As opposed to automata, the transitions of a Kripke structure are not labelled. The transition relation only indicates which states can be reached from a given state, but not why or how. There is also a set of Boolean variables, say, \mathcal{V} , and a labelling function that maps each state of the Kripke structure onto a subset of \mathcal{V} . The effect of this mapping is to associate with each state a number of Boolean variables which are considered to be true in that state. This is useful to write Boolean expressions representing sets of states of the Kripke structure, e.g. $x_1 \wedge x_2$, which denotes all states in which both variables x_1 and $x_2 \in \mathcal{V}$ are true.

While the definition of a Kripke structure as described above is independent of any automaton, we can also define a special type of structure to suit our specific needs:

Definition 4. (Kripke Structure of an Automaton).

Given an automaton $\mathcal{A} = \langle \Sigma, Q, \delta, q^0, M \rangle$ representing the product of a plant and a specification, we define its associated Kripke structure $\mathcal{K}_{\mathcal{A}} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ over the Boolean variables $\mathcal{V}_{\mathcal{A}} := \{x_q \mid q \in Q\} \cup \{x_b, x_m, x_u\}$ as follows:

- $\mathcal{S} := Q \times \{0, 1\}$
- $\mathcal{I} := \{(q^0, 0), (q^0, 1)\}$
- $\mathcal{R}((q, 0), (q', 0)) := \Leftrightarrow \exists \sigma \in \Sigma_u. \delta(q, \sigma, q')$
- $\mathcal{R}((q, 1), (q', 1)) := \Leftrightarrow \exists \sigma \in \Sigma. \delta(q, \sigma, q')$
- $\mathcal{L}((q, 0)) := \{x_q, x_u\} \cup \begin{cases} \{x_b\} & \text{if } q \text{ is bad} \\ \{\} & \text{otherwise} \end{cases}$
- $\mathcal{L}((q, 1)) := \{x_q\} \cup \begin{cases} \{x_m\} & \text{if } q \in M \\ \{\} & \text{if } q \notin M. \end{cases}$

Here, $\Sigma = \Sigma_c \cup \Sigma_u$ (see Section 2), \mathcal{S} is a 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} . Finally, \mathcal{L} assigns a set of labels from $\mathcal{V}_{\mathcal{A}}$ to each state, thereby enabling us to address sets of states through Boolean expressions. Note that the Kripke structure can be constructed from the automaton in time $O(|Q| |\Sigma|)$.

As an example, suppose the automaton in Figure 2 represents the product of some plant and specification. The composite numbers of the states have been replaced by singletons for simplicity. States 3 and 5 are assumed to be bad, and the event set is partitioned into $\Sigma_c = \{\alpha, \lambda\}$ and $\Sigma_u = \{\beta, \gamma\}$. The two halves

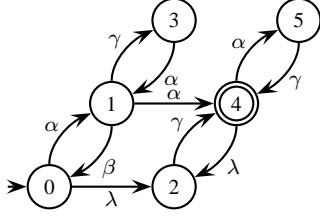


Figure 2. Example automaton

of the associated Kripke structure are shown in Figure 3. Each state (q, i) is labelled with the variable x_q . 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

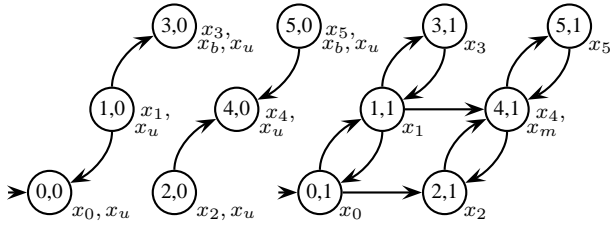


Figure 3. The associated Kripke structure

from the automaton, with the label x_m identifying the state that corresponds to a marker state. 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.

The following definitions give syntax and semantics of μ -calculus formulas:

Definition 5. (Syntax of μ -Calculus). Given a set of variables \mathcal{V} , the set of μ -calculus formulas over \mathcal{V} is defined as the least set \mathcal{L}_μ 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 $\varphi, \psi \in \mathcal{L}_\mu$
- $\text{EX } \varphi, \text{EY } \varphi \in \mathcal{L}_\mu$
- $\kappa_\pi(\varphi) \in \mathcal{L}_\mu$, provided that $\varphi \in \mathcal{L}_\mu$
- $\mu x. \varphi \in \mathcal{L}_\mu$, provided that $\varphi \in \mathcal{L}_\mu$.

