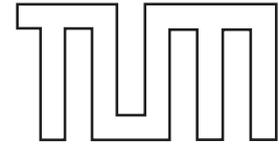


TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Software & Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy



SPES 2020 Deliverable D1.1.A

Initiale Modellierungstheorie



Software Plattform Embedded Systems 2020

Author: Judith Thyssen, TUM-SSE
Version: 1.0
Date: 31.12.2009
Status: Released

Inhaltsverzeichnis

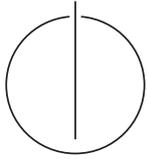
| | | |
|----------|-----------------------------|-----------|
| 1 | Einführung | 3 |
| A | Deliverable D1.1.A-1 | 4 |
| B | Deliverable D1.1.A-2 | 36 |
| C | Deliverable D1.1.A-3 | 49 |

1 Einführung

Ziel des Deliverable D1.1.A ist die Bereitstellung einer initialen Modellierungstheorie für das Projekt SPES durch die TUM-SSE. Auf Basis dieser initialen Version sind im weiteren Projektverlauf Änderungswünsche zu erfassen und eine überarbeitete Version der Modellierungstheorie zu erstellen. Das Deliverable D1.1.A umfasst drei Teildokumente:

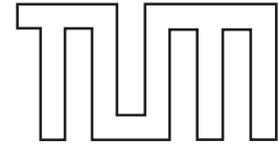
1. D1.1.A-1 „Modeling Theory – Motivation and Formal Foundations of a Comprehensive Modeling Theory for Embedded Software Systems“ (siehe Anhang [A](#))
Dieses Dokument stellt den Kernbestandteil des vorliegenden Deliverables dar; es enthält eine ausführliche Beschreibung der initialen Version der Modellierungstheorie.
2. D1.1.A-2 „Towards an Integrated Modeling Theory – A First Step in the Integration of the Semantic Foundations of Focus, Rich Components, and Mechatronic UML“ (siehe Anhang [B](#))
In diesem Dokument wird die vorgeschlagene Modellierungstheorie mit weiteren bei SPES-Partnern existierenden Ansätzen verglichen.
3. D1.1.A-3 „Literature Survey –Identification of Relevant Literature for SPES 2020 “ (siehe Anhang [C](#))
Dieses Dokument umfasst eine erste Literaturrecherche für das ZP-AP 1.

A Deliverable D1.1.A-1



TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Software & Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy



SPES 2020 Deliverable D1.1.A-1

Modeling Theory

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



Software Plattform Embedded Systems 2020

Author: Alexander Harhurin
Judith Hartmann
Daniel Ratiu
Version: 1.2
Date: 31.08.2009
Status: Released

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 FOCUS¹, 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.

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

Contents

| | | |
|----------|---|-----------|
| 1 | Motivation | 4 |
| 2 | Characteristics of the Addressed Systems | 5 |
| 2.1 | Addressed Aspects | 5 |
| 2.2 | Not (yet) Addressed Aspects | 6 |
| 3 | Requirements on the Modeling Theory | 7 |
| 4 | Comprehensive Modeling Theory | 8 |
| 4.1 | Service Hierarchies and Component Networks | 9 |
| 4.2 | Underlying Model of Computation | 11 |
| 4.3 | Stream Processing Function | 12 |
| 4.3.1 | Streams | 13 |
| 4.3.2 | Input/Output Channels and Channel Histories | 13 |
| 4.3.3 | Specification of Stream Processing Functions | 15 |
| 4.4 | Function Structuring | 18 |
| 4.4.1 | Composition | 18 |
| 4.4.2 | Combination | 20 |
| 4.5 | (Interface) Abstraction and Refinement | 24 |
| 5 | Basic Modeling Concepts: State Machines | 28 |
| 5.1 | From State Machines to Stream Processing Functions and Back Again | 28 |
| 5.2 | Composition of State Machines | 29 |
| 5.3 | Refinement of State Machines | 30 |
| | References | 30 |

