T U M

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

Motivation and Formal Foundations of a Comprehensive Modeling Theory for Embedded Systems

Alexander Harhurin, Judith Hartmann and Daniel Ratiu

TUM-I0924
September 09

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

TUM-INFO-09-I0924-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

About the Document

In this document, we present a *comprehensive modeling theory* for the seamless development of software for embedded systems. We start the document by describing the need for a comprehensive theory to provide a formal and thorough foundation for the software development in different process phases. What exactly a theory should contain, strongly depends on the systems that need to be described. Therefore we characterize the types of systems to be modeled and starting from this we derive a set of requirements that need to be fulfilled by the modeling theory. Based on these requirements, we shortly introduce the core elements of FOCUS[1], that is a comprehensive modeling theory for specification of embedded systems. Next, we present how do basic modeling concepts like state machines relate to the core theory.

Outline. In Section 1 we motivate the need of a comprehensive modeling theory to allow the thorough and seamless formalization of development artifacts. In Section 2 we define the scope of our work by investigating the classes of systems that need to be described within our modeling theory. Starting from these system classes we derive a set of requirements to the modeling theory in Section 3. In Section 4 we present the comprehensive modeling theory by firstly presenting intuitively its main features and then by introducing the minimal mathematical model that makes up the theory. In Section 5 we extend our mathematical model by presenting state machines as basic techniques for the specification of the operational semantics.

[1] A thorough formal foundation of the FOCUS modeling theory can be found in [BS01] and [BKM07]

Contents

1 Motivation

Model-based development aims at the use of models as main development artifacts in all phases of the development process. It promises to increase the productivity and quality of the software by raising the level of abstraction at which the development is done as well as the degree of automation with the help of models that are tailored and adequate for specific development tasks. Even if adopted in practical development of embedded systems today, model-based development approaches often fall short due to the lack of powerful enough modeling theories and missing integration of theories, methods and tools. The models applied in the development process are based on separate and unrelated modeling theories (if foundations are given at all), which makes the transition from one model to another unclear and error-prone.

Below we enumerate two main deficiencies of the status quo of model-based development approaches with respect to the semantic infrastructure on which they are based (illustrated in Figure 1):

1. *Fragmentation and isolation of semantic foundations — lack of horizontal integration.* In practice, there is a plethora of different modeling techniques which can be applied for the specification of the same artifacts of a system. For example, functional requirements (i. e., interaction patterns between the system and its environment) can be specified with use cases diagrams, sequence charts, state machines, etc. Even if there are formalizations that rigorously describe the meaning of individual diagrams, *once a system is described by different diagrams, it is not specified how they are combined and integrated.* Today, the fragmentation and isolation of the semantic foundations lead to the impossibility to deeply integrate different models and to verify their consistency or to generate parts of one from another.
2. *Lack of seamless transition between different abstractions – lack of vertical integration.* The main aim of a seamless model-based development approach is the usage of models at different stages in the process and at different abstraction levels. The pervasive use of models allows engineers to raise the level of abstraction at which systems are developed, to abstract from implementation details, and to add more and more implementation details step by step. *Today an integrated framework of abstraction layers is missing which leads to the isolation of the models at different stages in the process and to unclear transitions*

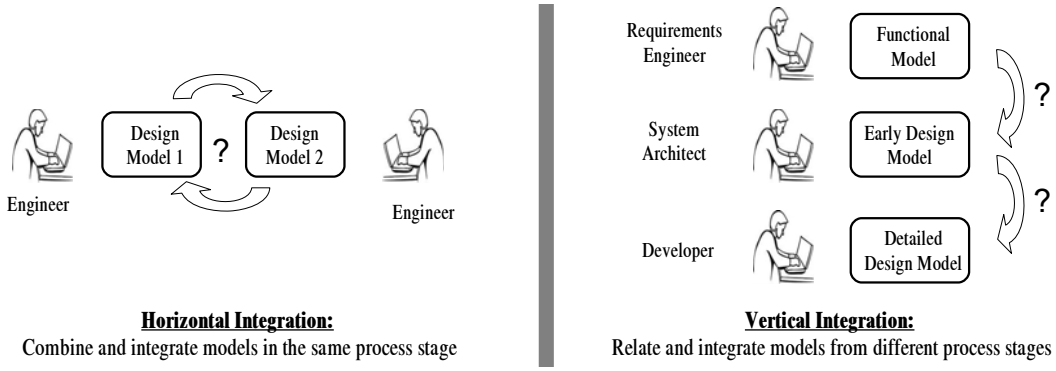

Figure 1: Horizontal vs. vertical integration problems.

between models. This subsequently leads to gaps between abstraction layers and thereby to a lack of automation and to difficulties with the management of intent and consistency between different layers.

All in all, both deficiencies lead to crucial consistency problems. Horizontal consistency problems prevent the integration of models created in the same development phase (e. g., two views showing a perspective of the design). Vertical consistency problems, prevent us to make sure that models from different phases of the development process are consistent to each other (e. g., requirements and design).

To address these problems we need a *comprehensive modeling theory* that provides a mathematical model of a system as a basis to ensure a thorough and seamless formalization of the software development process. This theory must be comprehensive and expressive enough to model all relevant views (aspects) of a system at different levels of abstraction, to allow the hierarchical decomposition of the system, and their modular development.

2 Characteristics of the Addressed Systems

How powerful should the modeling theory be? The choice of a modeling theory directly affects the limits of what can be explicitly described in the models. To answer this question, we have to answer the question which types of systems we want to address. Depending on the characteristics of the systems that are described, and thereby on what we need to express within our models, the modeling theory has to be more or less expressive within a certain respect. We enumerate a set of characteristics of the embedded systems that can be modeled with our modeling theory in Section 2.1 and a set of yet unaddressed aspects in Section 2.2.

2.1 Addressed Aspects

Multi-Functional and Complex Systems. Today, many software-intensive systems provide a wide range of functionalities, i. e., they offer a variety of different user-observable functions that interact with each other. We call such systems *multi-functional*. With functionality we mean the characteristic, observable behavior of a system, or more precisely its reaction (outputs) to stimuli (inputs). We also speak of *functionally complex* systems where complex refers to the complex interplay of functionalities and not (necessarily) to complex algorithms.

Interactive and Reactive Systems. Systems can be divided into two categories according to the way they transform input into output: transformational in contrast to interactive and reactive systems. Transformational systems, which transform input completely available at the beginning of the execution into output without any interaction with the user, are not in the focus of our approach. *Embedded systems are generally interactive or reactive systems.* Consequently we concentrate on these system types which are characterized by a constant interaction and synchronization between the system and its environment. While for interactive systems the interaction is determined by the system, the interaction of reactive systems is determined by the environment.

Distributed Systems. Today's embedded systems are generally no longer realized within a single ECU, but distributed over a network of distributed logical or physical components which heavily interact with each other in order to realize the desired functionality. These components execute their computations *simultaneously* on multiple cores in the same chip, different threads on the same processor, or on physically distributed systems.

Discrete Controllers and Continuous Physical Processes. Embedded systems are frequently used to control processes and devices that consist of physical (e. g., mechanical, electrical) components and exhibit time-continuous behavior. The controller, however, implemented in software, is generally some form of (finite-)state machine on a digital processor and is, therefore, *asynchronous* and *time-discrete*.

Real time constraints. Often the systems to be built must meet real-time requirements, namely there are *operational deadlines* from the occurrence of specific events (inputs) until the system produces a response (output). In this case, the construction of systems should allow for the verification of real-time constraints.

2.2 Not (yet) Addressed Aspects

Besides the characteristics listed above, there are also aspects which are not addressed in the initial version of the modeling theory.

Continuous / Hybrid modeling. In order to model continuous processes, we need a model of continuous time. A time-discrete model with a discretization function whose fine granularity is enough can be used to approximate most continuous processes. Therefore, in the first proposal of our modeling theory, we do not consider continuous time. However, we are aware of the relevance of modeling continuous and hybrid aspects in the domain of embedded systems. Thus, we are planning to elaborate appropriate concepts and to extend the modeling theory accordingly in a second step.

Probabilistic Systems. Various non-functional aspects like efficiency (time/space complexity) as well as various forms of failure are naturally expressed and reasoned about probabilistically. Even if some attempts exist, to the best of our knowledge none of them is satisfying for modeling the probabilistic aspects of systems with a well-formed and trunctable theory. Therefore, we agreed to refrain from probabilistic aspects for now.

Dynamic Reconfiguration. In various application domains, the dynamic reconfiguration of systems might be of interest. However, in the first version of our modeling theory we also back away from dynamic systems.

