

Building Components from Functions

Bernhard Schätz

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

Abstract

In the domain of embedded software systems the increasing complexity of the functionality as well as the increase in variations caused by product lines requires a modular design process, separating function-based and component-based design. As a consequence, functional integration becomes a central task in the development process, to avoid unforeseen interaction. While currently functional integration often is delayed to module integration, leading to a late detection of interactions, here we suggest a methodical approach to the early integration of functions to construct a logical component-oriented architecture.

1 Introduction

Substituting control hardware by software has led to a new level of customizability in the domain of embedded systems; especially in application domains like automotive industry with customer-related areas like comfort electronics, the possibility to offer variant combinations of *functions* (or *features*) like power window, child protection, or blocking-detection, has become an important market factor. To meet the demand for increasing numbers of variations combined with a reduced time-to-market, a development process explicitly supporting the combination of functionalities into components is necessary. Therefore, domain-specific engineering approaches like [12] explicitly distinguish between a *Functional Architecture* and a *Logical Architecture*.

While a modular description of functionality enables a flexible combination of functions, it requires an explicit integration of these functions to deployable components to avoid unwanted interactions of functions. However, approaches like [12] do not provide a clear distinctions between functions and components. Currently, functions are often treated like components, requiring the manual introductions of additional ‘coordinator’ components to ensure the compat-

ibility of the integrated functions.¹ In general, testing is used as the main method of quality assurance. Thus, unwanted interactions of functions are often detected after shipment, even within functions with safety-related aspects, as, e.g., central locking. In the approach presented here, we

- introduce of common formalism for functions and components
- define constraints for the integration of functions into components
- introduce a mechanism for an integration on the descriptive level respecting these constraints

To that end, in Section 2, we introduce a state-based description for functional and logical architectures, and define a suitable interpretation along the lines of [4] and [11]. In Section 3 we introduce structural and behavioral constraints for the integration. In Section 4 we give a mechanism for integrating functions on the descriptive level, respecting structural and behavioral constraints. Finally, in Section 5, we address tool support issues, and relate it to other approaches dealing with the integration of functions.

2 Preliminaries

Since we are interested in supporting a constructive approach to build components from modular pieces of behavior, in this section we introduce building blocks called *functions*. To that end, we first informally contrast functions to components from a methodical perspective; we then give a formal and compositional definition of functions based on [4].

2.1 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 [6], [4], or [11] this is defined as *input enabledness*, *input permissiveness*, or *input completeness*. As introduced in [11], 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

¹ See, e.g., [7] for the construction of such a coordinator component.

[11]. However, as a result, the combinations of functions (e.g., manual control function of a car window, switch-off function at final position of window) may lead to conflicts (e.g., upward movement of window by manual control vs. stop of movement by switch-off a highest position) 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.

2.2 Semantics: State-Based Functions

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 introduced in [5]²; they support the treatment of undefined behavior along the lines introduced in [11]. 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 2.2.4. In the following, Fun corresponds to the set of function terms, starting from basic functions and using operators to form more complex descriptions.

2.2.1 Basics

The structural aspects of a function are defined by its input ports In , its output ports Out – with $In \cap Out = \emptyset$ –, its variables Var – with $In \cup Out \subseteq Var$ as special monitored and controlled variables – as well as its control locations Loc . To describe the behavior of a function, we use the notions

State: A state $s \in S = Var \rightarrow Val$ maps variables to their current values.³

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 an infinite sequence t of states, corresponding to a non-terminating execution starting at location a .

Behavior: The behavior of a function is the set Obs its observations.

The most simple function is the trivial function nil with $Var(nil) = Loc(nil) = Obs(nil) = \emptyset$. For a state $s : Var \rightarrow Val$ with $Var' \subseteq Var$ we use notation $s \upharpoonright Var'$ for restrictions $(s \upharpoonright Var')(v) = s(v)$ 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 .

² [5] defines the data interface via ports, the control interface via connectors.

³ For reasons of brevity, we assume that all ports and variables are of the same type.

