A compositional approach for functional requirement specifications of automotive software systems

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

{hartmanj, rittmann, wildd}@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 will introduce a simple but formal service description language (ForSeL) together with a methodology for its application in software engineering. ForSeL helps to bridge the gap between informal functional requirements and formal models in the subsequent design phase. Though its semantics is formally defined and therefore well suited for development techniques like behavioural refinement, code generation and verification, the language is very easy to use for software engineers. Using our approach, requirements can be formulated precisely and without any contradictions.

The basic notion in ForSeL is an action. A ForSeL specification is the composition of a finite number of functional requirements. Each (functional) requirement describes a system "re"-action that is triggered by a set of input actions – but only if an additional precondition holds. A pragmatic and at the same time adequate notation for this triple is given by so-called reaction tables.[1]

1 Introduction

During requirements engineering for embedded, interactive systems as often found in the automotive industry, (functional) requirements are usually obtained in a textual and therefore informal way. This often leads to imprecise, ambiguous, and contradictory descriptions of the system under consideration. Moreover, the transition from the informal requirements to the formal models in the design phase often causes problems.

In this article, we will introduce a formal service description language called ForSeL (Formal Specification Language) which describes functional requirements in terms of (formal) services. Here a service is a piece of functionality, which is visible for the user of the system under development. ForSeL together with its methodology is a means to bridge the aforementioned gap between the informal requirements and the formal design models.

A specification in ForSeL describes the system on a very high level of abstraction, i.e. implementation details are not considered yet and the system under development is regarded as a black box. Hence, a ForSeL specification easily supports reuse and variant handling.

Furthermore, ForSeL allows for the partial specification of the system behaviour. The specification specialist is not forced to define the system totally already at the early phase of requirements engineering. Therefore, it is possible to concentrate on the characteristic ("most important") behaviour of the system or parts of it. The result then is a (partial or total) formal specification which is precise, unambiguous, and consistent.

1.1 Related Work

In this subsection we relate our contribution to other work.

The SCR (Software Cost Reduction) method [5] also makes use of tables for the specification of

[1] The work presented stems from the MobilSoft- project which is partially funded by the Bavarian Government under grant number IuK 188/001.

systems. However, SCR does not aim at specifications of such an abstract level that our approach is based on. Moreover, as SCR makes explicit use of states, it is less black-box oriented. In our approach we focus on the system behaviour as it can be observed from the outside of the system.

In [4], causal and temporal dependencies between actions are caught and graphically represented in form of so-called timing diagrams. A semantic foundation is given by the mapping to LTL formulas. The focus of [4] is centred on the topic of verification. Our approach focuses on the provision of a methodology.

In [1], 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 [1] and augment it by a lean notation technique and a methodology.

Whereas Statecharts and related approaches [7, 8] aim at the system specification during the software design phase, ForSeL helps the user in describing the system on a more abstract level. ForSeL expressions can be seen as automata fragments which can be integrated into automata. Therefore, ForSeL expressions can be seen as a more abstract specification technique or as a preliminary stage in the development process to describe the system behaviour.

1.2 Outline

In this contribution, we introduce syntax and semantics of ForSeL, a lean notation technique for requirements engineering of automotive software systems, and its methodology. A running example of the automotive domain, which is introduced in Section 2, shall help the reader to understand the concepts of ForSeL. In Section 3, we focus on the syntax and semantics of ForSeL. A convenient notation technique based on reaction tables for specifying with ForSeL is given in Section 4. The methodology of how to use ForSeL is described in Section 5. Finally, we conclude in Section 6 and give an outlook about future work.

2 Running Example

In order to introduce both specification concepts and methodology, we make use of a running example from the automotive domain: power windows as they can be found in almost all modern cars. In this section, the textually (informally) given requirements of the considered system are listed.

In order to reduce complexity and to improve readability we limit ourselves to only one single window that is described as follows:

(1) **Opening** and **closing** a window.
  (1.1) On pressing the open toggle switch (close toggle switch), the window is moved up (down).
  (1.2) The window keeps on moving as long as the toggle switch is pressed. It stops as soon as the toggle switch is released or if the respective end position is reached.

(2) **Crush protection**. The crush protection is intended to prevent someone/something from being clamped and possibly hurt between the window and the window frame.
  (2.1) A sensor detects if something is being clamped (between the window and the window frame) while the window is moving upwards.
  (2.2) If a clamped obstacle has been detected, the window immediately has to open completely.
  (2.3) If the toggle switch is pressed again, the window has to go upwards despite the crush protection.