3 Requirements on the Modeling Theory

As already mentioned in the beginning, the modeling theory must be powerful enough to express all relevant properties of a system. In order to enable the modeling of systems with the characteristics mentioned in the previous section, the modeling theory needs to fulfill the following requirements:

Modularity and Compositionality. Compositionality is the property of a theory that allows to deduce the interface behavior of a system from the interface behavior of its sub-systems. It is the basis for the incremental and modular development of (sub-)systems.

- Modular development of sub-systems reduces the complexity through "divide and conquer" – instead of building monolithic systems in one step, compositionality enables to incrementally build systems out of modularly specified parts.
- In order to obtain the overall system, individually developed subsystems must be integrated. Thereby, we need to make the new without having to change the old – we need to assure that the properties of the subsystems, which are composed, are preserved. We aim at replacing the traditional a posteriori validation of systems by building systems that are correct by construction. In other words, the compositionality needs to be property preserving.
- Besides, compositionality is essential for the reuse of existing components. In the case of non-compositional theories, the validation costs grow exponentially with the number of components integrated in a system (by adding a new component all others need be checked for interferences). Compositionality, instead, allows to easily integrate existent components.

Abstraction and Refinement. Abstraction and refinement are two important concepts of a modeling theory. Refinement enables the transformation between a more abstract model into a more concrete one without loosing the properties of the abstract model. The concept of refinement allows us to start with high-granular descriptions and to incrementally refine them into more detailed ones. Refinement is especially useful in relating models from different abstraction levels: when the more concrete model is a refinement of a more abstract model, we are sure that the refining model guarantees all the properties of the abstract model. Thus, *refinement is a basic property of modeling theories needed to establish a uniform formal basis for a seamless model-based development along different abstraction layers.*[2]

(Interface) abstraction enables modeling and specifying properties of a system that are needed for its usage without paying attention to how the system is implemented (i. e., black-box specification). In particular *the modeling theory must enable to specify the interface behavior* – i. e., all properties that are relevant for the use of the system in any context.

[2] In [FHH+ar] we describe the need and use of 'abstraction layers' for developing complex embedded systems.

Explicit Modeling of Time. A modeling theory adequate for describing real-time systems must directly include timing aspects. Thus, the modeling theory should implement a timing model that allows us to reason about timing properties of the modeled systems.

Supporting Different Views. A modeling theory should support diverse, integrated views on the system under development – e. g., the structure, behavior, data. By this, different aspects of the system can be specified and analyzed separately.

4 Comprehensive Modeling Theory

The essential goal of the presented modeling theory is a mathematical foundation for the structured modeling of system functions at different abstraction layers (see [FHH+ar] for details). Thereby, our approach aims at modeling two fundamental, complementary views onto multi-functional systems. These views are address the two most significant dimensions of structured modeling of systems in the analysis and design phases of development:

- User Functionality Hierarchy: A structured view onto the overall functionality offered by a multi-functional system by decomposing the system functionality into a hierarchy of interrelated user functions. We speak of the *functional specification*.
- Logical Component Architecture: Decomposition of the system into a network of components that mutually cooperate to generate the behavior specified by the user functionality hierarchy. We speak of the *design*.

Section 4.1 intuitively introduces the fundamental notions: service hierarchies (used to model the user functionality), and hierarchical component networks (used to model the logical component architecture). Section 4.2 introduces the underlying model of computation. The basic building blocks – of service hierarchies as well as component networks – are stream processing functions, which are introduced in Section 4.3. We define two operators to combine modularly specified functions in Section 4.4 and methodologically important refinement relations in Section 4.5.

Running Example The formal definitions introduced in the remainder of the paper will be illustrated by a simple example, namely a boolean evaluator. Based on the user's selection (received via the channel `switch`), the system outputs the disjunction or the conjunction of two boolean input values (received via the input channels $\mathtt{i}_1$ and $\mathtt{i}_2$). The evaluator and its in- and output channels are depicted in Figure 2. All further details will be described at the appropriate places.

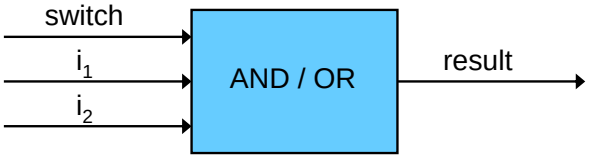

Figure 2: Running Example: Boolean Evaluator

4.1 Service Hierarchies and Component Networks

In this section we introduce two fundamental notions: service hierarchies and hierarchical component networks.

The *service hierarchy* aims at capturing all software-based functionality (*services*) offered by a system to its users. This hierarchy specifies the system behavior from a black-box perspective by capturing the family of services offered by the system. It aims at understanding how the services are structured and how they depend on and interfere with each other [BKPS07].

In a *component network*, the system functionality is specified as a distributed system of interacting components. Via their interaction, the components realize the observable behavior described in the service hierarchy.

Service Hierarchy. A service hierarchy consists of a set of services and relationships between them. Each service specifies a piece of functionality that is offered by the system and observable at the outer boundaries of the system. More formally, a service captures the interaction of the system with its environment by defining a (partial) relation between a set of inputs and outputs of the system. Usually, services describe system reactions only for a subset of the inputs. Thus, services are formally described by *partial* stream processing functions as introduced in Definition 5 below.

In general, multi-functional systems offer a bunch of services to their users. These services can be specified modularly and subsequently combined into an overall service hierarchy by means of the *combination* operator introduced in Definition 11 below (see Figure 3 for an informal representation).

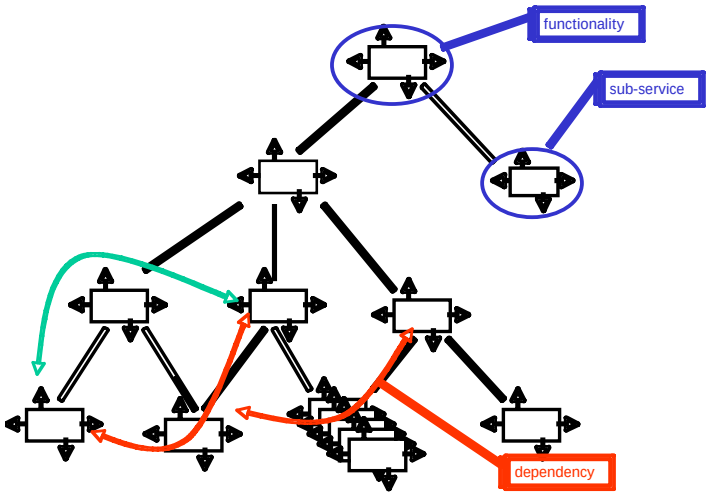

Figure 3: Service Hierarchy [Bro07]

✍ **Example 1** (Service Hierarchy of the Evaluator)

The introduced boolean evaluator comprises two sub-services. The sub-service AND calculates the conjunction of the input values if the corresponding input is received via the channel `switch`. The sub-service OR outputs the disjunction of the input values if the user wishes so. Figure 4 illustrates the simple service hierarchy of the boolean evaluator.

□

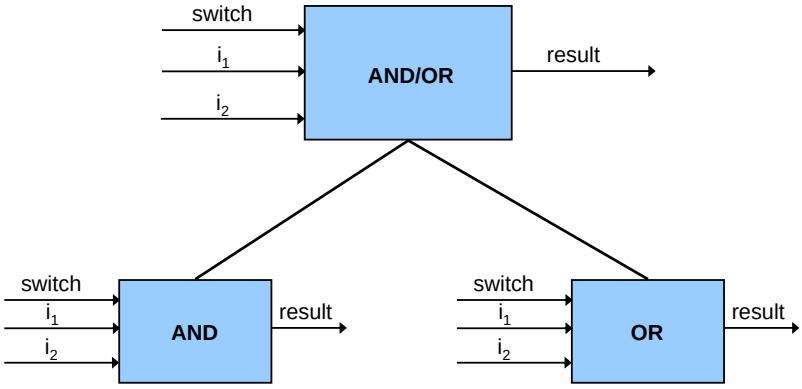

Figure 4: Service Hierarchy of the Evaluator

Component Network. The component network defines the architectural view of the system and decomposes the functionality into a network of communicating components. While the service hierarchy provides a structure of services observable at the outer boundaries of the system, the component network focuses on the internal component based structure of the system (cf. Figure 5).