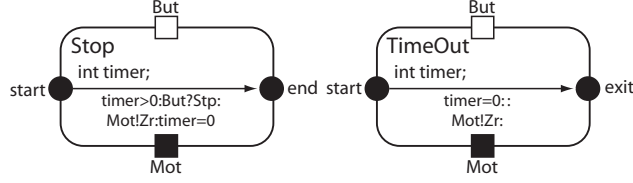


Fig. 1. Basic Functions Stop and Abort

2.2.2 Basic Functions

The most basic function performs only one step of computation. When entered through its entry location, it reads the currently available input; it produces some output, depending on its current variable state and the available input, and changes its variable state; it then terminates by exiting via its exit location. To describe a basic function, we use the notation described in [5]. Figure 1 shows such a basic function **Stop** with input port **But**, output port **Mot**, variable **timer**, entry location **start**, and exit location **end**. Its behavior is described by a labeled transition from **start** to **end** with a structured label $\text{timer} > 0 : \text{But?Stp} : \text{Mot!Zr} : \text{timer} := 0$. The first part of the label, its pre-part, states that whenever the data condition $\text{timer} > 0$ is true and signal **Stp** is received via port **But**, then the transition is enabled. The second part of the label, its post-part, states that, whenever the transition is triggered, in the next state signal **Zr** is sent via output port **Mot** and the data-condition $\text{timer} = 0$ is established. These parts correspond to terms $\text{timer} > 0 \wedge \text{But} = \text{Stp}$ and $\text{Mot}' = \text{Zr} \wedge \text{timer}' = 0$ with unprimed variables from Var for values prior to execution of the transition, primed variables from Var' for values after its execution. The interface of **Stop** is defined by $In = \{\text{But}\}$, $Out = \{\text{Mot}\}$, its variables by $Var = \{\text{timer}\} \cup In \cup Out$, and its locations by $Loc = \{\text{start}, \text{end}\}$. Abstracting from a concrete graphical representation, a basic function is described as the structure $(a, pre, post, b)$ with entry location a , exit location b , pre-condition pre over S , and post-condition $post$ over $S \times S$.⁴ The behavior of **Stop** is the set consisting of all observations $(\text{start}, t, \text{end})$ such that $t = \text{before} \circ \text{after}$ is a sequence of two states before and after with $\text{before}(\text{timer}) > 0 \wedge \text{before}(\text{But}) = \text{Stp}$ as well as $\text{after}(\text{timer}) = 0 \wedge \text{after}(\text{Mot}) = \text{Zr}$. Note that, as shown in case of function **Abort** in Figure 1, a transition label may be underspecified, e.g., by leaving out the input-condition and the post-condition.

2.2.3 Alternative Combination

Similar, e.g., to *Or*-combination used in Statecharts [3], we use alternative combination to describe sequential behavior. The behavior of an alternative

⁴ pre and $post$ are obtained from the corresponding terms by interpretation over Var , and (Var, Var') , resp.

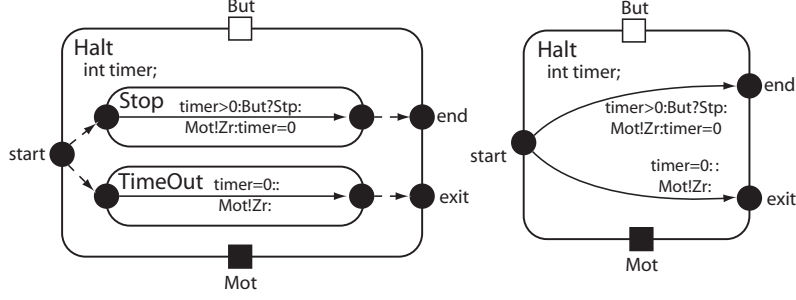


Fig. 2. Alternative Combination Halt and Simplified Representation

combination of two functions corresponds to the behavior of either function. Figure 2 shows the alternative combination **Halt** of functions **Stop** and **Abort**. It shares all the structural aspects of either function, and thus uses input port **But**, output port **Mot**, and variable **timer**. Furthermore, by means of the common entry location **start**, either **Stop** or **Abort** can be executed. Due to disjoint exit locations, **Halt** is either terminated through exit location **end** of **Stop** or exit location **exit** of **Abort**. Formally, the alternative combination of two functions A and B results in a function described by $A + B$ that

- uses the input and output ports as well as variables of each function: $In(A + B) = In(A) \cup In(B)$, $Out(A + B) = Out(A) \cup Out(B)$, $Var(A + B) = Var(A) \cup Var(B)$
- 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. To ensure a well-defined function, we require that for two functions A and B conditions $In(A) \cap Out(B) = \emptyset$ and $In(B) \cap Out(A) = \emptyset$ must hold to be alternatively composable. Obviously, functions $A + B$ and $B + A$, $A + A$ and A , as well as $A + nil$ and A are each equivalent in the sense of having the same interface and behavior .

