An Application of Generalized Supervisor Synthesis to the Control of a Call Center

Roberto Ziller
University of Karlsruhe
Institute of Computer Science and Engineering
Prof. Dr. D. Schmid
P.O. Box 6980, 76128 Karlsruhe, Germany
E-Mail: ziller@informatik.uni-karlsruhe.de

Abstract

The Ramadge-Wonham model for the control of discrete systems is a powerful formal design tool for a large class of systems. Its main feature is the ability to synthesize a supervisor for a system, so that the behavior of the latter is confined according to a given specification under minimal restrictions. Until now, the allowed specifications were limited to safety and liveness properties. However, it is not uncommon to also find persistence and fairness properties in the description of reactive systems. The present paper shows an application of a generalized synthesis approach that is able to deal with all these properties in a uniform way. Theoretical aspects are kept short in order to make room for the thorough description of an example, originally solved by means of conventional supervisor synthesis. The original solution does not ensure fairness and can thus be improved with the new approach.

1 Introduction

The results herein come from a larger work by the same author [Zil05], in which an application example of supervisor synthesis by Seidl et al. [SKWP02] was improved by means of a new approach to the synthesis problem. The main theoretical aspects of this approach were also described in [ZS03b, ZS03a, ZS05]. These three publications show how supervisor synthesis [CL99, RW87, RW89, Won04, WR87] and μ-calculus model checking [AC88, CE81, CGP99, EC80, Koz82, Koz83, Pra81, Sch03] can be combined in order to give origin to a new and more powerful synthesis tool. Due to lack of space, however, they do not contain a larger application example. This paper aims to close this gap – at the cost of a brief exposition of the theory. Readers not familiar with supervisor synthesis and model checking are invited to look at the above references.

The following classification of system properties [MP88, MP90, Sch03] is important to evaluate the capabilities of a formal synthesis approach:

- *Safety properties* describe restrictions to be obeyed whenever the system is running. These properties are expressed in terms of forbidden states, like for example the state in which the traffic lights at a road crossing are green at the same time.
- *Liveness properties* require that some given state is reachable, no matter how often. This can be used to identify the accomplishment of a task by the system.
- *Persistence properties* are related to stabilization, e.g. requiring that a buffer in a manufacturing cell does not become empty once a certain production stage has been reached.
- *Fairness properties* require that some states remain reachable, independently of how often they are visited. An example is the requirement that the users of some system should always be able to log in.

Modern verification methods allow designers to check a given specification for properties like the above. In order to do this, the designer specifies a model of the system and writes down an expression for each property to be verified. A model checking algorithm then either confirms a desired property 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 specification of the model for the system, a task whose degree of difficulty should not be underestimated. Ideally, the specification itself should be generated by a tool that takes high-level requirements and either outputs a correct specification or rejects them if they cannot be enforced.

A solution that takes the latter approach is given by the Ramadge-Wonham model for supervisor synthesis. The system is modeled by a finite automaton $\mathcal{A}_\mathcal{P}$ called *plant*, whose transition relation is only partially defined. It reflects exactly the event sequences that might occur in the system, thus describing both wanted and unwanted features. A second automaton $\mathcal{A}_\mathcal{E}$, called *specification*, is used to restrict the the system's behavior to something useful. This is done by means of the automata product, so that $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ represents the desired behavior for the system. Ideally, the control action is achieved by running the plant in parallel with the automaton $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$. The latter watches the events occurring in the plant and tracks its state accordingly. After each transition, the plant is told the set of events for which a transition is defined in the state it just entered by $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$. The plant is supposed to choose its next event from this set. The automaton used to restrict the plant's action – in this case, $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ – is called *supervisor*.

The model takes into account that not every event in the system can be prevented from occurring. In general, commands, e.g. to turn something on or off or to set a timer, can be inhibited by the supervisor in order to avoid unwanted states. However, messages originating from the system, like sensor or alarm signals and timeouts, cannot be prevented from occurring. Accordingly, the model distinguishes between *controllable* and *uncontrollable* events. It is therefore not always possible to use $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ as a supervisor. This happens when some state (p, q) from $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ does not define a transition for an uncontrollable event that might occur in state p of $\mathcal{A}_\mathcal{P}$. Such states are called *bad states*. If an automaton with bad states were used as a supervisor, an uncontrollable event could happen unexpectedly in a bad state, and it would lose control of the system.

Due to the large size of the state spaces, it is not uncommon that $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ ends up with some bad states, even if the specification $\mathcal{A}_\mathcal{E}$ is apparently sound. In this case, a *supervisor synthesis* procedure takes the automata $\mathcal{A}_\mathcal{P}$ and $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ and computes a third automaton $\mathcal{A}_\mathcal{S}$ from them by eliminating the bad states. The procedure also prevents livelocks and deadlocks by eliminating states that do not have a path to an accepting state. The result corresponds to the largest non-blocking solution that does not violate the desired behavior. The idea is that $\mathcal{A}_\mathcal{S}$ can be used as a supervisor, provided that the restrictions imposed by the synthesis procedure are still tolerable.

In spite of these advantages, the synthesis approach also has some limitations. Among them is the fact that specifications are restricted to safety and liveness properties, while persistency and fairness are often needed in reactive systems. The generalization of supervisor synthesis introduced in [ZS03a, ZS05, Zil05] treats all these properties uniformly.