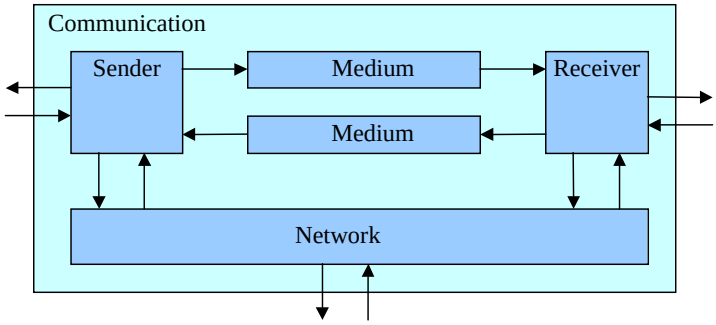

Figure 5: Component Network [Bro07]

A component network consists of a set different components, which are connected via channels. Each component can be further decomposed into a component network, itself. A component is total in its nature and therefore specified by a total function as will be introduced in Definition 4. In a component network, several components have to collaborate to realize functions

observable at the system boundary. Therefore, the components are connected by channels, via which they can exchange messages. Formally, we use the *composition* operator introduced in Definition 7 below to incrementally compose components to more complex ones, until we get the overall system.

✍ **Example 2** (Component Network of the Evaluator)

The overall evaluator can be decomposed into three interacting components. The component AND calculates the conjunction of the input values, the component OR the disjunction of the input values, and the component $SWITCH$ decides which of the results is shown on the output channel. Figure 6 depicts the component network of the evaluator.

□

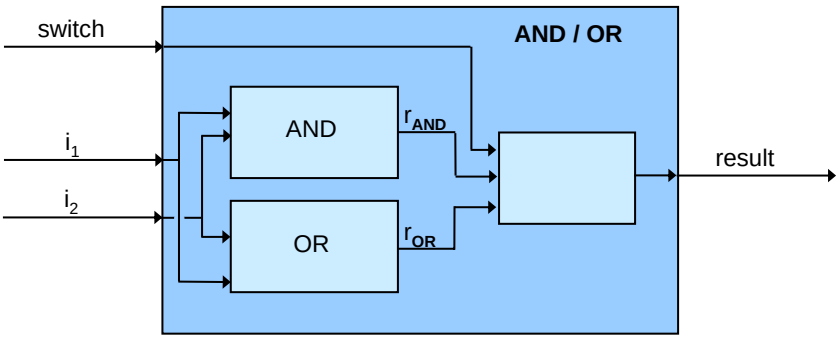

Figure 6: Component Network of the Evaluator

4.2 Underlying Model of Computation

In this section we put our modeling theory in a larger context according to its underlying timing model and its communication/synchronization mechanisms between different components.

Communication Paradigm. Regarding communication – i. e., the way information can be exchanged between components – one can distinguish between *implicit communication* (e. g., via shared variables), *explicit event based synchronous communication* and *explicit message based asynchronous communication*.

Our modeling theory is based on explicit message based asynchronous communication. In this class there is an explicit communication mechanism with clear distinction between sender and receiver. The communicating partners are decoupled, i. e., while the receiver has to wait until a message is available to be read, the sender can write a message without delay. We base our assumption on the fact that distributed systems are usually connected by a bus which never blocks the sender. In contrast to synchronous systems, when embedding a system into its environment, no (output) blocking has to be considered. In this sense, asynchronous message exchange allows unrestricted compositionality of the components without considering technical details as deadlocks or interference.

Timing Model. Considering possibilities at what point in time "relevant" activities are allowed to happen during the system run, the following classification can be done: In *time continuous systems*, relevant activities can evolve continuously. In *time discrete systems*, the time line is divided into not necessarily but possibly equal intervals and the relevant activities only occur at points of time of the time grid. In *event discrete systems*, events can occur at any point of a real time line. The different timing models are vividly illustrated in Figure 7.

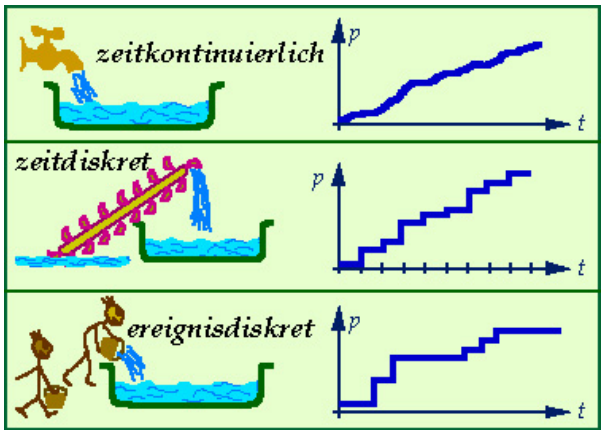

Figure 7: Illustration of Different Timing Models [Kru]

Embedded systems can contain a large amount of control algorithms which require a continuous model of time. However, such systems play only a minor role and can be abstracted by discrete algorithms most of the time. Furthermore, the resulting control system which is to be developed will run on a digital computer and therefore will be time-discrete.

Our modeling theory assumes a simple model of discrete time, where time advances in ticks. Therefore, events within a single tick can not be differentiated. In our experience, most frequent situations can be modeled and analyzed in this time model, as long as the length of a single time interval (tick) is chosen sufficiently small. The limitation to this discrete time model simplifies both the understandability and analyzability of as well as the creation of models and supporting tools. However, extending the theory to more complex models of real or continuous time can be easily achieved following [Bro97].

Synchronicity of Time. Different theories differentiate whether there is a global clock which defines a global time for all parts of the system, or if the different components have their local clock. In the second case further synchronization mechanisms are necessary. *Our modeling theory assumes a global clock.*

4.3 Stream Processing Function

In the FOCUS approach [BS01], a system is described by a *stream processing function*, which defines its syntactic interface as well as its behavior. Furthermore, the FOCUS approach of-

fers composition operators which allow derive a larger system (the composed system) out of modularly defined functions.

4.3.1 Streams

Basically, the FOCUS theory is based on the idea of timed data streams which are used to model the asynchronous interaction between a function and its environment. Streams represent histories of communication of data messages in a given time frame. Intuitively, a timed (data) stream can be thought of as a chronologically ordered sequence of data messages.

Definition 1 (Timed Stream) *Given a set M of data messages, we denote a timed stream of elements from M by a mapping*

$$s : \mathbb{N} \to M.$$

For each time $t \in \mathbb{N}$, $s(t) = s.t$ denotes the message communicated at time t in a stream s and $s \downarrow t$ the prefix of the first t messages in the timed stream s, i. e., the messages communicated until (and including) time t.[3]

We have chosen so-called timed data streams that allow us to flexibly include the timing issues of functions whenever required. We base our approach on a simple notion of discrete time: we assume a model of time consisting of an infinite sequence of time intervals of equal length. Thus, time can be simply represented by the natural numbers $\mathbb{N}_+$. In each time interval a message $m \in M$ can be transmitted.

✍ **Example 3** (Timed Stream)

A exemplary timed stream s over the data set $\mathbb{B} = \{0, 1\}$ is defined by the function $\forall t \in \mathbb{N} : s(t) = 1$. This means that in each time interval the stream contains the value 1, i. e., $s = (1\ 1\ 1\ \ldots)$.

□

4.3.2 Input/Output Channels and Channel Histories

Every stream processing function is connected to its environment by channels. The channels of a stream processing function are divided into disjoint sets of input channels $I = \{i_1, \ldots i_n\}$ and output channels $O = \{o_1, \ldots o_n\}$. Channels are used as identifiers for streams.

With every channel c, we associate a data type $\mathit{Type}(c)$ indicating the set of messages sent along this channel. To that end, we define the channel type by the following function:

$$\mathit{Type} : C \to \texttt{Type},$$

[3] The theory is originally defined for stream processing functions $s : \mathbb{N}_+ \to M^*$, which assign a sequence of messages to each time interval. In order to keep the paper at hand as understandable as possible, we decided for the simplification that only one message can be communicated within each time interval. A proper description of the original theory can be found in [BS01, BKM07]

Figure 8: Graphical representation of a FOCUS function and its typed I/O channels

which maps each channel $c \in C$ to a data type $t \in \texttt{Type}$ from the set of all possible data types $\texttt{Type}$.

✍ **Example 4** (Channel Types)

In our running example, the channel types are defined as follows. $\mathit{Type}(\texttt{result}) = \mathit{Type}(\texttt{i}_1) = \mathit{Type}(\texttt{i}_2) = \mathbb{B} = \{0, 1\}$ and $\mathit{Type}(\texttt{switch}) = \{\wedge, \vee\}$.