2.2.4 Simultaneous Composition

Besides alternative combination, functions can be combined using simultaneous combination similar, e.g., to *And*-composition in Statecharts to describe parallel execution. The behavior of a simultaneous combination of two functions corresponds to the joint behavior of both functions. Figure 3 shows the simultaneous combination **Hold** of functions **Stop** and **Abort**. Its interface

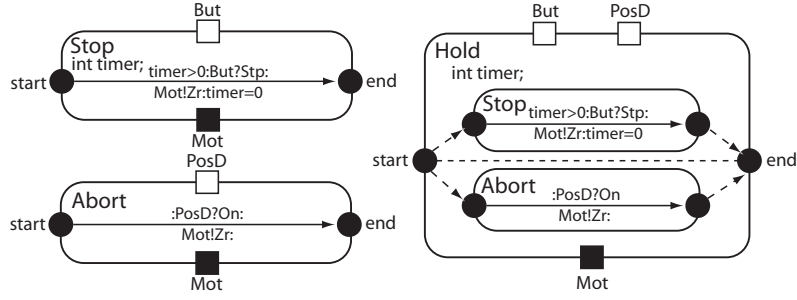


Fig. 3. Stop and Abort and their Simultaneous Combination

consists of input ports $In = \{\text{But}, \text{PosD}\}$ of **Stop** and **Abort** as well as output port $Out = \{\text{Mot}\}$; its locations $Loc = \{\text{start}, \text{end}\}$ are the shared locations of these functions; its variable $Var = \{\text{timer}\}$ is the corresponding variable of **Stop**. Formally, the simultaneous combination of two functions A and B results in a function described by $A \mid B$ that

- use the input and output ports as well as variables of each function: $In(A \mid B) = In(A) \cup In(B) \setminus Out(A \mid B)$, $Out(A \mid B) = Out(A) \cup Out(B)$, $Var(A \mid B) = Var(A) \cup Var(B)$
- accesses their *shared* control locations: $Loc(A \mid B) = Loc(A) = Loc(B)$
- exhibits the combined behavior of each function: $(a, t, b) \in Obs(A \mid B)$ if $(a, t \uparrow Var(A), b) \in Obs(A)$ and $(a, t \uparrow Var(B), b) \in Obs(B)$; $(a, t) \in Obs(A \mid B)$ 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. To ensure a well-defined function, we require condition $Loc(A) = Loc(B)$ for functions A and B to be simultaneously composable. Note that unless we require the standard interface constraint $(Var(A) \setminus In(A)) \cap (Var(B) \setminus In(B)) = \emptyset$ imposed for the composition of components, 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. Obviously, $A \mid B$ and $B \mid A$ as well as $A \mid A$ and A are each equivalent in the sense of exhibiting the same interface and behavior.

2.2.5 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. Formally, by hiding a location l from a function A we obtain a function described by $A \setminus l$ that

- uses the input and output ports and variables of A : $In(A \setminus l) = In(A)$, $Out(A \setminus l) = Out(A)$, $Var(A \setminus l) = Var(A)$

- accesses the control locations of A excluding l : $Loc(A \setminus l) = Loc(A) \setminus \{l\}$
- exhibits the behavior of A if entered/exited through locations excluding l and continuing execution at l : $(a, t_1 \circ \dots \circ t_n, b) \in Obs(A \setminus l)$ if $(a, t_1, l), (l, t_n, b) \in Obs(A)$ as well as $(l, t_i, l) \in Obs(A)$ for $i = 2, \dots, n - 1$; $(a, t_1 \circ t_2 \circ \dots) \in Obs(A \setminus l)$ if $(a, t_1, l) \in Obs(A)$ and $(l, t_i, l) \in Obs(A)$ for $i > 1$.

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

2.2.6 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 $A \setminus v$ that

- uses the input and output ports and variables of A excluding v : $In(A \setminus v) = In(A) \setminus \{v\}$, $Out(A \setminus v) = Out(A) \setminus \{v\}$, $Var(A \setminus v) = Var(A) \setminus \{v\}$
- accesses the control locations of A : $Loc(A \setminus v) = Loc(A)$
- exhibits the behavior of A for arbitrary v : $(a, t \uparrow Var(A), b) \in Obs(A \setminus v)$ if $(a, t, b) \in Obs(A)$; $(a, t \uparrow Var(A)) \in Obs(A \setminus l)$ if $(a, t) \in Obs(A)$.

