# Property-Driven Scenario Integration

Jewgenij Botaschanjan and Alexander Harhurin
*Technische Universität München Department of Informatics*
*Boltzmannstr. 3, 85748 Garching, Germany*
*{botascha,harhurin}@in.tum.de*

***Abstract*—Scenario-based specifications have gained wide acceptance in requirements engineering. However, scenarios are not appropriate to describe global, system-wide invariants. Thus, a specification often consists of scenarios and universal properties. In order to obtain a consistent specification, the scenarios must be integrated in a way which does not violate the properties. However, manual integration of scenarios is an error-prone and laborious process.**

**In the presented paper we suggest a synthesis algorithm for automatic integration of system scenarios to an overall specification with guaranteed satisfaction of system-wide safety properties. The main idea is to compute inter-scenario priorities, which disable certain scenarios if they violate a property.**

***Keywords*-Scenario Integration, Automated Scenario Prioritization, Feature Interaction, Model Merging, Controller Synthesis**

## I. Introduction

Rapid increase in the amount and importance of different software-based functions as well as their extensive interaction are just some of the challenges that are faced during the development of reactive systems. Most serious errors in safety-critical systems occur due to inconsistencies in specifications, which may result from different stakeholders' views, different hypothetical scenarios, etc. The more complex the systems become, the more important it is to support validation and analysis already in the requirements engineering phase.

Due to their intuitive notation, scenario-based specifications of system behavior have gained wide acceptance in early development stages. A scenario specifies a sequence of events, that the system can receive/send from/to its environment. One of the advantages of such specifications is that scenarios enable the definition of only the relevant (partial) behavior of the system. Instead of having to study the complete specification of a system, we can focus on particularly interesting aspects of the behavior. Scenarios do not communicate directly: interactions only occur between the system and its environment. The reason for this is that the specification model should abstract from realization details like internal interfaces or protocols and describe the system from the black-box view.

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "SPES2020, 01IS08045A" and by the German Research Foundation (DFG), grant "Inserve III, BR 887/19-3". The responsibility for this article lies with the authors.

Scenarios provide examples of the intended system behavior, i.e., interaction patterns the system might show. The way how these patterns are integrated into an overall behavior is described by universal properties (or system invariants), which constrain the set of behaviors specified by scenarios. These properties, e.g., given as LTL formulas, concern the overall behavior rather than a single scenario and have to be fulfilled by any behavior. One could say, the properties define relationships between single scenarios. These relationships can be expressed in terms of *prioritizations* of one scenario over another in certain situations.

In light of these observations, we introduce a mathematical framework to model scenario-based specifications of multifunctional reactive systems. In our approach, the system functionality is specified as a set of partial models of the overall behavior. In other words, the overall behavior is specified from different viewpoints by a set of scenarios. These scenarios are formalized by means of I/O automata. The combination of these scenarios according to their priorities yields the overall specification of the system.

In the presented paper, we focus on the automatic synthesis of priorities between scenarios according to safety properties. Thereby, the main goal is to ensure that all safety properties always hold in the overall specification. Given a set of scenarios and a set of properties, our synthesis algorithm automatically determines whether the properties hold. Otherwise, it yields a set of priorities between scenarios such that their combined behavior fulfills all properties.

*Running Example:* The concepts introduced in the remainder of the paper will be illustrated on a fragment of a specification originally written and implemented for "Advanced Technologies and Standards" of Siemens, Sector Industry. The considered bottling plant system comprises several distributed subsystems: to transport empty bottles from a storehouse to the bottling plant, fill bottles with items, seal them, and transport them back to the storehouse. All these systems are operated by a central control unit (CU) which provides a user interface to receive commands and display the system status, as well as a device interface to send/receive control signals to/from the subsystems. Although there are over 40 scenarios of system behavior, in this paper we consider a small subset only, concerning the interplay between the CU and the conveyor belt. Among other things, the user can start and stop the conveyor. There is also an emergency brake available. When the emergency brake is activated, the CU immediately switches the conveyor off. In this case, the CU is not allowed to switch the system on and the emergency lamp flashes red until an abolition of the emergency command is received.

*Outline:* This paper is organized as follows. In Sec. II, we compare our work to related approaches. In Sec. III, the operational semantics of the proposed scenario-based specification is presented. Sec. IV is the core of the paper. There, we introduce our algorithm for the automatic synthesis of priorities between scenarios according to given safety properties. Finally, we show how our approach is implemented in a CASE tool in Sec. V and we conclude the paper in Sec. VI.

## II. Related Work

The presented work is based on a theoretical framework introduced by Broy [1] where the notion of partial behavior, decomposition, and refinement are formally introduced. This framework proposes to model behaviors as partial stream processing functions. However, it does not cover several relevant issues such as an operational semantics, combination of partial behaviors of the same system, or automatic prioritization of functions.

The idea of property-ensuring synthesis as first introduced by the Ramadge-Wonham framework [2] for discrete event systems (DES) comprises the search for a component (a *supervisor*), which controls a given component (a *plant*) according to a certain property. The supervisor is synthesized by composing the plant and the property automaton. The main idea of the framework has been incorporated into various further approaches, some of which are discussed below. It also serves as the foundation of our work. However, the particular problem statements differ. A supervisor is constructed to feed inputs into the plant and observe outputs. Our prioritization observes inputs of the system and activates or deactivates system functions.

The feature integration as a conventional synthesis problem is presented in [3]. There, the base system (plant) must be controlled in order to fulfill a given set of specifications. The features (supervisors) synthesized for this purpose can be composed with the plant according to a given linear priority order. Then, a specification is satisfied by the composition only when it does not conflict with a higher-prioritized one. In contrast, our method provides a controller which always satisfies all properties.

The framework for scheduler synthesis presented by Altisen et al. [4] resembles the synthesis problem of our work. Schedulers are synthesized according to the time behavior of system tasks, time constraints, and/or general scheduling policies. The scheduler decides when a task should be activated driven by certain events, e.g., timeouts, and does not influence the inputs of its tasks. However, the scheduler synthesis can guarantee the compliance with time constraints only since it is based solely on a timing model of system functions (tasks). The proposed approach ensures the fulfillment of more general functional system properties.

The closest approach to our work is the framework introduced by Uchitel et al. [5], who support synthesis of behavior models from *both* property and scenario models. Similar to our approach, their combination is defined as a minimal common observable refinement of both models. However, this combination is defined for consistent models only, i.e., both models must have a common refinement. In general, this is not guaranteed in the 3-valued semantics of [5]. Our work does not has such limitations. In the case of inconsistencies, prioritizations between scenarios are synthesized to ensure that the overall behavior matches the desired properties. Sabetzadeh et al. introduced a framework in [6], which merges models w.r.t. given *relations* between them, which can not be violated. However, these merges are not synthesized automatically and result from a manual negotiation process.

Damas et al. [7] also consider scenarios and safety properties as an input to synthesis. Their goal, similar to numerous further scenario-to-automata synthesis approaches, is to interleave scenarios in a way, which respects given properties. Inconsistencies between properties and scenarios do not reduce the number of possible system behaviors. In contrast, our approach is able to reveal and avoid inconsistencies between scenarios and/or properties.

## III. Service-Oriented Development

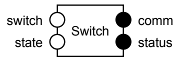
This section introduces a formal framework for modeling specifications of a system. Thereby, the basic building block of the model is a *service* – a formal representation of a scenario[1] (cf. Sec. III-A). The services can be *combined* to service-based specifications (cf. Secs. III-B and III-C). The property satisfaction problem of a service-based specification is dealt with in Sec. III-D.

### A. Service

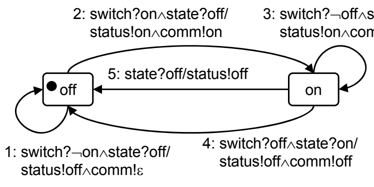
A service has a *syntactic interface* consisting of the sets of typed input and output ports. Fig. 1(a) depicts the syntactic interface of a service from our running example. There, input and output ports are depicted by empty and filled circles, resp.