□

To describe the function's communication with its environment, each channel is associated with a stream which represents the messages communicated over this channel (cf. Figure 8). A mapping that associates a stream to any channel from a set of channel C is called *(channel) history* of C.

Definition 2 (Channel History) *Let C be a set of typed channels. A channel history h is a mapping*

$$h : C \to (\mathbb{N} \to M),$$

such that $h(c)$ is a stream of the type $\mathit{Type}(c)$ for each $c \in C$.

The set of all histories over the set of channels C is denoted by $\mathbb{H}(C)$.

In the following, we also use $h.c$ instead of $h(c)$ to refer to the stream associated to a channel $c \in C$ by a history h.

✍ **Example 5** (Channel Histories)

One of the possible input histories of the evaluator from our running example (i. e., a history over its input channels) is given by the mapping h:

$$h : \begin{pmatrix} \texttt{switch} \\ \texttt{i}_1 \\ \texttt{i}_2 \end{pmatrix} \mapsto \begin{pmatrix} \wedge & \wedge & \vee & \vee & \wedge & \ldots \\ 0 & 1 & 0 & 1 & 0 & \ldots \\ 1 & 1 & 1 & 1 & 1 & \ldots \end{pmatrix}.$$

$h(\texttt{switch}) = (\wedge\ \wedge\ \vee\ \vee\ \wedge\ \ldots)$ results in the corresponding stream of the switch button, and $h(\texttt{switch}).3 = \vee$ denotes that the switch button is turned to $\vee$ in the third time interval.

□

4.3.3 Specification of Stream Processing Functions

The black-box specification of a stream processing function consists of a syntactic interface and its semantics.

A stream processing function is connected to its environment exclusively by its syntactic interface consisting of input/output channels. This interface indicates which types of messages can be exchanged.

Definition 3 (Syntactic Interface) *The syntactic interface of a function is denoted by*

$$(I \blacktriangleright O),$$

where I and O denote the sets of typed input and output channels, respectively.

✍ **Example 6** (Syntactic Interface)

The syntactic interface of the evaluator $(I \blacktriangleright O)$ comprises three input channels $I = \{\texttt{switch}, \texttt{i}_1, \texttt{i}_2\}$ and one output channel $O = \{\texttt{result}\}$.

□

For a function with syntactic interface $(I \blacktriangleright O)$, the set of all syntactically correct history pairs is denoted by

$$\mathbb{H}(I) \times \mathbb{H}(O).$$

However, the syntactic interface tells nothing about the interface behavior of the function.

The behavior (semantics) of the stream processing function is given by the mapping of histories of the input channels to histories of the output channels. Thereby, we distinguish between total and partial functions. While the behavior of a total function is defined for *all* syntactically correct inputs, the behavior of a partial functions is defined for a *subset* of the inputs. As mentioned in Section 4.1, partial stream processing functions are the basic building blocks of service hierarchies while total stream processing functions are used to specify the components of component networks.

Definition 4 (Semantics) *The semantics of a total stream processing function with syntactic interface $(I \blacktriangleright O)$ is given by a relation*

$$F : \ \mathbb{H}(I) \to \mathcal{P}(\mathbb{H}(O))$$

that fulfills the following timing property for all its input histories.

Let be $x_1, x_2 \in \mathbb{H}(I), y_1, y_2 \in \mathbb{H}(O)$, and $t \in \mathbb{N}_+$. The timing property is specified as follows:

$$x_1 \downarrow t = x_2 \downarrow t \Rightarrow \{y_1 \downarrow (t+1) : y_1 \in F(x_1)\} = \{y_2 \downarrow (t+1) : y_2 \in F(x_2)\}.$$

By mapping into the powerset of $\mathbb{H}(O)$, Definition 4 (as well as Definition 5) allows to specify *nondeterministic* behavior. For an input history, there is a set of output histories that represent all possible reactions of the function to the input history. If a function defines exactly one output history for every input history, the function is called deterministic; if the set of output histories has several members for some input history, then the function is called nondeterministic.

The timing property expresses that the set of possible output histories for F for the first $t+1$ intervals only depends on the inputs of the first t time intervals. In other words, the processing of messages within a function takes at least one time interval. Functions that fulfill this timing property are called time-guarded or *strictly causal*. Strict causality is a crucial prerequisite for the composability of functions.

If we replace the expression $(t+1)$ by t in Definition 4 above (i. e., the outputs in the first t intervals depend on the inputs in the first t intervals), messages are processed by the function without time delay. Such functions are called *weakly causal*. We also allow for the specification of weakly causal functions. For the composition, however, it must be assured that each feedback loop contains at least one strictly causal function. An important consequence of the causality assumption is, that a causal function leads either to a defined output for all its input histories (i. e., $F(x) \neq \emptyset$ for all $x \in \mathbb{H}(I)$) or an empty output for all input histories (i. e., $F(x) = \emptyset$ for all $x \in \mathbb{H}(I)$). In the first case, we call the function *total*, in the second case we call it *paradoxical*. Partial functions which are only defined for a subset of the input histories are not covered by Definition 4.

✍ **Example 7** (Semantics of Total Stream Processing Function)

The semantics of our evaluator F_E is given by the following equation:

$$y \in F_E(x) \Leftrightarrow \forall t \in \mathbb{N}_+ : y(\texttt{result}).(t+1) = \begin{cases} x(\texttt{i}_1).t \wedge x(\texttt{i}_2).t, & \text{if } x(\texttt{switch}).t = \wedge, \\ x(\texttt{i}_1).t \vee x(\texttt{i}_2).t, & \text{if } x(\texttt{switch}).t = \vee. \end{cases}$$

The behavior is as expected. If the switch is turned to $\wedge$, the conjunction of the input values received at the channels $\texttt{i}_1$ and $\texttt{i}_2$ is displayed at the output channel `result` in the next time interval. Otherwise, the disjunction is displayed.

A possible *valid* input/output pair is given by

$$h_{in} : \begin{pmatrix} \texttt{switch} \\ \texttt{i}_1 \\ \texttt{i}_2 \end{pmatrix} \mapsto \begin{pmatrix} \wedge & \wedge & \vee & \vee & \wedge & \dots \\ 0 & 1 & 0 & 1 & 0 & \dots \\ 1 & 1 & 1 & 1 & 1 & \dots \end{pmatrix} \text{ and}$$

$$h_{out} : \begin{pmatrix} \texttt{result} \end{pmatrix} \mapsto \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & \dots \end{pmatrix}.$$

The evaluator function F_E is strictly causal. The output in time interval $t+1$ only depends on the inputs received in time interval t. The calculated result is visible at the output channel exactly in the next time interval. Furthermore, the evaluator function is non-deterministic. The output in the first time interval is not further specified. Exemplary, for the input history h_{in} there exists a second valid output history h'_{out} which differs from h_{out} only in the first time interval.

$$h'_{out} : \begin{pmatrix} \texttt{result} \end{pmatrix} \mapsto \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 & \dots \end{pmatrix}.$$

□

Especially during the early phases of the development process, it is helpful to allow for partial specification. This allows the developer to specify all information that is already known about the system while not bothering about the situations which are not important for the moment.

Definition 5 (Semantics of Partial Stream Processing Function) *The semantics of a partial stream processing function with syntactic interface* $(I \blacktriangleright O)$ *is given by a relation*

$$F : \mathbb{H}(I) \rightharpoonup \mathcal{P}(\mathbb{H}(O))$$

that fulfills the timing property only for the input histories with nonempty output set.

Let be $x_1, x_2 \in \mathbb{H}(I), y_1, y_2 \in \mathbb{H}(O)$*, and* $t \in \mathbb{N}_+$*. The timing property is specified as follows:*

$$F(x_1) \neq \emptyset \neq F(x_2) \wedge x_1 \downarrow t = x_2 \downarrow t \Rightarrow \{y_1 \downarrow (t+1) : y_1 \in F(x_1)\} = \{y_2 \downarrow (t+1) : y_2 \in F(x_2)\}.$$

The set

$$dom(F) = \{x \in \mathbb{H}(I) : F(x) \neq \emptyset\}$$

is called the function domain. It characterizes those input streams for which F *is defined. The set*

$$ran(F) = \{y \in F(x) : x \in dom(F)\}$$

is called the function range and comprises the possible outputs for all valid inputs.

The definition of partial stream processing functions is analogous to the definition of total functions. A partial stream processing function defines a non-empty set of output streams only for a subset $dom(F) \subset \mathbb{H}(I)$ of all possible inputs and the timing property (implying strong or weak causality) must only hold for these input streams.