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

3 Structure and Behavior

As mentioned in Section 1, the presented approach supports the constructive composition of complex functionality from simpler functions. Thus, it is necessary to support the composition on the *descriptive* as well as on the *behavioral* level. In the following we define criteria that ensure that a composition of functions maintains both the structure and the behavior as much as possible. For that purpose, we use a *structural* mapping between function descriptions, taking function (sub-)terms out of Fun to function (sub-)terms.

3.1 Maintaining Structure

Especially in the description of embedded functions, hierarchic descriptions play an important role: sub-functions are often identified with modes of operations; entering and leaving those modes corresponds to activating and terminating the associated functions. Thus, for the practical application during the explicit composition of functions the hierarchical structure of these functions should be maintained.

Therefore, we use the existence of a structural mapping between the description of the resulting composition and the descriptions of the composed functions as an additional constraint for the creation of such an explicit compo-

sition. While this approach also carries over to functions using simultaneous composition as structural element, for reasons of brevity here we focus on alternative composition. To define such a structural constraint, we use the concept of *structural integration* of one description of a function into another.

Definition 3.1 (Structural Integration) The description of a function A is called *structurally integrated* within the description of a function C if a mapping $f : Fun \cup Loc \rightarrow Fun \cup Loc$ exists with

- $f(B + D) = f(B) + f(D)$ for all function terms B and D
- $f(B \setminus l) = f(B) \setminus f(l)$ for all function terms B and all locations l
- $f(B \setminus v) = f(B) \setminus v$ for all function terms B and all variables v
- $f(a, pre, post, b) = (f(a), pre', post', f(b))$ for basic functions $(a, pre, post, b)$ and some pre' and $post'$

with $A = f(C)$ using the equivalences of Subsection 2.2.3 and $Loc(A) = f(Loc(C))$. ◦

Intuitively, a description of a function A is structurally integrated within the description of a function C , if the structure of A in form of hierarchy and composition can be gained by structural abstraction from the structure of C , i.e., by removing elements from C and reordering the remainder according to the equivalences. The **Manual** control function with the graphical representation shown in Figure 4 is structurally integrated into the description of function **Window** in Figure 6. The corresponding mapping is obtained by projecting all locations to their second part, e.g., $hi \times up$ to hi ; furthermore, the labels of basic functions are obtained by removing the parts related to **PosU** or **PosD**, e.g., $:PosU?On,But?Stp:Mot!Zr$ is cut down to $:But?Stp:Mot!Zr$.

3.2 Maintaining Behavior

Obviously, maintaining the structure of its constituting functions is only one aspect when constructing an explicit description of the combination of the descriptions of two functions; furthermore, the behavior of each of the functions integrated in that combined description must be maintained in the behavior associated with the combined description. Here, we use a version similar to the one introduced [11]; it can also be checked mechanically, but in contrast to the former it offers better scalability for non-toy-size systems of functions since it does not require an explicit construction of the complete state space of the system. To that end, we use a stronger version with an abstracted version of the state space.

Definition 3.2 (Full Integration) The description of a function A is called a *fully integrated* within the description of a function C if there exists a mapping f structurally integrating A into C , and additionally for all pre_1, \dots, pre_n as well as all $post_1, \dots, post_n$ with $f(a_i, pre_i, post_i, b_i) = (a, pre, post, b)$ it holds that $pre \wedge post \Leftrightarrow (pre_1 \wedge post_1) \vee \dots \vee (pre_n \wedge post_n)$. ◦

Intuitively, full integration ensures that the elements of C removed during structural integration do not influence the behavior of A . Thus, e.g., basic functions leading from $\text{hi} \times \text{up}$ to $\text{stop} \times \text{idle}$ with labels $(\text{PosD?On} \wedge \text{But?Stp}, \text{Mot!Zr})$ as well as $(\text{PosD?Of} \wedge \text{But?Stp}, \text{Mot!Zr})$ of function `Window` of Figure 6 are equivalent to the basic function leading from up to idle with label $(\text{But?Stp}, \text{Mot!Zr})$ in function `Manual` of Figure 4, since the corresponding signal is either `On` or `Of`.

4 Integrating Functions

As functions describe modules of behavior, their combination is the essential part of the design of a functional architecture; while alternative combination is used to model the activation/deactivation of functions, simultaneous combination is used to model concurrently active functions. As mentioned in Section 2, the simultaneous combination of functions does not correspond to the composition of components since

- functions may share variables including output ports, while components may only share input ports,
- and as a result combined functions may exhibit undefined behavior where their constituting sub-functions do not, e.g., due to output conflicts.