Definition 5 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 $\pi : 2^S \rightarrow 2^S$ in the computations. In particular, we will define a function π to map states of one of the above substructures to the other.

Definition 6. (Semantics of 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 $\llbracket \Phi \rrbracket_{\mathcal{K}} \subseteq \mathcal{S}$ by the following rules:

- $\llbracket 0 \rrbracket_{\mathcal{K}} := \{\}$ and $\llbracket 1 \rrbracket_{\mathcal{K}} := \mathcal{S}$
- $\llbracket x \rrbracket_{\mathcal{K}} := \{s \in \mathcal{S} \mid x \in \mathcal{L}(s)\}$ for all $x \in \mathcal{V}$
- $\llbracket \neg\varphi \rrbracket_{\mathcal{K}} := \mathcal{S} \setminus \llbracket \varphi \rrbracket_{\mathcal{K}}$
- $\llbracket \varphi \wedge \psi \rrbracket_{\mathcal{K}} := \llbracket \varphi \rrbracket_{\mathcal{K}} \cap \llbracket \psi \rrbracket_{\mathcal{K}}$
- $\llbracket \varphi \vee \psi \rrbracket_{\mathcal{K}} := \llbracket \varphi \rrbracket_{\mathcal{K}} \cup \llbracket \psi \rrbracket_{\mathcal{K}}$
- $\llbracket \kappa_\pi(\varphi) \rrbracket_{\mathcal{K}} := \pi(\llbracket \varphi \rrbracket_{\mathcal{K}})$, $\pi : 2^S \rightarrow 2^S$ monotone
- $\llbracket \text{EX } \varphi \rrbracket_{\mathcal{K}} := \{s \in \mathcal{S} \mid \exists s' \in \mathcal{S}. \mathcal{R}(s, s') \wedge s' \in \llbracket \varphi \rrbracket_{\mathcal{K}}\}$
- $\llbracket \text{EY } \varphi \rrbracket_{\mathcal{K}} := \{s' \in \mathcal{S} \mid \exists s \in \mathcal{S}. \mathcal{R}(s, s') \wedge s \in \llbracket \varphi \rrbracket_{\mathcal{K}}\}$
- $\llbracket \mu x. \varphi \rrbracket_{\mathcal{K}} := \bigcap \{Q \subseteq \mathcal{S} \mid \llbracket \varphi \rrbracket_{\mathcal{K}_x^Q} \subseteq Q\}$.

The last expression in Definition 6 gives the least set of states $Q \subseteq \mathcal{S}$ such that $Q = \llbracket \varphi \rrbracket_{\mathcal{K}_x^Q}$ holds, where \mathcal{K}_x^Q is the Kripke structure where exactly the states Q are labelled with the variable x . If φ is a monotonic function of x , then $\mu x. \varphi$ is its least fixpoint (Tarski, 1955). 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 $\nu x. \varphi(x) := \neg \mu x. \neg\varphi(\neg x)$. The latter can be shown to be the greatest fixpoint of φ .

Together, Definitions 5 and 6 enable us to write Boolean expressions to denote sets of states, as described informally at the beginning of the current subsection. For example, for the Kripke structure of Figure 3, $\llbracket x_u \wedge (x_3 \vee x_5) \rrbracket_{\mathcal{K}_A} = \{(3, 0), (5, 0)\}$, and $\llbracket \text{EX } x_m \rrbracket_{\mathcal{K}_A} = \{(1, 1), (2, 1), (5, 1)\}$.

In order to apply the above definitions to Kripke structures stemming from automata according to Definition 4, we define $\pi(Q) := \{(q, \neg i) \mid (q, i) \in Q\}$, which 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, thereby enabling us to switch from one substructure to the other. For simplicity, we write just κ for κ_π from this point on. Further, we have:

- $\llbracket x_b \rrbracket_{\mathcal{K}_A} := \{(q, 0)\}$ for all q that violate cond. 1
- $\llbracket x_m \rrbracket_{\mathcal{K}_A} := \{(q, 1)\}$ for all $q \in M$
- $\llbracket x_q \rrbracket_{\mathcal{K}_A} := \{q\} \times \{0, 1\}$ for all $q \in Q$
- $\llbracket x_u \rrbracket_{\mathcal{K}_A} := \{(q, 0)\}$ for all $q \in Q$
- $\llbracket \neg x_u \rrbracket_{\mathcal{K}_A} := \{(q, 1)\}$ for all $q \in Q$

Any formula Φ over the variables \mathcal{V}_A 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 7. (Kripke Structure State Projection).

