Formal incremental requirements specification of service-oriented automotive software systems

J. Hartmann, S. Rittmann, D. Wild
Software & Systems Engineering
Technische Universität München
Boltzmannstraße 3
D-85748 Garching bei München

{hartmanj, rittmann, wild}@in.tum.de

P. Scholz
Fachbereich für Informatik
Fachhochschule Landshut
Am Lurzenhof 1
D-84036 Landshut

peter.scholz@fh-landshut.de

Abstract

In this paper, we introduce a simple but formal service description language (ForSeL) for model-based requirements engineering.

The basic notion in ForSeL is a service representing a functional requirement. Each service describes a system "re"-action that is triggered by a set of input actions – (but only) if an additional precondition holds. The functional part of a specification is then obtained by the combination of a finite number of services. We pay special attention to two kinds of preconditions which are often mixed up in practice: sufficient and necessary preconditions. Moreover, we present refinement concepts for services that enable a stepwise development of functional requirements.[1]

1 Introduction and Related Work

During the requirements engineering phase, (functional) requirements are usually obtained *iteratively* and documented in an *informal* way, i.e. they are described textually or by means of use cases. This often leads to imprecise and contradictory requirement specifications. Another problem is the gap between informal descriptions of the requirements engineering phase and the *formal* models of the design phase.

In our opinion, a promising approach to handle these intricacies is service orientation. In this paper we present our current research on a **for**mal ***se**rvice description **l**anguage* (ForSeL) that describes *functional requirements* in terms of formal services. Hereby, a service is a system reaction that is triggered by a set of input actions *if* (or *only if*) an additional precondition holds. We pay special attention to two different kinds of preconditions which are often mixed up in practice – namely sufficient and necessary preconditions. Moreover, we show how services can be refined (iteratively) during the development process.

We give both syntax and semantics for the formal service description language, the two types of preconditions and their refinement, respectively.

Related Work. In ForSeL, a set of requirement specifications can be alternatively written down as tables, as we already presented in [4]. The SCR (Software Cost Reduction) method [5] makes use of tables for the specification of systems. However, the SCR specification is done on a much more concrete abstraction level, i.e. implementation details are not abstracted away.

In [6], causal and temporal dependencies between actions are caught and represented graphically by so-called timing diagrams. A semantic foundation is given by the mapping to LTL formulas. The focus of [6] is centred on the topic of verification. In contrast, our approach focuses on the specification of systems.

In [2], a very formal theory on service-oriented development is presented. Services are defined as partial stream processing functions. We base our work on the concepts of [2] and augment it by a lean notation technique and a methodology.

Outline. The remainder of this paper is structured as follows: Syntax and semantics of ForSeL are given in Sec. 2. Sec. 3 presents syntactic transformation rules of ForSeL. Sec. 4 and 5 contain syntactic rules for behavioral refinement of services specified with ForSeL. Finally, we give a summary and an outlook in Sec. 6.

[1] The presented work is current research of the Mobilsoft project funded by the Bavarian Government (see www.mobilsoft.info).

2 Syntax and Semantics of System Reactions in ForSeL

2.1 Basics

ForSeL's semantics are based on stream processing functions that describe the system behavior by finite or infinite system reactions. The concepts of our approach are formally founded on the FOCUS [3] and the JANUS theory [2]. In FOCUS, a system is considered as a stream processing function on messages. This stream processing function relates input messages to output messages which are exchanged between the environment and the system. Thus, the system behavior is described by a black box view on the system. *Streams* of (typed) messages represent the communication history within a (finite) time frame. Given a set of data messages M (or better "actions" in our setting), M^* (M^∞) denotes the set of finite (infinite) streams (=sequences of elements of M). M^ω denotes $M^* \cup M^\infty$. A stream can be represented by a function $s: N_0 \to M$, where s(t) contains the message processed in the stream s at the point in time t. In JANUS, a component is a *total* behavior (function) whereas a service is a *partial* function. As a consequence, a service is only defined for a subset of all possible streams.

