Improving evolvability of a patient communication control system using state-based supervisory control synthesis[☆]

R.J.M. Theunissen[*], D.A. van Beek, J.E. Rooda

Department of Mechanical Engineering, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract

Supervisory control theory enables control system designers to specify a model of an uncontrolled system in combination with associated control requirements, and subsequently to use a synthesis algorithm for automatic controller generation. The use of supervisory control synthesis can significantly reduce development time for supervisory controllers as a result of the unambiguous specification of control requirements, and synthesis of controllers that—by definition—are nonblocking and satisfy the control requirements. This is especially important for evolving systems, for which requirements change frequently.

For successful industrial application, the specification formalism should be expressive and intuitive enough to be used by domain experts, who define control requirements, and by software experts, who implement control requirements and synthesize controllers. This paper defines such a supervisory control specification formalism that consists of automata, synchronizing actions, guards, updates, invariants, and independent and dependent variables, where the values of the dependent variables can be defined in terms of functions of the independent variables.

We also show how the formalism enables systematic, compositional specification of a control system for a patient communication system of an MRI scanner. We show that our specification formalism can deal with both event-based and state-based interfaces. To support systematic, modular specification of models for supervisory control synthesis, we introduce state trackers that record sequences of events in terms of states. The synthesized supervisor has been successfully validated by means of interactive user guided simulation.

Keywords: supervisory control synthesis, compositional specification, evolvability, industrial application, simulation, interfaces.

1. Introduction

1.1. Supervisory control

Controlled systems can be divided into two parts: the physical components (hardware) and the control components; see Figure 1. The physical components, typically sensors, actuators and the connecting structure, provide the means for controlling the machine. The interactions between the physical components result in the so-called uncontrolled behavior of the machine. The combination of the physical components is known as the system under control or *plant*. The control components interact with the sensors and actuators to achieve the required functionality of the machine, which results in the controlled behavior of the machine. The controlled behavior should be such that the machine fulfills its functions, i.e., meets its predefined requirements. The combination of the physical components together with the control components is the *controlled system*.

The control components can be divided into multiple levels of control [1]; as shown in Figure 1:

- *Resource control*, also known as regulative or low-level control, assures that a system reaches the desired position or the desired state in the desired way.
- *Supervisory control*, also known as logic control, assures that a system correctly performs its function by determining and executing allowed sequences of tasks on (possibly dependent) resources.
- *User interface* provides the interaction with the users of the system. A user of the system can be a human operator, but also another system.

At each level of control, the principles of feedforward and feedback control can be applied. In this paper, we focus on the design of the supervisory control level of the control systems. At the supervisory control level, many systems can be regarded as discrete event systems [2]. A discrete event system has discrete states, and the system switches between these states by means of events. Discrete event systems can be modeled by means of automata, i.e., finite state machines. For the purpose of supervisory control, the combination of all components of the system—other than the supervisory controller shown in Figure 1—is called the *plant*.

It is current practice to design discrete event controllers by hand, based on the (generally informal) requirements of the

[☆]This work has been carried out as part of the DARWIN project at Philips Healthcare under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economics Affairs under the BSIK program.

[*]Corresponding author.

Email addresses: `r.j.m.theunissen@tue.nl` (R.J.M. Theunissen), `d.a.v.beek@tue.nl` (D.A. van Beek), `j.e.rooda@tue.nl` (J.E. Rooda)

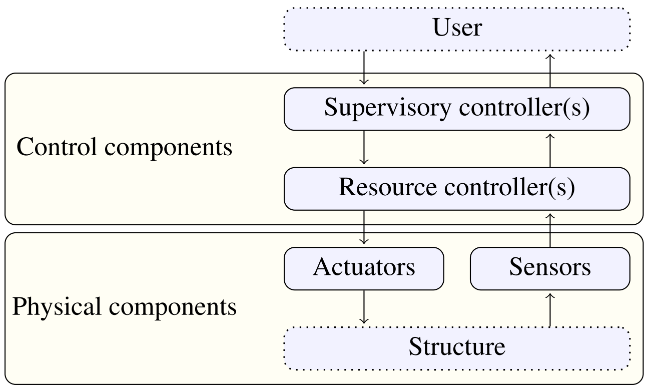


Figure 1: System view.

desired behavior of the system. Based on this design, an implementation of the supervisor is made. Usually, this is done by a software developer who manually encodes the requirements in a program. When the implementation is ready, it is integrated with (a model of) the hardware, and tested, possibly by means of simulation, to validate its correctness. A supervisor is considered to be correct if the system behaves in conformance with the requirements, and is both deadlock and lifelock free. If it turns out that the system does not behave correctly, the manually coded supervisor needs to be corrected by fixing the design and/or the implementation. Then the supervisor can be tested again. This loop is repeated until there is enough confidence that the system will behave correctly. However, this approach has several disadvantages:

- Informal requirements are usually ambiguous and incomplete, leaving room for differing interpretations.
- The relation between the original requirements, the design and the implementation code is often hard to find, making it difficult to change the design and implementation after changes in the requirements.
- Only during the testing phase can the system be validated, resulting in the late discovery of errors.
- Testing the system thoroughly, and finding and fixing errors is difficult, time-consuming, and unpredictable, potentially causing delays in market introduction.

These disadvantages are especially true for *evolving* systems, where requirements change frequently. Making new (or updating) designs and implementations is time-consuming, cost-intensive, and short time frames may compromise the overall quality of the system over time.

To support the evolvability of systems, the design process of supervisory controllers can be automated. By means of Supervisory Control Theory (SCT), initiated by Ramadge and Wonham [2, 3], supervisors can be synthesized, i.e., generated. The input for this method are models of the plant and models of the control requirements. The models of the plant model the behavior of the system without control. The control requirements specify the restrictions on the behavior of the controlled system. The control requirements also include rules to prevent both deadlock and lifelock. Based on these models, algorithms synthesize a supervisor that ensures that the controlled system satisfies the control requirements. This approach has the following advantages:

1. The need to design the supervisor by hand is eliminated.
2. The models of the plant and of the control requirements are unambiguous.
3. The supervisor is correct (w.r.t. the models of the plant and the control requirements) and nonblocking by construction, which eliminates the need for exhaustive testing and verification of the supervisor implementation.
4. The synthesized supervisors are suitable for code generation. This makes the design to a large extent independent of the implementation technology.

1.2. *Different flavors of supervisory control theory*

In centralized event-based supervisory control synthesis, the plant and control requirements are each modeled by means of a set of finite automata, synchronizing on shared event labels; see [2, 3]. The synthesis algorithm leads to a single supervisor. Although this form of supervisory control synthesis is conceptually simple, it is practically infeasible to handle large complex systems due to the exponentially high computational complexity resulting from synchronous composition of local component and requirement models. To cope with the complexity issue, many new event-based techniques have been introduced, such as interface-based hierarchical synthesis [4], and distributed aggregative synthesis [5]. For a more complete overview, see [6]. In interface-based synthesis, the plant consists of several components located at various levels. The components interact via interfaces. The components are controlled by local supervisors that can be locally synthesized, as long as all interfaces can remain fixed during synthesis. In distributed aggregative synthesis, local supervisors are also used, but the main technique used to avoid high synthesis complexity is model abstraction. The common denominator among event-based synthesis methods is the use of automata to model the plant and control requirements. Control requirements are thus specified as sequences of events.

A different method of supervisory control synthesis proposes the use of state tree structures as the underlying model of the plant, and state-based expressions as control requirements. The STS symbolic synthesis [7] algorithm utilizes binary decision diagrams to manipulate states. It is highly efficient and results in a single, efficiently encoded, supervisor. State tree structures are a superset of automata. They enable hierarchical modeling and support efficient storage. State-based control requirements restrict the behavior of the plant by forbidding combinations of states of the parallel components. These state-based control requirements are referred to as *mutual state exclusion* requirements. There are also *state-transition exclusion* requirements, which specify that events are disabled for a given combination of states. Apart from these state-based control requirements, the method also allows so-called dynamic state feedback, in which automata are used to specify control requirements as sequences of events, by transforming such automata to a form suited for state-based supervisory control. By means of this dynamic state feedback, event-based control requirements can also be defined in the state-based supervisory control framework. However, the allowed input form of the state-based control requirements as implemented in the STSlib [8] tool is quite restricted. Mutual state exclusion requirements are defined as the negation of a conjunction of location references, where a location reference is the name of a location of an automaton. The predicate over the state in a state-transition exclusion requirement is defined as a conjunction of location references.