The semantics of a service is described by an I/O automaton. This is a tuple $S = (V, \mathcal{I}, \mathcal{T})$ consisting of variables $V$, initial states $\mathcal{I}$, and a transition relation $\mathcal{T}$. $V$ consists of mutually disjoint sets of typed variables $I$, $O$, $L$. The type of a variable $v \in V$ is denoted by the function $ty(v)$, which maps $v$ to the set of all possible valuations. The variables from $I$ and $O$ are the input and output ports of the service

[1]The synthesis of single services from scenarios (e.g., in form of MSCs) is not in scope of this paper. The interested reader is referred to [8].



(a) Interface



(b) Automaton

Figure 1. Service `Switch`

interface, resp. $L$ is the set of local variables. A *state* of $S$ is a valuation $\alpha$ that maps every variable from $V$ to a value of its type. $\Lambda(V)$ is the set of all type-correct valuations for a set of variables $V$, i.e., for all $\alpha \in \Lambda(V)$ and all $v \in V$ holds $\alpha(v) \in ty(v)$.

We define the following relations on variable valuations: for $\alpha, \beta \in \Lambda(V)$ and $Z \subseteq V$, $\alpha \overset{Z}{=} \beta$ denotes the equality of variable valuations from $Z$, i.e., $\forall v \in Z : \alpha(v) = \beta(v)$. For an assertion $\Phi$ with free variables from $V$ and $\alpha \in \Lambda(V)$, we say that $\alpha$ *satisfies* $\Phi$, written as $\alpha \vdash \Phi$, iff $\Phi$ yields true after replacing its free variables with values from $\alpha$. Finally, the priming operation on a variable name $v$ yields a new variable $v'$ (the same applies to variable sets). Priming of valuation functions yields a mapping of equally valued primed variables, i.e., for given $\alpha \in \Lambda(V)$, $\alpha'$ is defined by $\forall v \in V : \alpha(v) = \alpha'(v')$. Priming is used to argue about the current and next state within the same logical assertion. For $\Phi$ with free variables from $V \cup V'$ and $\alpha, \beta \in \Lambda(V)$ we also write $\alpha, \beta' \vdash \Phi$ to denote that $\Phi$ yields true after replacing free unprimed variables by values from $\alpha$ and primed ones by values from $\beta$.

$\mathcal{I}$ is an assertion over $L \cup O$ characterizing the *initial states* of the system. $\mathcal{T}$ is a transition assertion over $V \cup V'$. In $\mathcal{T}$ the satisfying valuations of unprimed variables describe the current state while the valuations of the primed ones constrain the possible successor states. By enabling several satisfying successor state valuations for one current state, we can model non-determinism. A transition is not allowed to constrain primed input and unprimed output variables. By this, we disallow a service to constrain its own future inputs, and enforce the clear separation between the local state (read/write) and the outputs (write only).

We instantiated the above service model for an extended version of I/O automata used in our CASE tool. As in the classical I/O automata [9], a transition leads from one control state to another and might consist of four logical parts: precondition, input pattern, post-condition and output pattern. In our concrete syntax `i?x` denotes an input pattern, which evaluates to true if the variable $i \in I$ has the value $x$ and `o!x` an output pattern, which is satisfied by an assignment of value $x$ to the output variable $o' \in O'$. Fig. 1(b) shows the specification of service `Switch` from our running example, which formalizes the following scenario of the CU. The user can switch the conveyor on/off, by putting one of the two commands (`on` or `off`) in. Additionally, the CU receives the state of the conveyor through the port `state`. If the conveyor is in state `off` and the user switches it on, in the next step the CU sends command `on` through its port `comm` to the conveyor, as well as message `on` through port `status` to the user display (cf. Transition 2). Note, Transition 5 does only reference two of the four existing ports. This means that the remaining ports are allowed to have arbitrary values within their respective type domains when the automaton executes this transition (we speak of underspecification). For example, the port `comm` may have one of the following values $ty(comm) = \{on, \mathit{off}, \varepsilon\}$. $\varepsilon$ denotes an empty message.

In order to be able to reason about transition steps, we define the successor state of some valuation $\alpha$ as $\mathrm{Succ}(\alpha) \overset{\text{def}}{=} \{\beta \mid \alpha, \beta' \vdash \mathcal{T}\}$. The predicate En yields true if a service can make a step: $\mathrm{En}(\alpha) \overset{\text{def}}{\Leftrightarrow} \mathrm{Succ}(\alpha) \neq \emptyset$. If $\mathrm{En}(\alpha)$ holds, we say that the service is *enabled* in state $\alpha$.

The language of a service automaton consists of valuation sequences $\langle \alpha_0 \alpha_1 \dots \rangle$, called *runs*, such that $\alpha_0 \vdash \mathcal{I}$ holds and for all $i \in \mathbb{N}$ either $\alpha_{i+1} \in \mathrm{Succ}(\alpha_i)$ or $\alpha_i$ is the last element in the sequence. By this, the semantics of our service is *input-disabled.* The prefix-closed set of all runs of a service $S$ is denoted by $\langle\!\langle S \rangle\!\rangle$. The set of all reachable states of $S$ is denoted by $\mathrm{Reach}(S)$. Finally, we lift the $\overset{(.)}{=}$-operator to runs and sets of runs/valuations. For $r_1, r_2 \in \langle\!\langle S \rangle\!\rangle$ and $W \subseteq V$ we define $r_1 \overset{W}{=} r_2$ as $\forall i \in \mathbb{N} : r_1.i \overset{W}{=} r_2.i$, where $r.i$ stands for the $i$th valuation in $r$. For two sets of valuations $A$, $B$ we define $A \overset{W}{\subseteq} B$ as $\forall a \in A : \exists b \in B : a \overset{W}{=} b$ and $A \overset{W}{=} B \overset{\text{def}}{\Leftrightarrow} A \overset{W}{\subseteq} B \wedge B \overset{W}{\subseteq} A$. The same holds for two sets of runs.

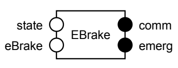
## B. Service Combination

The combination of services yields a service again. This directly reflects the idea that each scenario adds a certain new aspect of the specified behavior. The combination of these fragmented aspects yields the overall system behavior.

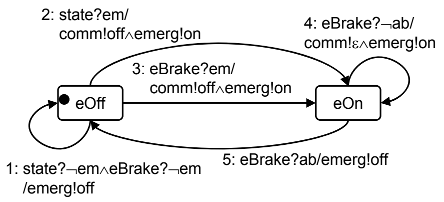
In our framework, the (parallel) service combination accepts all inputs, which the single services can deal with as long as the outputs produced by these services are unifiable (not contradictory). The reaction of the combination coincides with the reactions specified by the single services.

The combination of two services is defined only if input ports of one service and output ports of the other do not overlap with each other and with local variables: $(I_1 \cup L_1) \cap (O_2 \cup L_2) = (O_1 \cup L_1) \cap (I_2 \cup L_2) = \emptyset$ and their common variables $V_1 \cap V_2$ have the same type. Then, we speak of *combinable* services.