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*

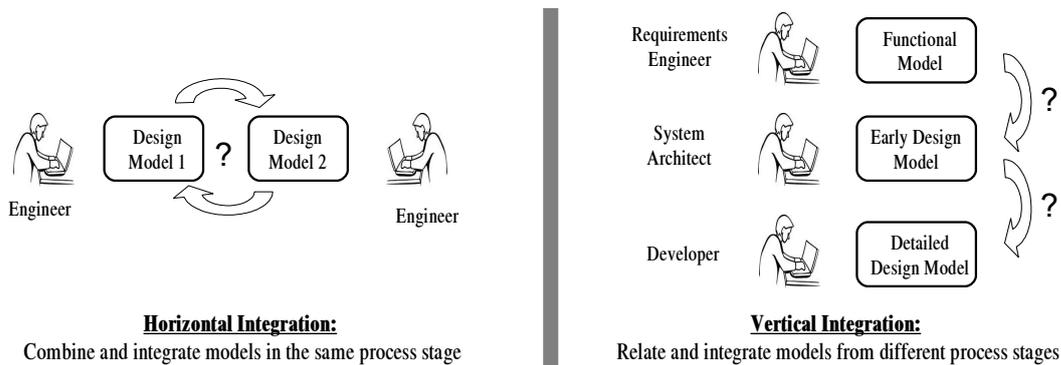


Figure 1: Horizontal vs. vertical integration problems.

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* [Bro06]. 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 tractable 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 losing 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.

²In [FHH⁺09] 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⁺09] for details). Thereby, our approach aims at modeling two fundamental, complementary views onto multi-functional systems. These views 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 `i1` and `i2`). The evaluator and its in- and output channels are depicted in Figure 2. All further details will be described at the appropriate places.

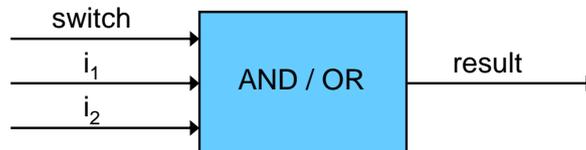


Figure 2: Running Example: Boolean Evaluator

 **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.

□

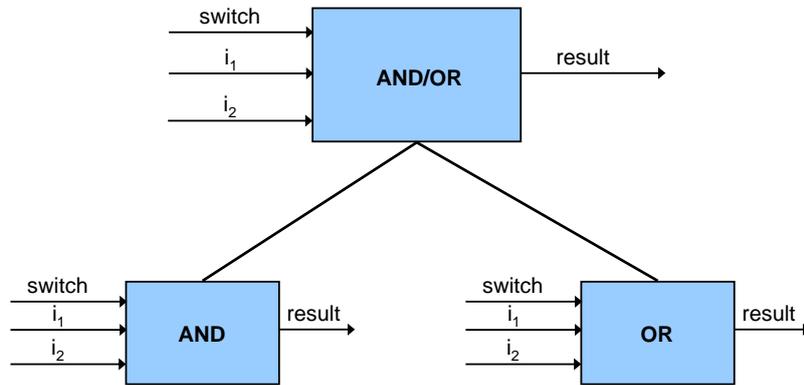


Figure 4: Service Hierarchy of the Evaluator

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).

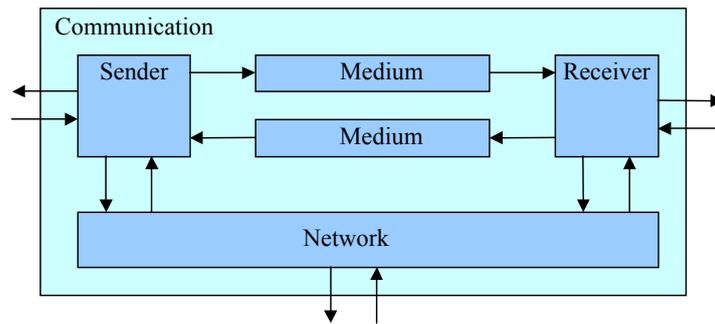


Figure 5: Component Network [Bro07]

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.

□

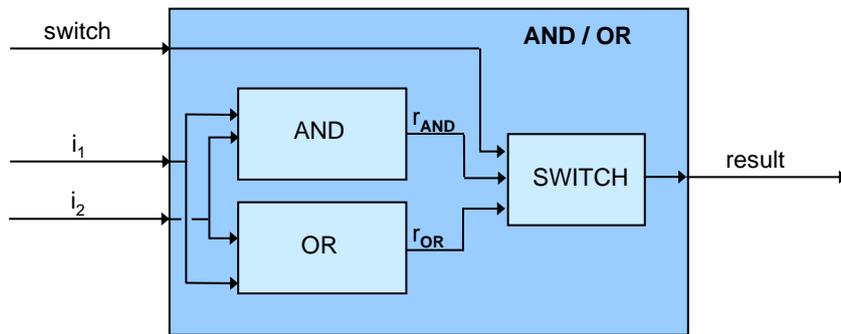


Figure 6: Component Network of the Evaluator

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.

