Modular Functional Descriptions

Bernhard Schätz

Technische Universität München, Fakultät für Informatik, Boltzmannstr. 3, D-85748 Garching bei München, Germany
schaetz@in.tum.de

Abstract

The construction of reactive systems often requires the combination of different individual functionalities, thus leading to a complex overall behavior. To achieve an efficient construction of reliable systems, a structured approach to the definition of the behavior is needed. Here, functional modularization supports a separation of the overall functionality into individual functions as well as their combination to construct the intended behavior, by using functional modules as basic paradigm together with conjunctive and disjunctive modular composition.

Keywords: Function, behavior, module, composition, decomposition, refinement, verification

1 Introduction

In many application domains reactive systems are becoming increasingly complex to cope with the technical possibilities and requested functionalities. The behavior provided by the system often is a combination of different functions integrated in an overall functionality; e.g., an embedded controller managing the movement of a car power window combines control of the basic movement, position control to restrict motor overload, as well as power management to avoid battery wear.

Implementing those combinations of individual functions is a complex and error-prone task. Since these functions in general influence each other, a modular development process ensures that the combined functionality respects the restrictions imposed by each individual function. Furthermore, due to the increased demand for possible variants of behavior, in general the development of reactive systems requires the recombination, restriction and extension of functionalities.

Here, the use of functional modules can improve the development process by supporting the modular definition of the basic functions as well as their combination into the overall functionality.

1.1 Contributions

Modular functional development aims at supporting the development process of multi-functional reactive systems by use of *modular composition of functions*. To

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

that end, we

- introduce *functions* as modular units of system construction, which provide a data flow interface describing the values observed and controlled by a function as well as a control interface describing the activation and deactivation of functions.
- use *disjunctive* and *conjunctive composition* as a means of combination, which allow to either alternatively or simultaneously combine functional behavior.
- provide automatic proof support to check the refinement between more descriptive and more constructive variants of specifications of funcionality.

1.2 Related Approaches

Functions are modules of behavior, used for the construction of complex behavior from basic functionality. They offer interfaces for both data and control flow in a similar fashion to the ports and connectors introduced in [6].

As generally used, e.g., in embedded systems, functions are intended for the description of signal-based reactive systems, using asynchronous communication unlike [5] or [10]. They use a communication paradigm similar to [9], [4], or [3]. Therefore, they provide similar forms of conjunctive and disjunctive compositions as provided for the modules introduced in [4] or the states introduced in [2]. However, while those are targeting the specification of reactive behavior in a constructive fashion, here a more descriptive from is used, using a more generalized form of (conjunctive) composition. In contrast to these constructive approaches, ruling out the introduction of partial behavior either syntactically by restricting compatible alphabets (e.g., [4]) or semantically using interleaving of interactions instead of synchronization (e.g., [2], functions with their less restricted composition allow a more natural modular form of specification.

Due to this form of composition, they are similar to services-oriented descriptions as used in [1]. In contrast to those rather descriptive approaches with a large number of different composition operators, however, functions provide a more constructive form of decomposition. Similar to [8], only conjunction and disjunction are used and a similar semantical models is used; but while there – due to its rather general form of specification without distinction between control and data flow – a more low-level formal form of description is needed, here by means of the explicit introduction ports and locations a more compact form of specification is possible.

Finally, the approach introduced here extends [11] by focusing on refinement rather than refactoring, and providing automated proof support by use of modelchecking.

2 Describing Functions

Functions form the building block of the approach presented here. Basically, functions are capsules of behavior, defined by their (external) interface in terms of data and control flow as well as their (internal) implementation. The data flow between the function and its environment is described in form of data signals exchanged between them, allowing the function to observe and control shared signals. The control flow between the function and its environment is described in form of control loca-

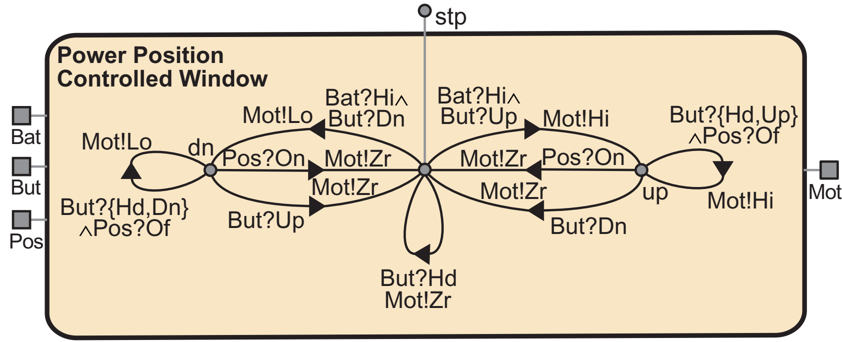

Fig. 1. Refactored Power Position Window Function

tions used to pass control between them, allowing the function to be activated and deactivated.

Figure 1 shows a function Power Position Controlled Window describing the functionality of a power window controller. The capsulated behavior is represented by a box, and identified by a function name (Power Window). Interface elements (e.g., Bat, stp) are attached to its border; its internal structure is depicted inside the box.

The function observes user interactions via the But signal (with Up, Hd, and Dn signaling the up, hold, and down position of the switch), the current battery status via the Bat signal (with Hi and Lo signaling high and low voltage), and the current position of the window via the Pos signal (with On and Of signaling intermediate or end positions of the window). It controls the motor of the window via the Mot signal (with Hi, Lo, and Zr signaling upward, downward, or no movement). As shown in Figure 1, the input and output signals accessed by a function are indicated by empty and filled boxes at the border of the function.

To control the activation and deactivation of the function, it can be entered and exited via the control location stp. As shown in Figure 1, while interface locations as well as internal control locations are indicated by outlined circles.