(3) **Child safety lock**. The child safety lock shall prevent children from playing around with the back windows. If the child safety lock is enabled the window must not move (even) if the respective toggle switch is pressed.

3 Syntax and Semantics

Before being able to define the concrete syntax and semantics of ForSeL, we first explain some preliminaries.

3.1 Basics

ForSeL's semantics is based on stream processing functions that describe the system behaviour by finite or infinite system reactions.

3.1.1 Stream processing functions (FOCUS)

The concepts of our approach are formally founded by the FOCUS theory [2]. FOCUS is a formal system model which serves very well for specifying reactive, distributed systems. 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[2]. Thus, the system behaviour is described by a black box view on the system.

[2] (and between components within the system)

In the following we only introduce the concepts of the FOCUS theory which are necessary to define the semantics of our approach formally.[3]

Finite and **infinite streams**. In FOCUS *streams* of (typed) messages represent the communication history of data messages 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 (which are sequences of elements of M). M^{ω} denotes $M^* \cup M^{\infty}$. Such a stream can be represented by the function $s : N_0 \rightarrow M$ where s(t) contains the message processed in the stream s at the point of time t.

For example, let s = <m1,m2,m3,m4> (s is a finite stream consisting of the messages m1 to m4). Then s(2)=m3 as m3 is the third message which is processed in the stream. To improve the readability of formulas we often write s.n instead of s(n). <m,m,m,m,m…> would denote the infinite stream consisting of infinitely many copies of the message m.

3.1.2 Actions

In our approach ForSeL, a specification describes a system by a set of valid streams $s \in Act^{\omega}$, whereas Act denotes the finite set of relevant actions for a system, respectively, i.e. those actions that can be observed at the system border. The set of valid streams is determined by predicates given by the formal requirements.

Actions are the basic elements out of which a formal service specification is comprised. An action is a user-visible input or output of the system on a logical level. Examples for actions are "press toggle switch", "window goes up", or "child safety lock enabled".

Formally spoken an action A is a variable of type Action = {0,1,↑,↓}. Considering a valid stream s, a declaration of each action has to be made at each point in time. An action A can have exactly one of the following assignments (at the point of time t):

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

[3] For more information on the comprehensive FOCUS theory, the interested reader is referred to [2].

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

For example, the child safety lock (CL) can be active (CL=1), not active (CL=0), currently activated (CL=↑), or deactivated at the moment (CL=↓).

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

Environment actions can not be influenced directly by the system as opposed to system reactions which are directly influenced by the system. This means that the system can start and stop these actions (system reactions). System reactions are primarily output actions, but can also serve as input actions.

Considering our running example, the pressing of the open toggle switch and the close toggle switch for instance, are environment actions which can not be influenced by the system. In contrast, the movement of the windows (upwards and downwards) are system reactions that are controlled by the system.

3.2 Syntax and semantics of ForSeL

3.2.1 Formal Requirements

In our approach, the system behaviour is described by means of formal expressions representing requirements. A requirement is given by the triple $R = [P](T \rightarrow S)$. A formal requirement of this form is called a service. The transformation from informal requirements into services however is not a 1:1-mapping, but requires real design work (cf. 5). A formal requirement describes how a system is to react on particular inputs. It is comprised of three parts, namely:

- the precondition P
- the triggering event T, and
- the system reaction S

Precondition P. The precondition describes the states of the system in which a certain triggering event leads to a certain reaction. Formally, P is a conjunction of assignments of actions, i.e.

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

Here, each A_i denotes an action which has an influence on the reaction pattern $T \rightarrow S$. Actions not influencing the reaction pattern $T \rightarrow S$ are not further specified.

Considering again our running example, the pressing of the (open or close) toggle switch causes the

window to move, only if the child safety lock is not active at the moment, the switch is pressed (and therefore the action is activated). Therefore the deactivated child safety lock (CL=0) is a precondition for this system reaction.

Triggering event T. The triggering event T defines which start or end events trigger the specified system reaction. This time the states of the actions are not considered, as it is assumed that only events cause system reactions. Formally, T is a conjunction of start and end events, i.e.

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

Actions B_i are those actions in Act whose start or end cause the system reaction S.

In our running example, the start of pressing a toggle switch or detecting a clamped obstacle are triggering events.