✍ **Example 8** (Semantics of Partial Stream Processing Function)

Let's assume that the type of the channel `switch` is defined as $Type(switch) = \{\wedge, \vee, \neg\}$, i. e., it is extended by the message $\neg$ compared to the original definition. Then, the function F_E, that is given in Example 7, is only partially defined. If an input history h contains the message $\neg$ there is no defined reaction to this input:

$$(\exists t \in \mathbb{N}_+ : h(switch).t = \neg) \Rightarrow (F_E(h) = \emptyset)\,.$$

The input history h is outside the domain of F_E. The domain is given as $dom(F_E) = \{h \in \mathbb{H}(I) : \forall t \in \mathbb{N}_x : h(switch).t \neq \neg\}$

□

4.4 Function Structuring

To cope with the complexity of today's systems, a modeling theory must comprise concepts to decompose the overall system into smaller building blocks which can be developed individually and easily integrated afterwards. Hereby, a central question in software engineering is how to adequately structure the functionalities of the system at different levels of abstraction. We introduce two operators, a composition and a combination operator, which allow two different kinds of decomposition of the functionality (cf. Section 4.1): Via the composition operator, we can derive the behavior of component networks containing interacting components. Via the combination operator, we can combine several services into a service hierarchy.

4.4.1 Composition

To deal with the complexity, functions are composed by parallel composition with feedback following the approach of Broy [BS01]. The composition operator combines a set of functions that mutually use each other into a network.

For the following definitions we firstly introduce the notion of history projection. A sub-history is the projection of a history with respect to a subset of its channel set and their restricted types. Given a set $C_1$of typed channels, to obtain the sub-history $x|C_1$, we keep only those channels and types of messages in the history x that belong to the typed channels in C_1.

Definition 6 (History Projection) *Let C_1 and C_2 be sets of typed channels with $C_1 \subseteq C_2$. For history $x \in \mathbb{H}(C_2)$, we define its restriction $x|C_1 \in \mathbb{H}(C_1)$ to the channels in the set C_1 and to the messages of the types of the channels in C_1 (denoted by the function $Type_{C_1}(.)$). For channels $c \in C_1$, we specify the restriction by the equation:*

$$(x|C_1).c = Type_{C_1}(c)@(x.c)$$

where $M@s$ deletes all messages $m \notin M$ from the stream s.

✍ **Example 9** (History Projection)

Given the history h_{in} from Example 7. It is defined over the channel set $I = \{switch, in_1, in_2\}$. Its projection to the channels in the set $I' = \{switch\}$ is given by

$$h_{in}|I' : \ \left(\ switch \ \right) \mapsto \left(\ \wedge \quad \wedge \quad \vee \quad \vee \quad \wedge \quad ... \ \right).$$

□

Now we can define the composition operator for stream processing functions.

The idea of the function composition is shown in Figure 9. In a composed function $F_1 \otimes F_2$, the channels in the channel sets C_1 and C_2 are used for internal communication. They are called feedback channels.

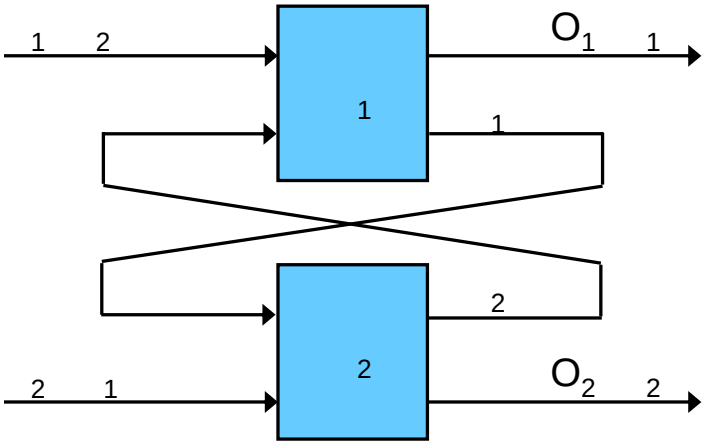

Figure 9: Function Composition with Feedback

Formally, the composition of two stream processing functions F_1 and F_2 is defined as the behavior observable at the black-box boundaries of the system, i. e., the behavior observable at the channels that are not used for internal communication ($I = (I_1 \cup I_2) \setminus (C_1 \cup C_2)$, $O = (O_1 \cup O_2) \setminus (C_1 \cup C_2)$). To check whether an output $y \in \mathbb{H}(O)$ is a correct response to an input stream $x \in \mathbb{H}(I)$, we must find a valuation of the feedback channels $c \in \mathbb{H}(C_1 \cup C_2)$ such that the projections of the overall history to the interfaces of the subfunctions F_1 and F_2 must represent valid behaviors according to the specification of the composed subcomponents. For the formal definition, the histories x, y, and c are combined into a history $z \in \mathbb{H}(I \cup O \cup C_1 \cup C_2)$ with $z|I = x$, $z|O = y$, and $z|(C_1 \cup C_2) = c$.

Definition 7 (Composition of Functions) *Given two functions F_1 with syntactic interface $(I_1 \blacktriangleright O_1)$ and F_2 with syntactic interface $(I_2 \blacktriangleright O_2)$ and $O_1 \cap O_2 = \emptyset$, we define a composition by the expression*

$$F_1 \otimes F_2.$$

The function $F_1 \otimes F_2$ has the syntactic interface $(I \blacktriangleright O)$ with $I = (I_1 \setminus C_2) \cup (I_2 \setminus C_1)$ and $O = (O_1 \setminus C_1) \cup (O_2 \setminus C_2)$ and internal feedback channels $C_1 = I_2 \cap O_1$ and $C_2 = I_1 \cap O_2$.

The semantics is defined as follows: Let $x \in \mathbb{H}(I)$, $z \in \mathbb{H}(I \cup O \cup C_1 \cup C_2)$, then

$$(F_1 \otimes F_2).x = \{z|O : x = z|I \wedge z|O_1 \in F_1(z|I_1) \wedge z|O_2 \in F_2(z|I_2)\}.$$

As mentioned before, strict causality assures that the composition of two components is well-defined. The composition is associative and commutative. With the composition operator the necessary prerequisites for describing architectures and modeling complex component networks as introduced in Section 4.1 are given.

✍ **Example 10** (Composition of Functions)

As depicted in Figure 6 on Page 10, the function `evaluator` is decomposed into three sub-functions. Functions `AND` and `OR` calculate the conjunction/disjunction, respectively. Function

`Switch` decides which result is sent on the output channel `result`. The syntactic interfaces of the functions are defined as follows:

$$I_{AND} = \{i_1, i_2\},\ O_{AND} = \{r_{and}\},$$
$$I_{OR} = \{i_1, i_2\},\ O_{OR} = \{r_{or}\},$$
$$I_{SWITCH} = \{switch, r_{and}, r_{or}\},\ O_{SWITCH} = \{result\},$$

with internal channels r_{and} and r_{or}.

To assure the same timing behavior as the composed function, we decided the functions `AND` and `OR` to be only weakly causal, i. e., the calculation of the intermediate result takes no time. The function `SWITCH`, however, is strictly causal. Then, the composition $AND \otimes OR \otimes Switch$ yields the expected result. Otherwise, the processing of the input data would have taken two time intervals (one for calculating the disjunction/conjunction) and one for deciding which of them is displayed on the output channel.

□

4.4.2 Combination

The combination relation allows to derive the combined complex behavior of a system based on simpler sub-specifications. In contrast to the composition, the combination allows us to horizontally combine different functions which define different aspects of a system but do not represent a structural decomposition. Thereby the functions are not internally connected but "overlayed". These functions formalize different (black-box) requirements or different aspects of the same system. They might be defined over the same I/O channels, i. e., they might define different aspects of the interaction with the environment on the same channels.

The idea of function combination is illustrated in Figure 10. Both functions F_1 and F_2 can be defined over overlapping interfaces, i. e., the functions can have common ports. On these ports, the specification of both functions must be fulfilled.