To allow a more intuitive way of defining control requirements, [9] proposes generalized state-based control requirements. Generalized state-based requirements extend the input format of state-based supervisory control synthesis by allowing general propositional logic in the mutual state exclusion requirements and in state-transition exclusion requirements. This is done by symbolic preprocessing of the requirements to obtain the form required for the STSlib tool. Using generalized state-based control requirements to define the control requirements for a coordinator of maintenance procedures of a high-tech printer by Océ show a considerable gain in the process of formalizing the informally specified control requirements; see [10].

Another way of extending event-based supervisory control with state-based functionality is by extending automata with variables, guards and updates. Multiple frameworks that extend automata with variables exist, such as [11, 12, 13, 14]. In [11], it is assumed that variables are updated by at most one automaton. In [12] automata extended with variables are used to implement a supervisor. In [13], it is assumed that variables are local to each automaton. In [14], variables are global and can be updated by any automaton. These automata are referred to as extended automata. The algorithms defined in [14] are implemented in the Supremica [15] tool.

1.3. Contribution

Experience with application of supervisory control synthesis in industry [16, 17, 9, 18] has shown that key aspects of the successful application of supervisory control synthesis in industry are the *expressiveness* and *user-friendliness* of the formalism for specification of plant models and control requirements. Currently, requirements are *defined* by domain experts, and the corresponding control code is *implemented* by software experts. For example, the required number of pages that a printer is supposed to print, or the action to be taken when a sheet is stuck in the printer, or the functionality of the user interface for an MRI scanner are all defined by experts on the design and operation of a printer, or MRI scanner, respectively. Software experts then implement these informal requirements as executable code.

To facilitate the use of formal models of the plant and control requirements as the basis of controller synthesis and subsequent real-time code generation, the specification formalism should be expressive and intuitive enough to support definition and understanding of the plant models and control requirements by both domain experts *and* software experts. Experience has demonstrated that the ability to refer to the *state* of the controlled system is a key aspect in the definition of formal control requirements by domain experts.

The contribution of this paper is the definition and use of a (subset of a) specification formalism for supervisory control synthesis that is both expressive and intuitive enough to be used by both domain experts and software experts. We also show how the language enables systematic, *state-based* and *event-based*, *compositional* specification of a control system for a patient communication system of an MRI scanner.

A modular approach is generally considered beneficial for the design of complex systems; see for example [19]. For the design of *evolvable* systems, modularity is especially valuable, since it implies decomposition of the system in a number of smaller, relatively independent subsystems, that interact via well-defined interfaces. In this way, changes in a subsystem do not affect other subsystems, as long as the interfaces between the subsystems remain unchanged. We show that our specification formalism can deal with both event-based and state-based interfaces. To support systematic, modular specification of models for supervisory control synthesis, we introduce the concept of *state trackers*. A state tracker records sequences of events in terms of states, where the states are represented by values of variables, and/or automaton locations, that are required for the specification of control requirements. For the variables, we introduce two classes: the independent and the dependent variables, where the values of the dependent variables can be defined in terms of functions of the independent variables.

Our specification formalism integrates concepts from supervisory control synthesis based on state tree structures; see [7], with concepts from CIF (see [20, 21]), extended automata (see [14]), and generalized state-based control requirements from [9].

The requirements modeled in this paper are a snapshot of the requirements for the actual communication system. At the time of writing, the system was still under development. The requirements were changing due to interaction with the customer.

This paper is structured as follows. The patient communication system is described in Section 2. Section 3 describes the specification formalism, and Section 4 discusses the control architecture used for modeling the patient communication system. The plant model, the requirements models and the supervisor model are presented in Sections 5, 6 and 7, respectively. Section 8 describes how changes from a state-based interface to an event-based interface can be readily incorporated into the control system by straightforward addition of plant components and control requirements. Sections 9 and 10 present the synthesized supervisor for the changed model and the used tool chain, respectively. Finally, Section 11 presents concluding remarks.

2. Description of the patient communication system

An MRI scanner is used mainly in medical diagnosis to render pictures of the inside of a patient non-invasively. The patient is positioned inside the scanner for examination. The MRI system is usually operated by two operators. One of the operators is in the examination room (OpEx) together with the patient. The other operator is in the control room (OpCo) that is separated from the examination room. The main controller of the MRI system is called the MRI host system. Figure 2 shows the

Figure 2: The patient communication system

operator in the control room and the MRI scanner with the patient in the examination room.

The patient communication system is responsible for the delivery of all audio signals in the patient environment. Audio signals originate from multiple sources. First, from the patient and the two operators. These signals are captured by microphones. Second, from an audio stream that can be played by the MRI host system to instruct the patient during a scan. For instance, the patient can be instructed to hold his or her breath during particular stages of the scan. Finally, from an auxiliary device, such as an MP3-player that can play music to give extra comfort to the patient. The audio signals are delivered to the patient and the two operators via speakers and headphones.

2.1. *Audio channels*

The audio signals traverse from the sources to the targets via audio channels. Figure 3 shows an overview of the audio channels. Audio sources are depicted by single line boxes, while audio source and sinks are depicted by double line boxes. The audio channels are depicted by arrows connecting the sources to the targets. Audio channels can be opened and closed. Audio channels are unidirectional, meaning that audio signals are transferred in one direction only. For two-way communication, an audio channel in each direction must be open.

2.2. *Communication modes*

The communication system can operate in various communication modes. In each mode, various entities communicate with one another. A communication mode may be active or inactive. The following communication modes are implemented in the system:

- `OpEx_ListenToPat`: The operator in the examination room listens to the patient.
- `OpEx_TalkWithPat`: The operator in the examination room talks with the patient (two way conversation).

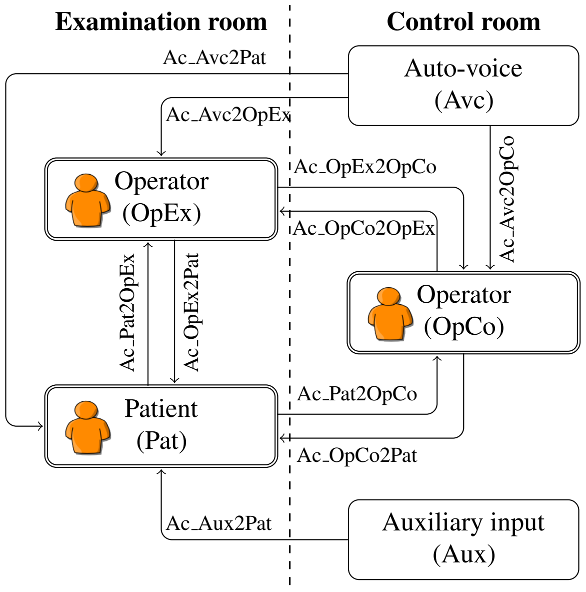


Figure 3: Overview of the audio channels

- `OpCo_ListenToPat`: The operator in the control room listens to the patient.
- `OpCo_TalkWithPat`: The operator in the control room talks with the patient (two way conversation).
- `OpCo_ListenToOpEx`: The operator in the control room listens to the operator in the examination room.
- `OpCo_TalkWithOpEx`: The operator in the control room talks with the operator in the examination room (two way conversation).
- `AutovoiceToAll`: The auto-voice messages are sent to the patient and both operators.
- `MusicToPatient`: The patient listens to music from the auxiliary audio source.

When a communication mode is active, the relevant channels must be open to allow the communication corresponding to that mode. Otherwise, the channels should be closed. Table 1 shows which channels are used for each communication mode.

Note that in principle all communication modes could be enabled at the same time. However, requirements may prevent some modes being enabled simultaneously. For instance, the requirement that the patient may only hear one source at a time prevents the modes `MusicToPatient` and `OpCo_TalkWithPat` to be active simultaneously. These conflicts are resolved by introducing priorities for the communication modes. In the case described above, the priorities ensure that mode `MusicToPatient` can only be active if mode `OpCo_TalkWithPat` is not active.