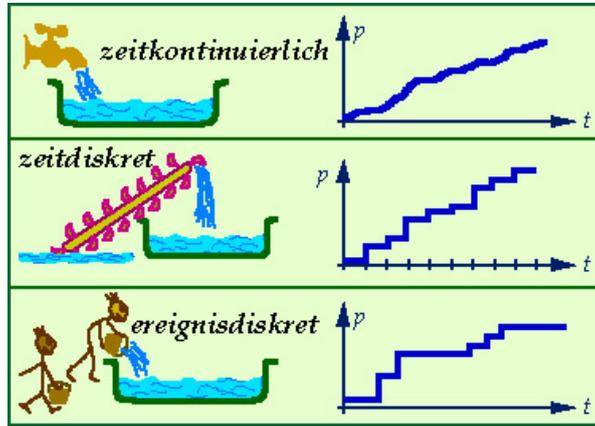


Figure 7: Illustration of Different Timing Models [Kru]

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*

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

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., $s = (1 \ 1 \ 1 \ \dots)$.

□

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 $I = \{i_1, \dots, i_n\}$ and output channels $O = \{o_1, \dots, o_n\}$. Channels are used as identifiers for streams.

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

$$Type : C \rightarrow \mathbf{Type},$$

³The theory is originally defined for stream processing functions $s : \mathbb{N}_+ \rightarrow M^*$, which assign a sequence of messages to each time interval. In order to keep the paper 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]

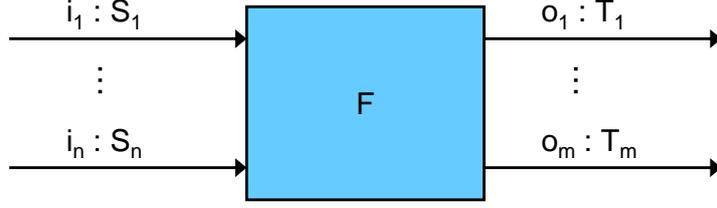


Figure 8: Graphical representation of a FOCUS function and its typed I/O channels

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

Example 4 (Channel Types)

In our running example, the channel types are defined as follows. $\text{Type}(\text{result}) = \text{Type}(i_1) = \text{Type}(i_2) = \mathbb{B} = \{0, 1\}$ and $\text{Type}(\text{switch}) = \{\wedge, \vee\}$.

□

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

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

such that $h(c)$ is a stream of the type $\text{Type}(c)$ 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 :

$$h : \begin{pmatrix} \text{switch} \\ i_1 \\ i_2 \end{pmatrix} \mapsto \begin{pmatrix} \wedge & \wedge & \vee & \vee & \wedge & \dots \\ 0 & 1 & 0 & 1 & 0 & \dots \\ 1 & 1 & 1 & 1 & 1 & \dots \end{pmatrix}.$$

$h(\text{switch}) = (\wedge \wedge \vee \vee \wedge \dots)$ results in the corresponding stream of the switch button, and $h(\text{switch}).3 = \vee$ 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 $I = \{\text{switch}, i_1, i_2\}$ and one output channel $O = \{\text{result}\}$. □

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*

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

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 $F_{AND/OR}$ is given by the following equation:

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

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

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

$$\begin{aligned} h_{in} : \begin{pmatrix} \mathbf{switch} \\ \mathbf{i}_1 \\ \mathbf{i}_2 \end{pmatrix} &\mapsto \begin{pmatrix} \wedge & \wedge & \vee & \vee & \wedge & \dots \\ 0 & 1 & 0 & 1 & 0 & \dots \\ 1 & 1 & 1 & 1 & 1 & \dots \end{pmatrix} \text{ and} \\ h_{out} : (\mathbf{result}) &\mapsto (1 \ 0 \ 1 \ 1 \ 1 \ 0 \ \dots). \end{aligned}$$

The evaluator function $F_{AND/OR}$ 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.

$$h'_{out} : (\text{result}) \mapsto (0 \ 0 \ 1 \ 1 \ 1 \ 0 \ \dots).$$

□

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*

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

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

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

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

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

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 $\text{dom}(F) \subset \mathbb{H}(I)$ 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 $\text{Type}(\text{switch}) = \{\wedge, \vee, \neg\}$, i. e., it is extended by the message \neg compared to the original definition. Then, the function $F_{AND/OR}$, 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:

$$(\exists t \in \mathbb{N}_+ : h(\text{switch}).t = \neg) \Rightarrow (F_{AND/OR}(h) = \emptyset).$$

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

□

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.