Given an automaton \mathcal{A} , its associated Kripke structure \mathcal{K}_A and a μ -calculus formula Φ over the variables \mathcal{V}_A , we define the projection of $\llbracket \Phi \rrbracket_{\mathcal{K}_A}$ onto the state set Q of \mathcal{A} as:

$$\llbracket \Phi \rrbracket_{\mathcal{A}} = \{q \in Q \mid (q, 0) \in \llbracket \Phi \rrbracket_{\mathcal{K}_A} \vee (q, 1) \in \llbracket \Phi \rrbracket_{\mathcal{K}_A}\}. \quad (3)$$

With this projection, we can construct an automaton from Φ by restricting the state set of the original automaton \mathcal{A} to $\llbracket \Phi \rrbracket_{\mathcal{A}}$. This completes the toolset to convert an automaton into a Kripke structure, compute a subset of states, and translate the result back. For example, the set of states of $\tau \subseteq \delta$ that are accessible only through states pertaining to τ is given by $\llbracket x_{Ac} \rrbracket_{\mathcal{A}}$, with:

$$x_{Ac} \stackrel{\mu}{=} \Phi_{\tau} \wedge (\text{EY } x_{Ac} \vee x_{q^0}), \quad (4)$$

where Φ_{τ} represents the states of τ . Similarly, the set of states of $\tau \subseteq \delta$ that are coaccessible only through states pertaining to τ is given by $\llbracket x_{Co} \rrbracket_{\mathcal{A}}$, with:

$$x_{Co} \stackrel{\mu}{=} \Phi_{\tau} \wedge (\text{EX } x_{Co} \vee x_m). \quad (5)$$

Of special interest is the *alternation depth* of a fixpoint expression (Emerson and Lei, 1986; Niwiński, 1986). Roughly speaking, this is the nesting depth of alternating μ and ν -operators whose computation depends on each other. Expressions with a single operator have alternation depth 1 and are also called *alternation-free*. The alternation depth of an equation system is 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 (Schneider, 2003) will be useful to assess the computational complexity of our solution¹:

Theorem 1. (Complexity of μ -Calculus Model Checking).

For every equation system E of alternation depth l , whose *length* $|E|$ is given by the sum of the lengths of the right sides of the equations in the system, and every Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$, there is an algorithm to compute its solution in time

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

Corollary 1. A system of equations of constant length and alternation depth l written for a Kripke structure associated to an automaton can be solved in time

$$O(|Q|^l |\Sigma|).$$

Proof. By construction, the Kripke structure from Definition 4 has $|\mathcal{S}| = 2|Q|$ and $|\mathcal{R}| \leq 2|Q||\Sigma|$. For constant $|E|$, the result follows immediately.

3.3 Solving SCP

We now apply the tools presented in Subsection 3.2 to the classical solutions of SCP (Wonham and Ramadge, 1987; Kumar and Garg, 1995). Our approach is closer to the one in (Kumar and Garg, 1995), because it collects bad and non-coaccessible states instead of eliminating them at each iteration, and collecting states is easier to represent with our Kripke

structure. We use Corollary 1 to confirm the quadratic complexity of the algorithm.

The solution for SCP is computed from the product $\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}$ by starting with the initial bad states (those that violate condition 1), and recursively adding the states that can reach some bad state through an uncontrollable transition. When no more bad states can be found, the remaining states are tested for coaccessibility and those found to be non-coaccessible are added to the set of bad states. Because removing bad states can destroy coaccessibility and removing non-coaccessible states can expose new bad states, the algorithm restarts and alternates between these two computations until a fixpoint is reached.

We form the Kripke structure associated to $\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}$ and collect the bad states into the set² x_B . The initial value for this set is x_b . We add a state to x_B when it has an uncontrollable transition leading into a state already found to be bad, or non-coaccessible, or both. This requires collecting also the non-coaccessible states, which is done in the set x_N .