2.3. *Requests*

The communication modes in the communication system are activated and deactivated via requests to activate or deactivate a mode. These requests are made by the host and the users. The

Communication mode	Channels to open
`OpEx_ListenToPat`	Ac_Pat2OpEx
`OpEx_TalkWithPat`	Ac_Pat2OpEx Ac_OpEx2Pat
`OpCo_ListenToPat`	Ac_Pat2OpCo
`OpCo_TalkWithPat`	Ac_OpCo2Pat Ac_Pat2OpCo
`OpCo_ListenToOpEx`	Ac_OpEx2OpCo
`OpCo_TalkWithOpEx`	Ac_OpEx2OpCo Ac_OpCo2OpEx
`AutovoiceToAll`	Ac_Avc2Pat Ac_Avc2OpEx Ac_Avc2OpCo
`MusicToPatient`	Ac_Aux2Pat

Table 1: Communication modes with the corresponding channels

User	Button	Communication mode
Patient	Nursecall	`OpCo_ListenToPat`
OpEx	Talk with patient	`OpEx_TalkWithPat`
	Listen to patient	`OpEx_ListenToPat`
	Talk to OpCo	`OpCo_ListenToOpEx`
	Patient music	`MusicToPatient`
OpCo	Talk with patient	`OpCo_TalkWithPat`
	Listen to patient	`OpCo_ListenToPat`
	Talk with OpEx	`OpCo_TalkWithOpEx`
	Listen to OpEx	`OpCo_ListenToOpEx`
	Patient music	`MusicToPatient`
Host		`OpCo_ListenToPat`
		`AutovoiceToAll`

Table 2: Relation between the user interface buttons and the communication modes

users of the system are the patient, the OpEx and the OpCo. The MRI host system makes requests via a network. The host is informed if a request is accepted or rejected via the same network. The users make requests via buttons. The users are informed about the internal state of the communication system via indicators (LEDs).

A communication mode can be active only if there is a corresponding request. If no other conflicting higher priority communication modes are active, the request is granted immediately; otherwise the requested communication mode is activated as soon as all other higher priority conflicting communication modes are inactive.

For each communication mode, there is an indicator that visualizes the current state of the mode. Each indicator distinguishes three cases:

- communication mode is active;
- communication mode is inactive and requested;
- communication mode is inactive and not requested.

The buttons and the associated indicators together form the user interface. Table 2 shows the buttons that are in the system, and for each button to which communication mode it relates. Furthermore, it shows which communication modes are related to the host.

3. Specification formalism

This section informally introduces a subset of the supervisory control specification formalism used to specify the plant model and the control requirements of the patient communication system. This formalism is based on a small subset of the general Compositional Interchange Format (CIF) for hybrid systems, see [20, 21]. The subset has several concepts in common with the *extended automata* defined in [14]; in particular, untimed automata with shared discrete variables, shared uncontrollable and controllable events, guards, and assignments. Differences are that our formalism has several additional concepts from CIF: the independent and dependent variables (referred to as algebraic variables in CIF), the invariants, and multiple initial locations.

3.1. Automata

An automaton consists of locations Q, variables V, alphabet Σ, and transitions E. The variables are divided into independent and dependent variables. They are discrete, and have a finite domain. The values of independent variables can be changed by means of assignments. An automaton can assign only its own variables. The value of a variable of another automaton can be referred to by prefixing such a variable by the name of the automaton that declares the variable. E.g. if a variable x is declared in an automaton A, then the value of that variable can be referred to as $A.x$ in another automaton. Independent boolean variables that are not explicitly initialized are initialized to false by default. The value of each dependent variable is obtained by evaluation of its defining expression. The events are divided into controllable and uncontrollable events. The supervisor can disable controllable events in certain states, whereas uncontrollable events cannot be disabled. The supervisor can prevent occurrence of uncontrollable events in certain states only by disabling controllable events in all states that may lead to occurrence of the uncontrollable event. The defined automata are composed in parallel and synchronize on shared events. An edge consists of a number of optional components: an event, a guard in the form of a predicate over variables and location references, and a multi-assignment on one or more variables.

A graphical representation of a supervisory control automaton is shown in Figure 4. The automaton has locations `Off`, `Requested`, `Accepted`, controllable events *av_reject, av_accept, av_abort*, uncontrollable events *host_av_request, host_av_done*, and guard expressions that are represented by *not av_ok* and *av_ok*.

Locations are modeled by vertices, and transitions are modeled by arrows. All states labels are unique. For instance, in Figure 4, `Ex.Off` refers to state `Off` in automaton `Ex`. Controllable and uncontrollable transitions are represented by solid and dashed arrows, respectively.

Dependent variables are declared as aliases:

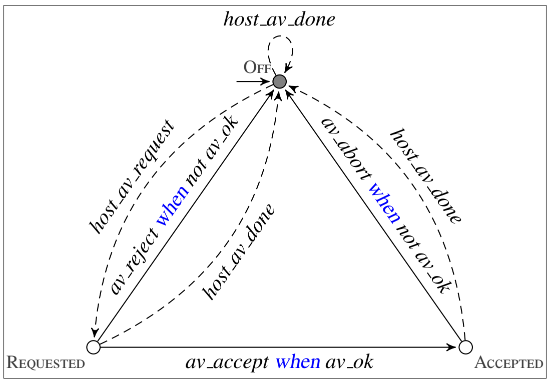


Figure 4: Example automaton Ex.

```
alias DEPENDENTVAR = EXPR
```

where EXPR is an expression over variables, location references and constants. Both independent and dependent variables can be used in EXPR as long as no circular dependencies are introduced.

A transition is labeled as follows:

```
EVENTLABEL when GUARD do MULTI-ASSIGNMENT
```

where:

- GUARD: Optional guard predicate over variables V and states Q. The predicates **P** are defined as follows:

  $$\mathbf{P} ::= true \mid \mathtt{s} \mid v = e \mid \mathtt{not}\ \mathbf{P} \mid \mathbf{P}\ \mathrm{op}\ \mathbf{P}$$

  where $\mathtt{s}$ is a location reference, $v \in V$ is an variable, e is a boolean or integer expression over variables and constants, and op $\in \{\, and, or, => \,\}$ is a logical operator. Note that we present only the subset of the language that is actually used in this paper.

- EVENTLABEL: An event label $\sigma \in \Sigma$.

- MULTI-ASSIGNMENT: Optional multi-assignment $v_1 := e_1, \cdots, v_n := e_n$ where v_i is an independent variable and e_i is an expression over variables (independent and/or dependent) and constants.

The guard and the keyword '`when`' are omitted when the guard is always true. The multi-assignment and the keyword '`do`' are omitted when the variables are not updated.

Execution of a transition transforms the state before execution, the *old* state, to a new state. The guards and the expressions e_i, including any dependent variables (if present), are evaluated in the old state. In the new state, the values of the assigned (independent) variables are equal to the values of the expressions e_i evaluated in the old state. The values of the independent variables that have not been assigned remain unchanged.

The well-known 'if then else' statement can be associated with an event by means of the following notation:

```
EVENTLABEL when GUARD do if GUARD′
                            then MULTI-ASSIGNMENT1
                            else MULTI-ASSIGNMENT2
                         end
```

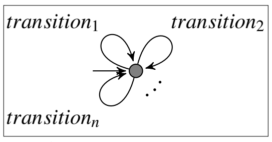


Figure 5: Graphical representation of an automaton with a single location.

```
event transition1
event transition2
  ...
event transitionn
```

Figure 6: Textual representation of an automaton with a single location.

which is an abbreviation for:

```
EVENTLABEL when GUARD and     GUARD′ do MULTI-ASSIGNMENT1
EVENTLABEL when GUARD and not GUARD′ do MULTI-ASSIGNMENT2
```

A textual representation is introduced for an automaton with a single location and self-loops, as shown in Figure 5. The textual model hides the single location, which is not relevant in this case. The textual representation of this automaton is shown in Figure 6. This textual representation is easer to read for models that have many self-loops.

3.2. *Requirement invariants*

A requirement invariant is a predicate **P** over variables and location references (as defined in Section 3.1) that must always evaluate to true. The invariant defines which states are safe in the controlled system. Requirement invariants are used in Section 8.4. However, plant invariants are not used in this paper.

4. Control architecture

The control architecture of the patient communication system is shown in Figure 7. MRI host H and user U generate uncontrollable events σ_{H_U} and σ_{U_U}, respectively, that activate or deactivate requests for communication modes in the state tracker T_{12}. This state tracker uses variables to keep track of whether a communication mode is requested. It consists of the two trackers T1 and T2, presented in Listings 1 and 2, respectively. The domain of the state of the supervisor is the union of the domain of the state x_T of the tracker and the domain of the state x_H of the host. The output states x_{UI} and x_A, that are made available to the indicators of the user interface U and to the audio channels A, respectively, are defined by functions on the state of the supervisor. The controllable events σ_{H_C} that are sent to the host H are enabled/disabled by the supervisor on the basis of the supervisor's state.

5. Plant model *P*

The plant model is a parallel composition of the models of the host H, user U, audio channels A, and tracker T_{12}.

5.1. *Host model H*

The host behavior consists of two parts: host notifications, and host auto-voice requests. These aspects of host behavior are described in the following subsections.

