T U M

I N S T I T U T F Ü R I N F O R M A T I K

Service-Based Specification of Reactive Systems

Jewgenij Botaschanjan, Alexander Harhurin, Leonid Kof

TUM-I0815
Mai 08

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-05-I0815-0/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©2008

Druck: Institut für Informatik der
Technischen Universität München

Service-Based Specification of Reactive Systems

Jewgenij Botaschanjan Alexander Harhurin* Leonid Kof
Technische Universität München
Department of Informatics
Chair of Software and Systems Engineering
Boltzmannstr. 3, 85748 Garching, Germany
{botascha,harhurin,kof}@in.tum.de

ABSTRACT

Requirement conflicts are very common. They can result from different stakeholders' views, different knowledge of the operating environment, different interpretations of the same concept, etc. Requirement conflicts are harmless, as long as they are detected and resolved directly in the requirements engineering phase. Software development practice shows, however, that the conflicts often remain unperceived until implementation. In this case conflict resolution becomes much more expensive.

In the presented paper we suggest an approach to formalization of functional requirements, based on the service concept introduced by Broy [5]. This allows us to model every functional requirement as a partial function and to identify conflicting functions. This results in a formally well-founded conflict detection method, applicable in the requirements engineering phase. Applicability of the method was tested on an industrial case study.

1. INTRODUCTION

Software plays a dominant role in todays embedded reactive systems domain. The rapid increase in the amount and importance of different software-based functions and their extensive interaction are just some of the challenges that are faced during the development of reactive systems. One of the essential problems in this domain is a tailored requirements engineering process [10]. The more complex systems become, the more important it is to support validation and verification, already in the requirements engineering phase.

In order to assure consistency of a specification, a precise semantics of the modeling techniques is inevitable. Based on a formal foundation, discrepancies between conflicting functionalities can be detected and resolved already in the early phases of the development process. Furthermore, such a formal specification represents the first model in a model-based system development along different abstraction levels. It serves as a formal basis for the construction and verification of the models in the consecutive phases.

We introduce an approach to model a reactive system during the early phases of a model-based development process. Thereby, we focus on the formal definitions of functional requirements and relations between them, and show how the proposed formalism is used to handle the aforementioned functional intricacy. The proposed notation specifies a system as a set of black-box descriptions of the system behavior (called services). In other words, the system behavior is specified as a set of causal relations between input and output messages without any internal implementation details. Integration of different requirements might cause unforeseen conflicts (known as feature interaction) and consequently lead to an inconsistent specification of the overall behavior. Informally, there is a conflict between two services if they impose conflicting requirements on the behavior of a system which cannot be simultaneously fulfilled. The earlier these conflicts are detected, the lower the cost to resolve them. According to Boehm [1], resolution of a conflict is 10 times more expensive in the design phase that in the requirements engineering phase. Thus, in the ideal case conflicts should be identified and resolved in the requirements engineering phase. We precisely show how the introduced concepts can be used to detect and to resolve conflicts.

According to the development process introduced in our previous work [11], a system is modeled at different levels of abstraction in a way that each further level gives a more detailed view of the system. Starting from a very abstract description of the system (a set of informal goals or features), a formal specification is built up. This specification provides the basis for the construction of a component architecture. This allows us to involve verification into the development process, which is valuable for the development of safety-critical systems. The fact that according to [11] both specification and architecture models are based on the same notation facilitates this process. Thus, our specification model integrates seamlessly at the top of a model chain closing the formal gap between requirements and design.

The presented paper is a continuation of the work begun in [13], where a denotational semantics of our specification technique was introduced. In contrast to that work, the operational semantics proposed here allows automatic analysis of functional requirements. By this, discrepancies between conflicting functionalities can be detected and resolved automatically. Secondly, a formal specification can be simulated in a CASE tool, and verified, e.g. by model checking, which is valuable for industrial application.

*This work was partially funded by Siemens, Sector Industry in the framework of the Flasco project. The responsibility for this article lies with the authors.

1.1 Running Example

The concepts introduced in the remainder of the paper will be illustrated on a simplified industrial example of a bottling plant [9] originally specified and implemented for "Advanced Technologies and Standards" of Siemens Automation and Drives [23]. The considered system comprises several distributed and partly safety-critical subsystems to transport empty bottles from a storehouse to the bottling plant, fill bottles with different 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 a lot of functional requirements on the system, in this paper we consider a safety-critical subset concerning the interplay between the CU and the conveyor belt only. Among other things, the user can start and stop the conveyor. There is also an emergency brake available. By putting the emergency brake on, the CU immediately switches the conveyor off. The CU is not allowed to switch the system on and the emergency lamp flashes red until an abolition of the emergency is received.

In the bottling plant example we have potentially conflicting requirements, such as: (1) If the operator puts the switch in the "conveyor on" position, the controller turns the conveyor belt on, and (2) If the controller unit (CU) puts the emergency brake on, no action can turn on the conveyor belt until the emergency brake is released. These requirements are potentially conflicting because they affect the same actuator, namely the motor of the conveyor belt and can potentially issue contradictory commands at the same time.

1.2 Outline

The rest of this paper is organized as follows: In Section 2, the operational semantics of the proposed Service-Based Specification is presented. In particular, we explain the formal specification of functional requirements by means of services, and an operator to combine modularly specified requirements. In Section 3, we concentrate on the consistency of a service-based specification. To that end, we formally define conflicts and describe how they can be detected and resolved based on the introduced formal concepts. In Section 4, we show how our formal approach can be integrated into a CASE tool. In particular, we introduce a concrete tool semantics for services and their compositions and describe algorithms to detect inter-service conflicts based on this semantics. Finally, we compare our service model to related approaches in Section 5 before we conclude the paper in Section 6.

2. SERVICE-BASED SPECIFICATION

A specification consists of a set of requirements, which usually deal with a specific part of the system functionality only, e.g. different features, as well as normal-case behaviors and exceptional situations are handled by separate requirements. Black-box view is the most appropriate level of abstraction to specify individual functional requirements on a system under development. It allows the functional specification to avoid any unnecessary constraints on its realization.

These considerations suggest a specification paradigm called Service-Based Specification (SBS). This specification consists of a set of partial and non-deterministic black-box descriptions of the system behavior, called *services*. Services are composed in such a way that they can share inputs and outputs. However, the services do not communicate directly: every service obtains its inputs directly from and sends outputs directly to the environment. According to Broy [5], a service specifies a functional requirement by defining a relation between certain inputs and outputs of the system. Due to the fact that this relation has no constraints on the implementation details, a service specifies solely the black-box behavior of a functionality. Moreover, services describe system reactions for a certain subset of the inputs only. This partial description allows the distribution of the system description over different services (requirements) and/or leaving certain inputs unspecified. The SBS describes the system behavior from the environment perspective. Thus, an implementation satisfies a service specification if it shows the same I/O behavior as specified by the SBS.

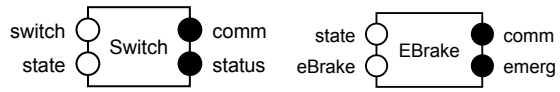
This section introduces an operational-style description of the proposed SBS. The operational semantics does not imply that design decisions are made in the specification phase. Rather, it is a means to describe, simulate, verify and validate a system specification. The corresponding black-box behavior can be extracted from this description in a schematic way [4]. Consequently, it can be realized by distinctively different design and implementation models.

The following section introduces a technique to specify a single functional requirement, a single service. Subsequently, we present an operator to combine modularly specified requirements, the service combination as well as its more general form, the prioritized service combination. As a result, we get a specification which consists of formally defined services and well-defined relationships between them.