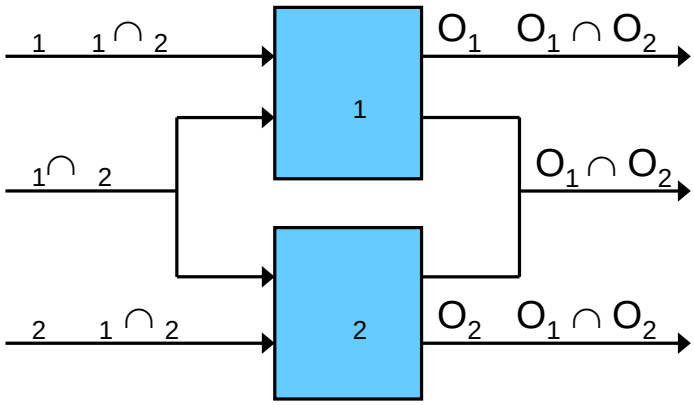

Figure 10: Function Combination

To define the combination of functions, we have to introduce some auxiliary notions beforehand. A fundamental notion is the sub-function relationship between functions. To formalize this relationship we introduce the projection of functions.

Our goal is to derive the complex overall behavior of a system based on less complex sub-behaviors or – the other way round – to split up the overall functionality into easier understandable and manageable pieces. Therefore, we introduce a technique to eliminate a set of input and output messages, i. e., the projection of a function F to a sub-interface. By eliminating a part of the inputs and/or outputs, the behavior projection allows us to concentrate only on a certain aspect of the behavior of a system, i. e., the influence of a defined set of inputs to a defined set of outputs.

Definition 8 (Behavior Projection) *Given syntactic interfaces* $(I_1 \blacktriangleright O_1)$ *and* $(I \blacktriangleright O)$ *with* $I_1 \subseteq I$ *and* $O_1 \subseteq O$*, we define for a function* F *with interface* $(I \blacktriangleright O)$ *its* behavior projection $F\dagger(I_1 \blacktriangleright O_1)$ *to the syntactic interface* $(I_1 \blacktriangleright O_1)$ *by the following equation (that has to hold for all input histories* $x \in \mathbb{H}(I_1)$*):*

$$F\dagger(I_1 \blacktriangleright O_1)(x) = \{y | O_1 \in \mathbb{H}(O_1) : \exists x' \in \mathbb{H}(I) : x = x' | I_1 \wedge y \in F(x')\}.$$

According to Definition 8, the behavior projection is only defined for the sub-interface $(I_1 \blacktriangleright O_1)$. Furthermore, an I/O-pair $(x_1, y_1) \in \mathbb{H}(I_1) \times \mathbb{H}(O_1)$ is valid for $F\dagger$ if and only if it is the projection of a valid I/O-pair $((x, y) \in \mathbb{H}(I) \times \mathbb{H}(O)$ of the original function F.

✍ **Example 11** (Behavior Projection)

Let $I' = \{i_1, i_2\} \subset I$, $O' = O$, and $h'_{in} \in \mathbb{H}(I')$ be the input history

$$h'_{in} : \begin{pmatrix} \mathtt{i_1} \\ \mathtt{i_2} \end{pmatrix} \mapsto \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & ... \\ 0 & 0 & 0 & 0 & 0 & ... \end{pmatrix}.$$

Here, the histories on the port `switch` are omitted. Then, $F_E\dagger(I' \blacktriangleright O')(h'_{in})$ results in the set of all output histories that arbitrarily contain either the conjunction or the disjuction of the values on $\mathtt{i_1}$ and $\mathtt{i_2}$ at any time interval. Exemplary, the output history

$$h'_{out} : \begin{pmatrix} \mathtt{result} \end{pmatrix} \mapsto \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & ... \end{pmatrix}$$

is correct, since there is a input history $h_{in} \in \mathbb{H}(I)$ such that $h_{in}(switch) = (\wedge \ \wedge \ \wedge \ \wedge \ \wedge \ )$. Besides, there are several other valid output histories, e. g.,,

$$h''_{out} : \begin{pmatrix} \mathtt{result} \end{pmatrix} \mapsto \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & ... \end{pmatrix}.$$

A corresponding valuation of `switch` is $(\vee \ \vee \ \wedge \ \wedge \ \wedge \ )$. However, the output history

$$h'''_{out} : \begin{pmatrix} \mathtt{result} \end{pmatrix} \mapsto \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & ... \end{pmatrix}$$

is not correct because there is no valuation of `switch` such that h'''_{out} would be a correct output for the original function F_E.

□

To master the complexity of a specification, we introduce another essential notion for functions, namely the sub-function relation. Note that the sub-function relation is the fundamental relationship between services and sub-services in service hierarchies (as presented in Section 4.1). Intuitively, the super-function "contains" the sub-function, i. e., the super-function imposes at least all those requirements that are already specified by the sub-function. However, since the super-function combines several sub-functions, it usually is more restricted than each of its sub-functions.

Definition 9 (Sub-Function) *A function F' with syntactic interface $(I' \blacktriangleright O')$ is a* sub-function *of a function F with syntactic interface $(I \blacktriangleright O)$, denoted by $F' \sqsubseteq_{sub} F$ if*

- $I' \subseteq I$, $O' \subseteq O$,
- $dom(F') \subseteq dom(F\dagger(I' \blacktriangleright O'))$, *and*
- $\forall x \in \mathbb{H}(I') : F\dagger(I' \blacktriangleright O')(x) \subseteq F'(x)$.

The second condition of Definition 9 states that the super-function F must be at least defined for all inputs which the sub-function is defined for. Since F combines several sub-functions, it might be defined for more inputs. The third condition formalizes that the super-function F is a refinement of the sub-function F'. This means that F preserves all requirements specified by F', but might be more restricted due to the fact that it combines several sub-functions. Note that the sub-function relation introduces a partial order on the set of functions. The relation is reflexive, transitive and antisymmetric.

✍ **Example 12** (Sub-Functions of the Evaluator)

As illustrated in Figure 4 on Page 9, the evaluator of our running example consists of two sub-functions, the function `AND` and the function `OR`. The syntactic interfaces of both sub-functions equal the syntactic interface of the combined function: $I = \{\mathtt{i_1}, \mathtt{i_2}, \mathtt{switch}\}$ and $O = \{\mathtt{result}\}$. The behavior of the function `AND` is given as follows. For all $x \in \mathbb{H}(I)$:

$$y \in F_{AND}(x) \Leftrightarrow \forall t \in \mathbb{N}_+ : x(\mathtt{switch}).t = \wedge \Rightarrow y(\mathtt{result}).(t+1) = x(\mathtt{i_1}).t \wedge x(\mathtt{i_2}).t.$$

Whenever the message $\wedge$ is received on the input channel `switch`, the values on $\mathtt{i_1}$ and $\mathtt{i_2}$ have to be conjuncted. Otherwise, an arbitrary value can be outputed. The behavior of the function `OR` is specified analogously. The behavior is only constrained if the message $\vee$ is received on the input channel `switch`. Obviously, both functions are defined only partially. The combination of both functions eliminates the non-determinism of both sub-functions and yields exactly the required behavior.

□

The introduced sub-function relation is rather straightforward. It allows to combine *independent* sub-functions: Given three functions F, F_1 and F_2, we call F_1 and F_2 independent in F if the following relation holds: $F_1 \sqsubseteq_{sub} F \ \wedge \ F_2 \sqsubseteq_{sub} F$. This means, there are no dependencies (or relationships) between F_1 and F_2 since both are real subfunctions of F. When combined to implement F, they do not influence each other.

Often, however, functions depend on each other and are thus not truly sub-functions according to Definition 9. More precisely, the choice of the concrete output of a sub-function may depend on messages outside of its domain. Therefore we introduce a more sophisticated relationship between functions, namely the restricted sub-function relation.

Definition 10 (Restricted Sub-Function) *F' is a* restricted sub-function *of F if there exists $R \subseteq dom(F)$ such that:*

$$F' \sqsubseteq_{sub} F|R.$$

For a function $F: \mathbb{H}(I) \to \mathcal{P}(\mathbb{H}(O))$ its restriction is defined as $F|R: \mathbb{H}(I) \setminus R \to \mathcal{P}(\mathbb{H}(O))$.

In other words, F' is not a true sub-function of F. There may be input streams such that F' is not a sub-function of F. But if we regard only a defined subset R of the inputs of F, F' is a sub-function of F.

✍ **Example 13** (Restricted Sub-Functions of the Evaluator)

We introduced two sub-functions in the forgoing example. However, a more natural way might be to abstract the input channel `switch` away and to look at the following function `AND'` with syntactic interface $(I' \blacktriangleright O') = (\{\mathtt{i_1}, \mathtt{i_2}\} \blacktriangleright \{result\})$. This function continuously calculates the conjunction of the values received on the channels $\mathtt{i_1}$ and $\mathtt{i_2}$. However, this function is no real sub-function of the overall evaluator functionality. It comprises I/O history pairs that are not conform to the overall specification. It is a correct sub-function if we regard only those input histories of F_E where `switch` is permanently turned to $\wedge$. According to Definition 10, the function `AND'` is a restricted sub-function of the function F_E with the set $R \subseteq dom(F_E)$ defined as $R = \{x \in dom(F_E) : \forall t : x(\mathtt{switch}).t = \wedge\}$.