For two combinable services $S_1$ and $S_2$, their combination $C \overset{\text{def}}{=} S_1 \parallel S_2$ is defined by $C \overset{\text{def}}{=} (V_C, \mathcal{I}_C, \mathcal{T}_C)$, where $I_C \overset{\text{def}}{=}$

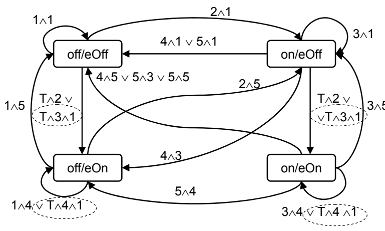


(a) Interface

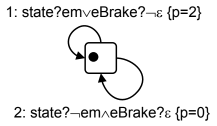


(b) Automaton

Figure 2. Service EBrake



(a) Service Combination



(b) Priority

Figure 3. Behavior Specifications

$I_1 \cup I_2$, $O_C \overset{\text{def}}{=} O_1 \cup O_2$, $L_C \overset{\text{def}}{=} L_1 \cup L_2$, $V_C \overset{\text{def}}{=} I_C \cup L_C \cup O_C$, $\mathcal{I}_C \overset{\text{def}}{=} \mathcal{I}_1 \wedge \mathcal{I}_2$. $\mathcal{T}_C$ is described by the successor function below. The combined automaton makes a step if either the current input can be accepted by both single services, and their reactions are not contradictory, or the input can be accepted by one of both services only. In the latter case, the local variables of the not enabled service (i.e., the service with $\neg\text{En}(\alpha)$) are not modified, and its output variables (not common with the first service) are unrestricted. Formally, the set of successor states of the combination is defined as follows:

$$\text{Succ}(\alpha) \overset{\text{def}}{=} \{\beta \mid \alpha, \beta' \vdash \mathcal{T}_1 \wedge \mathcal{T}_2\}$$
$$\bigcup_{i,j \in \{1,2\}, i \neq j} \{\beta \mid \alpha, \beta' \vdash \mathcal{T}_i \wedge \neg\text{En}_{S_j}(\alpha) \wedge \alpha \overset{L_j}{=} \beta\},$$

where $\text{En}_{S_j}(\alpha)$ is true iff $S_j$ is enabled in state $\alpha$.

To illustrate the concept of combination, we consider a further scenario concerning the emergency brake from our example. The CU switches the system off if the user puts the emergency brake on (message em on port eBrake) or a critical state message is received from the conveyor (message em on port state). The CU is not allowed to switch the system on and the emergency lamp flashes until an abolition of the emergency (ab) is received on eBrake. The service in Fig. 2 formalizes this scenario.

The combination of services Switch and EBrake results in the automaton in Fig. 3(a) (without transitions marked by dashed ovals). There, the labels of transitions are of the form $t_s \wedge t_e$, where $t_s$ and $t_e$ are the transition numbers from Fig. 1(b) and 2(b), resp. A label of the form $\top \wedge t_e$ identifies situations where service Switch is not enabled. A transition with a label $l_1 \vee l_2$ is an abbreviation of two transitions with labels $l_1$ and $l_2$, resp.

## C. Prioritized Combination

Usually, some events or scenarios explicitly have a higher priority in specifications than others. For example, the system reaction in the case of emergency has higher priority than the normal-case behavior. In order to be able to reflect this in our service model, we introduce the notion of a *prioritized combination*. It allows an individual service to take control over other services depending on specific input histories. Thereby, we can express different relationships between services without any modifications on them.

The prioritized combination temporally allows a service to take priority over another service. Therefore the combination is controlled by a special service $S_P$ with the interface consisting of all input ports of both services. Transitions of $S_P$ might prioritize one of both services. If the current input enables a transition of $S_P$ and this transition prioritizes service $S_2$, only $S_2$ is *executing* – the local state of $S_1$ remains unmodified, the output variables exclusively controlled by $S_1$ are not subject to any restrictions. If the current input does not enable any transition of $S_P$ or the enabled transition prioritizes no service, the combination behaves like the unprioritized one. Thus, the priority determines certain inputs for which the system behavior should coincide with the behavior of one of both services only.

The prioritized combination $PC \overset{\text{def}}{=} S_1 \|^P S_2$ is defined for a pair of combinable services $S_1$, $S_2$ and a special *priority service* $P \overset{\text{def}}{=} (V_P, \mathcal{I}_P, \mathcal{T}_P, p)$ with $V_P \overset{\text{def}}{=} I_P \uplus L_P$ and $L_P \cap (L_1 \cup L_2) = \emptyset$ by $PC \overset{\text{def}}{=} (V_{PC}, \mathcal{I}_{PC}, \mathcal{T}_{PC})$, where $I_{PC} \overset{\text{def}}{=} I_1 \cup I_2 \cup I_P$, $O_{PC} \overset{\text{def}}{=} O_1 \cup O_2$, $L_{PC} \overset{\text{def}}{=} L_1 \cup L_2 \cup L_P$, $V_{PC} \overset{\text{def}}{=} I_{PC} \cup L_{PC} \cup O_{PC}$, $\mathcal{I}_{PC} \overset{\text{def}}{=} \mathcal{I}_P \wedge \mathcal{I}_1 \wedge \mathcal{I}_2$. The function $p\colon \Lambda(V_{PC}) \times \Lambda(V_{PC}) \to \mathcal{P}(\{0,1,2\})$ defines whether $S_1$, $S_2$, or no service is prioritized by a certain transition step of $P$. For example, $p(\alpha, \beta) = \{0, 2\}$ with $\alpha, \beta \in \Lambda(V_{PC})$ and $\alpha, \beta' \vdash \mathcal{T}_P$ means that none of services or service $S_2$ may be prioritized. In other words, the combined step of both services as well as the sole step of service $S_2$ belong to the behavior of $PC$. Even if service $S_1$ can take the step $(\alpha, \beta)$ in isolation (i.e., $\beta \in \text{Succ}_{S_1}(\alpha)$), this step does not belong to the behavior of $PC$. Finally, if $p(\alpha, \beta) = \emptyset$, the behavior is no transition of $PC$.

$\mathcal{T}_{PC}$ is described by the successor function below. It is defined over the transition set $\mathcal{T}_C$ of the unprioritized combination of $S_1$ and $S_2$:

$$\begin{aligned}\mathrm{Succ}(\alpha) &\overset{\mathsf{def}}{=} \{\beta \mid (\alpha,\beta' \vdash \mathcal{T}_C \wedge \neg\mathcal{T}_P) \wedge \alpha \overset{L_P}{=} \beta\} \\ &\cup \{\beta \mid (\alpha,\beta' \vdash \mathcal{T}_C \wedge \mathcal{T}_P) \wedge 0 \in p(\alpha,\beta)\} \\ \bigcup_{i,j\in\{1,2\},i\neq j} &\{\beta \mid (\alpha,\beta' \vdash \mathcal{T}_i \wedge \mathcal{T}_P) \wedge i \in p(\alpha,\beta) \wedge \alpha \overset{L_j}{=} \beta\}.\end{aligned}$$

If $P$ is not enabled or its enabled transition is prioritized by 0, the behavior of $S_1 \parallel^P S_2$ coincides with the behavior of $S_1 \parallel S_2$. In the case when an enabled transition of $P$ prioritized by $i \in \{1,2\}$, the behavior of $S_1 \parallel^P S_2$ coincides with the common behavior of $S_i$ and $P$.

Returning to our running example, it makes sense to prioritize the emergency break signals `eBrake?em` and `state?em`. We require that the combined system must behave like service `EBrake` if one of these signals arrives. The priority service which prioritizes emergency signals is depicted in Fig. 3(b). While Transition 1 prioritizes service `EBrake`, Transition 2 prioritizes no service. The prioritized combination of `Switch` and `EBrake` with regard to the priority service results in the automaton in Fig. 3(a) (including formulas enclosed in dashed ovals). Thereby, the transitions of the form $t_1 \wedge t_2$ (not enclosed) belong to the unprioritized behavior (i.e., the priority automaton executes Transition 2). The transitions of the form $\top \wedge t_2 \wedge 1$ (enclosed) belong to the prioritized behavior (i.e., the priority automaton executes Transition 1). Whenever an emergency signal has arrived, this combination behaves like service `EBrake` (transitions marked by dashed ovals), otherwise the behavior is identical to the unprioritized combination from the last section.