2.1 Single Service

The formalism applied for the specification of the service behavior is an adapted form of state transition systems from [2, 4] and rule systems from [20].

A service has a *syntactic interface* consisting of the sets of typed input and output ports. Figure 1 depicts the syntactical interfaces of two services from our running example. There, the empty circles represent the input ports and the solid circles the output ports.



(a) `Switch` (b) `EBrake`

Figure 1: Syntactical Interfaces

The semantics of a service is described by a *service automaton*. This is a tuple

$$A = (V, \mathcal{I}, T) \tag{1}$$

consisting of variables V, initial states $\mathcal{I}$, and a transition relation T. V consists of mutually disjoint sets of typed variables, I, O, L. The variables from I and O are the input and output ports of the service interface, respectively. L is a set of local variables. A type of a variable describes a set of all its possible valuations. A *state* of A is a valuation α 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. state space of A is a subset of $\Lambda(V)$.

In the remainder of this work we use the following relations and predicates on variable valuations. Two valuations $\alpha, \beta \in \Lambda(V)$ are equal on a subset of common variables $Z \subseteq V$ if:

$$\alpha \overset{Z}{=} \beta \overset{\text{def}}{\Leftrightarrow} \forall v \in Z : \alpha(v) = \beta(v).$$

A set of valuations A is a subset of another set of valuations B with regard to a subset of common variables Z:

$$A \overset{Z}{\subset} B \overset{\text{def}}{\Leftrightarrow} \forall \alpha \in A : \exists \beta \in B : \alpha \overset{Z}{=} \beta$$
$$\wedge\, \exists \beta \in B : \nexists \alpha \in A : \alpha \overset{Z}{=} \beta.$$

The function *loc* denotes the set of all possible input and output variable valuations for a given local state:

$$loc(\alpha) \overset{\text{def}}{=} \{\beta \in \Lambda(V) \mid \beta \overset{L}{=} \alpha\}.$$

Finally, we define the priming operation on variable names: If v is a variable, then priming yields a new variable v' of the same type. The same applies for sets of unprimed variables, i.e. if $v \in V$, then $v' \in V'$. For a given valuation $\alpha \in \Lambda(V)$ we write $\alpha' \in \Lambda(V')$ in order to denote a new valuation such that $\forall v \in V : \alpha(v) = \alpha'(v')$. We use priming in order to be able to argue about the current state and the next state within the same logical formula.

In the following, we will describe the behavior of services by means of assertions. An assertion Φ is a logical formula, which may have free occurrences of variables from V. Then, a valuation α satisfies an assertion Φ, denoted by $\alpha \vdash \Phi$, if and only if the assertion yields true after replacing the free variables of Φ by the corresponding values from α. The other type of assertions is defined over the variables from $V \cup V'$. We write $\alpha, \beta' \vdash \Phi$ in order to denote that a pair of valuations $\alpha \in \Lambda(V)$ and $\beta' \in \Lambda(V')$ satisfies Φ. We also write $\alpha \vdash \Phi$ for $\alpha \in \Lambda(V \cup V')$.

$\mathcal{I}$ from (1) is an assertion characterizing the *initial states* of the system. It must be satisfiable by at least one valuation of V and it is allowed to constrain output and local variables only, i.e. the set of possible initial inputs is not constrained by $\mathcal{I}$.

T from (1) is a set of *transitions*. A transition $t \in T$ relates states to their respective successor states. Formally, a transition is an assertion over $V \cup V'$, where 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 separation between the local state and the outputs.

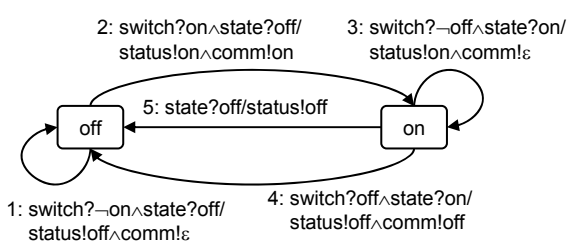


Figure 2: I/O Automaton of `Switch`

Figures 1(a) and 2 show the specification of service `Switch` from our running example. In Figure 2, we use a special syntax to describe transition assertions: A transition goes from one control state to another. A control state is a value of a dedicated local variable, which encodes the control state. The other local variables, which encode the data state of a service can be explicitly manipulated by the transition assertion. A transition consists of four parts: precondition, input patterns, post-condition and output patterns. A precondition contains conditions on the input and on the current values of the local variables. Input and output patterns describe the values of the expected inputs (unprimed variables) and produced outputs (primed variables), respectively. In our concrete syntax `i?v` denotes an input pattern, which evaluates to true if the variable $i \in I$ has the value v and `o!v` an output pattern, which is satisfied by an assignment of value v to the output variable $o' \in O'$. A special form of the input pattern `i?¬v` is satisfied by all type-correct valuations of i, except for v. The individual input patterns are separated by the logical AND and followed by an AND-separated output pattern list with a slash in between. The post-condition is an assignment of values to the variables in L'. In our simplified example the local state of service automata consists of the control state only ($L = \{cs\}$). Thus, e.g. , the pre- and post-conditions of Transition 2 from Figure 2 are $cs = \mathit{off}$ and $cs' = on$, respectively. Since the service automata from our running example do not contain further local variables, pre- and post-conditions are omitted in the figures. A transition can be fired if for the given inputs and the current local state the precondition and the input pattern yield true. The result of a transition step is a valuation that satisfies the output pattern and the post-condition.

The service in Figure 2 formalizes the following requirement on 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 `command` to the conveyor, as well as message `on` through port `status` to the user display (cp. Transition 2 in the automaton). The remaining transitions are defined analogously. Note, Transition 5 does not reference two of the four existing ports. This means that the outputs of these ports are not subject to any restrictions by the service automaton and thus allowed to have arbitrary values within their respective type domains. In other words, if the CU receives state message `on` from the conveyor, then it sends message `on` to the display, independently from the messages on the other ports.

In order to be able to reason about transition steps, we define a set of *I-enabled* transitions originating from a given state α. An *I-enabled* transition can process the provided input in the current local state:

$$IEn(\alpha) \overset{\text{def}}{=} \{t \in T \mid \exists \beta \in \Lambda(V) : \alpha, \beta' \vdash t\}.$$

For reasoning about valuation sequences, we define for every valuation the set of its successors. Formally,

$$\mathrm{Succ}(\alpha) \overset{\text{def}}{=} \{\beta \mid \exists t \in T : \alpha, \beta' \vdash t\}.$$

The valuations reachable after n steps starting from some valuation α are then recursively defined as

$$\mathrm{Succ}^0(\alpha) \overset{\text{def}}{=} \{\alpha\} \quad \text{and} \quad \mathrm{Succ}^{n+1}(\alpha) \overset{\text{def}}{=} \mathrm{Succ}(\mathrm{Succ}^n(\alpha))$$
$$\text{with} \quad \mathrm{Succ}(A) \overset{\text{def}}{=} \bigcup_{\gamma \in A} \mathrm{Succ}(\gamma).$$

Finally, the set of all reachable states of a service S is defined as

$$\mathrm{Attr}(S) \stackrel{\text{def}}{=} \bigcup_{n\in\mathbb{N},\alpha\vdash\mathcal{I}} \mathrm{Succ}^n(\alpha).$$

A *run* of a service automaton is an infinite sequence of states, $\langle\beta_0\beta_1\beta_2\ldots\rangle$, where $\forall i \in \mathbb{N} : \beta_{i+1} \in \mathrm{Succ}(\beta_i)$ and $\beta_0 \vdash \mathcal{I}$. The semantics of a service automaton $\langle\langle\mathcal{S}\rangle\rangle$ is the set of all its runs.