The computations use the two substructures of the Kripke structure. Because we are interested only in predecessors of bad 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 have to switch from one substructure to the other, using the set $\kappa(x_N)$. The above description translates naturally into expression 9 below.

To derive an expression for x_N , we first set $\Phi_{\tau} = \neg \kappa(x_B)$ in equation 5 to keep only coaccessible states that are not bad:

$$x_{Co} \stackrel{\mu}{=} \neg \kappa(x_B) \wedge (\text{EX } x_{Co} \vee x_m). \quad (6)$$

Next, we complement expression 6 to obtain the non-coaccessible states. Note this makes the expression a *greatest* fixpoint:

$$\neg x_{Co} \stackrel{\nu}{=} \kappa(x_B) \vee \text{AX } \neg x_{Co} \wedge \neg x_m. \quad (7)$$

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 . The expression for x_N is therefore obtained by restricting equation 7 to $\neg x_u$. This completes our equation system:

$$\begin{cases} x_N \stackrel{\nu}{=} \kappa(x_B) \vee \neg x_u \wedge \text{AX } x_N \wedge \neg x_m & (8) \\ x_B \stackrel{\mu}{=} \text{EX } (x_B \vee \kappa(x_N)) \vee x_b. & (9) \end{cases}$$

¹ This holds when the function κ_{π} can be computed in time $O(\mathcal{R})$, which we assume.

² To improve readability, we will write just ‘the set x_B ’ to signify ‘the set $\llbracket x_B \rrbracket_{\mathcal{A}}$ represented by x_B ’, as long as the meaning is clear from the context.

The accessible component of the transition relation that results when the bad states and the non-coaccessible states are eliminated can be found by setting $\Phi_\tau = \neg(\kappa(x_B) \vee x_N)$ in equation 4, again restricting the result to $\neg x_u$:

$$x_{Ac} \stackrel{\mu}{=} \neg(\kappa(x_B) \vee x_N) \wedge (\text{EY } x_{Ac} \vee x_{q^0} \wedge \neg x_u). \quad (10)$$

The solution for SCP is then given by restricting the automaton $\mathcal{A}_P \times \mathcal{A}_E$ to the states $\llbracket x_{Ac} \rrbracket_{\mathcal{A}_P \times \mathcal{A}_E}$. The complexity of the overall computation is given by Corollary 1. Since the solution has $l = 2$, we get $O(|Q|^2 |\Sigma|) = O((|Q_P| |Q_E|)^2 |\Sigma|)$, which is the known complexity of the original algorithm (Ramadge and Wonham, 1989; Cassandras and Lafortune, 1999). An example of the computation of the fixpoints is given in (Ziller and Schneider, 2003).

4. REDUCING COMPLEXITY

In this section we identify a special class of supervisory control problems that covers many practical situations. These are characterised by the absence of a particular connected component in the product $\mathcal{A}_P \times \mathcal{A}_E$, which we call *hidden livelock component*, or HLC for short. This class contains all problems that use prefix-closed languages (which can already be solved in linear time), but also many problems which the algorithms from (Wonham and Ramadge, 1987; Kumar and Garg, 1995) can solve only in quadratic time. Our main result is a set of equations that solves HLC-free problems in linear time. The equations also apply to all other problems, in which case complexity automatically changes to quadratic. Efficiency is nonetheless improved, because the number of iterations is kept smaller.

In the solution presented in Section 3, equations 8 and 9 form a system of alternation depth 2. According to Corollary 1, complexity decreases with the alternation depth. We therefore suggest a new approach to collect non-coaccessible states that leads to a least fixpoint instead of a greatest. We start from the set of states that are initially not coaccessible in $\mathcal{A}_P \times \mathcal{A}_E$, which we denote x_n . To compute it, we set $\Phi_\tau = 1$ in equation 5, complement it, and restrict the result to the substructure identified by $\neg x_u$:

$$x_n \stackrel{\nu}{=} \neg x_u \wedge \text{AX } x_n \wedge \neg x_m \quad (11)$$