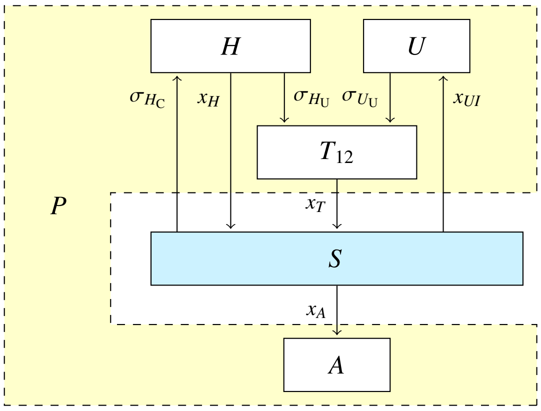


Figure 7: Control architecture

	event	contr.
σ_{H_U}	`host_scan_started`	No
	`host_scan_stopped`	No
	`host_tablemove_started`	No
	`host_tablemove_stopped`	No

Table 3: Events from host to tracker

Host notifications

The interface σ_{H_U} (see Table 3) from host H to tracker T_{12} is event-based. The host informs the communication system about host state changes that might require actions to be taken by the communication system; For instance, to improve the workflow, or to warn that a hazardous situation might occur. The communication system is notified when a scan starts and stops (workflow), and when the table starts and stops moving (hazard; the patient's fingers might become trapped), see the events specified in Table 3. The host itself does not restrict the occurrences of these host notification events. Therefore, this aspect of the host behavior is not explicitly modeled. Note that the absence of restrictions for some events can be modeled as a single state automaton with self-loops for each of the events. Adding such an automaton does not change the meaning of the model if the event already occurs in other automata.

Host auto-voice requests

The host can request enabling of the communication mode `AutovoiceToAll`. In this communication mode, an auto-voice message is played to the patient and to the operators. In certain circumstances, such an auto-voice request from the host may be ignored by the communication system. Therefore, the communication control system must notify the host whether the request is accepted or rejected. This is done according to the events listed in Table 4. The host behavior is specified by automaton `H` as presented in Figure 8.

Initially, the automaton is in mode `Off`. After a request from the host, the automaton changes to mode `Requested`. The request is accepted or rejected via the controllable events `autovoice_accept` and `autovoice_reject`, respectively. An accepted `AutovoiceToAll` request from the host, which results

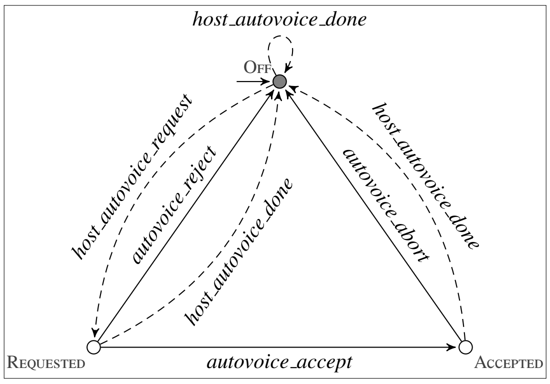


Figure 8: Host plant automaton `H`

	event	contr.
σ_{H_C}	`autovoice_reject`	Yes
	`autovoice_accept`	Yes
	`autovoice_abort`	Yes

Table 4: Events from supervisor to host

in playing of the auto-voice message, can be aborted by the supervisor via event `autovoice_abort`. The host can also terminate `AutovoiceToAll` requests by means of the uncontrollable event `host_autovoice_done`. The state x_H (see Table 5) of the host automaton is made available to the supervisor in the form of location references: the boolean expression `H.`ℓ, where ℓ can be any of the three location names of automaton `H`, is true if and only if that location is active.

5.2. *User model U*

The interface σ_{U_U} (see Table 6) from user U to state tracker T_{12} is event-based. When a user pushes a button, an event is generated which is sent to the tracker. The user interface has two kinds of buttons: push-buttons and hold-buttons. A push-button generates an event `<button>_pushed` when it is pushed. A hold-button generates two events: `<button>_pressed` and `<button>_released`. The user interface itself does not restrict the occurrence of these events in any way. Therefore this interface is not explicitly modeled.

The interface between the supervisor S and the user U is state-based. The state of the indicators in the user interface is a function of the state of the supervisor. Each indicator has three states: inactive, hold and active. Therefore, an indicator is modeled by a dependent enumerated variable with the values $\{Inactive, Hold, Active\}$. The relevant variables are given in Table 7.

	automaton location
x_H	`Off`
	`Requested`
	`Accepted`

Table 5: Locations in interface from host to supervisor

	event	contr.
σ_{U_U}	bt_opex_talkwithpat_pushed	No
	bt_opex_listentopat_pushed	No
	bt_opex_talktoopco_pushed	No
	bt_opex_patmusic_pushed	No
	bt_opco_listentopat_pushed	No
	bt_opco_listentoopex_pushed	No
	bt_opco_patmusic_pushed	No
	bt_opco_talkwithpat_pressed	No
	bt_opco_talkwithpat_released	No
	bt_opco_talkwithopex_pressed	No
	bt_opco_talkwithopex_released	No
	bt_pat_nursecall_pushed	No

Table 6: Events from user interface to tracker

	output variable
x_{UI}	Indc_OpEx_ListenToPat
	Indc_OpEx_TalkWithPat
	Indc_OpCo_ListenToPat
	Indc_OpCo_TalkWithPat
	Indc_OpCo_ListenToOpEx
	Indc_OpCo_TalkWithOpEx
	Indc_AutovoiceToAll
	Indc_MusicToPatient

Table 7: User interface output variables

5.3. *Tracker model T_{12}*

The tracker T_{12} of Figure 7 is modeled by two automata T1 and T2, shown in Listings 1 and 2, respectively. The trackers receive uncontrollable events from the host H and the user U, and relate these events to request for communication modes. The requests are modeled by the variables shown in Table 8.

Note that the trackers do not impose any restrictions on the occurrence of the events. Therefore, the state trackers could also have been introduced as part of the control requirements instead of as part of the plant. There would be no difference for the resulting synthesized supervisor.

The trackers are essential for evolvability of the control system, because they relate sequences of input actions to values of the request variables. These request variables were considered to be highly intuitive by the domain specialists, because they allow reasoning about the system at a higher level of abstraction. All changes to input action sequences required for communication mode requests can remain local to the tracker specifications, as long as the names of the request variables do not change.

	requested communication mode
x_T	Rq_OpEx_ListenToPat
	Rq_OpEx_TalkWithPat
	Rq_OpCo_ListenToPat
	Rq_OpCo_TalkWithPat
	Rq_OpCo_ListenToOpEx
	Rq_OpCo_TalkWithOpEx
	Rq_MusicToPatient

Table 8: Tracker request variables

Tracker T1

Tracker T1 declares the variables from Table 8, apart from variable Rq_OpCo_ListenToPat, which is declared in tracker T2. The variables are of type boolean. If the value of a request variable is true, the associated communication mode is requested, otherwise no action is taken. Note that omission of the initial value of a variable of a certain type, defaults to initialization of that variable to the default initial value for that type. For boolean variables, the default initial value is false. Therefore, the request variables are initially false. The requests that are modeled by tracker T1 in Listing 1 are toggled or always set to true or false when an event occurs.

Note that the events bt_opex_talkwithpat_pushed, host_scan_started and host_scan_stopped are included in both Listings 1 and 2. Due to the synchronous composition of these two automata, occurrence of an aforementioned event causes multiple requests for modes to be enabled (or disabled) simultaneously. The two automata T1 and T2 could be combined in one automaton. In that case, the assignments associated to the events bt_opex_talkwithpat_pushed, host_scan_started and host_scan_stopped in both automata would have to be combined as multi-assignments.

The variable Rq_OpEx_TalkWithPat is declared in tracker T1, where its value is updated. The value is used in tracker T2, where it is referred to as T1.Rq_OpEx_TalkWithPat.

Tracker T2

Listing 2 shows how the requests for communication mode OpCo_ListenToPat are modeled. The OpCo listens to the patient when the patient pushes the nursecall-button or when table is moving. Furthermore, the OpCo overhears the conversation between the OpEx and the patient. Finally, the OpCo can activate and deactivate the mode OpCo_ListenToPat whenever he or she likes. The four distinct cases must not influence one another. Therefore, four variables are used to model the request for communication mode OpCo_ListenToPat, each modeling a sub-request. The sub-request can only be deactivated by the same entity that activated it, with exception of the OpCo who can deactivate all sub-requests. The required behavior is modeled by the dependent variable (*alias*) Rq_OpCo_ListenToPat, which is defined in terms of the four independent variables Rq_OpCo_ListenToPat_i, $i \in$ {opco, opex, host, nursecall}, thus resulting in one request variable for this mode that can be used in the remainder of the specifications. The suffix of the variables indicates —for each submode—which entity enabled the request.