A service is called *valid* if there exists at least one successor for each reachable local state:

$$\forall\alpha \in \mathrm{Attr}(S) : \mathrm{Succ}(loc(\alpha)) \neq \emptyset.$$

Otherwise, the service is inconsistent. It is quite easy to see that the set of runs (consisting of infinite runs only) of a valid service is not empty.

2.2 Service Composition

Individual services that have been specified independently can be combined to a composite service. This directly reflects the idea that all single functional requirements on a system must be satisfied by a valid implementation simultaneously. Thus, we define the semantics of a composite service as being a container of all *simultaneously* operating sub-services. Thereby, services are allowed to share input and output ports.

Unlike the "classical" notion of the composition, which reduces the number of possible behaviors of individual automata [8, 17], we are interested in obtaining a mechanism for the *extension* of the system functionality. The service composition should accept all inputs, the individual services can deal with as long as the outputs produced by these services are unifiable (not contradictory). Thereby, the reaction of the composition should accord with reactions specified by the single services. In other words, our service composition accepts the union of the inputs and produces the intersection of the outputs.

We define the composition of two service only if their I/O ports and local variables are mutually disjoint: $(I_1\cup L_1)\cap(O_2\cup L_2) = (O_1\cup L_1)\cap(I_2\cup L_2) = \emptyset$ and their homonymous variables $(V_1 \cap V_2)$ have the same type. We speak about *composable* services if they fulfill these properties.

If two services S_1 and S_2 are composable, their composition $C \stackrel{\text{def}}{=} S_1||S_2$ is defined by $C \stackrel{\text{def}}{=} (V_C, \mathcal{I}_C, T_C)$, where

- $I_C \stackrel{\text{def}}{=} I_1 \cup I_2$, $O_C \stackrel{\text{def}}{=} O_1 \cup O_2$, $L_C \stackrel{\text{def}}{=} L_1 \cup L_2$,
- $V_C \stackrel{\text{def}}{=} I_C \cup L_C \cup O_C$,
- $\mathcal{I}_C \stackrel{\text{def}}{=} \mathcal{I}_1 \wedge \mathcal{I}_2$,
- T_C is defined via the successor relation, see Equation (2) below.

The composition 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. This means, the other service is not I-enabled. In the latter case, the local variables of the not I-enabled service are not modified, and its output variables (not common with the first service) are not subject to any restrictions. Formally, the set of successors in valuation sequences of a service composition is defined as follows:

$$\begin{aligned}\mathrm{Succ}(\alpha) \stackrel{\text{def}}{=} &\{\beta \mid \exists t_1 \in T_1, t_2 \in T_2 : \alpha, \beta' \vdash t_1 \wedge t_2\}\\ &\cup \{\beta \mid \exists t_1 \in T_1 : \alpha, \beta' \vdash t_1 \wedge IEn_2(\alpha) = \emptyset \wedge \alpha \stackrel{L_2}{=} \beta\}\\ &\cup \{\beta \mid \exists t_2 \in T_2 : \alpha, \beta' \vdash t_2 \wedge IEn_1(\alpha) = \emptyset \wedge \alpha \stackrel{L_1}{=} \beta\}.\end{aligned} \tag{2}$$

A valuation β is a successor of the valuation α if there exist both a transition t_1 of service S_1 and a transition t_2 of S_2 such that they are satisfied by α and β' simultaneously. Otherwise, β is a successor of α if there is no I-enabled transition t_2 of S_2 in α and there is a transition t_1 of S_1 such that it is satisfied by α and β'. Thereby, all local variables of S_1 must remain unaltered. The analogous behavior is exhibited by the transitions of S_2 when S_1 is not I-enabled.

To illustrate the concept of composition, we consider a further requirement concerning the emergency brake from our running example. The CU immediately 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 red until an abolition of the emergency (`ab`) is received on port `eBrake`. The service shown in Figures 1(b) and 3 formalizes this requirement. In the normal mode, as long as no emergency signal is received, the emergency lamp is off and port `switch` may have an arbitrary type-correct valuation, i.e. , any message of the port type is allowed (Transition 1). Once the emergency signal is received, the command on port `comm` will be `off` and the signal on port `emerg` to the emergency lamp will be `on` (Transitions 2 or 3). Then, no further commands are allowed and the signal to the lamp is `on` as long as no abolition message is received on port `switch` (Transition 4). After receiving an abolition, message `off` is sent to the lamp, whereas, port `switch` is not subject to any restrictions (Transition 5).

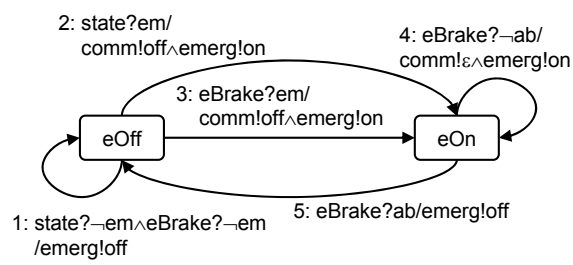


Figure 3: I/O Automaton of `EBrake`

The composition of services `Switch` and `EBrake` results in the automaton depicted in Figure 4. There, the labels of transitions are of the form $t_s \wedge t_e$, where t_s is the identity number of a transition from Figure 2, and t_e is a transition number from Figure 3. A label of the form $T \wedge t_e$ identifies a situation where service `Switch` is not I-enabled. A transition with a label $l_1 \vee l_2$ is an abbreviation of two transitions with labels l_1 and l_2, respectively. According to the definition, contradictory transition pairs do not belong to the transition set of the composition. For example, Transition 1 of `Switch` and Transition 3 of `EBrake` are contradicting because of contradictory messages on output port `comm`.

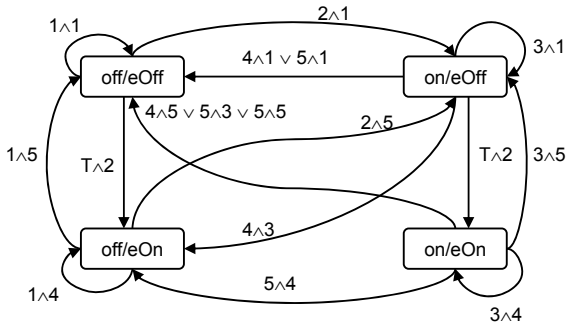


Figure 4: Service Composition

The composition defined above is well-defined in the sense that for composable services it yields a service again. It is also commutative, associative and idempotent, which is shown next.