As introduced in [11], functions are *consistent* if no new partiality is introduced by their (simultaneous) combination. In the semantic setting introduced in Section 2, consistency can be defined as follows.

Definition 4.1 (Consistency) Functions A and B are called *consistent* if $\{(a, t \uparrow \text{In}(X), b) \mid (a, t, b) \in \text{Obs}(X)\} \subseteq \{(a, t \uparrow \text{In}(X), b) \mid (a, t) \in \text{Obs}(A \mid B)\}$ and $\{(a, t \uparrow \text{In}(X)) \mid (a, t) \in \text{Obs}(X)\} \subseteq \{(a, t \uparrow \text{In}(X)) \mid (a, t) \in \text{Obs}(A \mid B)\}$, for $X \in \{A, B\}$. \circ

Due to the structural constraints imposed for the composition of components, components are consistent by construction. Therefore, when moving from the functional design phase to the architectural design phase

- the synchronous combinations of functions not corresponding to architectural compositions must be substituted,
- undefined behavior introduced by conflicts in the combination must be identified.

In the following subsection, we introduce a constructive approach to resolve a description constructed using synchronous combination while maintaining as much structure and behavior as possible, basically using the product construction for automata. Furthermore, we show how to identify possible conflicts that may cause additional undefined behavior.

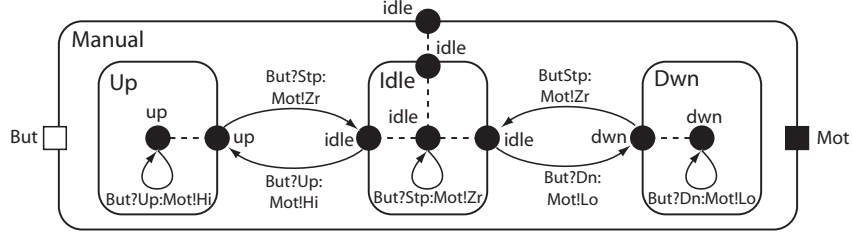


Fig. 4. Manual Control Function

4.1 Unfolding a Combination

Subsection 2.2.4 basically defines simultaneous composition as the product of the behaviors of the combined functions; similarly, approaches from [2] or [11] uses the product construction on a (state-based) *semantical model* to support mechanical analysis of functional descriptions. In contrast, here we are rather interested in using a mechanism on the *notational level* to integrate descriptions of function. Nevertheless, we use the product construction to construct an integrated, ‘unfolded’ version of simultaneously combined functions.

To demonstrate the basic principles of function integration, we use a simple example from automotive chassis electronics; window control often depends on the class of car or national regulations; therefore its final functionality is often constructed from basic functions.

Figure 4 shows the control function **Manual** for manual control with sub-functions **Up**, **Idle**, and **Down**. Initially in **Idle** with stopped window movement **Mot!Zr**, upward and downward movement is initiated via direction buttons **But?Up** and **But?Dn**, resulting in a corresponding **Mot!Hi** and **Mot!Lo** signal. The movement is maintained in functions **Up** and **Down** with pressed buttons, until deactivated via **But?Stp**. Figure 5 shows the control function **Position** for position control with sub-functions **Hi**, **Stop**, and **Low**. Initially in sub-function **Idle** with stopped window movement **Mot!Zr**, upward or downward movement causes a change to functions **Hi** or **Low**, with window movement checked for absence of the signal for end positions **PosU?Of** and **PosD?Of**. Termination of the functions results in stopping the movement **Mot!Zr**.

While both functions **Manual** and **Position** cover one part of the functionality of controlling the window, we are interested in defining a common functionality for both aspects, corresponding to their simultaneous combination. As, however, both functions share **Mot** as common output port, this combination must be adapted when moving from a functional-based to component-based architecture. Thus, we unfold these functions into a control function **Window**.