5.4. *Audio channel model A*

The audio channels are controlled by directly setting the state of the channels to opened or closed. Each channel is modeled by a boolean variable. If the value is true, the channel is opened, otherwise it is closed. The variables are given in Table 9.

```
plant T1:
  uncontrollable
    host_scan_started, host_scan_stopped,
    bt_opex_listentopat_pushed, bt_opex_talkwithpat_pushed,
    bt_opex_talktoopco_pushed, bt_opex_patmusic_pushed,
    bt_opco_listentoopex_pushed, bt_opco_patmusic_pushed,
    bt_opco_talkwithopex_pressed, bt_opco_talkwithopex_released,
    bt_opco_talkwithpat_pressed, bt_opco_talkwithpat_released;

  var bool Rq_OpEx_ListenToPat,  Rq_OpEx_TalkWithPat,
           Rq_OpCo_ListenToOpEx, Rq_OpCo_TalkWithOpEx,
           Rq_MusicToPatient,    Rq_OpCo_TalkWithPat;

  // host:
  event host_scan_started              do Rq_OpEx_ListenToPat  := false
                                        , Rq_OpEx_TalkWithPat  := false;
  event host_scan_stopped              do Rq_OpEx_ListenToPat  := true;

  // user interface buttons:
  event bt_opex_listentopat_pushed     do Rq_OpEx_ListenToPat  := not Rq_OpEx_ListenToPat;
  event bt_opex_talkwithpat_pushed     do Rq_OpEx_TalkWithPat  := not Rq_OpEx_TalkWithPat;
  event bt_opex_talktoopco_pushed      do Rq_OpCo_ListenToOpEx := not Rq_OpCo_ListenToOpEx;
  event bt_opex_patmusic_pushed        do Rq_MusicToPatient    := not Rq_MusicToPatient;
  event bt_opco_listentoopex_pushed    do Rq_OpCo_ListenToOpEx := not Rq_OpCo_ListenToOpEx;
  event bt_opco_patmusic_pushed        do Rq_MusicToPatient    := not Rq_MusicToPatient;
  event bt_opco_talkwithopex_pressed   do Rq_OpCo_TalkWithOpEx := true;
  event bt_opco_talkwithopex_released  do Rq_OpCo_TalkWithOpEx := false;
  event bt_opco_talkwithpat_pressed    do Rq_OpCo_TalkWithPat  := true;
  event bt_opco_talkwithpat_released   do Rq_OpCo_TalkWithPat  := false;
end
```
Listing 1: State tracker T1

```
plant T2:
  uncontrollable
    host_tablemove_started, host_tablemove_stopped, bt_pat_nursecall_pushed, bt_opco_listentopat_pushed;

  var bool Rq_OpCo_ListenToPat_nursecall, Rq_OpCo_ListenToPat_host,
           Rq_OpCo_ListenToPat_opex,      Rq_OpCo_ListenToPat_opco;

  alias bool Rq_OpCo_ListenToPat = Rq_OpCo_ListenToPat_opco
                                  or Rq_OpCo_ListenTopat_opex
                                  or Rq_OpCo_ListenToPat_host
                                  or Rq_OpCo_ListenToPat_nursecall;

  // host:
  event T1.host_scan_started           do Rq_OpCo_ListenToPat_opco      := false;
  event T1.host_scan_stopped           do Rq_OpCo_ListenToPat_opco      := true;
  event host_tablemove_started         do Rq_OpCo_ListenToPat_host      := true;
  event host_tablemove_stopped         do Rq_OpCo_ListenToPat_host      := false;

  // user interface buttons:
  event bt_pat_nursecall_pushed        do Rq_OpCo_ListenToPat_nursecall := true;
  event T1.bt_opex_talkwithpat_pushed  do Rq_OpCo_ListenToPat_opex      := not T1.Rq_OpEx_TalkWithPat;
  event bt_opco_listentopat_pushed  do if Rq_OpCo_ListenToPat
                                          then
                                            Rq_OpCo_ListenToPat_nursecall := false,
                                            Rq_OpCo_ListenToPat_host      := false,
                                            Rq_OpCo_ListenToPat_opex      := false,
                                            Rq_OpCo_ListenToPat_opco      := false
                                          else
                                            Rq_OpCo_ListenToPat_opco      := true
                                          end;
end
```
Listing 2: State tracker T2

	output variable
x_A	Ac_Pat2OpEx_Opened
	Ac_Pat2OpCo_Opened
	Ac_OpEx2Pat_Opened
	Ac_OpEx2OpCo_Opened
	Ac_OpCo2Pat_Opened
	Ac_OpCo2OpEx_Opened
	Ac_Sys2Pat_Opened
	Ac_Sys2OpEx_Opened
	Ac_Sys2OpCo_Opened
	Ac_Mus2Pat_Opened

Table 9: Audio interface output variables

	Resources (Targets)		
Boolean variable	Patient	OpCo	OpEx
OpCo_TalkWithPat	✓	✓	
OpCo_ListenToPat		✓	
OpCo_TalkWithOpEx		✓	✓
OpCo_ListenToOpEx		✓	
OpEx_TalkWithPat	✓		✓
OpEx_ListenToPat			✓
AutovoiceToAll	✓	✓	✓
MusicToPatient	✓		

Table 10: Communication mode priorities

6. Control requirements model

Based on the values of the variables and location references, supervisor S generates control outputs. The outputs can be event-based or state-based, depending on the interface for the control output. The supervisor enables and disables events used in tracker T_{12} and host H. The supervisor directly sets the indicator variables in U and the variables for the audio channels in A.

6.1. Communication mode priorities

The operators and the patient can each listen to multiple audio sources. For clarity, no more than one audio source may be heard at any one time. This implies that for each target, at most one of the audio channels to that target may be in the state open at one time. Therefore, communication modes may only be enabled at the same time if all corresponding channels can be opened at the same time.

Table 10 defines a partial order on the priorities of the communication modes. The tick marks in the table indicate the required resources for each mode. There is an ordering among communication modes only if they share resources. For such modes, the highest listed mode in Table 10 has the highest priority. However, if communication modes do not share resources, they can be active at the same time.

The requirements that define when each communication mode is enabled are derived from the priority table. A mode is enabled when its associated request variable is true, and when there is no higher priority mode enabled that shares one or more resources. For instance, mode OpEx_TalkWithPat is enabled only when the modes OpCo_TalkWithPat and OpCo_TalkWithOpEx are not active, whereas the modes MusicToPatient and OpCo_TalkWithOpEx are unordered: they can be active at the same time. These priority requirements are specified using aliases that define the values of the (dependent) communication mode variables in terms of the (independent) request variables and other, already defined, communication mode variables. In this way, the value of each communication variable is ultimately defined as a function of request variables. This is essential for the evolvability of the control system. Changes in the required priority rules can remain local to the alias definitions, because the only interaction with the other parts of the control system specification is by means of the names of the requests and the communication modes.

```
alias bool
  OpCo_TalkWithPat   = T1.Rq_OpCo_TalkWithPat,
  OpCo_ListenToPat   = T2.Rq_OpCo_ListenToPat
                       and not OpCo_TalkWithPat,
  OpCo_TalkWithOpEx  = T1.Rq_OpCo_TalkWithOpEx
                       and not OpCo_TalkWithPat
                       and not OpCo_ListenToPat,
  OpCo_ListenToOpEx  = T1.Rq_OpCo_ListenToOpEx
                       and not OpCo_TalkWithPat
                       and not OpCo_ListenToPat
                       and not OpCo_TalkWithOpEx,
  OpEx_TalkWithPat   = T1.Rq_OpEx_TalkWithPat
                       and not OpCo_TalkWithPat
                       and not OpCo_TalkWithOpEx,
  OpEx_ListenToPat   = T1.Rq_OpEx_ListenToPat
                       and not OpCo_TalkWithOpEx
                       and not OpEx_TalkWithPat,
  AutovoiceToAll     = H.Accepted
                       and Autovoice_OK,
  MusicToPatient     = T1.Rq_MusicToPatient
                       and not OpCo_TalkWithPat
                       and not OpEx_TalkWithPat
                       and not AutovoiceToAll;
```

where `Autovoice_OK` is defined as:

```
alias bool Autovoice_OK = not OpCo_TalkWithPat
                          and not OpCo_ListenToPat
                          and not OpCo_TalkWithOpEx
                          and not OpCo_ListenToOpEx
                          and not OpEx_TalkWithPat
                          and not OpEx_ListenToPat;
```