□

Based on the relations introduced so far, we now explain how the combination of sub-functions is defined.

Definition 11 (Combination of Functions) *Given a function F_1 with interface $(I_1 \blacktriangleright O_1)$ and a function F_2 with interface $(I_2 \blacktriangleright O_2)$, the combination $F = F_1 \oplus F_2$ is given as a function F with*

$$F_1 \sqsubseteq_{sub} F \quad \textit{and} \quad F_2 \sqsubseteq_{sub} F.$$

The syntactic interface of F is given as the union of the syntactic interfaces of F_1 and F_2:

$$(I \blacktriangleright O) = ((I_1 \cup I_2) \blacktriangleright (O_1 \cup O_2)).$$

The combination of the sub-functions yields the least function F such that F_1 and F_2 are still sub-functions of F.

Regarding the combination of two functions with a relationship in between, we define the following combination operator.

Definition 12 (Combination of Depending Functions) *Given a function F_1 with $(I_1 \blacktriangleright O_1)$ and a function F_2 with $(I_2 \blacktriangleright O_2)$, the combination $F = F_1 \oplus F_2$ with a dependency between F_1 and F_2 is given as a function F with*

$$\exists R_1, R_2 \in dom(F) : F_1 \sqsubseteq_{sub} F|R_1 \quad \textit{and} \quad F_2 \sqsubseteq_{sub} F|R_2.$$

The definitions given above can easily be extended to the combination of more than two sub-functions.

✍ **Example 14** (Function Hierarchy of the Evaluator)

The combination of the (independent) sub-functions `AND` and `OR` introduced in Example 12 yields exactly the behavior of the overall evaluator function F_E:

$$AND \oplus OR = F_E.$$

In Figure 4 on Page 9 the function hierarchy of the evaluator is depicted.

□

4.5 (Interface) Abstraction and Refinement

From a practical point of view, it is impossible to transfer an abstract model of a large system to a concrete one in just one step. Instead, we need a stepwise development process whereby the requirements specification is refined into its implementation via a number of intermediate models (cf. [FHH+ar]).

According to [BS01], FOCUS offers two basic refinement relations, namely *property* and *interaction* refinement. Property refinement allows us to add further behavior details to a specification. Interaction refinement allows us to change the representation of the communication histories, in particular, the granularity of the interaction as well as the number and types of the channels of a component. By this, interaction refinement supports relating different levels of abstraction (as presented in [FHH+ar]). In fact, these notions of refinement describe the steps needed in an idealistic view of a strictly hierarchical top-down system development. They are formally defined and explained in detail in the following.

Property Refinement. Property refinement allows the horizontal refinement of models, i. e., the stepwise enrichment of models on the same abstraction layer. By property refinement, we are able to relate functions with the same syntactical interface (i. e., the same channesl). It allows us to replace an interface behavior by another one having additional properties or in other words, to impose additional requirements. Formally, the property refinement of a stream-processing function is given as follows:

Definition 13 (Property Refinement) *Let F and F' be stream-processing functions with syntactic interface $(I \blacktriangleright O)$. F' is a property refinement of F if*

$$\forall x \in \mathbb{H}(I) : F'(x) \subseteq F(x).$$

The property refinement of F by F' is denoted by $F \rightsquigarrow F'$.

Property refinement guarantees that any I/O history of the refined function F' (more concrete behavior) is also an I/O history of the given function F (more abstract behavior). However, non-determinism of F can be reduced in F'. This way an interface behavior is replaced by a more restricted one.

✍ **Example 15** (Property Refinement)

In Example 12 we introduced the function `AND`. The overall evaluator function F_E has the same syntactic interface as `AND`. However, F_E determines the behavior more precisely than the `AND`. While the function `AND` allows totally non-deterministic behavior if the message $\vee$ is received on the channel `switch`, F_E defines the output also in these cases (disjunction of the inputs). Thus, F_E is a property refinement of `AND`: `AND` $\rightsquigarrow F_E$.

□

Obviously, property refinement is a partial order and therefore reflexive, antisymmetric, and transitive. If we look at hierarchically decomposed systems (cp. Section 4.4), compositionality of property refinement guarantees that separate refinements of sub-services/sub-components of a system lead to a refinement of the composed system, i. e.,

$$(F_1 \rightsquigarrow F_1') \wedge (F_2 \rightsquigarrow F_2') \implies F_1 \otimes F_2 \rightsquigarrow F_1' \oplus F_2'$$

In our case, the proof of the compositionality of property refinement is straightforward.

Interaction Refinement. Changing the granularity of interaction and thus the level of abstraction is a classical technique in software system development. Interaction refinement is a generalization of property refinement and allows us to change

- the number and names of the input and output channels of a function as well as
- the types of the messages on this channels determining the granularity of the communication.

In the following, interaction refinement is described formally. Interaction refinement between an abstract function F and a concrete function F' with syntactic interfaces $(I \blacktriangleright O)$ and $(I' \blacktriangleright O')$, respectively, is described by a pair of functions A and R with

$$A : \mathbb{H}(I' \cup O') \to \mathcal{P}(\mathbb{H}(I \cup O))$$
$$R : \mathbb{H}(I \cup O) \to \mathcal{P}(\mathbb{H}(I' \cup O'))$$

that relate the interaction on an abstract level with corresponding interaction on the more concrete level as illustrated by Figure 11. A is called the *abstraction* and R is called the *representation*. R and A are called a *refinement pair*. Calculating a representation for a given history and then its abstraction yields the old history again. Using sequential composition, this is expressed by the requirement:

$$A \circ R = \{Id\},$$

where Id denotes the identity relation and $\circ$ the sequential composition defined as follows:

$$(R \circ A)(x) = \{y \in A(z) : z \in R(x)\}.$$

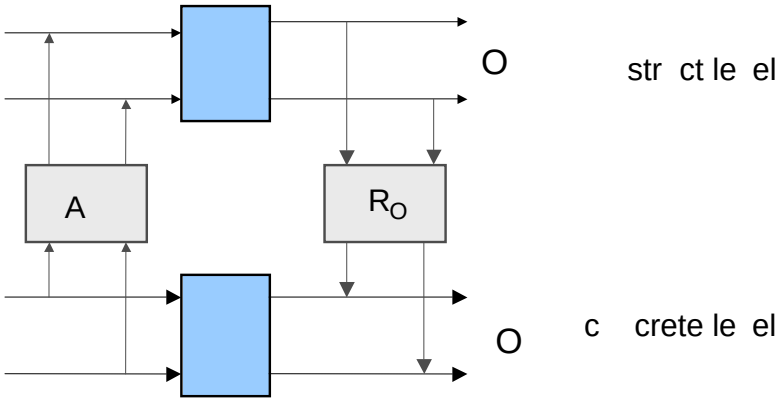

Figure 11: Interface Refinement of a Stream Processing Function

Definition 14 (Interaction Refinement) *A stream-processing function F' with syntactic interface $(I' \blacktriangleright O')$ is an interface refinement of a function F with interface $(I \blacktriangleright O)$ if there exists a refinement pair*

$$A_I : \mathbb{H}(I') \mapsto \mathcal{P}(\mathbb{H}(I))$$
$$R_O : \mathbb{H}(O) \mapsto \mathcal{P}(\mathbb{H}(O'))$$

such that F' is a property refinement of $R_O \circ F \circ A_I$, i. e.,

$$F'(x') \subseteq (R_O \circ F \circ A_I)(x').$$

This formula essentially expresses that, for every "concrete" input history $x' \in \mathbb{H}(I')$, every concrete output $y' \in F'(x')$ can be also obtained by translating x' onto an abstract input history $x \in A_I(x')$ such that we can choose an abstract output history $y \in F(x)$ with $y' \in R_O(y)$.

✍ **Example 16** (Interaction Refinement)

Let's assume that the channel `result` is replaced by two channels $\texttt{result}_\wedge$ and $\texttt{result}_\vee$. The types of both channels are defined as $Type(\texttt{result}_\wedge) = Type(\texttt{result}_\wedge) = \mathbb{B} \cup \{\epsilon\} = \{0, 1, \epsilon\}$. The type of the channel `switch` is extended by the message ϵ: $Type(\texttt{switch}) = \{0, 1, \epsilon\}$. The types of the other channels are not modified. The resulting syntactic interface of F'_E is depicted in Figure 12.