PROPOSITION 1. *The service composition is commutative, associative and idempotent.*

PROOF SKETCH. *The commutativity is a straight forward issue: the definition of composition is symmetrical for both arguments.*

We will sketch the associativity in one direction, i.e. that $\langle\langle (S_1 \parallel S_2) \parallel S_3 \rangle\rangle \subseteq \langle\langle S_1 \parallel (S_2 \parallel S_3) \rangle\rangle$ *for some pairwise composable* S_1, S_2 *and* S_3. *The proof for the opposite direction goes analogously. For this case it is sufficient to show that for state* α *holds* $\mathrm{Succ}_{(S_1 \parallel S_2) \parallel S_3}(\alpha) \subseteq \mathrm{Succ}_{S_1 \parallel (S_2 \parallel S_3)}(\alpha)$ *for every valuation. Let us assume that there exists* β *with*

$$\beta \in \mathrm{Succ}_{(S_1 \parallel S_2) \parallel S_3}(\alpha) \quad \textit{and} \quad \beta \notin \mathrm{Succ}_{S_1 \parallel (S_2 \parallel S_3)}(\alpha)$$

According to Definition (2)*, the successor relation builds upon three subsets. Because* $\wedge$ *is associative, we obtain a contradiction if* β *belongs to the first one (i.e. for all* $i \in \{1, 2, 3\}$ *there are transitions* $t_i \in T_i$ *such that* $\alpha, \beta' \vdash t_1 \wedge t_2 \wedge t_3$*). Since the remaining two cases are symmetrical, we will exemplary consider the case where* $IEn_3 = \emptyset$*. In this case there exist* $t_i \in T_i$ *for* $i \in \{1, 2\}$ *such that* $\alpha, \beta' \vdash t_1 \wedge t_2$*. Due to the last subset construction of* (2)*, we have* $\beta \in \mathrm{Succ}_{S_2 \parallel S_3}(\alpha)$ *and obtain a contradiction also in this case.*

Finally, for $S \parallel S$ *only the first subset in the definition of composition will be non-empty. The idempotence then follows from the idempotence of* $\wedge$. $\square$

2.3 Prioritized Composition

Usually, in specifications some events or behaviors explicitly have a higher priority than others. For example, the system reaction in the case of emergency has higher priority than the normal-case behavior. Another construction commonly found in specifications is the presence of operational modes, in which the behavior of the system changes. In order to be able to reflect these circumstances in our service model, we introduce the notion of a *prioritized composition*. It allows an individual service to take control over other services depending on certain input situations. By this, we can express different relationships between services, as for example service relations from [6], without any modifications on them. This means, for example, that a service does not have to be aware that it can be "deactivated" by some other service. Thus, the prioritized composition preserves the modularity of service specifications.

The prioritized composition allows service S_2 to take priority over service S_1 in the composite system. Thereby, the prioritized composition is controlled by a special service S_P with the interface containing all input ports $I_1 \cup I_2$ and no output ports. If the current input enables a transition of S_P, only service S_2 is *efficacious* – the local state of S_1 remains unmodified, the output variables controlled by S_1 are not subject to any restrictions. Otherwise, the composition behaves like the un-prioritized one. Thus, the priority service S_P determines certain input states for which the behavior of the system should coincide with the behavior of S_2 only.

The prioritized composition $PC \stackrel{\text{def}}{=} S_1 \parallel^{S_P} S_2$ is defined for a pair of composable services S_1 and S_2 and a special prioritization service $S_P \stackrel{\text{def}}{=} (I_P \uplus L_P, \mathcal{I}_P, T_P)$ by $PC \stackrel{\text{def}}{=} (V_{PC}, \mathcal{I}_{PC}, T_{PC})$, where

- $(I_1 \cup I_2) \subseteq I_P$,
- $I_{PC} \stackrel{\text{def}}{=} I_P$, $O_{PC} \stackrel{\text{def}}{=} O_1 \cup O_2$, $L_{PC} \stackrel{\text{def}}{=} L_1 \cup L_2 \cup L_P$,
- $V_{PC} \stackrel{\text{def}}{=} I_{PC} \cup L_{PC} \cup O_{PC}$,
- $\mathcal{I}_{PC} \stackrel{\text{def}}{=} \mathcal{I}_P \wedge \mathcal{I}_1 \wedge \mathcal{I}_2$,
- T_{PC} is defined via the successor relation, see Equation (2) below.

Let T_C be the transition set of the un-prioritized composition of S_1 and S_2, i.e. $S_1 \parallel S_2 = (V_C, \mathcal{I}_C, T_C)$, then the prioritized one contains the following valuation states

$$\begin{aligned}\mathrm{Succ}(\alpha) \stackrel{\text{def}}{=} & \\ & \{\beta \mid \exists t \in T_C : \forall t_P \in T_P : (\alpha, \beta' \vdash t \wedge \neg t_P) \wedge \alpha \stackrel{L_P}{=} \beta\} \\ & \cup \{\beta \mid \exists t_2 \in T_2, t_P \in T_P : (\alpha, \beta' \vdash t_2 \wedge t_P) \wedge \alpha \stackrel{L_1}{=} \beta\}.\end{aligned}$$

The first subset describes the case when S_P is inefficacious (not I-enabled). Then, the behavior of $S_1 \parallel^{S_P} S_2$ coincides with the behavior of $S_1 \parallel S_2$. The other subset contains the common behaviors of S_2 and S_P, i.e. the reactions of S_2 to the inputs enabled by S_P.

In our running example, it makes sense to prioritize the emergency break signals `eBrake?em` and `state?em`. Additionally, we require that the composite system must behave like service `EBrake` if one of these signals arrives, otherwise, the behavior is identical to the composition from Figure 4. The priority service which prioritizes emergency signals is depicted in Figure 5. The priori-

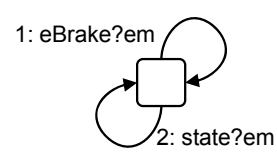


Figure 5: Priority Service

tized composition of `Switch` and `EBrake` with regard to the priority service results in the automaton from Figure 6. Whenever an emergency signal has arrived, this composition behaves like service

EBrake (transitions of the form $p2 \wedge t_e$), otherwise it behaves like the composition from Figure 4.

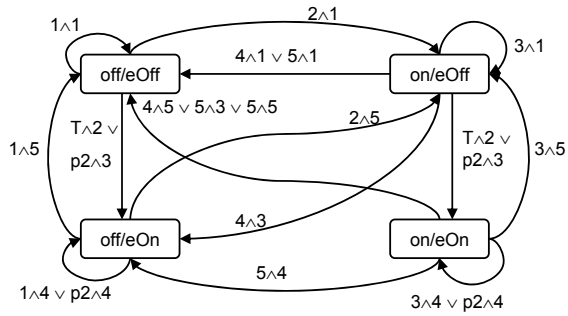


Figure 6: Prioritized Composition

The un-prioritized composition from the previous section is a special case of the prioritized one, namely $S_1 \parallel S_2 \stackrel{\text{def}}{=} S_1 \parallel^{S_\perp} S_2$, where $S_\perp$ is the service yielding "false" for any pair of valuation states (α, β'). More precisely, we define $S_\perp \stackrel{\text{def}}{=} (I_1 \cup I_2, \mathcal{I}_1 \wedge \mathcal{I}_2, \emptyset)$.

The prioritized composition is in general non-commutative and non-associative. However, we can generalize the associativity result from the previous section for the prioritization by the same priority service: $(S_1 \parallel^{S_P} S_2) \parallel^{S_P} S_3 = S_1 \parallel^{S_P} (S_2 \parallel^{S_P} S_3)$. We also obtain the distributivity of the prioritized composition by the following proposition.