In the sequel, Section 2 presents the generalization of the synthesis approach. The main section is Section 3, which is based on an application of supervisor synthesis to the control of a call center due to Seidl et al. [SKWP02]. A thorough analysis of their solution shows that it cannot always ensure that users will be granted access to the system as expected. This problem can be eliminated by adding a fairness property to the specification for the system's desired behavior. This makes the problem an instance of the generalized supervisor synthesis problem, which can be solved with the new synthesis approach[1].

[1]This fact does not exclude the possibility of solving the problem by making modifications to the system model, still using conventional supervisor synthesis. The solution presented here is an alternative that illustrates the power of the new synthesis approach.

2 Generalized Supervisor Synthesis

The generalization of the synthesis procedure is achieved by translating the synthesis algorithm into the μ-calculus, as illustrated in Figure 1. In a first step, a Kripke structure is derived from the automaton $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$. The synthesis algorithm is replaced by a μ-calculus equation system, whose solution is the set of states that corresponds to the supervisor on the Kripke structure. Finally, a projection of this state set onto $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ yields the state set of the supervisor.

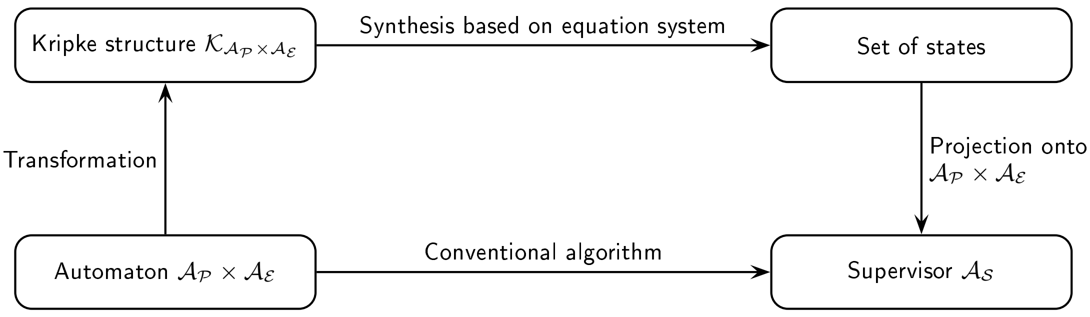

Kripke structure $\mathcal{K}_{\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}}$
Synthesis based on equation system
Set of states
Transformation
Projection onto $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$
Automaton $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$
Conventional algorithm
Supervisor $\mathcal{A}_\mathcal{S}$

Figure 1: Translation of Supervisor Synthesis into Model Checking

The following definitions specify the translation of the automaton $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ into a Kripke structure.

Definition 1 (Labeling of $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$) *Given an automaton $\mathcal{A} := \mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ with state set $Q_\mathcal{A}$, let $\mathcal{V}^x := \{x_{k-1}, \ldots, x_0\}$ and $\mathcal{V}^y := \{y_{k-1}, \ldots, y_0\}$ be two sets of Boolean variables such that $k \geq \lceil log_2(|Q_\mathcal{A}|) \rceil$. Further, let $\lambda^x : Q_\mathcal{A} \to 2^{\mathcal{V}^x}$ and $\lambda^y : Q_\mathcal{A} \to 2^{\mathcal{V}^y}$ be unique labelings of the states of $Q_\mathcal{A}$ with the variables of $\mathcal{V}^x$ and $\mathcal{V}^y$, respectively. The function $\mathcal{L}_\mathcal{A} : Q_\mathcal{A} \to 2^{\mathcal{V}_\mathcal{A}}$ then labels each state $q \in Q_\mathcal{A}$ with the variables from $\mathcal{V}_\mathcal{A} := \mathcal{V}^x \cup \mathcal{V}^y \cup \{x_m, x_b\}$, according to the following:*

$$\mathcal{L}_\mathcal{A}(q) := \lambda^x(q) \cup \lambda^y(q) \cup \begin{cases} \{x_m\} & \textit{if } q \in M \\ \{\bar{x}_m\} & \textit{else} \end{cases} \cup \begin{cases} \{x_b\} & \textit{if } q \textit{ is a bad state} \\ \{\bar{x}_b\} & \textit{else.} \end{cases}$$

Definition 2 (Kripke structure of an automaton) *Let $\mathcal{A} = \langle \Sigma, Q, \delta, q^0, M \rangle$ be an automaton whose states have been labeled according to Definition 1, and let $\Sigma_u \subseteq \Sigma$ be the set of uncontrollable events. Then $\mathcal{K}_\mathcal{A} := \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ is the Kripke structure associated to $\mathcal{A}$, with:*

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

The translation of the conventional synthesis algorithm into the μ-calculus leads to the equation system [Zil05]

$$\left\{ \begin{array}{lcl} u_c & \overset{\mu}{=} & (\Diamond u_c \vee x_m \bar{x}_u) \wedge u_{cg} \\ u_g & \overset{\nu}{=} & \Box u_g \wedge \bar{x}_b x_u \wedge \kappa(u_c) \\ u_{cg} & \overset{\nu}{=} & \kappa(u_g) \end{array} \right. \tag{1}$$

Here, the state set represented by u_{cg} corresponds to the states of the supervisor, seen on the Kripke structure. The projection of this state set back onto the automaton $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ is as follows:

Definition 3 (Projection of a Kripke structure onto an automaton) *Given an automaton $\mathcal{A}$ with state set $Q_{\mathcal{A}}$, as well as its associated Kripke structure $\mathcal{K}_{\mathcal{A}} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ and a set $P \in \mathcal{S}$, the projection of P onto $Q_{\mathcal{A}}$ is*

$$\{q \in Q_{\mathcal{A}} \mid (q, 0) \in P \vee (q, 1) \in P\}.$$

It is important to note that, as usual with model checking, the automata $\mathcal{A}_{\mathcal{P}}$ and $\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}$, as well as the Kripke structure $\mathcal{K}_{\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}}$, can be constructed using symbolic representation [BCM90a, BCM$^+$90b]. The same applies to the synthesis process. This allows to deal with state spaces much larger than those treated by conventional synthesis algorithms. In spite of this advantage, the specification properties are still limited to safety and liveness, because the requirements for controllability and blocking freedom are fixed in the Ramadge-Wonham approach. While controllability is always needed, blocking freedom can be enhanced by other properties specified by the designer.

The main result in [Zil05, ZS05] states that the synthesis process can be enhanced to deal with all specification properties in the same way. Because this includes blocking freedom, the enhanced approach is a generalization of the original one. This is achieved by allowing the designer to specify any desired property expressible in the μ-calculus. Since it may be difficult to do this directly, temporal logics like CTL, CTL*, or LTL can be used as an intermediate step. Any expression in these logics can be translated into an equation system of the form [Dam94, Sch03]:

$$\begin{cases} u_n & \stackrel{\sigma_n}{=} & \Phi_n \\ & \vdots & \\ u_1 & \stackrel{\sigma_1}{=} & \Phi_1. \end{cases}$$

This translation of the user-defined requirements can then be integrated in Equation System 2, whose solution is then projected onto $\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}$ according to Definition 3. This yields a supervisor that ensures controllability as well as the requirements specified by the designer.

$$\begin{cases} u_n & \stackrel{\sigma_n}{=} & \Phi_n \wedge \bar{x}_u \wedge z \\ & \vdots & \\ u_1 & \stackrel{\sigma_1}{=} & \Phi_1 \wedge \bar{x}_u \wedge z \\ u_g & \stackrel{\nu}{=} & \Box u_g \wedge \bar{x}_b x_u \wedge \kappa(u_1) \\ z & \stackrel{\nu}{=} & \kappa(u_g) \end{cases} \tag{2}$$

3 Application Example

The example presented below is based on work from Seidl et al. [SKWP02], who used conventional supervisor synthesis to derive a controller for the coordination of different components of a call center. Their publication contains the automata used to compute one of the supervisors employed. That way, it is possible to reproduce their results with the μ-calculus based supervisor synthesis outlined in Section 1. By construction, the system under supervision is non-blocking in the sense that it can always reach a marker (accepting) state. However, it is possible to show that this condition is not sufficient to guarantee that an operator of the call center will be able to log in to the system under all circumstances. This problem can be eliminated by adding a fairness property to the specification of the system's desired behavior. The new synthesis problem can then be solved with the generalized synthesis approach described above.

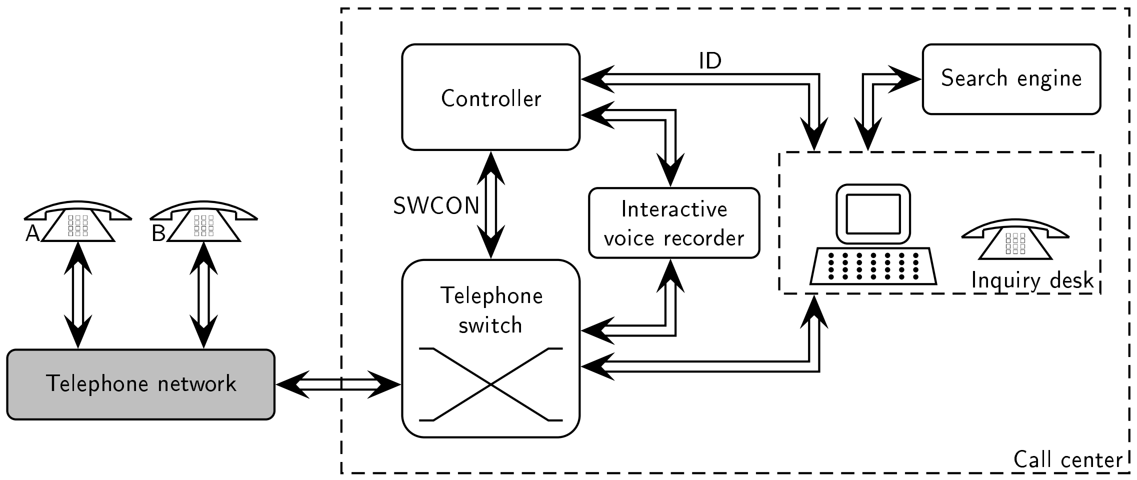


Figure 2: Representation of the call center

3.1 System Description

A call center consists of a number of *inquiry desks*, each of which is handled by an operator who receives calls from the public telephone network. These calls enter the call center through a telephone switch, as shown in Figure 2.