In our approach, we are only interested in the interaction of the system with its environment as observable at the system interface. Hence, the system is defined by the set of streams that characterize valid system runs. These streams can be represented by streams over actions $s \in Act^\omega$, whereas Act denotes the finite set of relevant actions for a system (=user-visible input or output that can be observed at the system border). Thinking of the requirements for an automotive power window system for example, actions could be "press toggle switch", "window goes up", or "child safety lock enabled".

Formally spoken, an action A is a variable of type Action = $\{0,1,\uparrow,\downarrow\}$. A can have exactly one of the following assignments (at the point in time t):

- A=1: A is active, i.e. $A \in s.t$
- A=0: A is not active, i.e. $\neg A \in s.t$
- A=↑: A starts, i.e. $A\uparrow \in s.t$, consequently $\neg A \in s.(t-1)$ and $(A \vee A\downarrow) \in s.(t+1)$
- A=↓: A stops, i.e. $A\downarrow \in s.t$, consequently $A \in s.(t-1)$ and $(\neg A \vee A\uparrow) \in s.(t+1)$

A↑ and A↓ are called *start event* and *end event,* respectively. A=0 and A=1 are the *states* of the action A.

Actions can be clearly divided into *environment actions* $Env \subset Act$ and *system reactions* $Sys \subset Act$, whereas $Act = Env \cup Sys$ and $Env \cap Sys = \emptyset$.

2.2 Syntax and semantics of system reactions

In our approach, we support the specification specialist in defining formal services representing functional requirements. A service describes how a system is to react on particular inputs if (or only if) an additional precondition holds. It is comprised of three parts, namely: a **precondition** P, a **triggering event** T, and a **system reaction** R.

Formally, a precondition P is a conjunction of assignments of actions, i.e. more precisely for a natural number n:

$$P = (A_1 = b_1) \wedge (A_2 = b_2) \wedge \ldots \wedge (A_n = b_n)$$
$$\text{with} \quad A_i \in Act, b_i \in \{0,1\}, i = 1, \ldots, n\,.$$

Each A_i denotes an action which has an influence on the reaction pattern T→R (see later).

We clearly distinguish between two different kinds of preconditions which are often mixed up in practice: sufficient and necessary preconditions. A *sufficient precondition* (**[P](T→R)**) describes the states of the system in which a certain triggering event *unavoidably* leads to a certain reaction. In case the precondition P is not fulfilled, the system behavior is not specified and any behavior is accepted. Therefore, sufficient preconditions allow for the specification of partial behavior. A *necessary precondition* (**⟨P⟩(T→R)**) describes the only possible states of the system in which a certain triggering event *may* cause a certain reaction. However, it is not required that the system reaction R has to occur at all, even if the precondition holds and the triggering event T occurs. The introduced patterns can be combined in order to express that *if and only if* the precondition is fulfilled the triggering event T causes necessarily the system reaction R (**{P}(T→R)**).

The triggering event T defines which start or end events trigger the specified system reaction R. Formally, T is a conjunction of start and end events:

$$T = (B_1 = e_1) \wedge (B_2 = e_2) \wedge \ldots \wedge (B_n = e_n)$$
$$\text{with} \quad B_i \in Act, e_i \in \{\uparrow, \downarrow\}, i = 1, \ldots, n$$

The B_i are those actions in Act whose start or end causes the reaction R. The reaction R denotes which start and end events are triggered. Formally, R is a conjunction of start and end events of system reactions.