PROPOSITION 2. *The prioritized composition is distributive.*

PROOF SKETCH. *First, we observe that the overall behavior of a transition set T can be described by a logical formula $[T] \stackrel{\text{def}}{=} \bigvee_{t \in T} t$. Moreover, we write $[T_{S_1 \parallel S_2}] = [T_{S_1}] \circledast [T_{S_2}]$, where $\circledast$ denotes the Boolean operation which corresponds to the un-prioritized composition. Due to Proposition 1 it is commutative, associative and idempotent. Then, the behavior of the prioritized composition $S_1 \parallel^{S_P} S_2$ denoted by $[T_{S_1 \parallel^{S_P} S_2}]$ reduces to*

$$[T_{S_1 \parallel^{S_P} S_2}] = ([T_P] \Rightarrow [T_2]) \wedge (\neg[T_P] \Rightarrow [T_{S_1}] \circledast [T_{S_2}]).$$

In order to prove that for pairwise composable S_1, S_2 and S_3 holds

$$S_1 \parallel^{S_P} (S_2 \parallel^{S_Q} S_3) = (S_1 \parallel^{S_P} S_2) \parallel^{S_Q} (S_1 \parallel^{S_P} S_3)$$

we must consider four cases: $([T_{S_P}] \wedge [T_{S_Q}])$,...,$(\neg[T_{S_P}] \wedge \neg[T_{S_Q}])$ and show for each of them that the behaviors of the both formulas are the same. We will do this exemplary for $\neg[T_{S_P}] \wedge \neg[T_{S_Q}]$: For the left-side formula we obtain $[T_{S_1}] \circledast [T_{S_2 \parallel S_3}]$ and for the formula on the right side $[T_{S_1 \parallel S_2}] \circledast [T_{S_1 \parallel S_3}]$ which are equal. □

The prioritized and un-prioritized composition operators with their properties enable the modular and distributed development of service models and facilitate reuse of service specifications.

3. CONSISTENT SPECIFICATION

The overall specification is the combination of modularly specified sub-functionalities. Thereby, different services can share the same I/O ports. Thus, the integration of different functions might cause unforeseen conflicts (known as *feature interaction*) and consequently lead to an inconsistent specification of the overall behavior. As a consequence, it becomes a central task during the function integration to detect and resolve conflicts in order to assure the consistency of the overall specification. In the following sections, we precisely define what we mean by *conflicts* and show how the introduced formal concepts can be used to detect and to resolve conflicts between functional requirements.

3.1 Conflict Detection

Intuitively, a conflict between two services exists when a service in the presence of other services is prevented from reacting to all inputs which can be suitably processed by the service in isolation. While the behavior of both services may be correct according to their intended behaviors, their interaction is undesired. However, this situation cannot be automatically assessed as a conflict because this restriction may be developer's purpose. Thus, *potential* conflicts need a further analysis by the developer. A prerequisite for a service conflict is that the two services have at least one common output port.

Formally, for a composition S, which combines a service S_1 with some other services, we gather all states in which the set of inputs accepted by S_1 is further restricted in the composition. This set of potentially conflicting states is defined as $\mathit{Confl}(S_1, S) \subseteq \Lambda(V)$ with

$$\alpha \in \mathit{Confl}(S_1, S) \stackrel{\text{def}}{\Leftrightarrow} \alpha \in \mathrm{Attr}(S) \wedge \mathrm{Succ}(\alpha) \stackrel{I_1}{\subset} \mathrm{Succ}_1(\alpha) \quad (3)$$

$\mathrm{Succ}(\alpha)$ is the set of successors of α in the composition and $\mathrm{Succ}_1(\alpha)$ is the successors of a projection of α in service S_1.

Then, two services are potentially conflicting within an un-prioritized composition in a state α, iff

$$\alpha \in \big(\mathit{Confl}(S_1, S_1 \parallel S_2) \cup \mathit{Confl}(S_2, S_1 \parallel S_2)\big).$$

Regarding potential conflicts in a prioritized composition where S_2 takes priority over S_1 with regard to S_P, a conflict might appear only if both services are efficacious, i.e. only if the current input cannot be processed by the priority service. This is the case when $\mathrm{Succ}_P(\alpha) = \emptyset$, according to the definition from Section 2.3. Otherwise, the behavior of the system coincides with the behavior of the prioritized service and thus, per definition, cannot cause a conflict. If both services are efficacious simultaneously, we use the same conflict definition as that of the un-prioritized composition. Formally, α is a potentially conflicting state in the composition $S_1 \parallel^{S_P} S_2$, iff

$$\alpha \in \mathit{Confl}(S_1, S_1 \parallel S_2) \cup \mathit{Confl}(S_2, S_1 \parallel S_2) \wedge \mathrm{Succ}_P(\alpha) = \emptyset.$$

The set of potentially conflicting states of two services within a composition is defined as:

$$\begin{aligned} \mathcal{C}_{\mathcal{P}}(S_1, S_2, S_P) \stackrel{\text{def}}{=} \{\alpha \mid & \\ & \alpha \in \mathit{Confl}(S_1, S_1 \parallel S_2) \cup \mathit{Confl}(S_2, S_1 \parallel S_2) \\ & \wedge \mathrm{Succ}_P(\alpha) = \emptyset\}. \end{aligned}$$

Please note, in the special case of an un-prioritized composition $\mathrm{Succ}_P(\alpha) = \emptyset$ is true by definition, i.e. $\mathcal{C}_{\mathcal{P}}(S_1, S_2, S_P) \subseteq \mathcal{C}_{\mathcal{P}}(S_1, S_2, S_\perp)$.

In our example, it is obvious that both services Switch and EBrake are potentially conflicting in the unprioritized composition. According to service EBrake, after an emergency switch

Service	Variable	Value				
$Switch$	$switch$	$\neg on$	on	$\neg off$	on	off
	$state$	off	off	on	off	on
	l_1	off	off	on	off	on
$EBrake$	$eBrake$	em	em	em	$\neg ab$	$\neg ab$
	l_2	$eOff$	$eOff$	$eOff$	eOn	eOn
Conflict Nr.		1	2	3	4	5

Table 1: Conflicting States

off, the CU is not allowed to switch the system on until an emergency abolition message is received. At the same time, according to service `Switch`, if the user switches the conveyor on, in the next step the CU has to switch the system on. Formally, let us consider a state α with $l_1 = off \wedge l_2 = eOn \wedge switch = on \wedge state = off \wedge eBrake \neq ab$. The output values are arbitrary. While there is no successor for this state in the composition, service `Switch` contains a successor of this state because of Transition 2 (see Figures 4 and 2). Because of $\mathrm{Succ}(\alpha) = \emptyset \overset{I_1}{\subset} \mathrm{Succ}_{\mathrm{Switch}}(\alpha) \neq \emptyset$, there is a potential conflict between both services. The set of all conflicting states $\mathcal{C}_{\mathcal{P}}(Switch, EBrake, S_\perp)$ is given by Table 1. The columns which specify values of input and local variables identify five conflicting states.

A special class of potential conflicts are *definite* conflicts. Services S_1 and S_2 are definitely conflicting if their composition does not yield a valid service, i.e. there is a deadlock in the composition. In other words, there is a reachable local state α in the composite automaton for which no successor state exists: $\mathrm{Succ}(loc(\alpha)) = \emptyset$. Obviously, every definite conflict satisfies Definition (3) for potential conflicts. Thus, every definite conflict is also a potential one.

3.2 Conflict Resolution

In order to obtain a consistent specification, all detected conflicts have to be resolved. It is possible to identify three sources of conflicts, namely (1) user requirements on the considered system are contradictory, (2) the formalizations of unifiable requirements are more restrictive than necessary, and (3) a desired priority between two requirements is omitted. Even though it is up to the developer to judge whether a potential conflict must be resolved or not, we propose two methods to resolve conflicts from classes (2) and (3). The first class of conflicts can be resolved by reviewing the affected requirements by the developer only, and therefore will not be further treated here. The second class can be resolved by abstracting the modular specification of at least one of the affected services. The third class can be resolved by introducing an additional priority between affected services.