The tasks that have to be accomplished in the call center are not limited to connecting the two participants of a conversation. It is also necessary to keep track of the state of each operator, so that the switch can always know to which inquiry desks it can send new incoming calls. To this end, each operator has to log in to the system at the beginning of his shift and to log out before leaving. This is accomplished through login and logout requests sent from the operator to the switch, which has to acknowledge these requests. Moreover, the operator is allowed to switch between the states *ready* and *not ready* for short breaks. These state changes are also initiated by requests sent from the operator to the switch and acknowledged by the latter. Moreover, the automata used to model the system also lead to the conclusion that an operator in the *not ready* state can be made *ready* by the switch without a previous request. This could be used for example to automatically activate an operator right after login. The switch may only send incoming calls to an operator that is logged in and ready. In order to ensure a correct functionality of the system, the communication between operator and switch goes through a controller, which is obtained through supervisor synthesis.

3.2 System Modeling

The system as viewed by the controller consists of the two interfaces named ID (*inquiry desk*) and SWCON (*switch control*). The state space of the problem remains small with respect to the size of the whole system, because it is only necessary to consider one operator for each call. Moreover, it turns out that the event set concerning the administrative tasks described above has no elements in common with the event set related to the switching tasks. The controller for each operator can therefore be split into two independent supervisors, which run simultaneously and independently from each other. In [SKWP02] the authors give the automata used to synthesize the supervisor for the administrative tasks, so that it is possible to reproduce these results. The automata for the coordination of the switching activities are not given completely in their paper and will receive no further attention here.

The interface ID between the controller and the inquiry desk is modeled by the automaton on the left side of Figure 3. Commands from the controller to the inquiry desk (shown in italics in the picture) are controllable, while requests sent by the operator are not. The interpretation of the events is as follows:

- idLoginReq : request from an operator to log in to the system.
- idLogin : command from the controller to log in the operator and set it *not ready*.
- idReadyReq : request from an operator to get ready.
- idReady : command from the controller to change the operator's state to *ready*.
- idNotReadyReq : request from an operator to get not ready.
- idNotReady : command from the controller to change the operator's state to *not ready*.
- idLogoutReq : request from an operator to log out of the system.
- idLogout : command from the controller to change the operator's state to *logged out*.
- idEOS (End of Service): request issued by an operator for long-term deactivation.

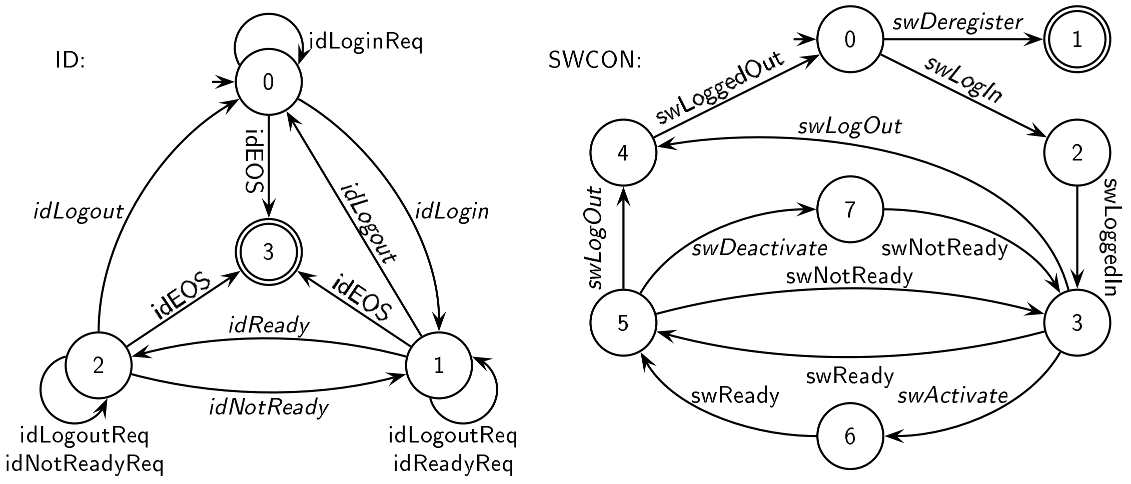


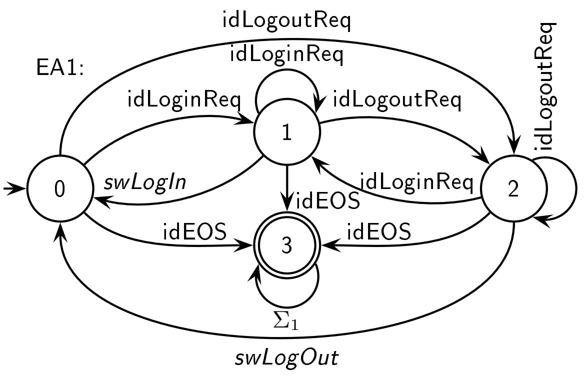
Figure 3: The interfaces ID and SWCON used to model the system

The interface SWCON is modelled by the automaton given on the right side of Figure 3. Commands from the controller to the switch are controllable, while the reactions of the latter are not. The interpretation of the events is as follows:

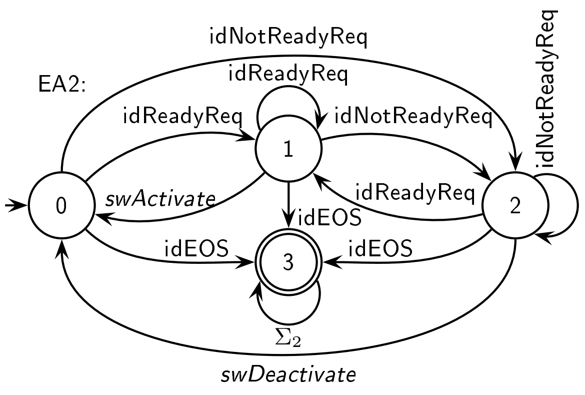
- swLogIn : command from the controller to change an operator's state to *logged in*.
- swLoggedIn : acknowledgment of the login command.
- swActivate : command from the controller to change an operator's state to *ready*.
- swReady : acknowledgment of activation command.
- swDeactivate: command from the controller to change an operator's state to *not ready*.
- swNotReady : acknowledgment of deactivation command.
- swLogOut : command from the controller to change an operator's state to *logged out*.
- swLoggedOut : acknowledgment of logout command.
- swDeregister: command from the controller to completely deactivate an operator.

The plant automaton $\mathcal{A}_{\mathcal{P}}$ is the result of the parallel composition of the automata for the interfaces ID and SWCON. The resulting automaton has 32 states and 152 transitions. Since the plant has to model all physically possible event sequences, this automaton allows not only the correct ones, but also many others that are of no use. For example, there could be an idLogout as a reaction to an idReadyReq, or an idLoginReq immediately followed by an idLogin, without the controller informing the switch of the login with the events swLogin and swLoggedIn in between.

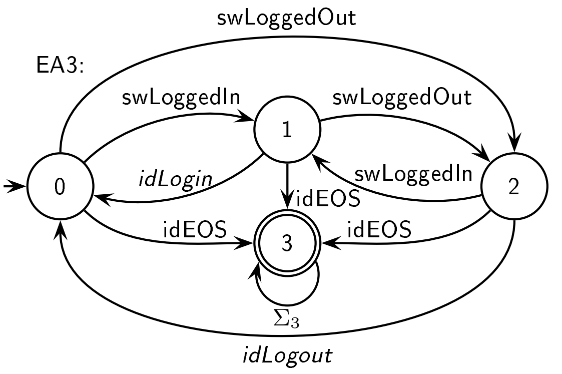
In [SKWP02], five other automata specify the system's desired behavior. The first four, EA1 to EA4, are shown in Figure 4. EA1 and EA2 restrict the communication possibilities from the operator towards the switch. Similarly, EA3 and EA4 impose restrictions on the communication in the opposite direction. EA1 and EA3 are concerned with the event sequences for logging in and out, while EA2 and EA4 are related to activation and deactivation. Apart from the transitions shown on the figure, each automaton has also selfloops in all states, labeled with the events that do not appear explicitly on the other transitions. This is necessary in order not to block these events in the product of the automata.



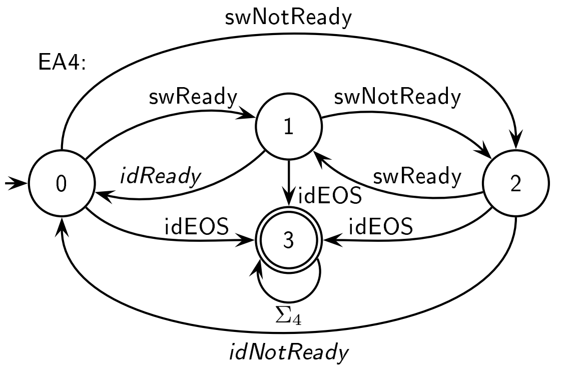
$\Sigma_1 = \{$idLoginReq, idLogoutReq, swLogIn, swLogOut$\}$



$\Sigma_2 = \{$idReadyReq, idNotReadyReq, swActivate, swDeactivate$\}$



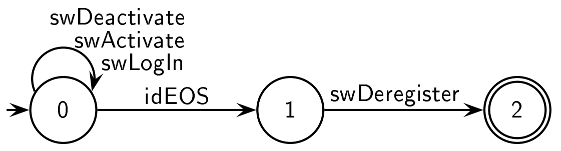
$\Sigma_3 = \{$swLoggedIn, swLoggedOut, idLogin, idLogout$\}$



$\Sigma_4 = \{$swReady, swNotReady, idReady, idNotReady$\}$

Figure 4: Specification automata EA1 to EA4: enforcing communication rules

The fifth automaton, EA5, is the one reproduced in Figure 5. It states that the events swLogIn, swActivate, and swDeactivate are no longer allowed after an idEOS request. Consequently, the only possible action following that event is a complete deactivation of the operator with swDeregister. Resetting the inquiry desk to the initial state is not part of the modeling and assumes some procedure not described in the paper. As before, each state has selfloops labeled with all transitions that do not appear explicitly.



Selfloops: $\Sigma \setminus \{$swLogIn, swActivate, swDeactivate, swDeregister, idEOS$\}$

Figure 5: Specification EA5: allowed behavior after event idEOS

The specification automaton $\mathcal{A}_\mathcal{E}$ consists of the product of the automata EA1 to EA5. Next, the product $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ is formed. The result is an automaton with 286 states and 1160 transitions. It turns out to have 23 bad states. $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ is therefore not useful as a supervisor, and supervisor synthesis is in order to compute its largest implementable subset.