System reaction S. The reaction S denotes which start and end events are triggered. Again, the states of the actions are not considered, but only events. Formally, S is a conjunction of start and end events of system reactions, i.e.

$$S = (C_1 = e_1) \wedge (C_2 = e_2) \wedge \ldots \wedge (C_n = e_2)$$
$$\text{with} \quad C_i \in \text{Sys}, e_i \in \{\uparrow, \downarrow\}, i = 1, \ldots, n$$

Those actions which are not influenced at all are not specified any further.

In our running example, there are only two actions which can be influenced by the system and therefore occur as reactions: „window up“ and „window down“.

Formal requirement [P](T→S). By the aid of the definitions above, the semantics of $[P](T \rightarrow S)$ can be defined as follows:

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

Informally spoken, $[P](T \rightarrow S)$ 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.

Each formal requirement defines a predicate that has to be fulfilled by the system. It filters out only the valid streams of all possible streams. Examples for such service formulas can be found in Section 5.

3.2.2 Formal Specification

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

$$\text{Spec} = R_1 \otimes R_2 \otimes R_3 \otimes \ldots\ldots \otimes R_n$$

A valid implementation has to fulfil each of the formal requirements R_i. For two requirements $R_1 = [P_1](T_1 \rightarrow S_1)$ and $R_2 = [P_2](T_2 \rightarrow S_2)$, their composition

$$R_1 \otimes R_2 = [P_1](T_1 \rightarrow S_1) \otimes [P_2](T_2 \rightarrow S_2)$$

is formally defined as follows:

$$[[[P_1](T_1 \rightarrow S_1) \otimes [P_2](T_2 \rightarrow S_2)]] =$$
$$\{s \in \text{Act}^{\omega} \mid \forall t \in N :$$
$$(((P_1 \in s.t \wedge T_1 \in s.t) \Rightarrow (\exists t' > t : S_1 \in s.t')) \wedge$$
$$((P_2 \in s.t \wedge T_2 \in s.t) \Rightarrow (\exists t' > t : S_2 \in s.t')))\}$$

Informally spoken: If P_1 is true at a certain point of time t and T_1 occurs, then the system has to react with S_1 and if P_2 is valid in a point of time t – independently of P_1, T_1, and S_1 – and T_2 occurs then the system has to react with S_2. If both of the preconditions are fulfilled and both of the triggering events occur, then also both of the reactions have to occur; however, not necessarily at the same point of time.

Analogously, the overall specification can be created by stepwise conjunction of the single requirements.

In the following we prove some properties of the composition operator which show that the formal definition of the composition operators goes along with the intuitive understanding.

Identical reaction patterns. If two requirements

$$R_1 = [P_1](T \rightarrow S) \text{ and } R_2 = [P_2](T \rightarrow S)$$

describe the same reaction pattern $T \rightarrow S$ for different preconditions P_1 and P_2, respectively, the composition $R_1 \otimes R_2$ has to make sure that the reaction pattern occurs both under the precondition P_1 and P_2, i.e. it is

$$[[[P_1](T \rightarrow S) \otimes [P_2](T \rightarrow S)]] = [[[P_1 \vee P_2](T \rightarrow S)]]$$

A simple consideration shows that this is fulfilled by our composition operator.

$$[[[P_1](T \rightarrow S) \otimes [P_2](T \rightarrow S)]]$$
$$= \{s \in \text{Act}^{\omega} \mid \forall t \in N :$$
$$(((P_1 \in s.t \wedge T \in s.t) \Rightarrow (\exists t' > t : S \in s.t')) \wedge$$
$$((P_2 \in s.t \wedge T \in s.t) \Rightarrow (\exists t' > t : S \in s.t')))\}$$
$$= \{s \in \text{Act}^{\omega} \mid \forall t \in N :$$
$$((\neg(P_1 \in s.t \wedge T \in s.t) \vee (\exists t' > t : S \in s.t')) \wedge$$
$$(\neg(P_2 \in s.t \wedge T \in s.t) \vee (\exists t' > t : S \in s.t')))\}$$
$$= \{s \in \text{Act}^{\omega} \mid \forall t \in N :$$
$$((\neg((P_1 \in s.t \wedge T \in s.t) \vee (P_2 \in s.t \wedge T \in s.t))$$
$$\vee (\exists t' > t : S \in s.t'))\}$$
$$= \{s \in \text{Act}^{\omega} \mid \forall t \in N : (((P_1 \in s.t \vee P_2 \in s.t) \wedge$$
$$(T \in s.t)) \Rightarrow (\exists t' > t : S \in s.t'))\}$$
$$= [[[P_1 \vee P_2](T \rightarrow S)]]$$