6.2. Auto-voice

Auto-voice requests are accepted only if the corresponding audio channels can be opened, indicated by Autovoice_OK. If the channels remain closed or be closed due to an active higher priority communication mode, the request is rejected or aborted, respectively. This behavior is defined by the control requirement AV defined below, in combination with the host model H presented in Figure 8.

```
requirement AV:
  event autovoice_accept when T1.Autovoice_OK;
  event autovoice_reject when not T1.Autovoice_OK;
  event autovoice_abort  when not T1.Autovoice_OK;
end
```

6.3. Channel open and close

Table 1 in Section 2.2 defines—for each communication mode—which channels must be open when the mode is active. Based on this table, the globally defined aliases below, define for each channel when it is open and when it is closed. The aliases define the interface between the communication modes in the control system, and the actual channels in the hardware specification, and in this way contribute to the evolvability of the control system design.

```
alias bool
  Ac_Pat2OpEx_Opened   = OpEx_TalkWithPat
                         or OpEx_ListenToPat,
  Ac_Pat2OpCo_Opened   = OpCo_TalkWithPat
                         or OpCo_ListenToPat,
  Ac_OpEx2Pat_Opened   = OpEx_TalkWithPat,
  Ac_OpEx2OpCo_Opened = OpCo_TalkWithOpEx
                         or OpCo_ListenToOpEx,
  Ac_OpCo2Pat_Opened   = OpCo_TalkWithPat,
  Ac_OpCo2OpEx_Opened = OpCo_TalkWithOpEx,
  Ac_Aux2Pat_Opened    = MusicToPatient,
  Ac_Avc2Pat_Opened    = AutovoiceToAll,
  Ac_Avc2OpEx_Opened   = AutovoiceToAll,
  Ac_Avc2OpCo_Opened   = AutovoiceToAll;
```

6.4. Indicators

For each communication mode, there is an indicator that visualizes the current state of the mode. The value of each indicator is specified by means of a conditional expression. An indicator is in state `Active` if the corresponding communication mode is active. It is in state `Hold` if the mode is requested but not active, and it is in state `Inactive` when the mode is not requested. For example for communication mode `OpEx_ListenToPat`:

```
alias LEDS
  Indc_OpEx_ListenToPat =
    if OpEx_ListenToPat
    then Active
    else if Rq_OpEx_ListenToPat then Hold else Inactive end
    end;
```

7. Supervisor model *S*

The supervisor consists of the aliases that define the states of the audio channels, the indicators, and the synthesized control functions for the controllable events. The control functions are synthesized using the method described in Section 10. The supervisor control functions for the control problem defined in Sections 5 and 6 are equivalent to the control requirements as defined in Section 6.2. Therefore, for this control problem, the control functions could be implemented by means of the control requirement automata.

Note that the control requirements defined in the preceding sections are the result of an iterative engineering process. In each iteration, the synthesized supervisor is tested by means of interactive, user-guided simulation to establish whether the defined control requirements define the behavior required by the customer. In these iterations, the control requirements can be blocking and/or uncontrollable (meaning that uncontrollable events are disabled by the control requirements). In such cases, synthesis is a valuable tool to help find errors in the control requirements by simulating the synthesized nonblocking and controllable supervisor on the plant model.

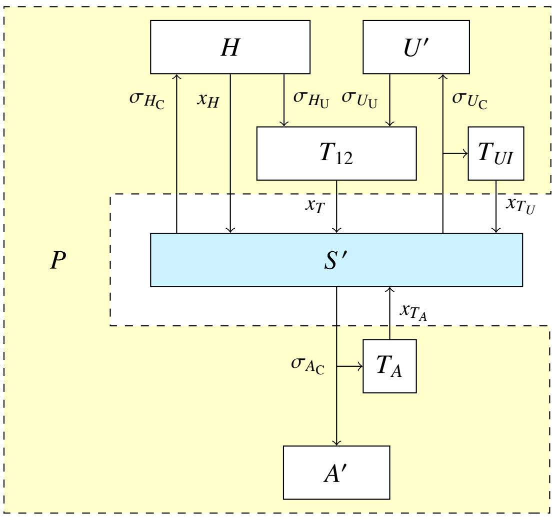


Figure 9: Control architecture for event-based output

8. Event-based output

In the preceding sections, the interfaces from the supervisor to the user and to the audio channels are state-based. The output of the supervisor is a function of the communication modes and the other variables of the model. In this section, the interface between the supervisor and the user and audio channels is changed to be event-based: instead of defining the output state as a function of the state of the plant and the tracker, events are used as outputs.

Figure 9 shows the new control architecture. The signals to users U' and audio channels A' are changed from variables ($x_{...}$) to events ($\sigma_{...}$), and two trackers are added to the plant. Tracker T_{UI} keeps track of the state of the indicators in U' by monitoring the events sent to U'. The state of T_{UI} is used by the supervisor to enable and disable events that are send to U'. Tracker T_A has the same purpose for the states and the events of A'.

The host model H remains the same. Also the events generated by the users U' remain the same. Therefore, the tracker model T_{12} also stays unchanged. In fact, the new model clearly demonstrates the evolvability of the control system design: the changes can be implemented by means of *additional* specifications, without changes to the original specifications. The specifications of the state-based output variables defined in Sections 6.3 and 6.4 are no longer required in an event-based output system, but the specifications need not be removed. The only component that changes is the supervisor S, which can be *generated* using a synthesis algorithm.

In the following sections, the changes to the plant model and control requirement model are discussed.

8.1. User interface model U'

The state of the indicators is changed by sending events to the user interface. Table 11 shows the events related to mode

	output event	contr.
σ_{U_C}	`indc_opexlistentopat_active`	Yes
	`indc_opexlistentopat_hold`	Yes
	`indc_opexlistentopat_inactive`	Yes

Table 11: New user interface output events

	output event	contr.
σ_{A_C}	`ac_opco2pat_open`	Yes
	`ac_opco2pat_close`	Yes

Table 12: New audio interface output events

`OpEx_ListenToPat`. For the other modes, similar events are used. The user interface accepts the events in any order, and is therefore not explicitly modeled.

8.2. *Audio channel model A'*

The state of the audio channels is also changed by sending events. Table 12 shows the event for the audio channel between the OpCo and the patient. For the other channels, similar events exist. The interface accepts the events in any order, and is therefore not explicitly modeled.

8.3. *Trackers T_{UI} and T_A*

The indicators in U' and the audio channels in A' do not provide feedback about the current state of the system. Therefore, trackers are added to the plant model. These trackers provide the current state of the indicators and audio channels for the supervisor. The indicator tracker model for the mode `OpEx_ListenToPat` is given below. The state of the indicator is modeled using an enumerated variable with the values `Inactive`, `Hold` and `Active`. The other indicators are modeled in a similar way.

```
enum LEDS = {Inactive,Active,Hold};

plant T_UI:
  var LEDS Indc_OpEx_ListenToPat = Inactive;

  event indc_opexlistentopat_active
        do Indc_OpEx_ListenToPat := Active;
  event indc_opexlistentopat_hold
        do Indc_OpEx_ListenToPat := Hold;
  event indc_opexlistentopat_inactive
        do Indc_OpEx_ListenToPat := Inactive;
  // ... and so on for other indicators
end
```

The plant model `T_A` is the tracker for the audio channels. The model below specifies only the channel between the Patient and the OpEx. The state of the audio channel is modeled using a boolean variable. The audio channel is opened and closed via controllable events `ac_pat2opex_open` and `ac_pat2opex_close`, respectively. The models of the other audio channels are similar.

```
plant T_A:
  var bool Ac_Pat2OpEx_Opened;
  event ac_pat2opex_open  do Ac_Pat2OpEx_Opened := true;
  event ac_pat2opex_close do Ac_Pat2OpEx_Opened := false;
  // ... and so on for other audio channels
end
```

8.4. *Control requirement models*

The control requirements defined in Section 6 remain unchanged. For event-based output, three types of requirements are added: requirements related to mutual exclusion of audio sources, to opening and closing of channels, and to setting of indicator states.

Mutual exclusion of audio sources

To prevent that the operators and the patient can hear multiple sources during transitions between control modes, a mutual exclusion requirement is added. This requirement prevents that channels with the same target may be open at the same time.