Both un- and prioritized combination operators are well-defined. The un-prioritized combination is a special case of the prioritized one. It is commutative and associative. The prioritized combination is in general non-associative and non-commutative, however, it is distributive. All these properties are shown in [10].

## *D. Property Satisfaction*

In the previous sections we introduced the notion of the system as a combination of services provided to its environment. However, in practice, there also exist requirements concerning the overall behavior rather than a single scenario. These requirements or properties of the system are orthogonal to its services as they involve the system as a whole. A scenario describes a behavioral pattern, which may be observed during a system run, while a property must always be satisfied by every system run. In the latter case, we speak of system invariants. For example, independently of a particular scenario, the CU has to switch the system off whenever the user puts the emergency brake on or a critical state message is received from the conveyor. In the following, we introduce the notion of a property and that of the property satisfaction and reduce the problem of the safety property satisfaction to the reachability problem.

A property characterizes a set of allowed behaviors. In our settings the behaviors are sequences of variable valuations from some variable set. We assume that every safety property $\varphi$ over variable set $V_\varphi = I \uplus L \uplus O$ is characterized by a *property service* $S_\varphi = (V_\varphi, \mathcal{I}, \mathcal{T})$. This assumption is based on the idea that for any temporal formula we can construct an automaton that accepts precisely the computations that satisfy the formula [11]. Since we consider safety properties only, the constructed automaton is finite. $S_\varphi$ is assumed to be deterministic and input complete. Formally, for all reachable valuations $\alpha,\beta,\gamma \in \mathrm{Reach}(S_\varphi)$ the following relations hold

$$\begin{aligned}&\big(\beta \models \mathcal{I} \wedge \gamma \models \mathcal{I}\big) \Rightarrow \beta \overset{L}{=} \gamma \\ &\wedge \big(\beta,\gamma \in \mathrm{Succ}_{S_\varphi}(\alpha)\big) \Rightarrow \beta \overset{L}{=} \gamma, \qquad (1)\\ &\mathrm{Succ}_{S_\varphi}(\alpha) \neq \emptyset. \qquad (2)\end{aligned}$$

The (internal) determinism is necessary in order to make the decision about the satisfaction of a property locally, i.e., if an error state is not reachable from a state $\alpha$, the system never violates the property after reaching $\alpha$. This is not necessarily true in the non-deterministic case. The determinization of a service is concerned in Sec. IV-C. An input-complete property service explicitly allows an arbitrary reaction to the inputs, for which the behavior is not constrained by the property. The input completeness can be easily achieved by complementing each state by missing inputs leading to any outputs (we speak of chaos completion).

A service satisfies a property if all its runs belong to the set of behaviors allowed by the property. Thereby, we consider properties, which constrain dependencies between input and output variables. Local variables of a service are not considered. We denote the satisfaction of the property $\varphi$ over $V_\varphi$ by the service $S$ over $V$ by $S \models \varphi$. The satisfaction is defined as

$$\langle\!\langle S\rangle\!\rangle \overset{V\cap V_\varphi}{\subseteq} \langle\!\langle S_\varphi\rangle\!\rangle, \qquad (3)$$

only if $S_\varphi$ and $S$ are combinable (cf. Sec. III-B). In the rest of the paper we always assume services and properties to be combinable. Next, we adapt the well-known reduction of the safety property satisfaction to a reachability problem for the notion of our services.

*Error State & Error Runs:* A run not contained in the allowed set violates the property. Thus, we complement property services by transitions leading to an *error state* (cf. Fig. 4 for the complemented emergency property). An error state is any state $\alpha$ in which the special *new* variable $e$ ($ty(e) = \mathbb{B}$, $e \notin V$) is mapped to *true*. Let $S = (V, \mathcal{I}, \mathcal{T})$ be a service, then, we define $\hat{S} \overset{\mathsf{def}}{=} (V \cup \{e\}, \hat{\mathcal{I}}, \hat{\mathcal{T}})$ for all $\alpha,\beta \in \Lambda(V \cup \{e\})$ as

$$\begin{aligned}\alpha \vdash \hat{\mathcal{I}} &\overset{\mathsf{def}}{\Leftrightarrow} \big(\alpha \vdash \mathcal{I} \Leftrightarrow \neg\alpha(e)\big), \\ \alpha,\beta' \vdash \hat{\mathcal{T}} &\overset{\mathsf{def}}{\Leftrightarrow} \big(\alpha,\beta' \vdash \mathcal{T} \wedge \neg\alpha(e) \wedge \neg\beta(e)\big) \\ &\quad \vee \big(\neg(\alpha,\beta' \vdash \mathcal{T}) \wedge \neg\alpha(e) \wedge \beta(e)\big).\end{aligned}$$

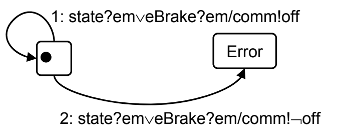

1: state?em∨eBrake?em/comm!off
Error
2: state?em∨eBrake?em/comm!¬off

Figure 4. Error Service `Emergency`

Any run of $S$ is obviously a run of $\hat{S}$. Also, any run contained in $\langle\!\langle \hat{S} \rangle\!\rangle$ but not in $\langle\!\langle S \rangle\!\rangle$ reaches a state $\alpha$ where $\alpha(e)$ holds. We call such runs *error runs*. Once in an error state, there is no transition leading back to a non-error one.

*Interface Reduction:* As already mentioned, properties describe the behavior of the system as a whole. However, single services are usually defined over subsets of the variables referenced by a property. The property satisfaction as defined in Eq. (3) takes this into account and projects system runs to the common variable subset. In order to reproduce this circumstance in our algorithmic analysis, we introduce the *interface reduction* of a service $S = (V, \mathcal{I}, \mathcal{T})$ by a variable set $W$ as $S|_W \overset{\text{def}}{=} (V \setminus W, \mathcal{I}|_W, \mathcal{T}|_W)$, where for all $\alpha, \beta \in \Lambda(V \setminus W)$ the following relations hold:

$$\alpha \vdash \mathcal{I}|_W \overset{\text{def}}{\Leftrightarrow} \exists \gamma \in \Lambda(V) : \gamma \vdash \mathcal{I} \wedge \alpha \overset{V \setminus W}{=} \gamma \text{ and}$$
$$\alpha, \beta' \vdash \mathcal{T}|_W$$
$$\overset{\text{def}}{\Leftrightarrow} \exists \gamma, \delta \in \Lambda(V) : \gamma, \delta' \vdash \mathcal{T} \wedge \alpha \overset{V \setminus W}{=} \gamma \wedge \beta \overset{V \setminus W}{=} \delta.$$

The relation $\langle\!\langle S \rangle\!\rangle \overset{V \setminus W}{=} \langle\!\langle S|_W \rangle\!\rangle$ between a service and its reduction is obvious. The accomplishment of a reduction is completely schematic. The constraints on variables from $W$ are replaced by *true*.

*Reduction to Reachability Problem:* We reduce the safety property satisfaction to a reachability problem. A property $\varphi$ over $V_\varphi$ is satisfied by $S = (V, \mathcal{I}, \mathcal{T})$ if and only if no error state is reachable in $C \overset{\text{def}}{=} \hat{S_\varphi}|_{V_\varphi \setminus V} \parallel S$ by $S$. In other words, whenever $S$ is enabled in $C$, $C$ never proceeds to an error state. Analogous to the projection on common variables in Eq. (3), the property service is reduced to constraints concerning $S$ only using the interface reduction. Formally, we define

$$S \models \varphi \overset{\text{def}}{\Leftrightarrow} \forall \alpha, \beta \in \text{Reach}(C) : \alpha \vdash \mathcal{I} \Rightarrow \neg\alpha(e)$$
$$\wedge \big(\text{En}_S(\alpha) \wedge \beta \in \text{Succ}_C(\alpha)\big) \Rightarrow \big(\alpha(e) \Leftrightarrow \beta(e)\big). \quad (4)$$

Obviously, Eq. (4) can be transformed into a reachability problem by simply marking states to which $C$ can transit while $S$ is enabled.