Identical precondition and reaction. If two requirements

$$R_1 = [P](T_1 \rightarrow S) \text{ and } R_2 = [P](T_2 \rightarrow S)$$

have the same precondition and reaction, i.e. that under the same precondition the same reaction is triggered by a different triggering event T_1 and T_2, respectively, the composition $R_1 \otimes R_2$ 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 S, i.e.

$$[[[P](T_1 \rightarrow S) \otimes [P](T_2 \rightarrow S)]] = [[[P]((T_1 \vee T_2) \rightarrow S)]]$$

An analogous consideration shows that our composition operator fulfils this requirement:

$$\begin{aligned}
&[[[P](T_1 \rightarrow S) \otimes [P](T_2 \rightarrow S)]] \\
&= \{s \in Act^{\omega} \mid \forall t \in N : \\
&(((P \in s.t \wedge T_1 \in s.t) \Rightarrow (\exists t' > t : S \in s.t')) \wedge \\
&((P \in s.t \wedge T_2 \in s.t) \Rightarrow (\exists t' > t : S \in s.t')))\} \\
&= \{s \in Act^{\omega} \mid \forall t \in N : ((P \in s.t \wedge \\
&(T_1 \in s.t \vee T_2 \in s.t)) \Rightarrow (\exists t' > t : S \in s.t'))\} \\
&= [[[P]((T_1 \vee T_2) \rightarrow S)]]
\end{aligned}$$

Identical Precondition and identical triggering event. If two requirements

$$R_1 = [P](T \rightarrow S_1) \text{ and } R_2 = [P](T \rightarrow S_2)$$

demand that under the precondition P, a triggering event T causes two different reactions S_1 and S_2, the composition $R_1 \otimes R_2$ shall make sure that both reactions S_1 and S_2 are caused. However, these two reactions do not have to occur at the same time but can occur at two arbitrary future points in time t' and t''. Formally it has to be proven that

$$[[[P](T \rightarrow S_1) \otimes [P](T \rightarrow S_2)]] = [[[P](T \rightarrow (S_1, S_2))]]$$

Here (S_1,S_2) means that both system reactions S_1 and S_2 occur, but independently from each other (as far as a concrete point in time is concerned). Again, a simple consideration shows that this requirement is fulfilled by the composition operator:

$$\begin{aligned}
&[[[P](T \rightarrow S_1)] \otimes [P](T \rightarrow S_2)]] \\
&= \{s \in Act^{\omega} \mid \forall t \in N : \\
&(((P \in s.t \wedge T \in s.t) \Rightarrow (\exists t' > t : S_1 \in s.t')) \wedge \\
&((P \in s.t \wedge T \in s.t) \Rightarrow (\exists t' > t : S_2 \in s.t')))\} \\
&= \{s \in Act^{\omega} \mid \forall t \in N : \\
&((\neg(P \in s.t \wedge T \in s.t) \vee (\exists t' > t : S_1 \in s.t')) \wedge \\
&(\neg(P \in s.t \wedge T \in s.t) \vee (\exists t' > t : S_2 \in s.t')))\} \\
&= \{s \in Act^{\omega} \mid \forall t \in N : \\
&(\neg(P \in s.t \wedge T \in s.t) \vee \\
&((\exists t' > t : S_1 \in s.t') \wedge (\exists t' > t : S_2 \in s.t')))\} \\
&= \{s \in Act^{\omega} \mid \forall t \in N : \\
&((P \in s.t \wedge T \in s.t) \Rightarrow \\
&((\exists t' > t : S_1 \in s.t') \wedge (\exists t'' > t : S_2 \in s.t'')))\} \\
&= [[[P](T \rightarrow (S_1, S_2)]]
\end{aligned}$$

No Precondition: Another special case is the specification of a ‚pure' reaction pattern $T \rightarrow S$ where there is no precondition specified. According to the formal definition, we obtain the following semantics:

$$\begin{aligned}
&[[T \rightarrow S]] = \\
&\{s \in Act^{\omega} \mid \forall t \in N : (T \in s.t \Rightarrow (\exists t' > t : S \in s.t'))\}
\end{aligned}$$

$T \rightarrow S$ means that the occurrence of T causes the system to react with S in each situation, no matter in what state the system currently is. The specification specialist has to be aware of the fact that this is a very ‚strong' requirement. If a precondition is added to the reaction patter $T \rightarrow S$ belatedly, this is not a refinement as the requirement is made weaker.