As mentioned above, when unfolding a simultaneous combination of functions, we want to maintain both structure and behavior. In the following we show the *hierarchic product* of two functions can be used to construct an unfolded description of those functions. For reasons of brevity, we use n-ary

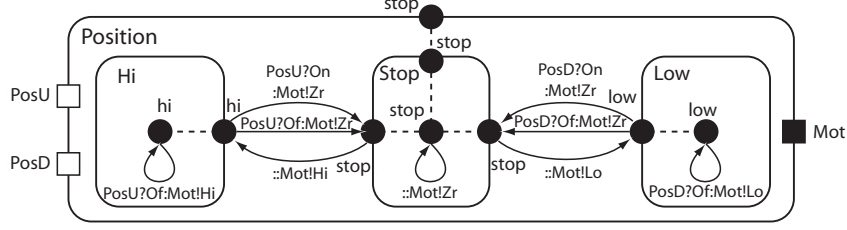


Fig. 5. Position Control Function

variants of the operators introduced in Subsection 2.2:

- Definition 4.2 (Structural Unfolding)** A description of a function F is called the *structural unfolding* of the description of two functions F_1 and F_2 if there exists a mapping $U : Fun \times Fun \rightarrow Fun$ with $F = U(F_1, F_2)$ and⁵
- (i) $U(F_{1,1} + \dots + F_{1,m}, F_{2,1} + \dots + F_{2,n}) = U(F_{1,1}, F_{2,1}) + \dots + U(F_{1,m}, F_{2,n})$
 - (ii) $U(F_1 \setminus \{a_1, \dots, a_m\}, F_2 \setminus \{b_1, \dots, b_n\}) = U(F_1, F_2) \setminus \{a_1 \times b_1, \dots, a_m \times b_n\}$
for locations a_1, \dots, a_m and b_1, \dots, b_n
 - (iii) $U(F_1 \setminus \{v_1, \dots, v_m\}, F_2 \setminus \{w_1, \dots, w_n\}) = U(F_1, F_2) \setminus \{v_1, \dots, v_m, w_1, \dots, w_n\}$
for variables v_1, \dots, v_m and w_1, \dots, w_n
 - (iv) $U((a_1, pre_1, post_1, b_1), (a_2, pre_2, post_2, b_2)) = (a_1 \times a_2, pre_1 \wedge pre_2, post_1 \wedge post_2, b_1 \times b_2)$
 - (v) $U(F_1, F_2) = nil$, otherwise

◦

Intuitively, the structural unfolding is obtained by construction of the product of the functions on each level of hierarchy (i, ii, iii), introducing product locations $a \times b$ for internal locations; basic functions are integrated by conjunction of their pre- and post-conditions (iv); incompatible levels are ignored. (v).

As shown in the description of the unfolded functions in Figure 6, we obtain products of sub-functions $Hi \times Up$, $Stop \times Idle$, and $Low \times Dwn$, including their locations $hi \times up$, $idle \times stop$, and $low \times dwn$. Additionally, we obtain products of basic functions, e.g., $:PosU?Of, But?Up:Mot!Hi$ looping from $hi \times up$ as the product of $:But?Up:Mot!Hi$ and $:PosU?Of:Mot!Hi$ looping from hi and up . By constructing the unfolded function $Window$, the hierarchic structure of $Manual$ and $Position$ was maintained according to definition 4.2.

However, besides maintaining the structure, the unfolded function must also correspond to the simultaneous combination of $Manual$ and $Position$. By construction, if F_1 and F_2 are structurally integrated into the structural unfolding F , the overall behavior remains unchanged, i.e., $obs(F) = obs(F_1 \mid F_2)$. As mentioned in Subsection 3.1, the corresponding mapping from the unfolded function to its constituting functions is obtained by projecting all product locations to their corresponding part, and furthermore removing those parts of

⁵ For sake of brevity, we assume a homogeneous form of hierarchies as well as uniqueness of hidden variables, leading to a shorter definition than in the more general case.

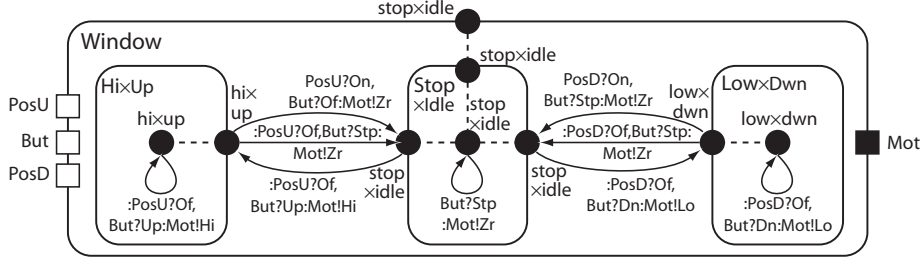


Fig. 6. Simplified Window Control Function

the labels of basic functions that are added by the other function.