To compute non-coaccessible states we initialise x_N with x_n and add a new state to x_N if it is not a marker state and all its outgoing transitions lead to states that were already found to be either bad or not coaccessible. This amounts to exchanging equation 8 by equation 12 in our equation system as follows:

$$\begin{cases} x_N \stackrel{\mu}{=} \text{AX } (\kappa(x_B) \vee x_N) \vee x_n & (12) \\ x_B \stackrel{\mu}{=} \text{EX } (x_B \vee \kappa(x_N)) \vee x_b & (13) \end{cases}$$

If equation 12 can capture all non-coaccessible states of \mathcal{S} , then the solution can be computed from equation 10 as before. Since the equation system above has alternation depth 1, the problem can be solved in time $O(|Q| |\Sigma|)$. However, the expression for x_N will fail to capture some non-coaccessible states from \mathcal{S} if they form a component as described in the following:

Definition 8. (Hidden Livelock Component (HLC)). A hidden livelock component is a subset of states of $\mathcal{A}_P \times \mathcal{A}_E$ satisfying the following conditions simultaneously:

- they are initially coaccessible, but become not coaccessible after some bad states collected in x_B are removed;
- none of them becomes a bad state;
- they form a connected component.

As an example, consider a specification like the one in Figure 4 (where event labels are not shown because they are irrelevant for the illustration). Suppose that all transitions are controllable and that state 5 is a bad state. Although states 2, 3, and 4 will become not coaccessible after 5 is removed, they will remain undetected because each of them has a transition leading into a state that can not be gathered while solving equations 12 and 13.

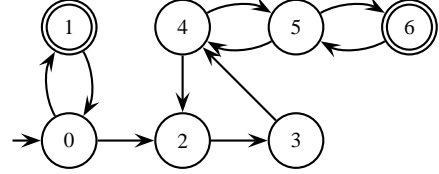


Figure 4. States 2, 3, and 4 form a HLC

Therefore, the states in $(\kappa(x_B) \vee x_N)$ may not be the only states that have to be removed. The only way to test this is to remove them and then check the result for coaccessibility, much like the algorithm from Section 3. By equation 5, the remaining coaccessible states are:

$$x_{Co} \stackrel{\mu}{=} \neg(\kappa(x_B) \vee x_N) \wedge (\text{EX } x_{Co} \vee x_m). \quad (14)$$

The complement of this expression, restricted to $\neg x_u$, includes the hidden livelock components exposed by the removal of the bad and the non-coaccessible states found so far. We denote this set by x_H . It is given by equation 15, which completes our equation system for the general solution of SCP. Equation 17 is the same as equation 13, while equation 12 has been augmented by the new set of non-coaccessible states x_H to form equation 16. Note that the first computation of x_H is identical to x_n , so the latter could be dropped. The solution for SCP is again given by restricting $\mathcal{A}_P \times \mathcal{A}_E$ to the states given by equation 10.

$$\begin{cases} x_H \stackrel{\nu}{=} (\kappa(x_B) \vee x_N) \vee \neg x_u \wedge \text{AX } x_H \wedge \neg x_m & (15) \\ x_N \stackrel{\mu}{=} \text{AX } (\kappa(x_B) \vee x_N) \vee x_H & (16) \\ x_B \stackrel{\mu}{=} \text{EX } (x_B \vee \kappa(x_N)) \vee x_b & (17) \end{cases}$$

The above system of equations has alternation depth 2. According to Corollary 1, the worst case complexity is thus $O((|Q_{\mathcal{P}}||Q_{\mathcal{E}}|)^2|\Sigma|)$. To see that this complexity appears only if $\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}$ is not HLC-free, consider that the solution starts by initialising $x_H = 1$ and $x_N = x_B = 0$. x_H is then computed once, which can be done in time $O(|Q_{\mathcal{P}}||Q_{\mathcal{E}}||\Sigma|)$. Next, a fixpoint for the subsystem formed by equations 16 and 17 is found. The key point is that x_N and x_B can be computed without reinitialising their values to 0, because they are both least fixpoints. In contrast, equations 8 and 9 required initialising $x_N = 1$ and $x_B = 0$ at each iteration. When x_H is evaluated a second time and there are no HLC's, the iteration stops, including all non-coaccessible states. In the presence of HLC's, the iteration continues automatically, only then yielding a quadratic complexity. This leads to the conclusion that, even in the presence of HLC's, the new solution will be more efficient than the existing ones, because only the non-coaccessible states that form HLC's will be collected in quadratic time. The special case of prefix-closed languages is also done in linear time, since if all states are marker states, there can be no HLC's. Equivalently, the full marking leads to $x_H = x_N = 0$, so it suffices to evaluate x_B .