3.3 Reasons for the lack of controllability

The following analysis will be useful to explain the improvements made later on. An event sequence that shows that $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ cannot be used as a supervisor can be found by picking out a path that leads from its initial state to one of the bad states. Figure 6 shows one such state, namely (10,196). In this state, the automata ID and SWCON on Figure 3 are in states 0 and 3, while the automata EA1 to EA4 on Figure 4 are in states 0, 0, 1, and 1, respectively. Automaton EA5 remains in state 0 and is not important here. Note that the uncontrollable event swReady can occur in state 10 of the plant $\mathcal{A}_\mathcal{P}$, but is not present in state (10,196) of the product $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$. This is also reflected in automaton EA4, which is in state 1 and cannot execute that event. This is a violation of the controllability condition.

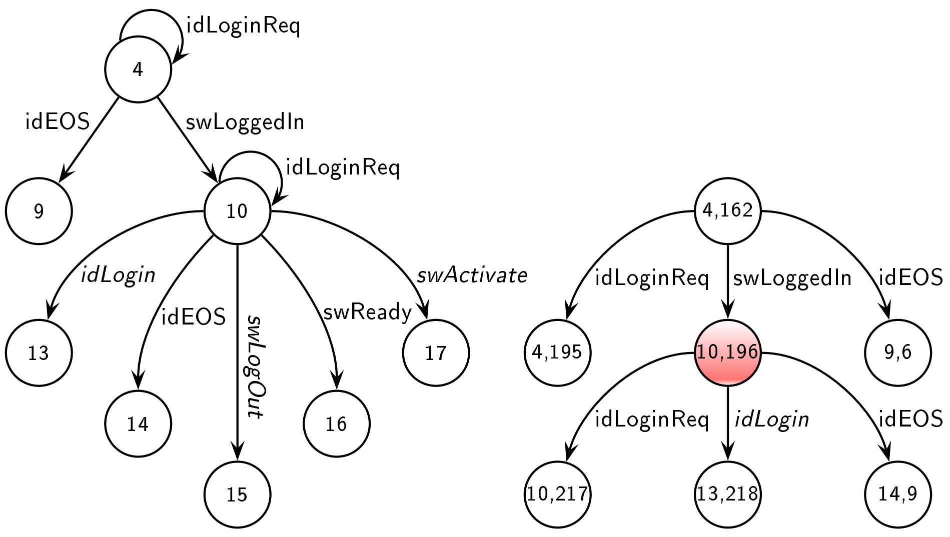


Figure 6: Parts of the automata $\mathcal{A}_\mathcal{P}$ (left side) and $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ (right side)

State (10,196) can be reached by the event sequence depicted in Figure 7. The last event in the time diagram is the swReady not present in state (10,196) of $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ and in state 1 of EA4.

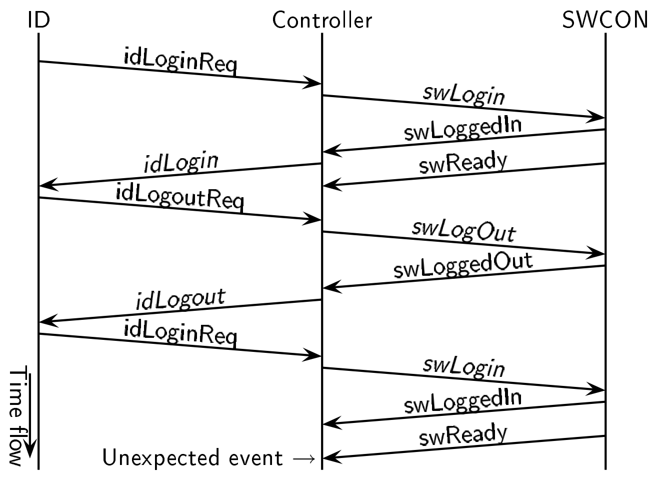


Figure 7: Time diagram showing a counterexample for the controllability of $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$

The interpretation of the events in the time diagram is as follows: an operator sends a login request idLoginReq to the controller, which then sends the command swLogin to the switch. The switch takes the appropriate measures to log the operator in and sends the acknowledgment swLoggedIn to the controller. Suppose further that the switch is configured to automatically activate the operator, so shortly after it also sends the event swReady. In the meantime, the controller has instructed the inquiry desk to finish the login procedure with the command idLogin. For some reason, the operator immediately wants to log out and sends the request idLogoutReq to the controller. The controller reacts with a swLogOut command to the switch. The switch confirms the operation with swLoggedOut, and the controller sends the command idLogout to the inquiry desk. From the point of view of the operator, the system appears now to be back to the initial state. However, not all automata are back to state 0. The exception is automaton EA4, which remains in state 1, waiting for the event idReady.

Eventually, the operator issues a new login request idLoginReq. Because EA4 does not depart from state 0, the controller will not visit the same states as before. After the event swLogin and the corresponding acknowledgment swLoggedIn, the automata $\mathcal{A}_\mathcal{P}$ and $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ are in the states 10 and (10,196) respectively, as shown in Figure 6. Automaton EA4 is still in state 1. Due to the automatic activation, the next event is an unexpected swReady coming from the switch.

3.4 Synthesis

The supervisor obtained in [SKWP02] can be reproduced using the μ-calculus based approach, solving Equation System 1. The result is an automaton with 239 states and 969 transitions. As the authors from [SKWP02] note, the supervisor ensures that both interfaces are managed correctly, no specifications are violated, and the restrictions to the system's behavior are kept as small as possible. Implementation of the controller can be done automatically, thereby increasing flexibility when a change in the behavior is needed.