Thus, from a development point of view, by unfolding a simultaneous combination we can adapt functional descriptions that do not respect the (structural) restrictions for component composition, without changing the overall behavior. As a result, unfolding helps to simplify the transition from the functional design to the component based design in the development process.

Obviously, the construction of F leads to a functional description that can be considerably simplified: due to clause [iv](#) of [Definition 4.2](#), the basic sub-functions of F are obtained by conjunction of the corresponding basic sub-functions of F_1 and F_2 . Thus, e.g., when combining the basic functions (dwn, (But?Dn, Mot!Lo), dwn) of function [Manual](#) and (hi, (PosU?Of, Mot!Hi), hi) of function [Position](#), this results in (hi \times dwn, (PosU?Of \wedge But?Dn, Mot!Lo \wedge Mot!Hi), hi \times dwn). As Mot!Lo \wedge Mot!Hi requires that at port Mot simultaneously signals Hi and Lo are sent, the combined basic function is not satisfiable since Hi \neq Lo. Therefore, this basic function does not contribute to the overall behavior of Window. To simplify the description of the unfolded function, sub-functions that do not contribute to the behavior of the system are removed from the description. By iteratively removing basic functions with unsatisfiable *pre* \wedge *post*, unreachable functions, or empty functions, a simplified version – as already shown in [Figure 6](#) – is obtained without changing its behavior.

Note that here we only use a *local criterion* for the detection of conflicts: we analyze the satisfiability of a transition without considering the actual state space of the combined functions. Obviously, local satisfiability is a necessary prerequisite of global satisfiability; as thus local unsatisfiability is a sufficient criterion for global unsatisfiability, the strategy of simplification is safe, but may miss unsatisfiable transitions.

4.2 Detecting Conflicts

As shown in the previous subsection, the construction of the product automaton in general leads to the introduction of non-executable transitions, which were removed from the description of the combined functions without changing the behavior. However, these conflicts may also be the cause for a lack of

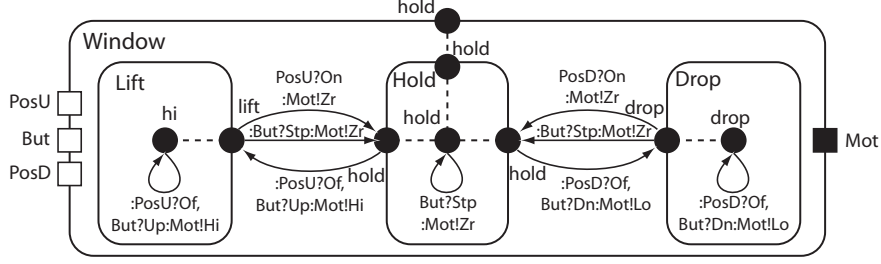


Fig. 7. Simplified Window Control Function with Resolved Conflicts

consistency of two combined functions, as described in Definition 4.1. Therefore, we are interested in the detection of those conflicts that do change the behavior of the combined system.

To detect those conflicts, we make use of the mapping used in the previous subsection to establish the structural integration of the constituting functions into the unfolded function, ensuring that the behavior of the unfolded function does indeed correspond to the behavior of the simultaneously combined functions. By checking that additionally this mapping establishes a full integration as defined in Definition 3.2, the absence of conflicts can be ensured.

To that end, as illustrated in Subsection 3.2, we check whether the label of the basic functions of **Manual** and **Position** are equivalent to their counterparts of **Window** defined by the mapping of the structural integration. When, e.g., relating the basic function leading from **up** to **up** with label $(\text{But?Up}, \text{Mot!Hi})$ in function **Manual** of Figure 4 to its counterpart leading from $\text{hi} \times \text{up}$ to $\text{hi} \times \text{up}$ with label $(\text{PosD?On} \wedge \text{But?Up}, \text{Mot!Hi})$ of the simplified function **Window** of Figure 6, a non-equivalence is detected. This is due to the conflict between $(\text{But?Up}, \text{Mot!Hi})$ in function **Manual** and $(\text{PosU?On}, \text{Mot!Zr})$ in function **Position**, leading to the elimination of the corresponding product function.

By changing the design through adding corresponding new basic functions for these conflicts, a complete description for the **Window** control function can be obtained, as shown in Figure 7 (using new location names).