5. IMPLEMENTATION

This section presents experimental results that illustrate how our new solution improves supervisor synthesis. Traditionally, an automaton is represented in computer memory by transition tables, and writing such programs for the classical and the new solutions would show the expected reduction from quadratic to linear time in a number of examples. However, even if the computational effort becomes linear in the number of states of $\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}$, the transition tables of most practical systems would be far too large to fit into the memory of any computer, especially due to the state explosion when computing the product $\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}$.

Therefore, we went one step further and chose a symbolic representation through binary decision diagrams (BDD's) (Bryant, 1986). This requires a new method to find the initial bad states (Ziller, 2002), and also affects computational complexity, since some operations can no longer be done in linear time. This means that we did not stick to linearity, but rather used the new solution to create a program that can handle large state sets and is faster than a similar program based on the classical solution from Section 3. For comparison, the following example has been solved with BDD-based implementations of both solutions.

5.1 The Nim Game

The example is about the synthesis of a winning strategy for the nim game. Given the matches arranged as in Figure 5, each player removes at least one match from one of the rows in each turn. The goal is to

force the opponent to take the last match. Events reflecting the moves of the wanted winner are controllable, while those of the opponent are not. The plant encompasses all possible states of the game, while the specification is derived from it by eliminating the state in which the wanted winner would lose. This results in an uncontrollable specification, whose supremal controllable sublanguage is either empty (when there is no winning strategy for the chosen player) or the wanted winning strategy. There is always a winning strategy for one of the players. The reader is referred to (Ziller, 2001; Ziller, 2002) for more details about modelling and solving the example.

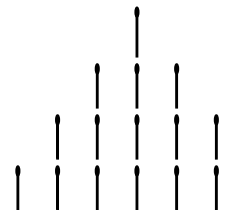


Figure 5. The nim game with 4 rows

Because removed matches do not return to the game, the product of plant and specification is loop-free and hence HLC-free. This means that equations 15 to 17 can be solved in one pass, while the classical solution will require several passes through equations 8 and 9 (see Table 1). Factoring out the effects of the BDD package, this means that this example can be solved in linear time with our new solution, while the existing ones require quadratic time.

We solved the example from 4 to 12 rows. These range from 5920 to 2.82527×10^{14} transitions, thereby reaching the size of real-world problems. The new solution has a clear advantage with respect to time and memory usage; the last case could not even be solved by the classical algorithm within available memory. Time is CPU user time, in seconds, needed to solve equations 8 and 9 (classical) and equations 15 to 17 (new solution). Memory usage is given in the peak number of BDD nodes (1 node = 16 bytes). The programs were written in C++ using CUDD³ 2.3.1, compiled with GNU g++ 3.2 and run under SUSE Linux 8.1 on a PC with 1 GB RAM and an Athlon XP 2200+ processor.

Table 1. Experimental results

Rows	Classical solution			New solution	
	Time	Node peak	Passes	Time	Node peak
4	0.01	18396	7	0.01	17374
5	0.07	61320	11	0.04	48034
6	0.45	231994	17	0.10	114464
7	3.95	1234576	24	0.40	266742
8	61.8	2498790	31	1.46	591738
9	1618	2587704	39	5.35	1211070
10	53106	6441666	49	49.2	2484482
11	1.86×10^6	32898180	58	1943	2474262
12				180262	3594374

³ A BDD package by Fabio Somenzi, University of Colorado at Boulder

6. CONCLUSION

The paper presents a new approach to the RW synthesis problem using μ -calculus expressions and refers to results from formal verification to assess its computational complexity. Before this work, only the special class of problems where all states are marked could be solved in linear time. These problems are seen to be strictly contained in the class of HLC-free problems. Our new approach can solve all HLC-free problems in linear time, thereby including many practical examples that the existing algorithms can solve only with quadratic effort. Moreover, an implementation of the new solution based on binary decision diagrams can handle industrial-size problems faster and with far less memory than the corresponding implementation of existing algorithms. Finally, the representation through μ -calculus formulas allows the problem to be solved by verification tools not necessarily conceived with supervisor synthesis in mind.