Each service defines a predicate over streams $s \in Act^\omega$ that has to be fulfilled by the system. It filters out only the valid streams of all possible streams. By the aid of the definitions above, the semantics of the abstract service patterns **[P](T→R), ⟨P⟩(T→R),** and **{P}(T→R)** can be formally define as follows:

$$[\![[P](T \to R)]\!] = \{s \in Act^\omega \mid \forall t \in N : ((P \in s.t \wedge T \in s.t) \Rightarrow (\exists t' > t : R \in s.t'))\}$$

Informally spoken, [P](T→R) means that if at some point in time t precondition P is true and triggering event T occurs, then reaction S has to occur at some later point in time t´>t.

$$[\![\langle P \rangle (T \to R)]\!] = \{s \in Act^{\omega} \mid \forall t \in N : ((T \in s.t \wedge (\exists t' > t : R \in s.t')) \Rightarrow P \in s.t)\}$$

⟨P⟩(T→R) means that if a triggering event T at some point in time t causes a reaction R at some later point in time t´>t, then precondition P must be true at the point in time t.

$$[\![\{P\}(T \to R)]\!] = \{s \in Act^{\omega} \mid \forall t \in N : (T \in s.t \Rightarrow ((\exists t' > t : R \in s.t' \wedge P \in s.t) \vee (\neg \exists t' > t : R \in s.t' \wedge \neg P \in s.t)))\}$$

{P}(T→R) means that if a triggering event T occurs at some point in time t, then it causes the reaction R at some later point in time t´>t, if and only if the precondition P is true at the point in time t.

3 ForSeL Calculus

A formal specification Spec is the conjunction of a finite number of services:

$$Spec = S_1 \otimes S_2 \otimes S_3 \otimes ..... \otimes S_n$$

A valid implementation must realize each of the services S_i. For two services S_1 and S_2, their combination $S_1 \otimes S_2$ is simply defined as conjunction of their constituting predicates, i.e. formally $S_1 \cap S_2$. Consequently, the composition operator is commutative and associative. The proof for these rules is straightforward [4].

Applying the composition operator to the different service patterns separately, we obtain the following definitions:

[P](T→R): $S_1 \otimes S_2 = [P_1](T_1 \to R_1) \otimes [P_2](T_2 \to R_2)$ is formally defined as follows:

$$[\![[P_1](T_1 \to R_1) \otimes [P_2](T_2 \to R_2)]\!] = \{s \in Act^{\omega} \mid \forall t \in N : (((P_1 \in s.t \wedge T_1 \in s.t) \Rightarrow (\exists t' > t : R_1 \in s.t')) \wedge ((P_2 \in s.t \wedge T_2 \in s.t) \Rightarrow (\exists t' > t : R_2 \in s.t')))\}$$

⟨P⟩(T→R): $S_1 \otimes S_2 = \langle P_1 \rangle (T_1 \to R_1) \otimes \langle P_2 \rangle (T_2 \to R_2)$ is formally defined as follows:

$$[\![\langle P_1 \rangle (T_1 \to R_1) \otimes \langle P_2 \rangle (T_2 \to R_2)]\!] = \{s \in Act^{\omega} \mid \forall t \in N : (((T_1 \in s.t \wedge \exists t' > t : R_1 \in s.t') \Rightarrow P_1 \in s.t) \wedge ((T_2 \in s.t \wedge \exists t' > t : R_2 \in s.t') \Rightarrow P_2 \in s.t))\}$$

{P}(T→R). As mentioned before, the composition of two services [P](T→R) and ⟨P⟩(T→R) results in {P}(T→R): $\{P\}(T \to R) = [P](T \to R) \otimes \langle P \rangle (T \to R)$

This composition is formally defined as follows:

$$[\![[P_1](T_1 \to R_1) \otimes \langle P_2 \rangle (T_2 \to R_2)]\!] = \{s \in Act^{\omega} \mid \forall t \in N : ((T_1 \in s.t \Rightarrow ((\exists t' > t : R_1 \in s.t' \wedge P_1 \in s.t) \vee (\neg \exists t' > t : R_1 \in s.t' \wedge \neg P_1 \in s.t))) \wedge (T_2 \in s.t \Rightarrow ((\exists t' > t : R_2 \in s.t' \wedge P_2 \in s.t) \vee (\neg \exists t' > t : R_2 \in s.t' \wedge \neg P_2 \in s.t))))\}$$

In the sequel, we analyze further properties of the composition operator.