Besides these special cases of composing requirements, the following rules apply as a consequence of the Boolean calculation rules:

Commutative law. The composition operator is commutative, i.e. for two arbitrary requirements R_1 and R_2 the following holds:

$$[[R_1 \otimes R_2]] = [[R_2 \otimes R_1]].$$

The proof for this rule is rather straightforward.

Associative law. Analogously, the application of the associative law for the Boolean operators proves the associativity of the composition operator for arbitrary requirements R_1, R_2, and R_3:

$$[[(R_1 \otimes R_2) \otimes R_3]] = [[R_1 \otimes (R_2 \otimes R_3)]].$$

In the last part of this sub-section we take a look at refinement. To that end, we describe our notion of refinement and show that the addition of further requirement to an existing specification leads to a refinement of the specification. Concretely we show that for the requirements $R_1 = [P_1](T_1 \rightarrow S_1)$ and $R_2 = [P_2](T_2 \rightarrow S_2)$ the following holds:

$$R_1 \otimes R_2 \Rightarrow R_1 .$$

Refinement. The refinement of a specification means that the set of accepted streams is reduced. It is obvious that $R_1 \otimes R_2$ is a stronger condition than R_1 and that consequently $R_1 \otimes R_2 \Rightarrow R_1$ holds as

$$\begin{aligned}
&[[[P_1](T_1 \rightarrow S_1) \otimes [P_2](T_2 \rightarrow S_2)]] = \\
&\{s \in Act^{\omega} \mid \forall t \in N : \\
&(((P_1 \in s.t \wedge T_1 \in s.t) \Rightarrow (\exists t' > t : S_1 \in s.t')) \wedge \\
&((P_2 \in s.t \wedge T_2 \in s.t) \Rightarrow (\exists t' > t : S_2 \in s.t')))\} \\
&\subseteq \\
&\{s \in Act^{\omega} \mid \forall t \in N : \\
&(((P_1 \in s.t \wedge T_1 \in s.t) \Rightarrow (\exists t' > t : S_1 \in s.t'))\} = \\
&[[[P_1](T_1 \rightarrow S_1)]]
\end{aligned}$$

The chosen notion of refinement can be described best by taking a look at the special cases. A refinement of a service can be achieved by

- Enhancing the service domain of a reaction pattern by introducing additional preconditions and/or triggering events (as done in the cases $[P_1](T \rightarrow S) \otimes [P_2](T \rightarrow S) = [P_1 \vee P_2](T \rightarrow S)$ and $[P](T_1 \rightarrow S) \otimes [P](T_2 \rightarrow S) = [P]((T_1 \vee T_2) \rightarrow S)$ respectively, or by
- Restricting/concreting the system reaction by defining a stronger reaction, as done with $[P](T \rightarrow S_1) \otimes [P](T \rightarrow S_2) = [P](T \rightarrow (S_1, S_2))$

We hereby use an alternating approach for refinement as for example known from interface automata [3]: a specification refines another specification, if it demands weaker input assumptions or stronger output guarantees.

3.2.3 Dependencies between actions

Besides the possibility to specify single requirements and to compose these modular requirements to obtain an overall specification, our approach also allows for the specification of dependencies between actions that are given by the environment or that have to be ensured by the system. These dependencies already have to be taken into account during the specification of the modular formal requirements and therefore restrict the set of the possible, correct requirements. Examples for such dependencies are explained in the following paragraphs.

Concurrency of actions. If two actions A and B are in the concurrency relation $A \succ B$, A is only allowed to occur simultaneously with B. Consequently, in each precondition both actions have to be active or A must not be active at the same point of time t. Analogously, the occurrence of a start (end) event of action A demands the start (end) event of the action B at the same point of time t. This has to be ensured by the specification. Formally this fact can be expressed by the equation

$$\begin{aligned}
&[[A \succ B]] = \forall [P](T \rightarrow S) : (Spec \Rightarrow [P](T \rightarrow S)) \Rightarrow \\
&((((P \Rightarrow (A = b)) \Rightarrow (P \Rightarrow (B = b))) \wedge \\
&(((T \Rightarrow (A = e)) \Rightarrow (T \Rightarrow (B = e))) \wedge \\
&(((S \Rightarrow (A = e)) \Rightarrow (S \Rightarrow (B = e)))) \\
&\text{with} \quad b = \{0,1\},\ e = \{\uparrow, \downarrow\}
\end{aligned}$$