REFERENCES

- Aziz, A., F. Balarin, R.K. Brayton, M.D. Dibeneditto, A. Sladanha and A.L. Sangiovanni-Vincentelli (1995). Supervisory control of finite state machines. In: *Conference on Computer Aided Verification (CAV)* (P.L. Wolper, Ed.). Vol. 939 of *LNCS*. Springer. Liege, Belgium. pp. 279–292.
- Bryant, R.E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers* **C-35**(8), 677–691.
- Cassandras, C. G. and S. Lafortune (1999). *Introduction to Discrete Event Systems*. Kluwer Academic Publishers. Boston, U.S.A. ISBN 0-7923-8609-4.
- Clarke, Jr, E. M., O. Grumberg and D. A. Peled (1999). *Model Checking*. The MIT Press. London, U.K. ISBN 0-262-03270-8.
- Cleaveland, R., M. Klein and B. Steffen (1992). Faster Model Checking for the Modal μ -Calculus. In: *Computer Aided Verification (CAV'92)* (G.v. Bochmann and D.K. Probst, Eds.). Vol. 663 of *LNCS*. Springer-Verlag. Heidelberg, Germany. pp. 410–422.
- Emerson, E.A. and C.-L. Lei (1986). Efficient model checking in fragments of the propositional μ -calculus. In: *IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press. Washington, D.C.. pp. 267–278.
- Kumar, R. and V. Garg (1995). *Modeling and Control of Logical Discrete Event Systems*. Kluwer Academic Publishers. ISBN 0-7923-9538-7.
- Kumar, R., S. Nelvagal and S. I. Marcus (1997). A discrete event systems approach for protocol conversion. *Discrete Event Dynamical Systems* **7**(3), 295–315.
- Lassez, J.-L., V.L. Nguyen and E.A. Sonenberg (1982). Fixed point theorems and semantics: a folk tale. *Information Processing Letters* **14**(3), 112–116.
- Niwiński, D. (1986). On fixed point clones. In: *International Colloquium on Automata, Languages and Programming (ICALP)*. L. Kott, Ed., vol 226 of *LNCS*, Springer-Verlag. pp. 464–473.
- Overkamp, A. (1997). Supervisory control using failure semantics and partial specifications. *IEEE Transactions on Automatic Control* **42**(4), 498–510.
- Ramadge, P. J. and W. M. Wonham (1987). Supervisory control of a class of discrete event processes. *SIAM J. of Control and Optimization* **25**(1), 206–230.
- Ramadge, P. J. and W. M. Wonham (1989). The control of discrete event systems. *Proceedings of the IEEE* **77**(1), 81–98.
- Schneider, K. (2003). *Verification of Reactive Systems – Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer. ISBN 3-540-00296-0.
- Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* **5**(2), 285–309.
- Thistle, J. G. and W. M. Wonham (1994). Control of infinite behavior of finite automata. *SIAM J. of Control and Optimization* **32**(4), 1075–1097.
- Wonham, W. M. (2002). Notes on control of discrete-event systems. Technical report. Dept. of Electrical and Computer Engineering, University of Toronto. <http://www.control.utoronto.ca/DES>.
- Wonham, W. M. and P. Ramadge (1987). On the supremal controllable sublanguage of a given language. *SIAM J. Control and Optimization* **25**(3), 637–659.
- Yevtushenko, N., T. Villa, R. K. Brayton and A. Petrenko and A. L. Sangiovanni-Vincentelli (2001). Solution of parallel language equations for logic synthesis. In: *Proceedings of the International Conference on Computer-Aided Design*. pp. 103–110.
- Ziller, R. and K. Schneider (2003). A μ -Calculus Approach to Supervisor Synthesis. In: *GI/ITG/GMM-Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*. pp. 132–143.
- Ziller, R. M. (2001). System Modelling Using Marker States in the RW-Framework. In: *GI/ITG/GMM-Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*. MoPress. pp. 121–130.
- Ziller, R. M. (2002). Finding Bad States during Symbolic Supervisor Synthesis. In: *GI/ITG/GMM-Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*. pp. 209–218.
- Ziller, R. M. and José E. R. Cury (1994). On the Supremal L-controllable Sublanguage of a non Prefix-Closed Language. *Anais do 10. Congresso Brasileiro de Automática e 6. Congresso Latino-Americano de Controle Automático* **2**, 260–265.