**Proposition 1.** *For a service and a property Eq.* (3) *holds if and only if Eq.* (4) *holds.*

*Proof Sketch: The proof follows from the observation that given S and $S_\varphi$ over V and $V_\varphi$, resp., then the set* $\{\alpha \in \text{Reach}(C) \mid \neg\alpha(e)\}$ *is equal to the set* $\{\alpha \mid \alpha \in \text{Reach}(S) \wedge \alpha \in \text{Reach}(S_\varphi)\}$ *over the variable set* $V \cap V_\varphi$. ∎

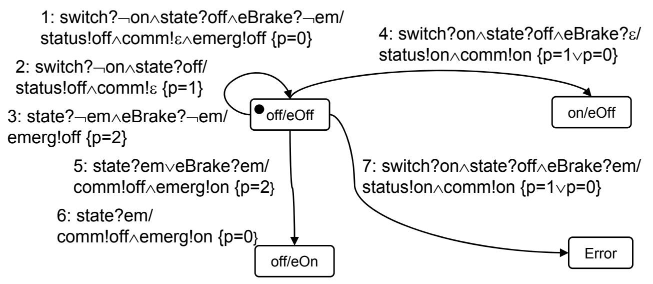


Figure 5. $($`Switch` $\parallel^{S_\top}$ `EBrake`$) \parallel$ `Emergency`

## IV. Prioritization Algorithm

For a pair of services and a property we provide an algorithm which synthesizes a priority between the services such that the prioritized service combination satisfies the property. The synthesized priority can disable one of both services in order to prevent the combination from violating the property. Thereby, we are interested in finding the most permissive prioritization. It should intervene only when the un-prioritized combination would potentially lead to the property violation. In this case, all possible non-violating prioritizations are allowed. This idea and its extensions for deadlock-free and modular prioritizations are formalized in the rest of this section.

The complete prioritization process can be outlined as follows. The corresponding deterministic versions of services $S_1$ and $S_2$ and the property service $S_\varphi$ are produced as described in Sec. IV-C. Then, the complete search space of all possible prioritizations is computed using the results of Sec. IV-A. Finally, the algorithm from Sec. IV-B, or its conflict-free modification from Sec. IV-D is applied to produce one of the following results: (1) $S_1 \parallel S_2 \models \varphi$, no prioritization is needed, (2) the priority service $P$ is computed, such that $S_1 \parallel^P S_2 \models \varphi$, or (3) $S_1$ and $S_2$ violate $\varphi$ under any prioritization. The determinization and search space computation can be accomplished iteratively along with the actual priority synthesis. This increases the efficiency of our method.

*Complexity Considerations:* Our synthesis procedure is linear in the product of the sizes of $S_1$, $S_2$, and $S_\varphi$. Thereby, the services are assumed to be deterministic. Determination presented in Sec. IV-C can cause an exponential blow-up in the worst case. However, in the practice that worst case rarely occurs for linear-time properties.

### A. Search Space

Given two services $S_1$, $S_2$ and a property $\varphi$ defined over $V_1$, $V_2$, and $V_\varphi$, resp. Our synthesis procedure starts with following service combination $C = (S_1 \parallel^{S_\top} S_2) \parallel \hat{S_\varphi}$.[2]

[2]Due to space limitations, we consider the matching variable sets only, i.e., we assume that $V_\varphi = V_1 \cup V_2$. The procedure and proofs for the more general case go analogous.

The service $S_\top$ is defined as follows: $S_\top \overset{\text{def}}{=} (I_1 \cup I_2 \cup I_\varphi, \mathcal{I}_1 \wedge \mathcal{I}_2 \wedge \mathcal{I}_\varphi, \mathit{True}, p)$ and $\forall \alpha, \beta : p(\alpha, \beta) = \{0, 1, 2\}$. $S_\top$ permits any un- and prioritized behavior of the combination – for any state pair $(\alpha, \beta)$ $S_1$, $S_2$ or none of them may be prioritized. The service $\hat{S_\varphi}$ ensures that $\beta(e)$ holds whenever the combination violates the property $\varphi$. This service combination is the search space of our synthesis algorithm.

For $C$ and its state $\alpha$ we write $\mathrm{Succ}_i(\alpha)$ with $\mathrm{Succ}_i(\alpha) \subseteq \mathrm{Succ}(\alpha)$ and $i \in \{0, 1, 2\}$ to denote sets of all successor states $\beta$ for which $i \in p(\alpha, \beta)$ holds. We call $\beta$ an *i-successor* of $\alpha$.

Fig. 5 shows a fragment of the combination automaton $(\texttt{Switch} \|^{S_\top} \texttt{EBrake}) \| \texttt{Emergency}$. Thereby, we only sketch the transitions going from the combined state `off/eOff`. According to the definition in Sec. III-C, the combination of Transitions 1 and 2 in Fig. 1(b), 1, 2 and 3 in Fig. 2(b), and both transitions in Fig. 4 yields 7 transitions displayed in Fig. 5. This combination allows for the unprioritized behavior ($p = 0$) as well as prioritization of service `Switch` ($p = 1$) or service `EBrake` ($p = 2$). Transition 7 obviously violates the `Emergency` property (Fig. 4) and because of Transition 2 in Fig. 4 it goes to the state `Error`.

**Algorithm 1** Priority Construction Algorithm

```
Ensure: $prio(A, Vis, Err) = (Pri, Vis^n, Err^n)$
 1: $Pri \leftarrow \emptyset$, $Err^n \leftarrow Err$, $Vis^n \leftarrow Vis$
 2: for all $\alpha \in A \setminus (Vis^n \cup Err^n)$ do
 3:   $Vis^n \leftarrow Vis^n \cup \{\alpha\}$
 4:   $(P_0, Vis^n, Err^n) \leftarrow prio(\mathrm{Succ}_0(\alpha), Vis^n, Err^n)$
 5:   if $\mathrm{Succ}_0(\alpha) \cap Err^n = \emptyset$ then
 6:     $Pri \leftarrow Pri \cup \{(\alpha, \beta, 0) \mid \beta \in \mathrm{Succ}_0(\alpha)\} \cup P_0$
 7:   else // prioritization needed
 8:     $(P_1, V_1, E_1) \leftarrow prio(\mathrm{Succ}_1(\alpha), Vis^n, Err^n)$
 9:     $(P_2, V_2, E_2) \leftarrow prio(\mathrm{Succ}_2(\alpha), Vis^n, Err^n)$
10:     $Err^n \leftarrow E_1 \cup E_2$, $Vis^n \leftarrow V_1 \cup V_2$
11:     if $\forall i \in \{1, 2\} : \mathrm{Succ}_i(\alpha) \cap Err^n \neq \emptyset$ then // an error is inevitable
12:       return $(Pri, Vis^n, Err^n \cup A)$
13:     end if
14:     for all $i \in \{1, 2\}$ do
15:       if $\mathrm{Succ}_i(\alpha) \cap Err^n = \emptyset$ then
16:         $Pri \leftarrow Pri \cup \{(\alpha, \beta, i) \mid \beta \in \mathrm{Succ}_i(\alpha)\} \cup P_i$
17:       end if
18:     end for
19:   end if
20: end for
21: return $(Pri, Vis^n, Err^n)$
```

## *B. Priority Service Synthesis*

Our algorithm pursues the following strategy. By means of appropriately chosen prioritization we try to prevent combination services from reaching an error state. The algorithm favors the unprioritized combination – if it has a choice between the unprioritized behavior ($p = 0$) and the prioritized one ($p \in \{1, 2\}$), it prefers the former. This directly reflects the main idea behind the service-based specification that each service extends the system behavior, i.e., a service should be enabled when possible. If in the state $\alpha$ every prioritization leads to an error state, we propagate the error to $\alpha$. If the error is propagated to the initial state, the property $\varphi$ is not satisfiable by the given service combination under any prioritization. If the synthesized priority never prioritizes a service (i.e., for each transition $p = 0$), the system always satisfies the property and the priority can be omitted.