In its elementary form, the behavior of the function is described in a state-transition manner. As shown in Figure 1 in case of Power Position Controlled Window, its internal control flow is described via locations dn, stp, and up (with stp being an internal control location as well as an interface location, indicated by the grayed-out line), as well as transitions between these locations. Transitions are influenced by observed signals and influence controlled signals. Furthermore, transitions might be influenced by values of local variables and influence local variables. Thus, if control resides in location stp, value Hi is received via the Bat signal (Bat?Hi), and a value Up is received via the But signal (But?Up), then value Hi is sent via the Mot signal (Mot!Hi) and control is transferred to location up.

As shown in Section 3, a single transition can be understood as the most basic form of a function. Its interface is defined by the observed and controlled signals (and variables) as well as by its start and end locations.

2.1 Decomposing Functionality

The functionality of Power Position Controlled Window shown in Figure 1 can be decomposed in simpler functionalities, addressing special aspects of the combined

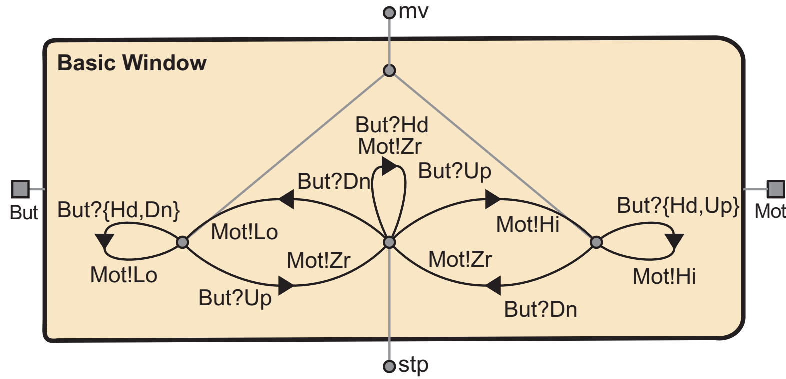

Fig. 2. Basic Window Function

behavior: *power control*, addressing issues of voltage-dependent window movement; *position control*, addressing issues of end position detection; and *basic window control*, addressing issues of button-controlled window movement.

As shown by the white boxes in Figure 3, to obtain an equivalent functional decomposition Power Position Window of function Power Position Controlled Window, five basic functions are used:

(i) Basic Window movement control, moving the window as requested by the interactions of the user,

(ii) Position Check, restricting window movement to positions in between end positions,

(iii) Position Override, halting the window if reaching an end position,

(iv) Power Check, restricting window movement to situations with sufficient initial voltage,

(v) Power Override, disabling window movement if lacking sufficient voltage.

Obviously, all five functions control the motor movement via the Mot signal, interacting to realize the overall behavior. However, their combined behavior does not support a modularization of the behavior of the controller, failing to reflect the separation of concerns into individual functions. Therefore, if restricted to constructive formalisms (like Statecharts [2] or Masaccio [4]), identifying these five functions in a modular fashion is not possible; as a result, ensuring that the overall behavior implements the intended interaction of these functions is requires the use of an additional property-language (e.g., temporal logic).

Figures 2, 4, and 5 show the corresponding basic functionalities.

Function Basic Window provides basic window movement functionality, in form of upward movement caused by a Hi-value for the Mot-signal initiated by a Up-value for the But-signal in location stp; holding (But?Up) or relasing (But?Hd) the button continues the upward movement (Mot!Hi), while changing the button (But?Dn) will stop the movement (Mot!Zr). The functionality for the downward movement is supplied in a similar fashion.

The function is activated and deactivated via interface location stp – corresponding to the internal control location representing a stopped motor – or via interface

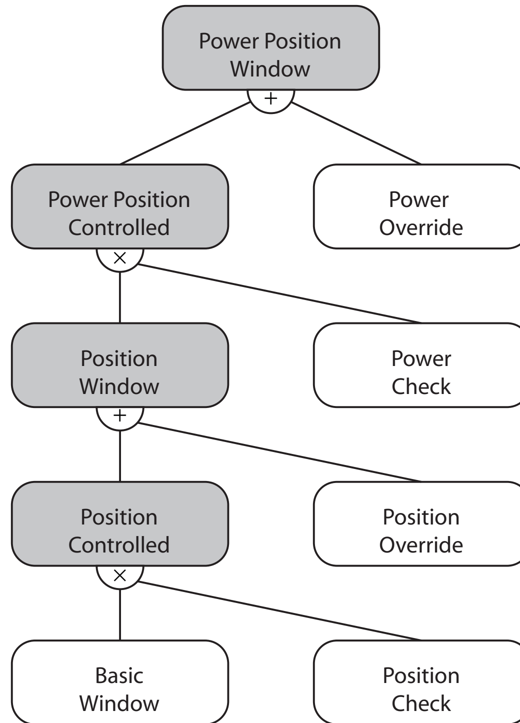

Fig. 3. Functional Decomposition of Power Position Controlled Window

location mv – corresponding to two internal control locations representing either downward or upward movement of the motor – shown at the interface of Basic Window.

Position control as shown in Figure 4, consists of Position Check, ensuring that the motor is restricted to intermediate positions, and Position Override, ensuring that the motor is stopped if an end-position is reached. Once activated by a motor movement (Mot!{Hi,Lo}), Position Check enforces that further movement requires a non-end position (Pos?Of:Mot!{Hi,Lo}) until deactivation (Pos?Of:Mot!Zr). Interface locations stp and mv correspond to a stopped a moving motor. Position Override provides an override functionality to stop a window movement when an end-position signal is detected.