Identical reaction patterns: If two services $S_1=[P_1](T \to R)$ and $S_2=[P_2](T \to R)$ describe the same reaction pattern $T \to R$ for different sufficient preconditions P_1 and P_2, respectively, the composition $S_1 \otimes S_2$ has to make sure that the reaction pattern occurs both under the precondition P_1 and P_2:

$$[\![[P_1](T \to R) \otimes [P_2](T \to R)]\!] = [\![[P_1 \vee P_2](T \to R)]\!]$$

The composition $S_1 \otimes S_2$ of two services $S_1=\langle P_1 \rangle (T \to R)$ and $S_2=\langle P_2 \rangle (T \to R)$ that describe different necessary preconditions P_1 and P_2 for the same reaction pattern T→R has to ensure that the reaction pattern occurs only if both preconditions P_1 and P_2 are fulfilled, i.e.

$$[\![\langle P_1 \rangle (T \to R) \otimes \langle P_2 \rangle (T \to R)]\!] = [\![\langle P_1 \wedge P_2 \rangle (T \to R)]\!]$$

Simple considerations show that these properties are fulfilled by our composition operator [4].

Identical precondition and reaction: If two services $S_1=[P](T_1 \to R)$ and $S_2=[P](T_2 \to R)$ have the same precondition and reaction, i.e. that under the same sufficient precondition the same reaction is triggered by a different triggering event T_1 and T_2, respectively, the composition $S_1 \otimes S_2$ has to make sure the following: if the precondition P is fulfilled, both the occurrence of T_1 and T_2 – independently from each other – has to trigger the reaction R, i.e.

$$[\![[P](T_1 \to R) \otimes [P](T_2 \to R)]\!] = [\![[P]((T_1 \vee T_2) \to R)]\!]$$

This property analogously applies for the composition of services with necessary or combined preconditions.

Identical Precondition and identical triggering event: If two services $S_1=[P](T \to R_1)$ and $S_2=[P](T \to R_2)$ require that under the precondition P, a triggering event T causes two different reactions R_1 and R_2, the composition $S_1 \otimes S_2$ shall make sure that both reactions R_1 and R_2 are caused. However, these two reactions can occur at two arbitrary future points in time t´ and t´´. It can easily be proven that

$$[\![[P](T \to R_1) \otimes [P](T \to R_2)]\!] = [\![[P](T \to (R_1, R_2))]\!]$$

Here (R_1, R_2) means that both system reactions R_1 and R_2 occur, but independently from each other (as far as a concrete point in time is concerned).

The composition $S_1 \otimes S_2$ of two services $S_1=\langle P \rangle (T \to R_1)$ and $S_2=\langle P \rangle (T \to R_2)$ that describe that both reaction patterns $T \rightarrow R_1$ and $T \rightarrow R_2$ implicate the necessary precondition P. Thus, the precondition P must hold if T triggers R_1 just as well as if T triggers R_2, i.e. it is

$[\![\langle P \rangle (T \rightarrow R_1) \otimes \langle P \rangle (T \rightarrow R_2)]\!] = [\![\langle P \rangle (T \rightarrow (R_1 \vee R_2))]\!]$.

The combination of two services with combined preconditions results in

$[\![\{P\}(T \rightarrow R_1) \otimes \{P\}(T \rightarrow R_2)]\!] = [\![\{P\}(T \rightarrow (R_1, R_2))]\!]$.

T triggers both reactions, R_1 and R_2 (temporarily independent of each other) if and only if the precondition P holds.

4 Refinement of ForSeL Specifications

This section addresses the deductive design of requirements specifications. A specification S2 is a *behavioral refinement* of a specification S1 (or "S2 refines S1" or briefly S1®S2) if and only if any observable behavior of S2 is also an observable behavior of S1, formally $[\![S2]\!] \subseteq [\![S1]\!]$. As a consequence, under-specification is always expressed by non-determinism on the semantic level and behavioral refinement is its stepwise reduction.

We show that the notion of refinement heavily depends on the preconditions. This confirms to clearly distinguish the different kinds of preconditions.