The idea sketched above is realized by Alg. 1, which computes a set of priorities $Pri$ and a set of (propagated) error states $Err$ for an initial state set $A$ of a combined service $C$. Thereby, a priority is of the form $(\alpha, \beta, i)$, which means that in the step $(\alpha, \beta)$ prioritization $i \in \{0, 1, 2\}$ is valid. $Vis$ marks the visited states. Note, in the rest of this section, we assume $C$ to be deterministic, i.e., for each state $\alpha$ and each prioritization $i \in \{0, 1, 2\}$ there is at most one successor state: $\forall \beta_1, \beta_2 \in \mathrm{Succ}_i(\alpha) : \beta_1 \overset{L \cup O}{=} \beta_2$. Determinism is achieved by the method from Sec. IV-C.

Given a set of states, the algorithm performs a *depth-first* search on $C$. For each unvisited and not erroneous state $\alpha \in A$ (Line 2 in the algorithm), the prioritization of its unprioritized successors $\mathrm{Succ}_0$ is recursively computed (Line 4). If all 0-successors do not lead to error states, no prioritization is required – for each $\beta \in \mathrm{Succ}_0(\alpha)$ the function $p(\alpha, \beta)$ is set to 0 (Line 6). Otherwise, the prioritizations of 1- and 2-successors are recursively computed (Lines 8 and 9). If in both prioritized successors an error state is reachable (Line 11), an error is inevitable – no prioritization can prevent the system from reaching an error state. In this case the error is propagated backwards to all states from $A$ (Line 12). Otherwise, the appropriate prioritization is synthesized (Line 16).

Given a service combination $C = (S_1 \|^{S_\top} S_2) \| \hat{S_\varphi}$, the set of prioritizations is computed by the following function:

$$(Pri, ., Err) = prio(\{\alpha \vdash \mathcal{I}_C\}, \emptyset, \{\alpha \in \Lambda(V_C) \mid \alpha(e)\}),$$

where $prio$ is defined by Alg. 1, $\{\alpha \vdash \mathcal{I}_C\}$ is the set of initial states, and $\{\alpha \in \Lambda(V_C) \mid \alpha(e)\}$ is the set of (initial) error states. The synthesized set $Pri$ is then converted to the property-preserving priority service $P \overset{\text{def}}{=} (I_1 \cup I_2 \cup I_\varphi \cup L_1 \cup L_2 \cup L_\varphi, \mathcal{I}_C, \mathcal{T}_P, p)$ as introduced in Sec. III-C. $\mathcal{T}_P$ is defined by the successor function: $\mathrm{Succ}(\alpha) \overset{\text{def}}{=} \{\beta \mid \exists (\alpha, \beta, i) \in Pri\}$. The function $p$ is defined as follows: $p(\alpha, \beta) \overset{\text{def}}{=} \{i \mid (\alpha, \beta, i) \in Pri\}$.

In our running example, the algorithm processes the automaton in Fig. 5 and yields the automaton partially depicted in Fig. 6. As already mentioned, both figures depict the transitions going from the state `off/eOff` only. Transitions 2

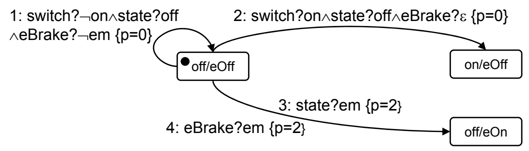

1: switch?¬on∧state?off ∧eBrake?¬em {p=0}
2: switch?on∧state?off∧eBrake?ε {p=0}
off/eOff
on/eOff
3: state?em {p=2}
4: eBrake?em {p=2}
off/eOn

Figure 6. Prioritization Service

and 3 in Fig. 5 are eliminated because the reaction to the same inputs is specified by unprioritized Transition 1 – this results in Transition 1 in Fig. 6. Transition 4 may prioritize service `Switch` or none ($p = 1 \vee p = 0$) – the algorithm keeps the unprioritized behavior only (Transition 2). Transition 7 is eliminated because it reaches the error state and the reaction to the same inputs is specified by Transitions 5 and 6 – this results in Transitions 3 and 4. Fig. 3(b) shows the complete priority service, which ensures satisfaction of the property depicted in Fig. 4 by services `Switch` and `EBrake`. For a better visual presentation all local states are merged to a single one since they have the same reaction to the same inputs. The same is done for equivalent transitions. This priority service is already explained in Sec. III-C.

Next, we prove the correctness of the presented method. By Prop. 2 we prove invariants (5) and (6) of Alg. 1 needed to show the correctness of our method (Prop. 3). We use the following notation: $\mathrm{Reach}(C, A)$ denotes the set of all states reachable in $C$ starting from some state subset $A$.

**Proposition 2** (Algorithm Correctness)**.** *Given a deterministic service $C$ with transitions marked by 0, 1, 2 and error states $E_C$. Alg. 1 called as $(Pri^n, Vis^n, Err^n) = prio(A, Vis, Err)$ is correct if the following assertions hold. (a) $\forall \alpha_1, \alpha_2 \in A : \alpha_1 \overset{L \cup O}{=} \alpha_2$ and (b) $\forall \alpha \in Err : \mathrm{Reach}(C, \{\alpha\}) \cap E_C \neq \emptyset$.*

*Proof Sketch: We show that Alg. 1 terminates with a correct result.*
*Termination* *The algorithm performs a depth-first search. Visited or erroneous states are not processed. Thus, the function prio always terminates for services over variables with finite domains.*
*Partial Correctness* *The following post-conditions hold after the termination of the algorithm. For all $\alpha \in A$*

$$\alpha \in Err^n \Leftrightarrow \forall p \in \{0,1,2\} : \mathrm{Reach}(C, \mathrm{Succ}_p(\alpha)) \cap E_C \neq \emptyset \tag{5}$$

$$(\alpha, \beta, p) \in Pri^n \Leftrightarrow p \in \{0,1,2\} \wedge \alpha, \beta \notin Err^n \wedge \beta \in \mathrm{Succ}_p(\alpha) \wedge p \neq 0 \Rightarrow \mathrm{Reach}(C, \mathrm{Succ}_0(\alpha)) \cap E_C \neq \emptyset \tag{6}$$

*Eqs. (5) and (6) can be shown by structural induction.* ■

**Proposition 3** (Synthesis Correctness)**.** *Given $S_1$, $S_2$, $\varphi$, and a priority service $P$ synthesized by Alg. 1, the following assertions hold. (a) $S_1 \parallel^P S_2 \models \varphi$ and (b) $P$ is the weakest property-preserving property, i.e., for any other priority $R$, which respects the prioritization order "0 before 1, 2", the following assertion holds: if $S_1 \parallel^R S_2 \models \varphi$ then $\langle\!\langle S_1 \parallel^R S_2 \rangle\!\rangle \subseteq \langle\!\langle S_1 \parallel^P S_2 \rangle\!\rangle$.*

*Proof: (a) According to Prop. 1 we need to show that Eq. (4) holds for $D = (S_1 \parallel^P S_2) \parallel \hat{S}_\varphi$. We observe that since $\langle\!\langle P \rangle\!\rangle \subseteq \langle\!\langle S_\top \rangle\!\rangle$, $\langle\!\langle D \rangle\!\rangle \subseteq \langle\!\langle (S_1 \parallel^{S_\top} S_2) \parallel \hat{S}_\varphi \rangle\!\rangle$ and, thus, Eq. (6) from Prop. 2 also holds for D. From Eq. (6) follows that $\langle\!\langle D \rangle\!\rangle$ contains only runs in which $S_1 \parallel^P S_2$ does not reach an error state.*