Similarly, power control consists of Power Check, ensuring that starting the motor movement requires sufficient voltage, and Power Override, ensuring that the motor is not activated in case of insufficient voltage.

2.2 Composing Functionality

To obtain the overall behavior of the controller of the power window, the functions introduced in Section 2.1 are combined. Figure 3 also shows, how Power Position Window is composed to obtain an equivalent functionality like Power Position Controlled Window: Basic Window and Position Check are combined by *conjunctive* composition – indicated by × – to function Position Controlled, which in turn is combined by *disjunctive* composition with function *Position Override* to function *Position Window*. Using the same pattern of composition, Position Window is combined by conjunctive composition with Power Check to obtain function Power Posi-

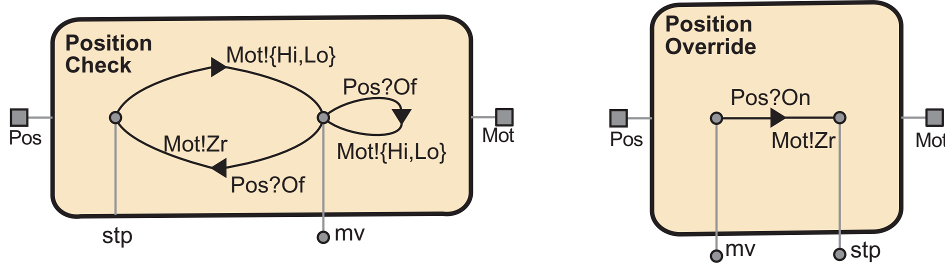

Fig. 4. Position Control Functions

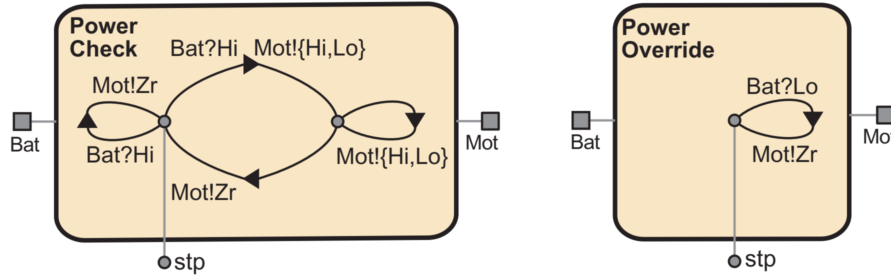

Fig. 5. Power Control Functions

tion Controlled, which in turn is combined by conjunctive composition with Power Override to function Power Position Window.

Intuitively, disjunctive composition corresponds to the *alternative* use of composed functions, while *conjunctive* composition corresponds to the simultaneous use of the composed functions. Obviously, disjunctive composition is not sufficient to obtained the intended functionality, since position control and power control are supposed to restrict basic window movement. Similarly, simple conjunctive composition does not lead to a reasonable behavior, since basic movement, position control, and power control as defined above are in conflict to each other. Therefore, a more sophisticated form of combination is needed, describing the priorities between these (sub-)functions.

Figures 6 and 7 describe these prioritized compositions. As shown in the left-hand side of Figure 6 , at the top level, the Power Position Window is realized by *disjunctive composition* – indicated by a light background used inside the box representing a function – of the Power Override function together with the Controlled Position Window, ensuring that a lack of voltage does result in a blocked window movement. Activation and deactivation of the disjunctive composition Power Position Window via the stp interface location corresponds to the activation and deactivation of either sub-function via stp. Furthermore, as Power Override and Power Position Controlled share the interface location stp, activation may pass from one to other and back. As the signal interfaces (Bat, But, Pos, and Mot) are linked to the corresponding interfaces of the sub-functions, signals are passed between the environment of Power Position Window and the currently activated sub-function.

As Controlled Position Window is obtained by *conjunctive composition* – indicated by a dark background used inside the box representing a function – of Power Check and Position Window, any window movement is only initialized in case of sufficient voltage. Activation and deactivation of the conjunctive composition Power Position