3.2.1 Abstracting Modular Specifications

In many cases, conflicts can be resolved by abstracting modular specifications without changing the original meaning of requirements. Thereby, we are guided by the fact that a textual requirement is the most abstract specification and the corresponding service is one of its refinements. A common source of inter-service conflicts is the specification of the reaction time of the system to a given input. A textual requirement "the system should react to the input i with the output o" ($i \rightarrow \mathrm{F}\, o$) is often interpreted as "the system should react to the input i with the output o in the next step" ($i \rightarrow \mathrm{X}\, o$)[1]. This refinement of the textual requirement is more restrictive than necessary and thus may cause a conflict with other services.

[1]We use the following temporal logic operators: X for next and F for eventually.

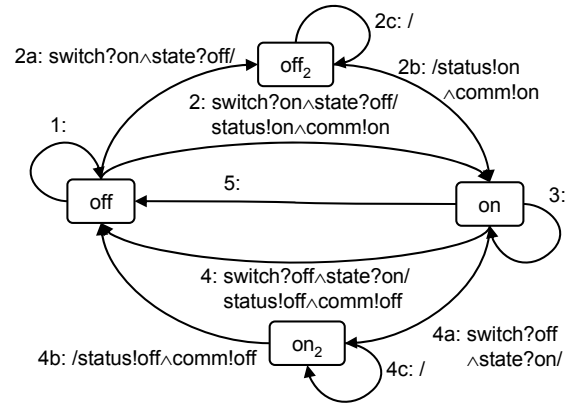


Figure 7: Abstraction of Service `Switch`.

Note, the introduced nondeterminism may cause new conflicts and consequently entails a repeated search and review of potential conflicts.

In our example, service `Switch` always specifies the system reaction to the user input in the next step (see Figure 2). Let us assume that the conveyor does not have to be switched on immediately after receiving the user signal ($switch?on \wedge state?off \rightarrow \mathrm{F}(status!on \wedge comm!on)$). Then, service `Switch` is a correct but too restrictive specification of this requirement. To resolve conflicts detected in the last section, additionally to Transition 2 we add three new transitions and a new local state off_2 (see Figure 7). Transition 2a is I-enabled whenever Transition 2 is I-enabled. However, it constrains no output variables. 2c is a loop without any restrictions on the I/O variables. 2b is always I-enabled and has the same restrictions on the output variables as Transition 2. The modified service now specifies the following requirement: $switch?on \wedge state?off \rightarrow \mathrm{F}(status!on \wedge comm!on)$. The same method is used to resolve conflicts involving Transition 4. By this, we have resolved conflicts 2, 4 and 5 from Table 1.

3.2.2 Additional Priorities

For most conflicts the procedure introduced so far is not adequate since changing the modular specification accordingly to the behavior of another services implies loss of modularity. Therefore, to resolve the source of conflicts (namely, the service interaction) we propose to introduce additional priorities. By this, we preserve the modularity of services and, furthermore, make functional dependencies explicit.

In order to resolve a subset of potential conflicts $RC \subseteq \mathcal{C}_{\mathcal{P}}(S_1, S_2, S_\perp)$ between two services S_1 and S_2 in favor of S_2, we synthesize a new priority service. In every conflicting state, this service prioritizes service S_2 and so resolves the detected potential conflicts. Formally, the priority service $S_{\mathcal{C}}(RC) \overset{\text{def}}{=} (V_{\mathcal{C}}, \mathcal{I}_{\mathcal{C}}, T_{\mathcal{C}})$ is defined as:

$$V_{\mathcal{C}} = I_{\mathcal{C}} \overset{\text{def}}{=} I_1 \cup I_2, \quad \mathcal{I}_{\mathcal{C}} \overset{\text{def}}{=} True \quad \text{and}$$

$$T_{\mathcal{C}} \overset{\text{def}}{=} \{(\gamma, \delta') \in \Lambda(I) \times \Lambda(I') \mid \alpha \in RC \wedge \gamma \overset{I_2}{=} \alpha\}$$

Transitions of the new service $T_{\mathcal{C}}$ are I-enabled exactly in the conflicting states of the unprioritized composition. Then, in $S_1 \parallel^{S_{\mathcal{C}}(RC)} S_2$ the set of potential conflicts is provably reduced by RC, which is shown next.

PROPOSITION 3. *For any services S_1, S_2 and a potential conflict state $\alpha \in \mathcal{C}_{\mathcal{P}}(S_1, S_2, S_\perp)$ holds: $\alpha \notin \mathcal{C}_{\mathcal{P}}(S_1, S_2, S_{\mathcal{C}}(\{\alpha\}))$.*

PROOF. *First, we observe that the domains of $S_1 \parallel S_2$ and $S_1 \parallel^{S_{\mathcal{C}}(\{\alpha\})} S_2$ are the same, i.e. $\alpha \in \Lambda(V_{S_1 \parallel^{S_{\mathcal{C}}(\{\alpha\})} S_2})$. A transition of $S_{\mathcal{C}}(\{\alpha\})$ is always enabled in α, thus, according to the definition of the parametrized composition, S_2 will always take the control over in α and by this $\mathrm{Succ}(\alpha) = \mathrm{Succ}_2(\alpha)$. Here, Succ is the successor function of $S_1 \parallel^{S_{\mathcal{C}}(\{\alpha\})} S_2$ and Succ_2 of S_2.* □

4. IMPLEMENTATION

Concepts introduced so far build a theoretical foundation for service-based specification. Below, we sketch how these concepts can be integrated into an existing CASE tool. The logical characterization of the operational service semantics permits on the one side validation of our models by executing them and on the other their verification and automatic inter-service conflict detection by techniques like model checking, SAT- and constraint solving, etc. For all these purposes a description technique and a tool-support are needed. To our knowledge, there is no tool-support for the formal specification of functional requirements and automatic detection of conflicts between them. Therefore, we aim at integration of our approach into the CASE tool AutoFOCUS [3].

AutoFOCUS is a tool for the component-based development of reactive systems. It supports graphical description of the developed system using different integrated diagram types (views). Here, we describe only two, for our approach relevant views. The description techniques applied there are similar to the diagrams used to describe our bottling plant example.

Architecture view: By using System Structure Diagrams (SSDs) users can define the component structure of the system. In particular, component interfaces consisting of I/O ports can be defined.

Behavior view: State Transition Diagrams (STDs) describe the behavior of a component in the system using an extended version of I/O automata.

The current purpose of AutoFOCUS is the development of the design (with the possibility of a subsequent code generation) of reactive systems. Its formally founded operational semantics allowed to realize a simulation environment and to integrate different verification tools, e.g. model checkers, constraint solvers, etc. We are currently extending this tool by a more abstract view dealing with the specification of functional requirements.

In Section 4.1, we introduce a concrete (and more concise) AutoFOCUS based semantics for services and their compositions, which allows to use STDs to describe the behavior of single services (cp. [16] for a detailed description of the STD semantics). In Section 4.2, based on this operational semantics and the definitions from Section 3, we describe algorithms to detect conflicts between services.

4.1 Simulation

A service is specified by an automaton $A = (V, \mathcal{I}, T)$, introduced in Section 2. A missing part, not discussed so far is a concrete scheme for the description of service transitions. In fact, the definition of the composition based on the Succ-relation give us no possibility to represent infinite-state models by finite relations. Also, for bigger finite-state models this description is not really handsome either. This concrete description of service transitions is the topic of the present section.