*(b) $S_\top$ is the super-set of all possible priority services, thus, $\langle\!\langle R \rangle\!\rangle \subseteq \langle\!\langle S_\top \rangle\!\rangle$ and also $\langle\!\langle S_1 \parallel^R S_2 \rangle\!\rangle \subseteq \langle\!\langle S_1 \parallel^{S_\top} S_2 \rangle\!\rangle$ for any R. Assume there is some R, which violates our second proof goal. Then, there must exist a run $\langle \alpha_1 \ldots \rangle \in \langle\!\langle S_1 \parallel^R S_2 \rangle\!\rangle \setminus \langle\!\langle S_1 \parallel^P S_2 \rangle\!\rangle$, which satisfies $\varphi$, and there must exist a valuation $\alpha_i$ in this run, such that $\alpha_i$ is reachable by $S_1 \parallel^P S_2$ but $\alpha_{i+1} \notin \mathrm{Succ}_{S_1 \parallel^P S_2}(\alpha_i)$ . If $\alpha_{i+1}$ was visited by Alg. 1, it must belong to the error set. Then, according to Eq. (5) there exists an input sequence starting at $\alpha_{i+1}$ which inevitably reaches an error state and we obtain a contradiction.*

*If $\alpha_{i+1}$ was not visited by Alg. 1, then according to (6) the priority of $(\alpha_i, \alpha_{i+1})$ is $> 0$ and $S_1 \parallel^P S_2$ has a transition starting from $\alpha_i$ with priority 0, i.e., either R disrespects the prioritization order or $\alpha_{i+1}$ is not reachable in $S_1 \parallel^R S_2$ either.* ■

## C. Determinization

Priorities control combined services based on the system inputs only. They reference neither outputs nor local variables of both services. Thus, every output or internal non-determinism is a behavior *not controllable* by the priority. These kinds of non-determinism must be eliminated. We replace every successor set containing more than one valuation without respect to inputs by exactly one successor. The powerset construction [12] is applied therefore. For a given service $S = (I \cup L \cup O, \mathcal{I}, \mathcal{T})$, a corresponding deterministic service $S_D = (V_D, \mathcal{I}_D, \mathcal{T}_D)$ is built as follows. $V_D \overset{\text{def}}{=} I \uplus \{l_D\}$ with $ty(l_D) \overset{\text{def}}{=} \mathcal{P}(\Lambda(L \cup O))$. The only local variable $l_D$ contains sets of local and output valuations of the original service. We define $subval$ to be a set, which contains all states $\gamma \in \Lambda(V)$ building one deterministic state $\alpha \in \Lambda(V_D)$:

$$\gamma \in subval(\alpha) \overset{\text{def}}{\Leftrightarrow} \gamma \overset{I}{=} \alpha \wedge \{\gamma\} \overset{L \cup O}{\subseteq} \alpha(l_D).$$

Thus, $subval$ associates every state of $S_D$ with the set of corresponding states of $S$. The successor state of $\alpha \in \Lambda(V_D)$ is computed by applying the successor function of $S$ on each state from $subval(\alpha)$. Formally, for all $\alpha, \beta \in \Lambda(V_D)$ we define

$$\alpha \vdash \mathcal{I}_D \overset{\text{def}}{\Leftrightarrow} \forall \gamma \in subval(\alpha) : \gamma \vdash \mathcal{I},$$

$$\alpha, \beta' \vdash \mathcal{T}_D \overset{\text{def}}{\Leftrightarrow} subval(\beta) = \{\delta \mid \gamma \in subval(\alpha) \wedge \gamma, \delta' \vdash \mathcal{T}\}.$$

The described procedure does not preserve the language of $S$. $S_D$ simulates $S$ but not vice versa. However, $\langle\langle S_D \rangle\rangle$ describes exactly the set of behaviors, which can be distinguished by our priority services since they are aware of inputs only. The same procedure can be applied for the reduction of internal non-determinism only if we redefine $V_D \overset{\text{def}}{=} I \uplus \{l_D\} \uplus O$, $ty(l_D) \overset{\text{def}}{=} \mathcal{P}(\Lambda(L))$, and $\gamma \in subval(\alpha) \overset{\text{def}}{\Leftrightarrow} \gamma \overset{I\cup O}{=} \alpha \wedge \{\gamma\} \overset{L}{\subseteq} \alpha(l_D)$.

## D. Deadlock-free Combination

The construction presented so far produces the priority service $P$ which guarantees the fulfillment of a given property $\varphi$ by a system $S_1 \|^P S_2$. However, in order to satisfy $\varphi$ $P$ may lead $S_1 \|^P S_2$ into a deadlock. Since deadlocks are undesired for reactive systems, we adapt our procedure for the synthesis of deadlock-avoiding priority services.

Not all deadlocks can be detected in the search space $(S_1 \|^{S_\top} S_2) \| \hat{S}_\varphi$ because during the determinization deadlocks may potentially be eliminated[3] and, consequently, become not observable during the priority synthesis. Thus, we search for deadlocks in the non-deterministic service combination and add them to the initial error set $Err$. A deadlock is a reachable local state without successor states: $\wedge_{\beta \in loc(\alpha)} \text{Succ}(\beta) = \emptyset$ with $loc(\alpha) \overset{\text{def}}{=} \{\beta \in \Lambda(V) \mid \beta \overset{L}{=} \alpha\}$. The initial error set is then defined as follows

$$\begin{aligned} Err \overset{\text{def}}{=} & \{\alpha \mid \alpha(e)\} \\ & \cup \{\alpha \mid \exists \beta \in subval(\alpha) : \forall \gamma \in loc(\beta) : \text{Succ}(\gamma) = \emptyset\}. \end{aligned}$$

Note, the computation of this set is not required in advance. The condition our algorithm relies on – whether some $\alpha$ belongs to $Err$ – can be checked individually for every $\alpha$ along with priority computation.

## E. Modular Prioritization

Modularity is an important issue since it allows to keep the problems manageable and to ensure scalability of the approach. In this section we deal with the satisfaction of a set of several properties. If a combination of two services $S_1$ and $S_2$ has to satisfy properties $\varphi$, $\psi$, ..., we can build priority services for every single one and combine them to a priority which ensures the satisfaction of the conjunction $\varphi \wedge \psi \wedge \ldots$

The combination of combinable input-complete priority services $P_1 = (V_1, \mathcal{I}_1, \mathcal{T}_1, p_1)$, $P_2 = (V_2, \mathcal{I}_2, \mathcal{T}_2, p_2)$ yields the priority service $P = P_1 \| P_2 \overset{\text{def}}{=} (V, \mathcal{I}, \mathcal{T}, p)$. The first three components of $P$'s tuple are defined as defined for regular services in Sec. III-B. The priority function $p$ is defined for all $\alpha, \beta \in \Lambda(V)$ as $p(\alpha, \beta) \overset{\text{def}}{\Leftrightarrow} p_1(\alpha, \beta) \cap p_2(\alpha, \beta)$. $P_1$ and $P_2$ have no outputs, thus, $P$ is input-complete also. However, only coinciding prioritization decisions of $P_1$ and $P_2$ are included into $p$. When $p_1$ and $p_2$ prioritize opposite services in some state $\alpha$, the behavior of a prioritized combination $S_1 \|^P S_2$ becomes undefined according to the definition in Sec. III-C. In other words, the system cannot react to $\alpha$. The following proposition shows that the above definition of priority combination is property preserving.

[3]Think of two states – a deadlock and a non-deadlock one – which are combined to one state during the powerset construction.