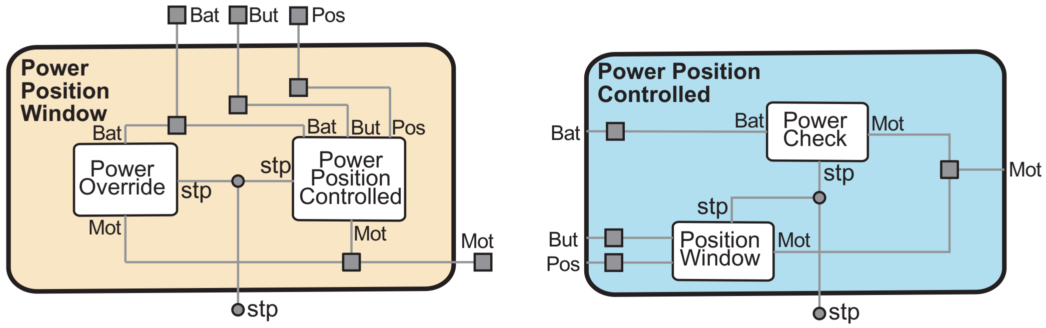

Fig. 6. Power Position Window and Controled Position Window Functions

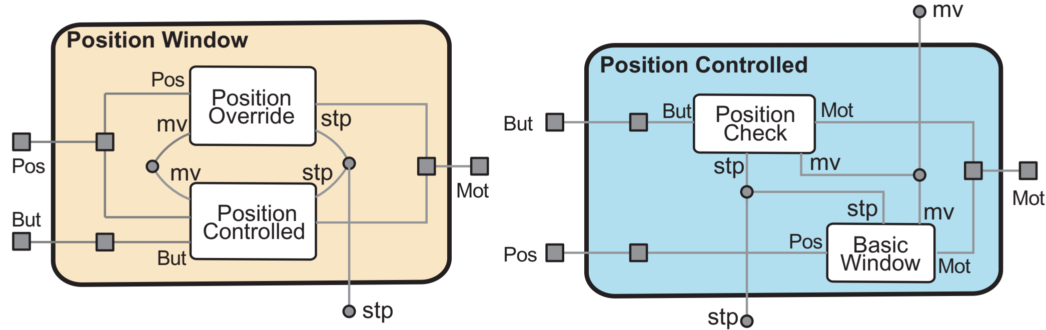

Fig. 7. Position Window and Controled Window Functions

Controlled via interface location stp corresponds to the simultaneous activation and deactivation of both sub-functions via stp. Furthermore, the signal interfaces (Bat, But, Pos, and Mot) of Power Position Controlled Power Check and Position Windoware simultaneously observed and controlled by the sub-functions via their corresponding linked signal interfaces.

A similar construction is applied to ensure position control. As Position Window is realized by disjunctive composition of Position Override and Controled Window, detection of an end position stops the movement of the window. By conjunctive composition of Position Check and Basic Window to form Controled Window, the basic window movement is restricted to intermediate positions of the window.

3 Modeling Functions

In this section, *functions* are introduced as building blocks for the construction of reactive behavior. Since functions are a generalization of components, the difference between functions and components from a methodical perspective is discussed, before giving a formal and compositional definition of functions based on [4].

3.1 Components and Functions

A component communicates with its environment via its interface. A component has a completely specified behavior: for each behavior of the environment (in form of a history of input messages received by the component) its reaction (in terms of histories of output messages) is defined. In approaches like [9], [4], or [12] this is defined as *input enabledness*, *input permissiveness*, or *input completeness*. As

introduced in [12], in contrast to a component, a function behavior needs not be totally defined. For a partial specification, it is possible to have a behavior of the environment where no behavior of the function is defined by the specification.

This distinction plays an important role when combining components or functions. Generally, syntactic restrictions (e.g., disjointness of output interfaces and data states), ensure that the composition of components results in a component (with input total behavior); e.g., [4] uses such a restriction. Due to their more general nature, such a restriction is not required for functions [12]. However, as a result, the combinations of functions (e.g., manual window control, position control) may lead to conflicts (e.g., upward movement of window by manual control vs. stop of movement by position control) resulting in undefined behavior.

To define a formal framework for the construction of functions, in the following subsection we introduce a basic model, and then supply some operators for the construction of complex functions from basic ones.

3.2 Semantics: State-Based Functions

Since functions are intended for the modular specification of components with input complete behavior, as semantical basis in the following we use a formalization similar to [4] to introduce a set Fun of functional descriptions as well as its interpretation; however in contrast to the former, we generalize it to support the description of functions with their partially defined behavior, especially allowing the introduction of new partially by simultaneous combination as defined in Subsection 3.2.4. In the following, Fun corresponds to the set of function terms, starting from basic functions and using operators to form more complex descriptions.

3.2.1 Basics

The structural aspects of a function are defined by its variables Var – used to transfer signal values between the function and its environment – as well as its control locations Loc – used to transfer execution control between the function and its environment.To describe the behavior of a function, we use the concepts

State: A state $s \in \overrightarrow{Var} = Var \to Val$ maps variables to their current values.

Observation: An observation is either a triple (a, t, b) consisting of a finite sequence t of states corresponding to an execution starting at location a and ending at location b, changing variables according to t; or it is a pair (a, t) consisting of a finite sequence t of states, corresponding to a partial execution, starting at location a. Since in the following only continuous functions are introduced, a restriction to finite observations is sufficient.

Behavior: The behavior of a function is the set Obs of all its observations. Consequently, Obs is prefix-closed, i.e., $(a, t, b) \in Obs$ implies $(a, t) \in Obs$, and $(a, t) \in Obs$ implies $(a, s) \in Obs$ for any prefix s of t.

For a state $s : Var \to Val$ with $Var' \subseteq Var$ we use notation $s \uparrow Var'$ for restrictions $(s \uparrow Var')(v) = s(v)$ for all $v \in Var'$. This restriction is extended to sequences of states through point-wise application. For sequences r and t we use the notation $r \circ t$ to describe the concatenation of r and t; furthermore, $\langle\rangle$ describes the empty

sequence.

3.2.2 Basic Functions

The most basic function performs one step of computation. When entered through its entry location, it reads the values of its variables; depending on this variable state, it then changes the variable state by writing new values and terminates by exiting via its exit location. To describe a basic function, we use the notation described in [6]. Figure 4 shows such a basic function Position Override with variables Pos and Mot, entry location mv, and exit location stp. Its behavior is described by a labeled transition from mv to stp with a label consisting of Pos?On and Mot!Zr. The first part of the label states that whenever signal On is received via variable Pos, then the transition is enabled. The second part of the label states that, whenever the transition is triggered, in the next state signal Zr is sent via signal Mot. These parts use a short-hand notation for reading pre-transtion values and writing post-transition values. They correspond to terms $\grave{\mathsf{But}} = \mathsf{Stp}$ and $\acute{\mathsf{Mot}} = \mathsf{Zr}$ using variables $\grave{v}$ with $v \in Var$ for values of v prior to execution of the transition, and variables $\acute{v}$ with $v \in Var$ for values of v after its execution. The interface of Position Override is defined by $Var = \{\mathsf{But}, \mathsf{Mot}\}$, and its locations by $Loc = \{\mathsf{mv}, \mathsf{stp}\}$. As shown in Figure 4, the data flow interface of a function is indicted by boxes connected to the function, while the control flow interface is described by circles connected to it. The transition itself is described by an arrow linking the corresponding control locations.