For compact notation of mutual exclusion requirements, a function is introduced. A set of mutually exclusive states imply that if any one of the states in a set is active, all other states must be inactive. The function is defined for a set of state predicates $\mathcal{SP}$:

$$mutex(\mathcal{SP}) = \bigwedge_{x \in \mathcal{SP}} \left(x \implies \left(\bigwedge_{y \in \mathcal{SP}, y \neq x} \neg y \right) \right) \tag{1}$$

For example $mutex(x, y, z) =$

$$(x \implies (\neg y \wedge \neg z)) \wedge (y \implies (\neg x \wedge \neg z)) \wedge (z \implies (\neg x \wedge \neg y))$$

At all times, the following mutual exclusion rules must be obeyed for the patient, the operator in the examination room and the operator in the control room:

```
requirement Channel_mutex:
  invariant
    mutex( Ac_OpCo2Pat_Opened
         , Ac_OpEx2Pat_Opened
         , Ac_Avc2Pat_Opened
         , Ac_Aux2Pat_Opened ),
    mutex( Ac_OpCo2OpEx_Opened
         , Ac_Pat2OpEx_Opened
         , Ac_Avc2OpEx_Opened ),
    mutex( Ac_Pat2OpCo_Opened
         , Ac_OpEx2OpCo_Opened
         , Ac_Avc2OpCo_Opened )
end
```

Opening and closing of the channels

The following requirement defines when a channel open or close event may occur. The other channels are opened and closed using similar rules.

```
requirement Audio_Channel_events:
  event ac_pat2opex_open
        when ( OpEx_TalkWithPat or OpEx_ListenToPat )
        and not Ac_Pat2OpEx_Opened;
  event ac_pat2opex_close
        when not ( OpEx_TalkWithPat or OpEx_ListenToPat )
        and Ac_Pat2OpEx_Opened;
end
```

Setting the indicator states

The following requirement defines when the output events for the OpEx listen to Pat indicator may occur. The other output events are defined using similar rules:

```
requirement Indicator_events:
  event indc_opexlistentopat_active
       when Rq_OpEx_ListenToPat and OpEx_ListenToPat
            and Indc_OpEx_ListenToPat != Active;
  event indc_opexlistentopat_hold
       when Rq_OpEx_ListenToPat and not OpEx_ListenToPat
            and Indc_OpEx_ListenToPat != Hold;
  event indc_opexlistentopat_inactive
       when not Rq_OpEx_ListenToPat
            and Indc_OpEx_ListenToPat != Inactive;
end
```

9. Supervisor model S'

As an example of the result of the synthesis procedure, the generated supervisory control rules for the audio channel from the patient to the OpEx in the form of a supervisory control automaton is shown below. The restrictions that the other channels must be closed before the channel can be opened are the result of the synthesis algorithm:

```
event ac_pat2opex_open
     when not ac_Pat2OpEx_Opened
          and not Ac_Avc2OpEx_Opened
          and not Ac_OpCo2OpEx_Opened
          and ( OpEx_TalkWithPat or OpEx_ListenToPat )
event ac_pat2Opex_close
     when ac_Pat2OpEx_Opened
          and not ( OpEx_TalkWithPat or OpEx_ListenToPat )
```

10. Toolchain

Figure 10 shows the toolchain that was used to synthesize the supervisor.

First, the automata of the plant model $P_{\mathbb{A}}$ are transformed into regular automata P_{A} by elimination of the variables, using the algorithm defined in [14]. The output of this algorithm are automata without variables P_{A}, and a mapping table M that maps the variables, states and events in the $\mathbb{A}$ automata to states and events in the regular automata.

Second, control requirement automata $R_{\mathbb{A}}$ are translated to (generalized) state-transition predicates $R_{\mathrm{STP}_{\mathbb{A}}}$ [9]. Note that all requirements are modeled by self-loops without multi-assignments. Therefore, the translation is straightforward. A transition:

```
  event EVENTLABEL when GUARD
```

is translated to the following state-transition predicate:

$$\rightarrow\!\text{EVENTLABEL} \Rightarrow \text{GUARD}\!\downarrow$$

which is equivalent to:

$$\neg\,\text{GUARD}\!\downarrow \Rightarrow \neg \rightarrow\!\text{EVENTLABEL}$$

meaning that the event EVENTLABEL is disabled in all states where the guard GUARD does not hold.

Third, state-transition predicates $R_{\mathrm{STP}_{\mathbb{A}}}$ and requirement invariants $R_{\mathrm{Inv}_{\mathbb{A}}}$ are mapped to the states of the plant without variables, resulting in $R_{\mathrm{STP}_{\mathrm{A}}}$ and $R_{\mathrm{Inv}_{\mathrm{A}}}$.

Fourth, regular plant automata P_{A} with the corresponding control requirements $R_{\mathrm{Inv}_{\mathrm{A}}}$ and $R_{\mathrm{STP}_{\mathrm{A}}}$ are used to synthesize the supervisor $S_{\mathrm{STP}_{\mathrm{A}}}$. The synthesis algorithm defined in [7], and the preprocessing transformation defined in [9] are used to synthesize the supervisor. These algorithms take as input the plant modeled as parallel automata and the requirements modeled as predicates over the states and events of the automata. The result of this synthesis method is a set of predicates $S_{\mathrm{STP}_{\mathrm{A}}}$, which define for each controllable event when the event is enabled. These predicates are defined over the states of the regular plant automata P_{A}.

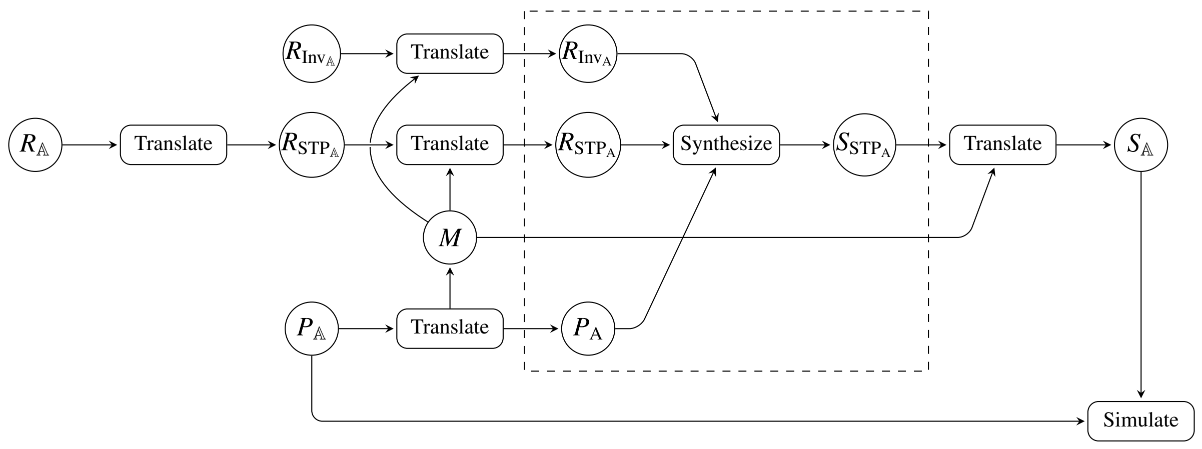
Using mapping M, the synthesized state-transition predicates $S_{\mathrm{STP}_{\mathrm{A}}}$, are mapped to predicates over the states and variables that are used in the plant model $P_{\mathbb{A}}$. The resulting state-transition predicates are transformed to automata $S_{\mathbb{A}}$. This supervisor can be simulated together with plant model $P_{\mathbb{A}}$.

11. Concluding remarks

First, we have defined the concepts of a specification formalism for supervisory control synthesis that is expressive and intuitive enough to be used by both domain experts and software experts. The formalism is based on untimed automata and invariants, that can each be of type ‘plant’ or ‘requirement’. Automata consist of locations, edges, and variables. An edge of an automaton consists of an event label, a guard, and a multi-assignment. The variables are divided in two classes: independent variables that can be assigned, and dependent variables whose values are obtained by evaluation of their defining expressions.

Second, we have applied the supervisory control specification formalism for the control system design of a patient communication system of an MRI scanner. The application illustrates how the division of the specification into a number of small, relatively independent components with well-defined interfaces increases the evolvability of the control system. Essential for the increased evolvability is the use of trackers and both dependent and independent variables. The trackers record the history of events, generated by the user interface and the host, in the independent variables that model the requests for specific communication modes. This decouples the recording of the input events, from the requirements that specify which communication mode can be activated. These requirements define the value of each (dependent) communication mode variable as a function of its associated request variable and the values of other, higher priority, communication variables. Thus, changes in the trackers and changes in the control requirements can be made independently of each other, as long as the *interface* between the tracker and the control requirements in terms of the request variables remains unchanged.