Single Service. For a service with n input ports $I = \{ip_1, \ldots, ip_n\}$, m output ports $O = \{op_1, \ldots, op_m\}$, and w local variables $L = \{l_1 \ldots, l_w\}$ a transition t is given by

$$pre \wedge \bigwedge_{j=1}^{n} ip_j = i_j \wedge \bigwedge_{j=1}^{m} op'_j = o_j \wedge \bigwedge_{j=1}^{w} l'_j = v_j$$

with the meaning that whenever actual input port values ip_j match the patterns i_j, and in addition, precondition pre evaluates to $true$, then updated values of output ports op'_j and of local variables l'_j have to match the patterns o_j and v_j, respectively. pre is defined over unprimed input ports and unprimed local variables only. The patterns o_j and v_j may depend on the values of unprimed input ports and local variables only.

Unprioritized Composition. If two services S_1 and S_2 are composable, the transition set of their composition $C \stackrel{\text{def}}{=} S_1 || S_2$ consists of the following three sets: $T_C \stackrel{\text{def}}{=} T_a \cup T_b \cup T_c$. T_a is a set of transitions which are conjunctions of transitions from S_1 and S_2:

$$T_a \stackrel{\text{def}}{=} \{(pre_1 \wedge pre_2) \wedge (in_1 \wedge in_2) \wedge (out_1 \wedge out_2) \wedge (ass_1 \wedge ass_2) \mid t_1 \in T_1 \wedge t_2 \in T_2\}.$$

T_b is a set of transitions from S_1 whose input patterns and preconditions are extended by an additional condition such that if the transition from S_1 is I-enabled then no transition from S_2 is I-enabled simultaneously. In this case, no local variables of S_2 are allowed to be updated. Output ports of S_2 are not subject to any restrictions:

$$T_b \stackrel{\text{def}}{=} \{(pre_1 \wedge \neg \bigvee_{t \in T_2} pre_t) \wedge (in_1 \wedge \neg \bigvee_{t \in T_2} in_t) \wedge out_1 \wedge (ass_1 \wedge \bigwedge_{l \in L_2} l' = l) \mid t_1 \in T_1\}.$$

Set T_c of transitions from S_2 is defined analogously to T_b.

Prioritized Composition. The transition set of prioritized composition $PC \stackrel{\text{def}}{=} S_1 \parallel^{S_P} S_2$ is defined by $T_{PC} \stackrel{\text{def}}{=} T_a \cup T_b$. T_a is a set of transitions from unprioritized composition $C \stackrel{\text{def}}{=} S_1 || S_2$ whose input patterns and preconditions are extended by an additional condition such that if the transition from C is I-enabled then no transition from S_P is I-enabled simultaneously. In this case, the local variables of S_P remain unaltered:

$$T_a \stackrel{\text{def}}{=} \{(pre_t \wedge \neg \bigvee_{p \in T_P} pre_p) \wedge (in_t \wedge \neg \bigvee_{p \in T_P} in_p) \wedge out_t \wedge (ass_t \wedge \bigwedge_{l \in L_P} l' = l) \mid t \in T_C\}.$$

T_b is a set of transitions which are conjunctions of transitions from S_2 and S_P. No local variable of S_1 is allowed to be updated:

$$T_b \stackrel{\text{def}}{=} \{(pre_2 \wedge pre_P) \wedge (in_2 \wedge in_P) \wedge out_2 \\ \wedge (ass_2 \wedge ass_P \wedge \bigwedge_{l \in L_1} l' = l)) \mid t_2 \in T_2\}.$$

The above specification scheme allows us to reuse the STDs of AutoFocus for the specification of the proposed service diagrams. We provided algorithms that can be used for the implementation of the simulation environment for services in AutoFocus as well as for the import of service models into different verification tools.

4.2 Conflict Detection

In order to ensure the correctness of safety-critical systems, validation by simulation is not sufficient. Often, formal proofs are required to show the consistency of the specifications. A model checking back-end provided by AutoFocus tool allows to prove highly general properties of a system based on its functional specification [22].

For finite-state models the logical characterization of services and inter-service conflicts can be formulated in the propositional calculus. Thus, we can use the established SAT solvers or Constraint Logic Programming [15] to detect inconsistencies between functional requirements automatically. However, the transformation to a SAT solver or CLP is not in the scope of this paper – it is precisely addressed e.g. in [16] for AutoFocus STDs. In the following, we express the conditions for potential and definite conflicts based on the concrete semantic scheme presented above. This representation allows us to find conflicts automatically, e.g. by representing the conflict condition as a further CLP constraint.

Deadlocks. According to the definition of the valid service from Section 2.1, there is a deadlock in the specification of a single service if there is a reachable local state for which no successor state exists. In other words, there is a transition in the service automaton such that satisfiability of its assertion implies unsatisfiability of all service transitions for any inputs:

$$\exists t \in T : ass_t \wedge \forall ip_1, \ldots, ip_n : \neg \bigvee_{i \in T} pre_i.$$

Then, all states from $loc(\alpha)$ with $\alpha \vdash ass_t$ are definite conflicts (deadlocks) of the service. Naturally, the same applies to the composite services.

Potential Conflicts. According to the definition from Section 3.1, two services are potentially conflicting if there is an input sequence which can be processed by one of the single services in isolation but cannot be processed by their composition. In other words, there is a transition t_1 of a single service S_1 such that there is no transition of the composition $S_C \stackrel{\text{def}}{=} S_1 \parallel S_2$ whose input patterns and precondition imply the input patterns and precondition of t_1:

$$\exists t_1 \in T_1 : \forall t_c \in T_C : \neg(pre_c \wedge in_c \Rightarrow pre_1 \wedge in_1).$$

Then, any α with $\alpha \vdash pre_1$ is a state in which service S_1 is potentially conflicting with S_2.

Regarding potential conflicts in a prioritized composition $S_{PC} \stackrel{\text{def}}{=} S_1 \parallel^{S_P} S_2$, a conflict might appear only if the current input cannot be processed by the priority service. In other words, additionally to the condition of an unprioritized conflict, satisfiability of the preconditions and input patterns of t_1 has to imply the unsatisfiability of the preconditions and input patterns of all transitions of S_P:

$$\exists t_1 \in T_1 : \forall t_{pc} \in T_{PC} : \neg(pre_{pc} \wedge in_{pc} \Rightarrow pre_1 \wedge in_1) \\ \wedge \neg \bigvee_{p \in T_P} (pre_1 \wedge pre_p \wedge in_1 \wedge in_p).$$

The above definitions of a precise operational semantics of services and inter-service conflicts in a uniform scheme allow us to extend an existing tool by a specification technique which can be used to validate and verify functional requirements. Also, based on this semantics consistency analysis can be performed automatically.

5. RELATED WORK

Formal Foundation in Requirements Engineering. The definition of formal semantics for requirement specifications is not new. This idea goes back to the work by Parnas and Madey [19], that proposes to specify a system as a black-box by means of mathematical functions. These functions describe the relations between variables of the considered system and its environment. In contrast to our work, this approach offers neither an operation semantics nor a method to specify single requirements modularly and therefore it is not able to support the scalability necessary for the state-of-the-art systems.

Our work was significantly inspired by the PlayEngine introduced by Harel et al. in [12], and particularly by their model chain from play-in scenarios to the system model via requirements. The PlayEngine provides interactive validation of interaction sequences. It takes a set of LSCs (Live Sequence Charts, an extention of Message Sequence Charts) as input and allows the user to send input signals to the system. Then, it picks at random an interaction sequence that suits the user input and shows the system output. In the case that the user registers an undesirable system reaction, she has to manually respecify the system. This approach, although valuable for interactive validation, does not offer automatic conflict detection, which is a central result of the presented paper.