Abstracting from a concrete graphical representation, a basic function is described as the structure (a, t, b) with entry location a, exit location b, and transition label t over $\overrightarrow{Var} \times \overrightarrow{Var}$. t corresponds to the conjunction of the pre- and the port-part of the label. Its behavior is the set of observations containing all elements

- $(a, \mathit{before} \circ \mathit{after}, b)$
- $(a, \mathit{before} \circ \mathit{after})$
- (a, before)
- $(a, \langle\rangle)$

with $t(before, after)$. Consequently, the behavior of Position Override is the set consisting of all observations $(\mathsf{mv}, \mathit{before} \circ \mathit{after}, \mathsf{stp})$, $(\mathsf{mv}, \mathit{before} \circ \mathit{after})$, $(\mathsf{mv}, \mathit{before})$, and $(\mathsf{mv}, \langle\rangle)$, such that $\mathit{before}(\mathsf{But}) = \mathsf{Stp}$ as well as $\mathit{after}(\mathsf{Mot}) = \mathsf{Zr}$.

3.2.3 Alternative Combination

Similar, e.g., to *Or*-combination used in Statecharts [2], we use alternative combination to describe sequential behavior. The behavior of an alternative combination of two functions corresponds to the behavior of either function. Function Position Window in the left-hand side of Figure 7 shows the alternative combination of functions Position Override and Position Controled. It shares all the structural aspects of either function, and thus uses variables But and Pos, as well as Mot. Furthermore, by means of the common interface location stp, either Position Override or Position Controlled can be activated and deactivated. Furthermore, by means of the shared internal control location mv, activation may be passed from Position Controlled to

Position Override. Formally, the alternative combination of two functions A and B results in a function described by $A + B$ that

- uses the variables of either function: $Var_{A+B} = Var_A \cup Var_B$
- accesses their control locations: $Loc_{A+B} = Loc_A \cup Loc_B$
- exhibits the behavior of either function: $(a, t, b) \in Obs_{A+B}$ if $(a, t \uparrow Var_A, b) \in Obs_A$ or $(a, t \uparrow Var_B, b) \in Obs_B$; $(a, t) \in Obs_{A+B}$ if $(a, t \uparrow Var_A) \in Obs_A$ or $(a, t \uparrow Var_B) \in Obs_B$

Intuitively, the combined function offers observations that can be entered and exit via one of its sub-functions. If the sub-functions share a common entry location, observations of either function starting at that entry location are possible; similarly, if they share a common exit location, observations ending at that common exit location are possible. Obviously, functions $A + B$ and $B + A$ are equivalent in the sense of having the same interface and behavior .

3.2.4 Simultaneous Composition

Besides alternative combination, functions can be combined using simultaneous combination. The behavior of a simultaneous combination of two functions corresponds to the joint behavior of both functions. [1] Function Position Controlled in the right-hand side of Figure 7 shows the simultaneous combination of functions Position Check and Basic Window. Its interface consists of variables But of Position Check and Pos of Basic Window as well as variable Mot of both sub-functions; its locations $Loc = \{\mathsf{stp}, \mathsf{mv}\}$ are the shared locations of these functions. Formally, the simultaneous combination of two functions A and B results in a function described by $A \times B$ that

- use the variables of each function: $Var_{A\times B} = Var_A \cup Var_B$
- accesses their *shared* control locations: $Loc_{A\times B} = Loc_A = Loc_B$
- exhibits the combined behavior of each function: $(a, t, b) \in Obs_{A\times B}$ if $(a, t \uparrow Var_A, b) \in Obs_A$ and $(a, t \uparrow Var_B, b) \in Obs_B$; $(a, t) \in Obs(A \times B)$ if $(a, t \uparrow Var_A) \in Obs_A$ and $(a, t \uparrow Var_B) \in Obs_B$

Intuitively, the combined functions offers observations that can be offered by both functions. Obviously, $A \times B$ and $B \times A$ are equivalent in the sense of exhibiting the same interface and behavior.

3.2.5 Hiding Locations

Hiding a location of a function renders the location inaccessible from the outside. At the same time, when reaching a hidden location the function does immediately continue its execution along an enabled transition linked to the hidden location. In Function Position Window in the left-hand side of Figure 7, control location mv is hidden to enable immediate position override. Formally, by hiding a location l from a function A we obtain a function described by $A\backslash l$ that

- uses the variables of A: $Var_{A\backslash l} = Var_A$

[1] Note that this differs essentially from *And*-composition in Statecharts describing interleaved composition.

- accesses the control locations of A excluding l: $Loc_{A\backslash l} = Loc_A \backslash \{l\}$
- exhibits the behavior of A if entered/exited through locations excluding l and continuing execution at l: $(a, t_1 \circ s_1 \ldots s_{n-1} \circ t_n, b) \in Obs_{A\backslash l}$ if $(a, t_1 \circ s_1, l)$, $(l, s_{n-1} \circ t_n, b) \in Obs_A$ as well as $(l, s_{i-1} \circ t_i \circ s_i, l) \in Obs_A$ for $i = 2, \ldots, n-1$, $t_j \in S^*$, and $s_j \in S$; $(a, t_1 \circ t_2 \circ \ldots) \in Obs_{A\backslash l}$ if $(a, t_1, l)) \in Obs_A$ and $(l, t_i, l) \in Obs_A$ for $i > 1$.

Obviously, $(S\backslash a)\backslash b)$ and $(S\backslash b)\backslash a)$ are equivalent in the sense of exhibiting the same interface and behavior. We write $A\backslash\{a, b\}$ for $(A\backslash a)\backslash b$.