3.5 An undesirable situation

In spite of these advantages, it is possible to find a subtle problem in the solution presented in [SKWP02]. The problem can be explained with the aid of Figure 8, which shows part of the supervisor's transition relation. Assume the first nine events in the diagram in Figure 7 – from idLoginReq to idLogout – occur in the sequence depicted there. The system then returns to its initial state, while the supervisor remains in state (0,93). This is again the situation in which the automata ID, SWCON, and EA1 to EA3 are back to state 0, while EA4 remains in state 1.

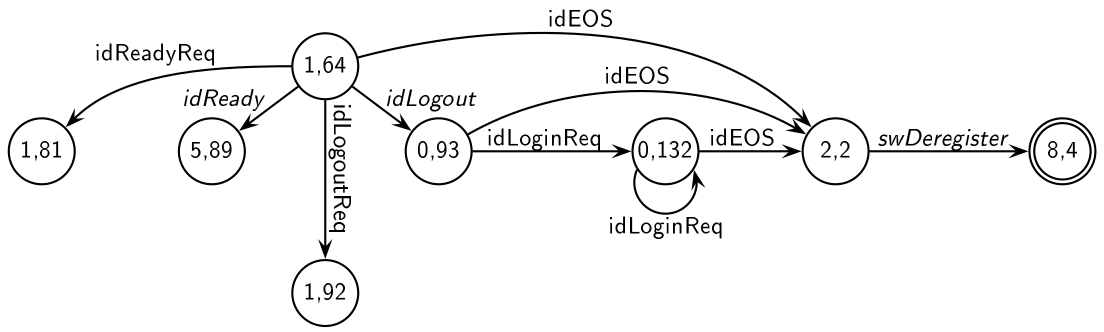


Figure 8: Part of the supervisor according to the original solution

From Section 3.3 we know that the automaton $\mathcal{A}_\mathcal{P} \times \mathcal{A}_\mathcal{E}$ would reach the bad state (10,196) in Figure 6 after the remaining events in the time diagram. In order to avoid this, the supervisor in Figure 8 inhibits the event swLogin in state (0,93). This prevents the swLoggedIn before last in the diagram from occurring, so the bad state becomes unreachable. However, inhibiting swLogin in state (0,93) also cuts off the path from this state back to the initial state. Since state (0,93) still has a path to the marker state (8,4) through idEOS and swDeregister, it is not removed by the synthesis algorithm. The operator, however, may not want to deregister, but rather to log in to the system again. However, when trying to do so with no matter how many idLoginReq requests, the supervisor gets trapped in the selfloop around state (0,132). The operator has the impression of a livelock, and is forced to issue an idEOS request. It is then deactivated by swDeregister, and returning to normal activity is only possible after a reset procedure that is not part of the automata model.

This problem could be avoided if states like (0,93), which lose their path back to the initial state during synthesis, were also removed. Within the conventional synthesis approach, however, such requirements can be ensured only in very special cases, for example if the product $\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}$ has only one marker state. In this case, avoidance of blocking is enough to ensure that every state in the supervisor retains a path back to that state. The automaton $\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}$, however, already has a marker state, namely (8,4). Therefore, marking the initial state too would not force the synthesis algorithm to remove state (0,93), because the latter has a path to state (8,4) and therefore fulfills the non-blocking criterion.

3.6 Improving the supervisor

In order to avoid the problem described above, the synthesis procedure must remove those states from $\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}$ whose path back to the initial state cannot be kept due to controllability reasons. Note, however, that it does not suffice to require that every state has a path to the initial state. The automaton $\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}$ has also some states that do not have such a path, which are nevertheless needed to allow the deactivation through the idEOS request when appropriate. Therefore, if a state that does not have a path to the initial state is found during synthesis, it has to be removed if and only if it did have such a path at the beginning of the computations. For all other states, it is enough to ensure absence of blocking in the sense of conventional synthesis. Of course, controllability has also to be ensured for all states.

As shown in [Zil05], translation of this informal specification into temporal logics expressions and then into a μ-calculus equation system in the form of Equation System 2 is quite straightforward. Solving the equation system and projecting the solution onto $\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}$ according to Definition 3 results in a new supervisor for the call center, this time with 218 states and 894 transitions. As could be expected, it is somewhat smaller than the former solution (239 states, 969 transitions), because now states like (0,93) in Figure 8 are no longer present.

The behavior of the system under the new supervisor can be checked using the time diagram in Figure 7 once more. Recall that, with the conventional solution, the first nine events lead to state (0,93) of the supervisor in Figure 8, and that this state should be avoided by the new supervisor. Figure 9 shows some states of the new supervisor to allow the comparison. After the eighth event in the sequence, both supervisors reach state (1,64). However, the new supervisor forbids the event idLogout in that state, thereby preventing states (0,93) and (0,132) from being reached. Instead, the controller is forced to choose the command idReady to the inquiry desk, which is the correct response to the swReady notification previously received. Only then the supervisor enables the command idLogout. This way, the supervisor returns to the initial state (0,0) through state (5,89), and the operator can log in to the system as often as needed. The correctness of the behavior can also be checked with the automata on Figures 4 and 5.