4.3 Establishing Completeness

As mentioned in Section 2.1, component behavior is generally expected to be completely defined. Thus, supporting the detection of partiality additionally eases the transition to the component-based architecture. To that end, we use an adaption of the completeness check in [11]: A function description F is considered *locally complete* if

$$\forall s \in S. \exists s' \in S. (\text{pre}_1(s) \wedge \text{post}_1(s, s')) \vee \dots \vee (\text{pre}_n(s) \wedge \text{post}_n(s, s'))$$

for all basic sub-functions $(a, \text{pre}_i, \text{post}_i, b_i)$ of F with a common entry location a in F . Similar to the approach used in [8], this establishes a sufficient condition for global completeness, enabling a safe and scalable check.

5 Conclusion and Related Work

The main contributions of the approach presented here target the constructive transition from function-based to component-based descriptions of systems; especially the presented approach

- illustrates a mechanism for an integration on the descriptive level
- introduces a corresponding mechanism to detect possible conflicts of simultaneously combined functions

with a focus on scalability.

5.1 Tool Support

To ease transition from the function- to the component-based architecture of a system using the approach presented here, tool support is needed, both for the unfolding of a combined description including the construction of the mapping used in the structural integration, as well as for the detection of conflicts.

Using the framework of `AUTOFOCUS`, a user-guided merging of state-based function descriptions has been developed, currently limited to non-hierarchical descriptions [10]. This merging includes checking for conflicts when merging the labels of basic functions, however restricted to a limited set of simplification strategies when checking the equivalence of conditions.

As those weak simplifications lead to less compacted versions of the unfolded descriptions as well as to more undecided conflicts, a stronger validity checker must be applied. Currently, `CVCL` [1] is applied to check unsatisfiability for the simplification, the validity of the equivalence condition for full integration, and the local completeness of a description. Due to the expressiveness of the description formalism for transition labels, the satisfiability of a transition is generally not decidable. In the context of simplifying a product automaton this does not pose an essential problem - we only obtain a less compact but semantically equivalent description by maintaining undecided cases. Similarly, during conflict detection, undecided cases are treated as possible conflicts, leading to more falsely identified conflicts.

5.2 Related Work

The combination of functions has traditionally been studied in the context of feature integration, e.g., [2]. However, those and other approaches like [8] focus mainly on the *semantical level* and *analytical techniques*. Here, in contrast, we are rather interested in supporting the modular development of control functions on the *descriptive level*; furthermore, we introduce a *constructive approach* that supports the developer in building component descriptions from a collection of functions. Other notationally oriented approaches

like [9] focus on the support of non-simultaneous composition. Finally, those approaches like [2] and [11] perform a precise analysis of the system under development, leading to non-scalability; in contrast, here, we use a limited technique ensuring correct development but supplying sufficient scalability in practical applications.

References

- [1] Barrett, C. and S. Berezin, *CVC Lite: A New Implementation of the Cooperating Validity Checker*, in: *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, 2004.
- [2] Calder, M. and E. H. Magill, “Feature Interaction in Telecommunication and Software Systems IV,” IOS Press, 2000.
- [3] Harel, D. and M. Politi, “Modeling Reactive Systems with Statecharts,” MacGraw-Hill, 1998.
- [4] Henzinger, T. A., *Masaccio: A Formal Model for Embedded Components*, in: *Proceeding of the First International IFIP Conference of Theoretical Computer Science (2000)*, LNCS 1872.
- [5] 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).
- [6] Lynch, N. and M. Tuttle, *An Introduction to Input/Output Automata*, CWI Quarterly **2** (1989).
- [7] Mutz, M., M. Huhn, U. Goltz and C. Krömke, *Model Based System Development in Automotive*, in: *Proceedings of the SAE 2002 World Congress*, Detroit, 2002.
- [8] Park, D. Y. W., J. U. Skakkebæk, M. P. E. Heimdahl, B. J. Czerny and D. L. Dill, *Checking properties of safety critical specifications using efficient decision procedures*, in: *Proc. Workshop on Formal Methods in Software Practice*, 1998.
- [9] Prehofer, C., *Feature-oriented programming: A new way of object composition.*, *Concurrency and Computation: Practice and Experience* **13** (2001).
- [10] Schätz, B., P. Braun, F. Huber and A. Wisspeintner, *Checking and Transforming Models with AutoFOCUS*, in: *Engineering of Computer-Based Systems ECBS’05* (2005).
- [11] 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.

- [12] Thurner, T., J. Eisenmann, U. Freund, R. Geiger, M. Haneberg, U. Virnich and S. Voget, *The EAST-EEA Project - A Middleware Based Software Architecture for Networked Electronic Control Units in Vehicles*, VDI Berichte 1 (2003), (In German).