Mutual exclusion of actions. Analogously, two actions can exclude each other mutually (A <≠> B), i.e. they never occur concurrently. Therefore, the concurrent occurrence of both actions can never be demanded as precondition, triggering events or reactions, for example. The formalization of this fact can be done analogously to the previous example.[4]

4 Notation

In order to apply our approach in practice, a pragmatic, concise notation for the specification is inevitable. An adequate notation technique for our ForSeL specifications are tables. Their structure is described in the following.

Each table is – like each specification term – divided into three parts: the preconditions, the triggering events, and the reaction. In each part an additional line is inserted for each possibly relevant action. As mentioned above, preconditions are statements over all actions $A \in Act$. Likewise, all actions $A \in Act$ have to be taken into consideration for the part of the triggering events. In contrast, when it comes to the part of the reactions, only those actions have to be inserted which are directly influenced by the system $A \in Sys$.

The assignments of the actions are inserted in the columns. Each column corresponds to a formal specification term R_1. For a system specification $Spec = R_1 \otimes \ldots R_n$ having the environments actions $Env = \{E_1, \ldots E_n\}$ and the system reactions $Sys = \{S_1, \ldots S_n\}$ we obtain the following table structure:

[4] Due to the limitation of space the formalization is not considered here.

	R_1	…	R_n
Precondition P:			
E_1			
…			
E_n			
S_1			
…			
S_n			
Triggering Event T:			
E_1			
…			
E_n			
S_1			
…			
S_n			
Reaction S:			
S_1			
…			
S_n			

Table 1: Schema of reaction tables

For example, the formal specification term $[(E_1 = 0) \wedge (S_1 = 1)](E_n = \uparrow \rightarrow S_1 = \downarrow)$ can be inserted in the table as follows:

	R_1
Precondition P:	
E_1	0
…	
E_n	
S_1	1
…	
S_n	
Triggering Event T:	
E_1	
…	
E_n	↑
S_1	
…	
S_n	
Reaction S:	
S_1	↓
…	
S_n	

Table 2: Exemplary ForSeL term in reaction table

The specification with a notion in table form has the following advantages over the specification with formal specification terms:

- The presentation is more concise and easier to understand.
- The table structure supports a systematic approach and helps in detecting holes in the requirements specification.
- In particular, the dependencies between the single formal specification terms are better understood. Therefore, the specification with help of tables supports the integration of the single specification terms and leads to a precise, consistent specification.
- As all possible actions and reactions are listed, the table supports the totalization step (which takes place later in the development process, cf. Section 5.3) and the transition to the operational system model. Cases/States/Behaviours which have not been considered can be detected automatically and added.
- The tables can easily be expanded. Both the introduction of additional requirements (insertion of a new column) and the introduction of further actions (insertion of new rows) are possible. Therefore, the tables support an iterative proceeding for the creation of the system specification.
- The table format services well as a basis for tool support; a prototype for this purpose is currently being developed.

Concluding it has to be remarked that the specification with tables demands a certain general knowledge about the system already at the beginning of the specification phase. The specification with help of tables does not seem to be useful until essential actions and basic requirements are identified.

5 Methodology

5.1 Creation of a formal specification based on textual requirements

In this sub-section a method for the systematic creation of a formal requirements specification based on single informal requirements is introduced. We start with a set of functional requirements given in textual form and aim at a concise, formal specification of these requirements. Basically, five steps have to be performed:

1. Identify the set of actions *Act.* (cf. 5.1.1)
2. Classify actions, i.e. divide *Act* into *Sys* and *Env* (cf. 5.1.2)
3. Construct the table structure (cf. 5.1.3).
4. Fill in the table entries iteratively and perform consistency checks (cf. 5.1.4).
5. Perform plausibility checks (cf. 5.1.5).

In the following these steps are explained with help of the previously introduced running example.