The output of the supervisor in terms of activation and deactivation of the communication channels can be defined as a function of the communication modes. As an illustration of the evolvability of the control system design, the specification is updated for event-based output, by adding events, trackers,



$R_{\mathrm{Inv}_{\mathbb{A}}}$
Translate
$R_{\mathrm{Inv}_{\mathrm{A}}}$
$R_{\mathbb{A}}$
Translate
$R_{\mathrm{STP}_{\mathbb{A}}}$
Translate
$R_{\mathrm{STP}_{\mathrm{A}}}$
Synthesize
$S_{\mathrm{STP}_{\mathrm{A}}}$
Translate
$S_{\mathbb{A}}$
M
$P_{\mathbb{A}}$
Translate
P_{A}
Simulate

Figure 10: Toolchain, where P = Plant, R = Requirement, S = Supervisor, and the subscripts $\mathbb{A}$ = Automata with variables, A = Regular automata, STP = State-Transition Predicate, and Inv = Invariant.

and requirement invariants that ensure mutual exclusion of communication channel activation. Thus, the defined communication mode interface decouples the output specification from the other control system requirements. The only component that is changed in the specification for event-based output is the supervisor, which can be generated using a supervisory control synthesis algorithm.

For implementation of the supervisory control synthesis algorithm, a sequence of steps connecting already existing supervisory control synthesis tools by means of various translations has been defined. These steps were manually executed to obtain the synthesized supervisors. Currently, the steps are automated. Future work is to define a synthesis algorithm that can directly manipulate the defined plant models and control requirements. This would eliminate the translation steps and reduce the complexity of the synthesis algorithm. A synthesis algorithm based on extended automata is defined in [14]. However, that algorithm does not support requirement invariants and results in a monolithic supervisor automaton without variables.

The automated steps shown in Figure 10 execute within seconds. Our experiences with the supervisory control synthesis algorithm based on state tree structures ([7]) on bigger examples are also quite positive. The supervisor for a patient support system of an MRI scanner, where the uncontrolled system consisted of 6.3 billion states, is generated in seconds. However, to deal with much bigger systems, we expect some form of modular supervisory control to be required.

One of the obstacles in realizing evolvable systems is the difficulty to capture the customers needs. Customers find it difficult to envision what the system's behavior will be and what exact behavior is desired, when the desired system is not yet available. Therefore, the customer cannot tell in advance what the exact requirements for the system are. This is one of the reasons that informal control requirements are usually incomplete and and/or ambiguous. Only after the system has been created, can the customer evaluate if the system obeys the exact customer needs. As a result, iterations are required in all development stages to make the customer needs clear.

For interaction with the customer, the synthesized supervisor can be interactively simulated together with the plant model. For this purpose, a computer visualization of the user interface, with clickable buttons, and a visualization of the state of the communication channels can be connected to the interactive simulation. Such an interactive simulation based on a generated supervisor allows for early user feed-back to establish whether or not the defined control requirements correspond to the behavior required by the customer. This interaction with the customer in the form of iterative testing of the synthesized controller by means of interactive simulation, and subsequent updates of the formal control requirements, allows the generation of a consistent set of requirements that meets the customer's expectation in a relatively short period of time. The same approach can also be used in the case of later changes if the system evolves over time.

We expect our approach to be especially suited to logic control via user interfaces, as in the patient communication application. As future work, we aim to apply the proposed design method in other industrial control system design applications. Our experiences with state-based supervisory control in industry are so far very positive; see for example [10, 9, 18].

Acknowledgment

We would like to thank Nico van Rooijen and Geran Peeren from Philips Healthcare for their support and cooperation. We would like to thank Geran Peeren in particular for showing us the importance of intuitive, user-friendly specifications combined with interactive simulation and visualization to support rapid iterative development of control requirements.

We would also like to thank the anonymous reviewers for their constructive and helpful remarks on the draft of the paper.

Finally, we would like to thank Dennis Hendriks, Albert Hofkamp and Michel Reniers for joint development of the supervisory control specification formalism used in this paper, and in particular Albert Hofkamp for his suggestion of defining variables locally to automata.

References

[1] R. H. Bishop, Mechatronic Systems, Sensors, and Actuators, 2nd Edition, CRC Press, 2008.

[2] C. G. Cassandras, S. Lafortune, Introduction to Discrete Event Systems, 2nd Edition, Springer, New York, 2007.

[3] W. M. Wonham, Supervisory control of discrete-event systems, Dept. Elect. Comput. Eng., Univ. Toronto, Toronto, ON, Canada, 2007.
URL `http://www.control.toronto.edu/DES/`

[4] R. J. Leduc, P. Dai, R. Song, Synthesis method for hierarchical interface-based supervisory control, IEEE Transactions on Automatic Control 54 (7) (2009) 1548–1560. `doi:10.1109/TAC.2009.2022101`.

[5] R. Su, J. H. van Schuppen, J. E. Rooda, Aggregative synthesis of distributed supervisors based on automaton abstraction, IEEE Transactions on Automatic Control 55 (7) (2010) 1627–1640. `doi:10.1109/TAC.2010.2042342`.

[6] R. Su, D. A. van Beek, J. E. Rooda, Deliverable D2.2.1 Report on methods and prototype tool for the supervisory control of complex systems, Tech. rep., MULTIFORM consortium (2010).
URL `http://www.multiform.bci.tu-dortmund.de/images/stories/multiform/deliverables/multiform_d221.pdf`

[7] C. Ma, W. M. Wonham, Nonblocking supervisory control of state tree structures, Vol. 317/ isbn = 978-3-540-25069-2,2005 of Lecture Notes in Control and Information Sciences, Springer, 2005.

[8] C. Ma, Stslib, `https://github.com/chuanma/STSLib` (2011).

[9] J. Markovski, K. G. M. Jacobs, D. A. van Beek, L. J. A. M. Somers, J. E. Rooda, Coordination of resources using generalized state-based requirements, in: 10th International Workshop on Discrete Event Systems, 2010, pp. 300–305.

[10] J. Markovski, D. A. van Beek, J. C. M. Baeten, L. J. A. M. Somers, Deliverable D-WP6-4 Implementation and industrial evaluation of the solutions, `http://www.c4c-project.eu/uploads/files/D-WP6-4.pdf` (2008).

[11] Y.-L. Chen, F. Lin, Modeling of discrete event systems using finite state machines with parameters, in: Proceedings of the IEEE International Conference on Control Applications, 2000, pp. 941–946. `doi:10.1109/CCA.2000.897591`.

[12] Y. Yang, R. Gohari, Embedded supervisory control of discrete-event systems, in: IEEE International Conference on Automation Science and Engineering, 2005, pp. 410–415. `doi:10.1109/COASE.2005.1506804`.

[13] B. Gaudin, P. H. Deussen, Supervisory control on concurrent discrete event systems with variables, in: American Control Conference, 2007, pp. 4274–4279. `doi:10.1109/ACC.2007.4282808`.

[14] M. Sköldstam, K. Åkesson, M. Fabian, Supervisory control applied to automata extended with variables - revised, Tech. Rep. R001/2008, Chalmers University of Technology, Sweden (2008).
URL `http://publications.lib.chalmers.se`

[15] Supremica, The official website for the supremica project (2010).
URL `http://www.supremica.org/`

[16] R. J. M. Theunissen, R. R. H. Schiffelers, D. A. v. Beek, J. E. Rooda, Supervisory control synthesis for a patient support system, in: Proceedings of the European control conference, Budapest, Hungary, 2009, pp. 4647–4652.

[17] E. Bertens, R. Fabel, M. Petreczky, D. A. van Beek, J. E. Rooda, Supervisory control synthesis for exception handling in printers., in: In Proceedings Philips Conference on Applications of Control Technology, 2009.

[18] S. T. J. Forschelen, J. M. van de Mortel-Fronczak, R. Su, J. E. Rooda, Application of supervisory control theory to theme park vehicles, in: Proceedings of the International Workshop on Discrete Event Systems, 2010, pp. 303–309.

[19] R. E. Fairley, Software Engineering Concepts, McGraw-Hill, New York, 1985.

[20] J. C. M. Baeten, D. A. van Beek, D. Hendriks, A. T. Hofkamp, D. E. Nadales Agut, J. E. Rooda, R. R. H. Schiffelers, Multiform Deliverable D1.1.2 Report describing the extended CIF functionality, `http://www.multiform.bci.tu-dortmund.de/images/stories/multiform/deliverables/multiform_d112.pdf` (2010).

[21] D. A. v. Beek, P. Collins, D. E. Nadales, J. E. Rooda, R. R. H. Schiffelers, New concepts in the abstract format of the Compositional Interchange Format, in: A. Giua, C. Mahuela, M. Silva, J. Zaytoon (Eds.), 3rd IFAC Conference on Analysis and Design of Hybrid Systems, Zaragoza, Spain, 2009, pp. 250–255.