3.2.6 Renaming Locations

Renaming a location of a function changes the interface of the function, possibly unifying control locations. As, e.g., shown in the left-hand side of Figure 2, the distinct control locations corresponding to the upward and downward movement of the window are renamed to the unique control location mv.

Formally, by renaming a location l in a function A to location m we obtain a function described by $A[l/m]$ that

- uses the variables of A: $Var_{A[l/m]} = Var_A$
- accesses the control locations of A excluding l and including m: $Loc_{A[l/m]} = Loc_A \backslash \{l\} \cup \{m\}$
- exhibits the behavior of A after renaming: for $a, b \neq l$, $(a, t, b) \in Obs_{A[l/m]}$ if $(a, t, b) \in Obs_A$ as well as $(a, t) \in Obs_{A[l/m]}$ if $(a, t) \in Obs_A$. Furthermore for $a, b \neq l$, $(a, t, m) \in Obs_{A[l/m]}$ if $(a, t, l) \in Obs_A$ and $(m, t, b) \in Obs_{A[l/m]}$ if $(l, t, b) \in Obs_A$, $(m, t, m) \in Obs_{A[l/m]}$ if $(l, t, l) \in Obs_A$, and finally $(m, t) \in Obs_{A[l/m]}$ if $(l, t) \in Obs_A$.

3.2.7 Hiding Variables

Hiding a variable of a function renders the variable unaccessible from the outside. Formally, by hiding a variable v from a function A we obtain a function described by $A\backslash v$ that

- uses the variables of A excluding v: $Var_{A\backslash v} = Var_A \backslash \{v\}$
- accesses the control locations of A: $Loc_{A\backslash v} = Loc_A$
- exhibits the behavior of A for arbitrary v: $(a, t \uparrow Var_A, b) \in Obs_{A\backslash v}$ if $(a, t, b) \in Obs_A$; $(a, t \uparrow Var_A) \in Obs_{A\backslash l}$ if $(a, t) \in Obs_A$.

Obviously, $(S\backslash v)\backslash w)$ and $(S\backslash w)\backslash v)$ are equivalent in the sense of exhibiting the same interface and behavior. We write $A\backslash\{v, w\}$ for $(A\backslash v)\backslash w$.

3.2.8 Renaming Variables

Renaming a variable of a function changes the interface of the function by changing the name of a signal variable. Here, renaming is restricted to a new name not already contained in the variables of function A. Formally, by renaming a variable v from a function A to w we obtain a function described by $A[v/w]$ that

- uses the variables of A excluding v and including w: $Var_{A[v/w]} = Var_A \backslash \{v\} \cup \{w\}$

- accesses the control locations of A: $Loc_{A\backslash v} = Loc_A$
- exhibits the behavior of A , substituting w for v: $(a, s, b) \in Obs_{A\backslash v}$ if $(a, t, b) \in Obs_A$ and $(a, s) \in Obs_{A\backslash l}$ if $(a, t) \in Obs_A$ where $s \uparrow Var_A\backslash v = t \uparrow Var_A\backslash v$ and $s \uparrow \{w\} = t \uparrow \{v\}$.

4 Applying Functions

As introduced in the previous sections, functions are intended to support the modular construction of complex functionalities in the development process by combining individual pieces of reactive behavior. However, while the descriptive form of general functional descriptions eases the combination and reuse of functions and the reasoning about the overall functionality, for the final implementation of the intended behavior in general more constructive forms of descriptions are used, as provided, e.g., by corresponding tools like or [2], [3], or [6]. As stated in Section 1, these descriptions correspond to a restricted form of functions, avoiding the introduction of partiality and ensuring input enabledness. On this constructive level, input enabledness is either established implicitly by completion (as, e.g., in [6]) or explicitly by analysis (as, e.g., in [3]).

However, to integrate these different applications of functional descriptions are integrated in a function-based development process, the more descriptive and more constructive forms must linked by an implementation relation, introduced in the following.

4.1 Implementation

As stated in Section 1, functions are intended to support the modular construction of complex functionalities in the development process by combining individual pieces of reactive behavior. Thus, the descriptive form of general functional descriptions eases the combination and reuse of functions and the reasoning about the overall functionality. However, for complex behavior it requires to analyze the description for absence of *potential partiality* to ensure the *implementability* of a function.

To ensure that no partiality is contained in the function to be implemented, the function must be

data flow compatible, avoiding to introduce partiality by adding contradicting functionality, and

control flow complete, avoiding to keep partiality by leaving out necessary functionality.

Here, a notion of implementability for a function is used which can be *checked statically*, i.e, without constructing the behavior of a function.

4.1.1 Data Flow Compatibility

For a structured function to be *compatible concerning its data flow*, it must be composed from sub-functions which are compatible concerning the exchange of signals via their variables. To define the compatibility concerning data flow, the variables of a function are classified as *input* and *output variables*. Input variables are used

to transfer signals from the environment to a function, while output variables are used to transfer signals from a function to its environment. The distinction between input and output variables is defined inductively over the structural composition of a function.

For a basic function A described by the structure (a, t, b), the input variables In_A are the variables v that are used as $\grave{v}$ in t. Similarly, the output variables Out_A of A are the variables v that are used as $\acute{v}$ in t. Considering the example of basic function Position Override shown in Figure 4, its variables are defined by $Var = In \cup Out$ with $In = \{\mathsf{But}\}$ and $Out = \{\mathsf{Mot}\}$.

For the alternative combination of two functions A and B, the function described by $A+B$ uses the input and output variables of either function: $In_{A+B} = In_A \cup In_B$, and $Out_{A+B} = Out_A \cup Out_B$. Therefore, in case of function Position Window in Figure 7, it uses input variables But of Position Override and Position Controled, input variable Pos of Position Controled, as well as output variable Mot of and Position Override and Position Controled.