5.1.1 Identify the set of actions *Act*

First, all relevant actions of the system under construction are identified. For our running example we obtain the following actions:

- Switch for closing the window is being pressed (SWO)
- Switch for opening the window is being pressed (SWC)
- Window is closing (WC)
- Window is opening (WO)
- Clamped obstacle is detected (CD)
- Window end stop top (WET)
- Window end stop bottom (WEB)
- Child safety lock is on (CL)

Hence, the set of actions is defined as *Act* = {SWO, SWC, WC, WO, CD, WET, WEB, CL).

5.1.2 Classify actions

The set of actions *Act* is then divided into the set of environment actions *Env* and the set of system reactions *Sys*:

- *Env*={SWO, SWC, CD, WET, WEB, CL}
- *Sys*={WC, WO}

Furthermore, the dependencies between the actions within *Sys* and *Env* are identified:

- *Env:* SWO <≠> SWC, WET <≠> WEB
- *Sys:* WC <≠> WO

5.1.3 Construct the table structure

As a next step, the table structure has to be determined (see Section 4).

5.1.4 Fill in the table entries iteratively and perform consistency checks

Determine reaction patterns. For each system reactions *Sys*, all triggering events are systematically determined that start or stop this system reaction. As far as our ForSeL terms are concerned, the specification specialist defines terms of the form [P](T→S), whereas S is the given system reaction, T is the triggering event currently determined, and P is a disregarded precondition (which is added later).The reaction pattern T→S is then inserted into the table (cf. Table 3).

	1	2	3	4	5	6	7	8	9
P									
SWO									
SWC									
CD									
CL									
WET									
WEB									
WO									
WC									
T									
SWO	↑		↓			↑	↓		
SWC									
CD		↑							
CL				↑				↑	
WET									↑
WEB					↑				
WO									
WC									
S									
WO	↑	↑	↓	↓	↓				
WC						↑	↓	↓	↓

Table 3: Reaction patterns of the running example

Determine preconditions. The preconditions that have been disregarded in the previous step are not determined. As it might be too difficult to reveal all preconditions for a certain reaction pattern directly, we introduce an intermediate step: we consider possible preconditions for each triggering event and system reaction, respectively. [P]T or [P]S are intermediate notations (and not valid ForSeL terms) meaning that P is a precondition for the triggering event T or the system reaction S.

	2a	2b	2c	2d
P				
SWO				
SWC				
CD				
CL			WO	
WET				
WEB			WO	0
WO		0	0	0
WC			CD	1
T				
SWO				
SWC				
CD	↑	↑	↑	↑
CL				
WET				
WEB				
WO				
WC				
S				
WO	↑	↑	↑	↑
WC				

Table 4: Preconditions for one system reaction

For example, [WC]CD denotes that the closing of the window (WC) is a precondition for the crush

detection (CD). Further examples are [CL]WC, [CL]WO, [WET]WC, and [WEB]WO.

After this intermediate step, the previously identified reaction patterns can be enriched with preconditions.

During this step, a possible tool support can help the specification specialist by allowing him to only add information which is consistent to the prior defined system behaviour and by proposing him possible preconditions.

	1	2	3	4	5	6	7	8	9
P									
SWO						0	0		
SWC	0								
CD				0					
CL	0					0			
WET					0	0			
WEB	0	0							
WO	0	0	1	1	1		0	0	0
WC		1	0	0	0	0	1	1	1
T									
SWO	↑		↓						
SWC						↑	↓		
CD		↑							
CL				↑				↑	
WET									↑
WEB					↑				
WO									
WC									
S									
WO	↑	↑	↓	↓	↓				
WC						↑	↓	↓	↓

Table 5: Reaction table of the running example

Exemplary Application. Considering the reaction pattern of column 2 of Table 3 (CD↑→WO↑), we explain step by step how the interaction with the program could look like (cf. **Fehler! Verweisquelle konnte nicht gefunden werden.**):

1. **Column 2a:** First, only the reaction pattern (disregarding any precondition) is inserted by the specification specialist.
2. **Column 2b:** In the precondition part CD is given a grey background as it is already marked as triggering event. As the start of WO is the system reaction, WO has to be 0 in the precondition part. This step can be tool-supported.
3. **Column 2c:** At this stage, all possibly relevant preconditions have to be checked. In the context of our example these are of the form […]CD and […]WO. As mentioned above, we encounter the following preconditions: [WC]CD, [CL]WO, [WEB]WO (cf. column 2c).
4. **Column 2d:** The specification specialist has to decide which of the proposed preconditions (of column 2c) are really relevant. Accordingly, 0 or 1 is inserted into the respective fields of the table. If no entry is made, the assignment of the action does not matter for the precondition under consideration.