F'_E outputs the conjunction of both values through the channel $result_\wedge$ and an ϵ through $result_\vee$ if it receives the message $\wedge$ through the channel *switch*. F'_E outputs the disjunction of both values through the channel $result_\vee$ and an ϵ through $result_\wedge$ if it receives the message $\vee$ through the channel *switch*. F'_E outputs a ϵ through both ports if if receives a ϵ through *switch*.

To show, that F'_E is an interaction refinement of F_E, we define an abstraction A and representation R. The abstraction $A_I : \mathbb{H}(I') \mapsto \mathcal{P}(\mathbb{H}(I))$ is defined by the following equation:

$$
\begin{aligned}
& x \in A_I(x') \Leftrightarrow \forall t \in \mathbb{N}_+ : \\
& (x'(switch).t \in \{\wedge, \vee\} \Rightarrow x(switch).t = x'(switch).t) \wedge \\
& (x'(switch).t = \epsilon \Rightarrow x(switch).t \in \{\wedge, \vee\}) \wedge \\
& (x(in_1).t = x'(in_1).t) \wedge (x(in_2).t = x'(in_2).t).
\end{aligned}
$$

This means, the messages $\wedge$ and $\vee$ on the channel `switch` as well as all messages on both channels i_1 and i_2 in F'_E are mapped to the same message on the same channels in F_E. The message ϵ on `switch` in F'_E is mapped to $\wedge$ or $\vee$ in F_E non-deterministically.

The refinement $R_O : \mathbb{H}(O) \mapsto \mathcal{P}(\mathbb{H}(O'))$ is given as:

$$
\begin{aligned}
& y' \in R_O(y) \Leftrightarrow \forall t \in \mathbb{N}_+ : \\
& y'(result_\wedge).t \in \{y(result).t, \epsilon\}) \wedge \\
& y'(result_\vee).t \in \{y(result).t, \epsilon\}).
\end{aligned}
$$

The output message on the channel `result` in F_E is non-deterministically mapped to the same message or an ϵ on both output ports $\texttt{result}_\wedge$ and $\texttt{result}_\vee$ in F'_E. With the so-defined abstractions and representation, the condition for an interaction refinement is fulfilled:

$$F'_E(x') \subseteq (R_O \circ F_E \circ A_I)(x').$$

□

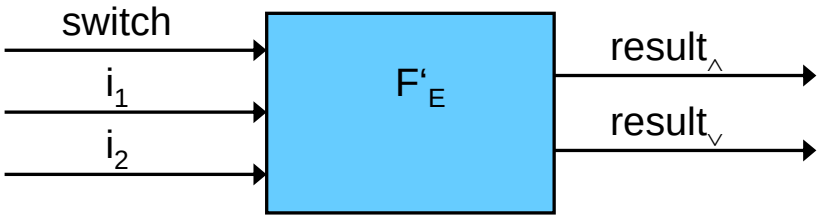

Figure 12: Interaction Refinement of the Evaluator

5 Basic Modeling Concepts: State Machines

One common way to model behavior, i. e., the behavior given by the stream processing functions, is to describe it by a state machine. This leads to a state view of systems.

A state machine consists of a state space and its state transitions, which represent one step of a system execution leading from a given state to another state of the system.

Definition 15 (State Machine) *Given a state space Σ and a set of messages M, a state machine (Δ, Λ) with inputs and outputs according to the syntactic interface $(I \blacktriangleright O)$ is given by a set $\Lambda \subseteq \Sigma$ of initial states as well as a state transition function*

$$\Delta : (\Sigma \times (I \to M)) \to \mathcal{P}(\Sigma \times (O \to M))$$

For each state $\sigma \in \Sigma$ and each valuation $u : I \to M$ of the input channels in I (the input messages received in a time interval t) we obtain a pair $(\sigma', s) \in \Delta(\Lambda, u)$ that contains the successor state σ' and a valuation $s : O \to M$ of the output channels consisting of the messages produced by the state transition. If the output depends only on the state we speak of a Moore machine, otherwise of Mealy machine. As we will see in the next subsection, each Moore machine induces a strictly causal stream processing function. Mealy machines, in contrast, allow to describe weakly causal functions.

5.1 From State Machines to Stream Processing Functions and Back Again

In this section we relate the afore introduced state machines to the basic concepts of the modeling theory, i. e., the stream processing functions. We first show how we may derive an interface abstraction for a state machine and then show how to construct a canonical state machine for an interface behavior.

From State Machines to Stream Processing Functions Each state machine describes a stream processing function. Given a state machine, the corresponding stream processing function (its interface abstraction) can be derived as follows.

Each state transition function

$$\Delta : (\Sigma \times (I \to M)) \to \mathcal{P}(\Sigma \times (O \to M))$$

induces a function

$$B_\Delta : \mathcal{P}(\Sigma) \to (\mathbb{H}(I) \to \mathcal{P}(\mathbb{H}(O))).$$

B_Δ provides the interface abstraction for the state transition function Δ, i. e., the corresponding stream processing function. For each state set $\Lambda \in \Sigma$ and each input channel valuation $x \in \mathbb{H}(I)$, let $B_\Delta(\Lambda)(x)$ be the set Λ by set of all output histories generated by computations of Δ for the input history x starting with a state in Λ. Based on these definitions we relate state machines to their interface abstractions.

Definition 16 (Interface abstraction) *$B_\Delta(\Lambda)$ provides the interface abstraction of the behavior of the state machine (Δ, Λ).*

From Stream Processing Functions to State Machines. Reversely, each stream processing function can be transformed into an abstract state machine. Given a stream processing function

$$F : \mathbb{H}(I) \to \mathcal{P}(\mathbb{H}(O))$$

we define the state space by

$$\Sigma = \mathbb{F}(I \blacktriangleright O),$$

where $\mathbb{F}(I \blacktriangleright O)$ denotes all possible stream processing functions for a given syntactic interface $(I \blacktriangleright O)$. Based on that, we get a state transition function

$$\Delta_F : (\Sigma \times (I \to M)) \to \mathcal{P}(\Sigma \times (O \to M))$$

by the following definition

$$\Delta_F(G, a) = \{(H, b) \in \mathbb{F}(I \blacktriangleright O) \times (O \to M) : \forall x \in \mathbb{H}(I) : G(< a >^x) = \{< b >^y : y \in H(x)\}\},$$

where $G \in \mathbb{F}(I \blacktriangleright O)$ and $a \in (I \to M)$.

Note that for $< b >^y \in G(< a >^x)$ the value b does not depend on x according to the strong causality assumption. The function H in the formula above is called a resumption. It represents the new state of the machine after the transition represented by an I/O-function. The set of initial states of the state machine is $\{F\}$.

5.2 Composition of State Machines

The composition of State Machines is simple defined as their parallel composition. An important property of the composition is that the interface abstraction of a composed state machine equals the composition of the interface abstractions of the sub state machines.

5.3 Refinement of State Machines

Property refinement can easily be extended to state machines by referring to their interface abstractions: A state machine S_1 is the refinement of another state machine S_2 if the interface abstraction of S_1 is an refinement of the interface abstraction of S_2 according to Section 4.5. The same applies to interaction refinement of state machines.

Acknowledgments We are grateful to Manfred Broy, Maria Victoria Cengarle, and David Cruz for their advice on early versions of the paper.

References

[BKM07] Manfred Broy, Ingolf Krüger, and Michael Meisinger. A formal model of services. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 16(1), 2007.

[BKPS07] M. Broy, I.H. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, Feb. 2007.

[Bro97] M. Broy. Refinement of time. In M. Bertran and Th. Rus, editors, *Transformation-Based Reactive System Development, ARTS'97*, number LNCS 1231, pages 44 – 63. TCS, 1997.

[Bro07] Manfred Broy. Two sides of structuring multi-functional software systems: Function hierarchy and component architecture. pages 3–12, 2007.

[BS01] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[FHH+ar] Martin Feilkas, Alexander Harhurin, Judith Hartmann, Daniel Ratiu, and Wolfgang Schwitzer. SPES 2020: Motivation and Introduction of a System of Abstraction Layers for Embedded Systems. Technical report, Technische Universität München, To appear.

[Kru] Heiko Krumm. Website "Algorithmenspezifikationsweise und Modellierung". http://www4.cs.uni-dortmund.de/MA/hk/OrdnerVertAlgo/Specification.html. Accessed at 08/08/2009.