The presented work is based on a theoretical framework introduced by Broy [5] where the notion of a service behavior and service composition are formally defined. This framework proposes to model system services as partial stream processing functions. It also describes, what behavior the composition, also a partial stream processing function, should exhibit. However, this framework does not automate conflict detection, either.

A further related approach can be found in the work by Schätz [21]. He provides a constructive approach to build components from modular building blocks called functions. Like a service in our approach, a function is not necessarily totally defined. However, in contrast to the presented work, these functions are not allowed to share common output ports. As a result, the behavior on a port cannot be specified by different requirements which makes the functions according to Schätz not a natural means to model requirements specifications.

The closest approach to our work is the work by Harhurin and Hartmann [13]. They focus on establishing the consistency of the spec-

ification of product lines. To that end, they define conflicts between requirements and describe how these conflicts can be detected. However, their approach uses a denotational semantics formalized in first-order logic, what renders automatic analysis of functional requirements impossible.

Feature Interaction. The central aim of our approach is to detect and resolve conflicts between single requirements in order to assure the consistency of the overall specification. A large body of research on a related problem, called feature interaction, was caused by the huge amount of software-based functions in telecommunications [7]. In this domain, features represent capabilities that are incrementally added to a telephony network. One of the most prominent works in this area is the approach by Jackson and Zave [14]. They introduce Feature Boxes to model a telecommunication system as a set of data passing components. In this sense, Feature Boxes are similar to the approach presented in this paper. However, they do not offer automatic conflict detection.

The goal-oriented approach to requirements refinement, as proposed by van Lamsweerde [24], aims at refining high-level goals to requirements and modeling conflicts between goals and/or requirements. Conflicts are manually detected and modeled. The presented approach, to the contrary, allows to validate specifications via simulation and offers systematic conflict detection.

Further related work can be found in the domain of embedded control systems, e.g. approaches by Metzger [18] or Wilson et al. [25]. All these approaches have in common that they focus on feature interaction in the design phase without supporting the inter-requirement conflict detection.

Automata Theories. Automata are a popular formalism for the formal foundation of different artifacts and models in the software engineering. Although there exists a variety of automata-based approaches, they all mainly deal with design and implementation of software systems. Automata-based approaches, like I/O automata by Lynch and Tuttle [17], or interface automata by de Alfaro and Henzinger [8], are not suitable to model requirements. Features like input-enabledness of the I/O-automata, or disjointness of output ports of the interface automata, would inevitably lead to unjustified design decisions made in the requirements engineering phase, which is methodically wrong.

6. CONCLUSION AND OUTLOOK

We have presented an automata-based modeling paradigm, which provides a natural way to capture and analyze the specifications of reactive systems. The proposed composition mechanisms combines single functional requirements, formalized as services, to an overall specification. Both requirements and overall specifications are described from the point of view of the system environment. Using our model, the developer does not have to take any internal details of the system into consideration such as component architecture or internal data and control flow. These details should be treated in the subsequent development phases. Another important feature of our model is the possibility to describe both input and output behaviors of a system only partially. These two features distinguish the proposed approach from any other formalism and bring two important advantages: They allow establishing a one-to-one correspondence between services and informal textual requirements. In addition, our notation eliminates the necessity for the developer to over-constrain the analytical specification model by implementation details.

Furthermore, the modularity in the proposed service-based development process is also facilitated by the properties of the composition, i.e. the associativity and commutativity of the un-prioritized composition and the distributivity of the prioritized one.

We see the application area of the proposed service-based specification in the requirements analysis phase, as part of the reactive systems development. With a structured textual specification as an input, this specification can be checked for consistency and fulfillment of stakeholders' goals by verification and simulation, respectively. These two tasks can be performed thanks to the presented formal operational semantics. For the former task we have formally described the conditions which characterize a conflicting state.

Automation and computer-support of different tasks in software engineering are as important as their foundation by formal methods. The combination of these two factors is the best way to produce correct systems, fulfilling the customer's needs, at affordable cost. For this purpose we presented a concrete scheme for the description of our service models in the CASE tool AutoFocus.

Last but not least, a formal specification model can serve as an acceptance criterion for the artifacts produced during the subsequent development phases, such as design or implementation. By integrating service models into AutoFocus, we aim at establishing a model stack, in which the correspondence between design and specification can be formally proved, e.g. by showing a simulation relation between both models.

There are several directions for further development of the presented approach. We are currently working on concepts to verify the satisfiability of the service-based specification by a component-based architecture. Beyond this, our future work includes the integration of the proposed service model into the tool AutoFocus.

Acknowledgments

We are grateful to Judith Hartmann for her advice on early versions of the paper.

7. REFERENCES

[1] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Upper Saddle River, NJ, USA, 1981.

[2] J. Botaschanjan. *Property-Preserving Deployment*. PhD thesis, TUM, 2008. To appear.

[3] P. Braun, H. Lötzbeyer, B. Schätz, and O. Slotosch. Consistent integration of formal methods. In *TACAS '00*. Springer-Verlag, 2000.

[4] M. Breitling and J. Philipps. Black box views of state machines. Technical Report TUM-I9916, Institut für Informatik, Technische Universität München, 1999.

[5] M. Broy. Service-oriented systems engineering: Specification and design of services and layered architectures. In *Engineering Theories of Software Intensive Systems*, pages 47–81. Springer Verlag, 2005.

[6] M. Broy. Model-driven architecture-centric engineering of (embedded) software intensive systems: modeling theories and architectural milestones. *ISSE*, 3(1):75–102, 2007.

[7] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and

considered forecast. *Comput. Networks*, 41(1):115–141, 2003.

[8] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC/FSE-9*, pages 109–120. ACM, 2001.

[9] Festo Didactic. Detailspezifikation der SmartAutomation Modellanlage, September 2006. Version 1.12.

[10] K. Grimm. Software technology in an automotive company: major challenges. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 498–503, Washington, DC, USA, 2003. IEEE Computer Society.

[11] A. Gruler, A. Harhurin, and J. Hartmann. Modeling the functionality of multi-functional software systems. In *Proceedings of ECBS'07*, 2007.

[12] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, pages 378–398, London, UK, 2002. Springer-Verlag.

[13] A. Harhurin and J. Hartmann. Towards consistent specifications of product families. In *FM: 15th International Symposium on Formal Methods*, volume 5014 of *LNCS*. Springer Verlag, 2008. To appear.

[14] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Trans. Softw. Eng.*, 24(10):831–847, 1998.

[15] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[16] H. Lötzbeyer and A. Pretschner. AutoFocus on Constraint Logic Programming. In *Proc. (Constraint) Logic Programming and Software Engineering*, 2000.

[17] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[18] A. Metzger. Feature interactions in embedded control systems. *Comput. Netw.*, 45(5):625–644, 2004.

[19] D. L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, 1995.

[20] A. Pretschner and W. Prenninger. Computing refactorings of state machines. *Software and Systems Modeling*, 6(4), 2007.

[21] B. Schätz. Building components from functions. In *Proceedings of FACS 2005*, volume 160 of *ENTCS*, 2005.

[22] B. Schätz and F. Huber. Integrating formal description techniques. In *World Congress on Formal Methods (2)*, pages 1206–1225, 1999.

[23] Siemens, Sector Industry. `http://www.industry.siemens.de/en/`.

[24] A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *RE '01: Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, pages 249–263. IEEE Computer Society, 2001.

[25] M. Wilson, E. H. Magill, and M. Kolberg. An online approach for the service interaction problem in home automation. In *IEEE Consumer Comm. and Networking Conf*, pages 251–256, 2005.