Proceeding like this for each column, we obtain **Fehler! Verweisquelle konnte nicht gefunden werden.**. This table now contains all system reactions with its triggering events that the specification specialist knows. It is therefore total in this regard. However, it is not total in the sense that (the reaction for) all possible input combinations are defined.

5.1.5 Plausibility checks

For each column which is additionally inserted it has to be checked if the column is contradictory to one of the other columns. In particular it has to be checked, if two (or more) triggering events which are not mutually exclusive lead to conflicting system reactions. In such a case the conflict might be resolved by defining the preconditions appropriately (if not already done).

5.2 Extension of specification terms /actions

So far, we assumed that the specification specialist already has knowledge about the part of the system to be specified. However, our approach also allows for changing the specification afterwards. Hereby, we can think of two situations: adding/deleting services or adding/deleting actions.

Adding formal requirements (services). The introduction of additional services corresponds to adding columns in the reaction table and can be done as explained in Section 4.

Deleting formal requirements (services). The deletion of a service (which corresponds to the deletion of a column in the reaction table) does not cause problems to the consistency of a specification. As the specification is the conjunction of all services (columns), a service (column) can be deleted without restricting the previously defined behaviour.

Adding actions. The introduction of actions is more complicated. It is assumed that the action to be added is already classified and that its dependencies to other actions are already defined. First, the new action has to be inserted (as a new line) in the part of the precondition and the part of the triggering action. If the

action is a system reaction a new row has to be inserted into the part of the system reactions, too. For each column it now has to be checked if the introduced action is a restriction on the previous specification and if it has to be considered in one ore more preconditions. Then it has to be checked if the action starts or stops one ore more system reactions. If so, respective columns have to be inserted as well and filled iteratively like discussed in 5.1.4. If the action is a system reaction all the other actions have to be looked at in order to find out if they start or stop the new action. Hereby, the introduction of additional columns might be necessary again.

Deleting actions. Deleting an action corresponds to deleting a row in the part of the preconditions and the part of the triggering events, respectively. If the action to be deleted is a system reaction, the corresponding row has to be deleted in the part of the system reactions. In this case it has to be checked for each column if there are any actions in the part of the triggering events and the system reactions. If not, the respective column can be deleted.

5.3 Totalization

As mentioned above, the formal specification is total with respect to the textual specification; however it is not total with regard to each possible input pattern. A tool could offer all cases which are not specified. As the amount of these under-specified cases is exponentially high, an appropriate algorithm has to be chosen which reduces the number of displayed cases. Making a specification total is a very important step during the requirements engineering phase and has to be investigated in detail. However, this is not scope of this paper.

6 Conclusion and Future Work

In this paper, we introduced our current work on the formal language ForSeL which aims at the specification of functional requirements of interactive systems. We gave syntax, semantics, and a pragmatic notation technique. Additionally, we introduced a methodology for the construction of precise, unambiguous, understandable system specifications with help of ForSeL. Furthermore, we explained how our approach can be supported by a tool. Finally we also outlined how a ForSeL specification can be turned into a total description of the system behaviour.

Currently we are concerned with how informal requirements can be translated into ForSeL expressions. To that end, textual patterns which are frequently used in informal requirement specifications are analysed and mapped to formal ForSeL expressions. Moreover, the transition from our formal requirements specification to the operational model of the design phase is currently investigated.. Additionally, we are working on a way to enrich our ForSeL specification with quantitative timing information as this is inevitable for the development of real time systems. In the line of this timed refinement, we are planning to adopt concepts of network planning techniques [6] to make statements about the expected timing behaviour such as buffer times (slack), critical paths, etc. of the system under construction. Case studies of medium-scale are being carried out in order to evaluate our approach and to investigate questions concerning scalability.

7 References

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

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

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

[4] 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.

[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] Eric Verzuh, *The Fast Forward MBA in Project Management*. John Wiley & Sons, 2005.

[7] P. Scholz, *Partitioning of Perfect Synchroneous Reactive Specifications to Distributed Processors using µ-Charts.* Software and Systems Modeling (SoSyM) Journal, Band 5, Nummer 1, Seite 13-25, Springer, April 2006.

[8] P. Scholz, *Incremental Design of Statechart Specifications.* Science of Computer Programming 40 (2001), Seiten 119 – 145, Elsevier Science B.V., 2001.