For the simultaneous combination of two functions A and B, the function described by $A \times B$ uses input and output variables: $In_{A\times B} = In_A \cup In_B \backslash Out_{A\times B}$, and $Out_{A\times B} = Out_A \cup Out_B$. Thus, in case of function Position Controlled shown in Figure 7, its data flow interface consists of input ports $In = \{\mathsf{But}\}$ of Position Check and $In = \{\mathsf{Pos}\}$ of Basic Window as well as of their common output port $Out = \{\mathsf{Mot}\}$ of both sub-functions.

For the hiding of a variable v from a function A, we obtain a function described by $A\backslash v$ that uses the input and output variables of A excluding v: $In_{A\backslash v} = In_A \backslash \{v\}$, $Out_{A\backslash v} = Out_A \backslash \{v\}$.

Finally, the hiding and renaming of control locations leaves the input and output variables unchanged: $In_{A\backslash l} = In_A$ and $Out_{A\backslash l} = Out_A$, as well as $In_{A[l/m]} = In_A$ and $Out_{A[l/m]} = Out_A$.

Based on the definition of the input and output ports of a structured function, the concept of *interface compatibility concerning data flow* is defined. Since basic functions as well as hiding and renaming of variables and locations only deal with a single function, no compatibility restrictions are imposed for these constructions.

In contrast, for the alternative combination $A + B$, we require that for two functions A and B

- $In_A \cap Out_B = \emptyset$ and
- $In_B \cap Out_A = \emptyset$

must hold to be alternatively composable;

For the simultaneous combination $A \times B$

Note that – unless we require the interface constraint generally imposed for the composition of components by $(Var_A \backslash In_A) \cap (Var_B \backslash In_B) = \emptyset$ – simultaneous combination of functions may result in output or variable conflicts, leading to the introduction of (additional) partiality in the behavior of the combined functions.

4.1.2 Control Flow Completeness

For a structured function to be *complete concerning its control flow*, it must be composed from sub-functions which are compatible concerning the transfer of control via their locations. To define the completeness concerning control flow, the locations of a function are assigned *entry* and *exit conditions*. Entry conditions are used to describe states that allow the transfer of control a function from its environment, while exit conditions are used to describe states that require the transfer of control from a function to its environment. The assignment of entry and exit conditions is defined inductively over the structural composition of a function.

For a basic function A described by the structure (a, t, b), the entry condition of location a is the set entry of states from $\overrightarrow{\mathsf{Var}_A}$ with $(a, \mathsf{entry}, \mathsf{exit}, b)$ in Obs_A for some exit from $\overrightarrow{\mathsf{Var}_A}$. The entry condition of location b is the empty set of state from $\overrightarrow{\mathsf{Var}_A}$. Symmetrically, the exit condition of location b is the set exit of states from $\overrightarrow{\mathsf{Var}_A}$ with $(a, \mathsf{entry}, \mathsf{exit}, b)$ in Obs_A for some entry from $\overrightarrow{\mathsf{Var}_A}$. The exit condition of location a is the empty set of state from $\overrightarrow{\mathsf{Var}_A}$. Considering the example of basic function $\mathsf{Position\ Override}$ shown in Figure 4, the entry condition of location mv is the set of all states *entry* with $entry(\mathsf{Pos}) = \mathsf{On}$; its exit condition is the empty set. Similarly, the exit condition of stp is the set of all states *exit* with $exit(\mathsf{Mot}) = \mathsf{Zr}$; its entry condition is the empty set.

For the alternative combination of two functions A and B, the function described by $A + B$ uses the union of the entry and exit conditions of locations of either function: $entry^a_{A+B} = entry^a_A \cup entry^a_B$, and $exit^a_{A+B} = exit^a_A \cup exit^a_B$ for joint locations a. For disjoint locations, entry and exit conditions are the entry and exit conditions of either function.

For the simultaneous combination of two functions A and B, the function described by $A \times B$ uses the intersection of the entry and exit conditions of locations of either function: $entry^a_{A\times B} = entry^a_A \cap entry^a_B$, and $exit^a_{A\times B} = exit^a_A \cap exit^a_B$ for joint locations a. For disjoint locations, entry and exit conditions are the entry and exit conditions of either function.

For the hiding of location l from a function A, the function described by $A\backslash l$ uses the entry and exit conditions of function A or its locations: $entry^a_{A\backslash v} = entry^a_A$, and $exit^a_{A\backslash v} = exit^a_A$ for location a from $Loc_{A\backslash l}$.

For the renaming of a location l of a function A to m, the function described by $A[l/m]$ uses the entry and exit conditions of function A, substituting l for m: $entry^a_{A[l/m]} = entry^a_A$, and $exit^a_{A[l/m]} = exit^a_A$ for location a from $Loc_A\backslash\{l\}$; $entry^a_{A[l/m]} = entry^a_A$, and $exit^a_{A[l/m]} = exit^a_A$ for location m.

For the hiding of a variable v from a function A, the function described by $A\backslash v$ uses the restriction of the entry and exit conditions of function A for its locations: $entry^a_{A\backslash v} = entry^a_A \upharpoonright Var_A\backslash\{v\}$, and $exit^a_{A\backslash v} = exit^a_A \upharpoonright Var_A\backslash\{v\}$ for location a from $Loc_{A[v/w]}$.