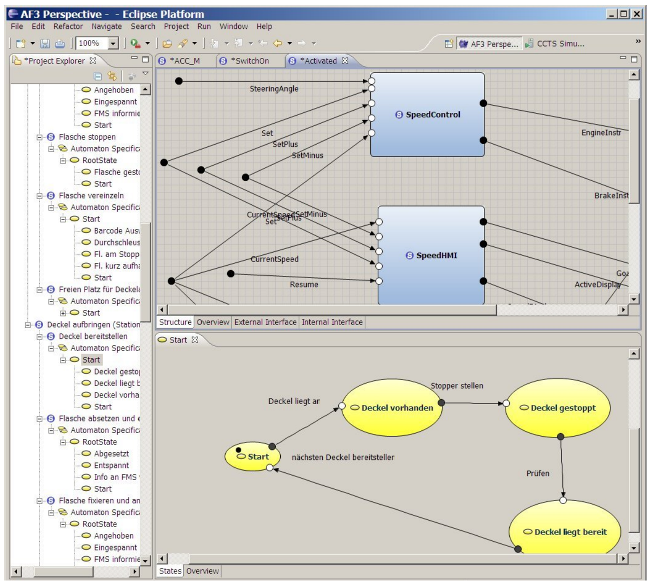


Figure 7. Services in AutoFocus

**Proposition 4.** *Given a pair of combinable services $S_1$, $S_2$, a pair of properties $\varphi$, $\psi$, and a pair of priority services $P_\varphi$, $P_\psi$, such that holds $S_1 \|^{P_\varphi} S_2 \models \varphi$ and $S_1 \|^{P_\psi} S_2 \models \psi$. Then, $S_1 \|^{P_\varphi \| P_\psi} S_2 \models \varphi \wedge \psi$ holds, too.*

*Proof Sketch: We define $S_{\varphi,\psi} = (V_{\varphi,\psi}, \mathcal{I}_{\varphi,\psi}, \mathcal{T}_{\varphi,\psi})$, the characterizing service of $\varphi \wedge \psi$, as $V_{\varphi,\psi} \overset{\text{def}}{=} V_\varphi \cup V_\psi$, $\mathcal{I}_{\varphi,\psi} \overset{\text{def}}{=} \mathcal{I}_\varphi \wedge \mathcal{I}_\psi$, $\mathcal{T}_{\varphi,\psi} \overset{\text{def}}{=} \mathcal{T}_\varphi \wedge \mathcal{T}_\psi$. $S_{\varphi,\psi}$ characterizes exactly all the runs which satisfy both properties: $\langle\langle S_{\varphi,\psi} \rangle\rangle = \langle\langle S_\varphi \rangle\rangle \cap \langle\langle S_\psi \rangle\rangle$. We define $C \overset{\text{def}}{=} (V_C, \mathcal{I}_C, \mathcal{T}_C) \overset{\text{def}}{=} S_1 \|^{P_\varphi \| P_\psi} S_2$, then we must show that $\langle\langle C \rangle\rangle \overset{V_{\varphi,\psi} \cap V_C}{\subseteq} \langle\langle S_{\varphi,\psi} \rangle\rangle$ or, equally. due to Prop. 1 that Eq. (4) holds for $C$ and $S_{\varphi,\psi}$. The conjunction of Prop. 1 for $\varphi$ and $\psi$ yields the desired property.* ∎

## V. Tool Support and Evaluation

We have integrated the service-based approach from Sec. III into an existing CASE tool. AutoFocus[4] is a tool for the component-based development of reactive systems. It supports graphical description of the developed system using different integrated diagram types.

We have extended this tool by a perspective dealing with service-based specifications. This perspective offers three views (cf. Fig. 7). In *Project Explorer* services are structured hierarchically. In *Service Structure Diagrams* syntactical interfaces are defined. *State Transition Diagrams* describe the

[4]http://af3.in.tum.de/

behavior of services using our I/O automata. In the adapted simulation and verification environments of AutoFocus service-based specifications can be validated and verified.

Our current work includes the implementation of Alg. 1 in the tool. Thereby, an existing NuSMV[5] back-end of the AutoFocus verification environment is applied for definition of the priority function $p$ and error propagation.

The presented approach is currently evaluated in an industrial project with Siemens, Sector Industry. 40 different scenarios of the control unit of the bottling plant informally specified on over 50 pages have been formalized in AutoFocus by approximately 30 modular and concise services (e.g., "filling bottles with items" or "singularizing bottles on the conveyor belt"). Also, 11 safety-critical properties involving the behavior of the whole system are identified (e.g., "at most one bottle in the filling station"). The first NuSMV models are manually created and priority services are synthesized.

## VI. Conclusion

We presented a mathematical framework to model scenario-based specifications of multifunctional reactive systems. In our approach, the system functionality is specified as a set of partial scenarios (services), which specify the overall system behavior from different viewpoints. The combination of these scenarios according to their priorities yields the overall system specification. We presented a novel synthesis method for the prioritization w.r.t. safety properties involving more than one scenario. Our synthesis algorithms automatically yield a set of priorities between scenarios such that their combined behavior fulfills all properties and remains conflict-free.

The proposed approach can be set in a broader context of model-based development. Thereby, we aim at a formal model-based development based on the same notion of function. With scenario-based specifications as an input, these specifications can be checked for consistency by verification and simulation [10], [13]. Subsequently, the specification is transformed into a component-based architecture [14]. Finally, the components are deployed onto a network of electronic control units [15].

We are currently working on the implementation of the presented approach in the CASE tool AutoFocus. A further promising research direction is the combination of our synthesis procedure with compositional verification techniques, which would allow the independent synthesis of distributed priority services.

*Acknowledgments:* We are grateful to Judith Thyssen, Leonid Kof, Christian Leuxner, and Philipp Neubeck for their advice on early versions of the paper.

[5]http://nusmv.irst.itc.it/

## References

[1] M. Broy, "Service-oriented systems engineering," in *Eng. Th. of SW Intensive Systems*. Springer, 2005.

[2] P. J. G. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 81–98, Jan. 1989.

[3] D. D'Souza and M. Gopinathan, "Conflict-tolerant features," in *CAV '08*, 2008, pp. 227–239.

[4] K. Altisen, G. Gößler, and J. Sifakis, "A methodology for the construction of scheduled systems," in *FTRTFT '00*. London, UK: Springer, 2000, pp. 106–120.

[5] S. Uchitel, G. Brunet, and M. Chechik, "Synthesis of partial behaviour models from properties and scenarios," *T-SE*, vol. 99, no. 1, 2009.

[6] M. Sabetzadeh, S. Nejati, S. Easterbrook, and M. Chechik, "A relationship-driven framework for model merging," in *MISE '07*. IEEE, 2007.

[7] C. Damas, B. Lambeau, and A. van Lamsweerde, "Scenarios, goals, and state machines: a win-win partnership for model synthesis," in *FSE'14*. ACM, 2006.

[8] R. Alur, K. Etessami, and M. Yannakakis, "Inference of message sequence charts," *IEEE Trans. Softw. Eng.*, vol. 29, no. 7, 2003.

[9] N. A. Lynch and M. R. Tuttle, "An introduction to i/o automata," *CWI-Quarterly*, vol. 2, no. 3, pp. 219–246, 1989.

[10] J. Botaschanjan, A. Harhurin, and L. Kof, "Service-based Specification of Reactive Systems," TU München, Technical Report TUM-I0815, 2008.

[11] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *LICS'86*. IEEE, 1986.

[12] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory*. Addison-Wesley, 1979.

[13] A. Harhurin and J. Hartmann, "Towards consistent specifications of product families," in *FM'08*, ser. LNCS, vol. 5014. Springer, 2008.

[14] J. Botaschanjan and A. Harhurin, "Integrating Functional and Architectural Views of Reactive Systems," in *CBSE'09*, ser. LNCS, vol. 5582. Springer, 2009.

[15] J. Botaschanjan, A. Gruler, A. Harhurin, L. Kof, M. Spichkova, and D. Trachtenherz, "Towards Modularized Verification of Distributed Time-Triggered Systems," in *FM'06*. Springer, 2006.