First, we show that the addition of a further requirement to an existing specification leads to a refinement of the specification. This means that the set of accepted streams is reduced. It is obvious that $S_1 \otimes S_2$ is a stronger condition than S_1 and that consequently $S_1 \otimes S_2 \Rightarrow S_1$ holds [3]. Together with the above introduced properties of the composition operator, this leads to different refinement rules. . It results in an alternating approach for refinement as for example known from interface automata [1]: a specification refines another specification, if it demands weaker input assumptions or stronger output guarantees. This can be achieved by:

(1) Converting a sufficient precondition or a necessary precondition into combined precondition:
$[P](T \rightarrow R)$ ® $\{P\}(T \rightarrow R)$ and
$\langle P \rangle (T \rightarrow R)$ ® $\{P\}(T \rightarrow R)$.
However, note that there is no refinement relation between necessary and sufficient precondition.
(2) Enhancing the service domain of a reaction pattern by introducing additional sufficient preconditions:
$[P_1](T \rightarrow R)$ ® $[P_1 \vee P_2](T \rightarrow R)$
(3) Restricting the system behavior by concreting a necessary (or combined) precondition:
$\langle P_1 \rangle (T \rightarrow R)$ ® $\langle P_1 \wedge P_2 \rangle (T \rightarrow R)$ and
$\{P_1\}(T \rightarrow R)$ ® $\{P_1 \wedge P_2\}(T \rightarrow R)$
(4) Enhancing the service domain of a reaction pattern by introducing additional triggering events:
$[P](T_1 \rightarrow R)$ ® $[P]((T_1 \vee T_2) \rightarrow R)$,
$\langle P_1 \rangle (T_1 \rightarrow R)$ ® $\langle P_1 \rangle ((T_1 \vee T_2) \rightarrow R)$ and
$\{P\}(T_1 \rightarrow R)$ ® $\{P\}((T_1 \vee T_2) \rightarrow R)$
(5) Restricting/concreting the system reaction by defining a stronger reaction, as done with
$[P](T \rightarrow R)$ ® $[P](T \rightarrow (R_1, R_2))$ and
$\{P\}(T \rightarrow R)$ ® $\{P\}(T \rightarrow (R_1, R_2))$
Again, this rule does not apply for necessary preconditions.

5 Summary and Future Work

In this contribution we gave both syntax and semantics for the formal service description language ForSeL. We paid special attention to two different notions of preconditions. Additionally, we presented refinement concepts which enable a stepwise refinement process.

As next steps we plan to investigate the transition from ForSeL specifications to subsequent design activities (e.g. mapping to logical system architectures). Moreover, we work on refinement concepts allowing for the enlargement also of syntactic interfaces. Furthermore, we work on pragmatic notation techniques and on a comprehensive methodology for ForSeL. A case study of a medium-scale automotive software system is currently performed to validate our concepts.

6 References

[1] L. de Alfaro and T. A. Henzinger: *Interface automata. Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering* (FSE), ACM Press, 2001, pp. 109-120.

[2] M. Broy: *Service-Oriented Systems Engineering: Specification and Design of Services and Layered Architectures – The Janus Approach.* In: Engineering Theories of Software Intensive Systems, pp. 47-81. Springer, 2005.

[3] M. Broy and K. Stølen: *Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement.* Springer, 2001.

[4] J. Hartmann, S. Rittmann, P. Scholz, and D. Wild: *A compositional approach for functional requirement specifications of automotive software systems.* Accepted at the International Workshop on Automotive Requirements Engineering (AuRE), 2006.

[5] C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords: *Tools for constructing requirements specifications: The SCR toolset at the age of ten*, International Journal of Computer Systems Science and Engineering, 20(1), January 2005, pp. 19-35.

[6] R. Schlör and W. Damm: *Specification and Verification of System-level Hardware designs using Timing Diagrams*, Proc. IEEE EDAC-EuroASIC'93, Paris (France), Feb. 1993, pp. 518-524.