For the renaming of a variable v from a function A to w, the function described by $A[v/w]$ uses the subsitution of the entry and exit conditions of function A for its locations: $entry^a_{A[v/w]} \upharpoonright Var_A\backslash\{v\} = entry^a_A \upharpoonright Var_A\backslash\{v\}$, and $entry^a_{A[v/w]} \upharpoonright \{w\} = exit^a_A \upharpoonright \{v\}$ for location a from $Loc_{A[v/w]}$; similarly, $exit^a_{A[v/w]} \upharpoonright Var_A\backslash\{v\} = exit^a_A \upharpoonright Var_A\backslash\{v\}$, and $exit^a_{A[v/w]} \upharpoonright \{w\} = exit^a_A \upharpoonright \{v\}$ for location a from $Loc_{A[v/w]}$.

Formally, a function F is said to be *complete* iff for all

- either $\exists b \in Loc_F.(a, t, b) \in Obs_F$,
- or $\forall i \in I.\exists o \in O.(a, t \circ (i, o)) \in Obs_F$

4.2 Refinement

While using the general descriptive form supports a structured development of the overall functionality, a more compact and constructive variant is generally more preferable for its effective implementation.

As shown in the examples, the definition of Power Position Window from the basic functions (Basic Window, Position Check, Position Override, Power Check, Power Override) leads to a more structured description. In contrast, the definition of Power Position Controlled Window is more suited for implementation using state-of-the-art tools. Thus, in a function-based development process, the former should be used in the early stages of defining the function under development, while the latter should be used in the latter stages. However, for a sound and integrated development process, it is furthermore necessary to establish an implementation relation between those functions.

Formally, a function F_1 is said to *implement* a function F_2 iff

- they provide the same closed signal interface: $In_{F_1} = In = In_{F_2}$ and $Out_{F_1} = Out = Out_{F_2}$ and $Var_{F_1} = In \cup Out = Var_{F_2}$
- they provide the same control interface: $Loc_{F_1} = Loc_{F_2}$
- every possible observation of F_1 is also a possible observation of F_2: $Obs_{F_1} \subseteq Obs_{F_2}$

Basically, functional refinement corresponds to standard trace inclusion. Since here continuous reactive systems are considered with simultaneous input/output actions using a signal-based communication with input enabledness, partial execution traces provide a suitable semantical basis. Obviously, this notion of implementation is transitive and reflexive. Furthermore, the operators introduced in Section 3 are monotonic with respect to this implementation relation.

Using this notion of implementation, Power Position Controlled Window is an implementation of Power Position Window and vice versa.

4.3 Proof Support

To effectively use the implementation relation in a sound development process, (automatic) support for the verification of the implementation relation between two functions is necessary. Since the behavior of functions is defined by (possibly infinite) sets of finite traces, and the implementation relation is defined by the inclusion relation over those sets, a trace-based formalism is best-suited.

Therefore, here WS1S (weak second order monadic structure with one successor function) is used to implement automatic proof support. This formalism is, e.g., supported by the modelchecker *Mona* [7]. Using WS1S, functions are specified by predicates over sets of traces. The operators introduced in Section 3 can be directly implemented, allowing a compositional construction of the corresponding

trace sets. Similarly, the implementation relation can be defined as a relation on trace sets. Besides proving the refinement relation between two functions, *Mona* can be used to generate a counter-example for functions violating the refinement relation.

5 Conclusion

The increasing complexity of reactive behavior integrating different interacting functionalities requires a construction process supporting the modular description of individual functions as well as their composition into the overall behavior.

Therefore, we suggest functional modular development using functions as construction units, with transitions as the most basic form, as well as disjunctive and conjunctive composition to combine modules. Offering separation of concern by modular composition of functions, reasoning about the overall behavior is simplified by conjunctive and disjunctive construction of functionalities. Additionally, reuse of modular functionalities is simplified when constructing variants of reactive behavior. Finally, using automatic proof support, the implementation of the integrated modular behavior through a more-constructive form of functional description can be established.

References

[1] Broy, M., I. H. Krüger and M. Meisinger, *A formal model of services*, ACM Trans. Softw. Eng. Methodol. **16** (2007).

[2] Harel, D. and M. Politi, “Modeling Reactive Systems with Statecharts,” MacGraw-Hill, 1998.

[3] Heitmeyer, C. L., *Software Cost Reduction*, in: J. J. Marciniak, editor, *Encyclopedia of Software Engineering*, 2002 .

[4] Henzinger, T. A., *Masaccio: A Formal Model for Embedded Components*, in: *Proceeding of the First International IFIP Conference of Theoretical Computer Science* (2000), pp. 549–563, lNCS 1872.

[5] Hoare, C. A. R., *Communicating sequential processes*, Communications of the ACM **26** (1983), pp. 100–106.

[6] Huber, F., B. Schätz and G. Einert, *Consistent Graphical Specification of Distributed Systems*, in: *Industrial Applications and Strengthened Foundations of Formal Methods (FME'97)*, LNCS 1313 (1997), pp. 122–141.

[7] Klarlund, N. and A. Møller, “MONA Version 1.4 User Manual,” BRICS, Department of Computer Science, University of Aarhus (2001).

[8] Lamport, L., “Specifying Systems,” Addison-Wesley, 2002.

[9] Lynch, N. and M. Tuttle, *An Introduction to Input/Output Automata*, CWI Quarterly **2** (1989), pp. 219–246.

[10] Milner, R., “CCS - A Calculus for Communicating Systems,” Lecture Notes in Computer Science **83**, Springer Verlag, 1983.

[11] Schätz, B., *Building Components from Functions*, in: *Proceedings 2nd International Workshop on Formal Aspects fo Component Software (FACS'05)*, Macao, 2005.

[12] Schätz, B. and C. Salzmann, *Service-Based Systems Engineering: Consistent Combination of Services*, in: *Proceedings of ICFEM 2003, Fifth International Conference on Formal Engineering Methods* (2003), lNCS 2885.