To make sure that the new restrictions imposed to $\mathcal{A}_{\mathcal{P}} \times \mathcal{A}_{\mathcal{E}}$ are still acceptable, it suffices to check whether a state in which the operator is logged in and active can be reached. This is clearly the case, since the first events in the time diagram lead to such a state. Alternatively, this can be confirmed by means of a simple model checking query on the Kripke structure.

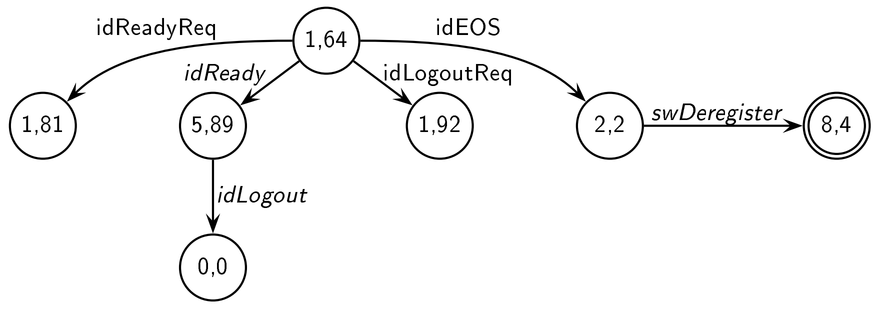


Figure 9: Part of the supervisor according to the improved solution

4 Conclusion

Conventional supervisor synthesis is limited to deal with specifications about safety and liveness properties. While these properties are important for the specification of reactive systems, their description may also involve requirements that translate to persitence and fairness properties. Supervisor synthesis can be extended to handle such properties when combined with model checking. As the example in this paper illustrates, the result is a powerful design tool for the design of discrete, event-driven systems. The method allows an easy translation of informal specifications given at the beginning of the design process into a formal representation. From that point on, the synthesis of the controller can be done automatically. As the result is given in form of a finite automaton, the designer is still free to chose between a software or a hardware implementation.

References

[AC88] A. Arnold and P. Crubille. A linear algorithm to solve fixed-point equations on transition systems. *Information Processing Letters*, 29(2):57–66, 1988.

[BCM90a] C. Berthet, O. Coudert, and J.C. Madre. New ideas on symbolic manipulations of finite state machines. In *International Conference on Computer Design (ICCD)*, pages 224–227. IEEE Computer Society, 1990.

[BCM+90b] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Symposium on Logic in Computer Science (LICS)*, pages 1–33, Washington, D.C., June 1990. IEEE Computer Society.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71, Yorktown Heights, New York, May 1981. Springer.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT, London, England, 1999.

[CL99] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems.* Kluwer Academic Publishers, Boston, U.S.A., 1999. ISBN 0-7923-8609-4.

[Dam94] M. Dam. Temporal logic, automata, and classical theories - an introduction. Notes for the Sixth Summer School in Logic, Language, and Information, 1994.

[EC80] E.A. Emerson and E.M. Clarke. Characterizing correctness properties of parallel programs as fixpoints. In *Colloquium on Automata, Languages and Programming (ICALP)*, volume 85 of *LNCS*, pages 169–181, Berlin, 1980. Springer.

[Koz82] D. Kozen. Results on the propositional μ-calculus. In *Colloquium on Automata, Languages and Programming (ICALP)*, pages 348–359, 1982.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, December 1983.

[MP88] Z. Manna and A. Pnueli. The anchored version of the temporal framework. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 428–437, Noordwigherhout, Netherland, May/June 1988. Springer.

[MP90] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Symposium on Principles of Distributed Computing*, pages 377–408, 1990.

[Pra81] V. Pratt. A decidable μ-calculus. In *Symposium on Foundations of Computer Science (FOCS)*, pages 421–427, New York, 1981. IEEE Computer Society.

[RW87] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, 1987.

[RW89] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[Sch03] K. Schneider. *Verification of Reactive Systems – Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003. ISBN 3-540-00296-0.

[SKWP02] M. Seidl, T. Kleinert, J. Wagner, and B. Plannerer. Systematischer Steuerungsentwurf zur Kontrolle lastintensiver Call-Center. *at - Automatisierungstechnik*, 50:19–27, June 2002.

[Won04] W. M. Wonham. Supervisory control of discrete-event systems. Technical report, Dept. of Electrical and Computer Engineering, University of Toronto, Jul. 2004. ECE 1636F/1637S 2004-05, http://www.control.utoronto.ca/DES.

[WR87] W.M. Wonham and P.J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal of Control and Optimization*, 25(3):637–659, May 1987.

[Zil05] R. Ziller. *Eine Verallgemeinerung der Überwachersynthese mit Hilfe des μ-Kalküls*. Dissertation, Universität Karlsruhe, 2005.

[ZS03a] R. Ziller and K. Schneider. A generalized approach to supervisor synthesis. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 217–226, Mont Saint-Michel, France, 2003. IEEE Computer Society.

[ZS03b] R. Ziller and K. Schneider. A μ-calculus approach to supervisor synthesis. In R. Drechsler, editor, *GI/ITG/GMM Workshop: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 132–143, Bremen, Germany, 24-26 February 2003. Shaker.

[ZS05] R. Ziller and K. Schneider. Combining Supervisor Synthesis and Model Checking. *ACM Transactions on Embedded Computing Systems*, 4(2):331–362, May 2005.