Thereby, 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 $Type_{C_1}(\cdot)$). 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

$$h_{in}|_{I'} : (switch) \mapsto (\wedge \ \wedge \ \vee \ \vee \ \wedge \ \dots) .$$

□

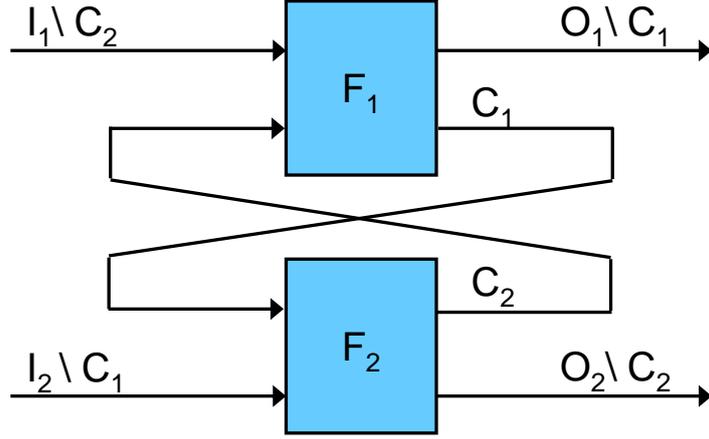


Figure 9: Function Composition with Feedback

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.

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 11, the function **AND/OR** 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:

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

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 “overlaid”. 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.

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.

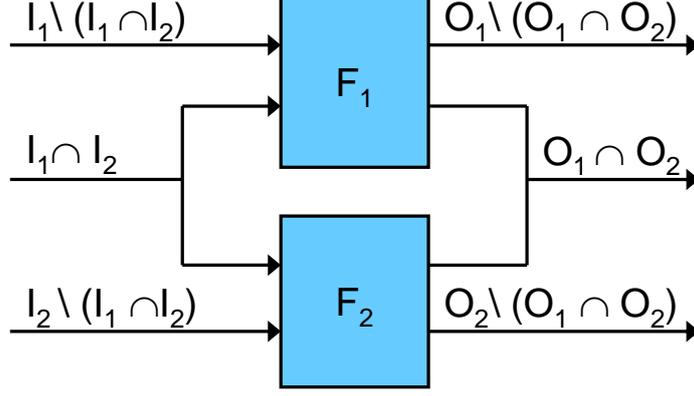


Figure 10: Function Combination

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

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

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

$$h'_{out} : (\text{result}) \mapsto (0 \ 0 \ 0 \ 0 \ 0 \ 0 \ \dots)$$

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

$$h''_{out} : (\text{result}) \mapsto (0 \ 0 \ 1 \ 0 \ 0 \ 0 \ \dots).$$

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

$$h'''_{out} : (\text{result}) \mapsto (1 \ 1 \ 1 \ 1 \ 1 \ 1 \ \dots)$$

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

□

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 10, 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: $I = \{\mathbf{i}_1, \mathbf{i}_2, \mathbf{switch}\}$ and $O = \{\mathbf{result}\}$. The behavior of the function **AND** is given as follows. For all $x \in \mathbb{H}(I)$:

$$y \in F_{AND}(x) \Leftrightarrow \forall t \in \mathbb{N}_+ : x(\mathbf{switch}).t = \wedge \Rightarrow y(\mathbf{result}).(t+1) = x(\mathbf{i}_1).t \wedge x(\mathbf{i}_2).t.$$

Whenever the message \wedge is received on the input channel **switch**, the values on \mathbf{i}_1 and \mathbf{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 $R \subseteq \text{dom}(F)$ such that:

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

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

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 foregoing 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 $(I' \blacktriangleright O') = (\{i_1, i_2\} \blacktriangleright \{result\})$. This function continuously calculates the conjunction of the values received on the channels i_1 and 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 $F_{AND/OR}$ where `switch` is permanently turned to \wedge . According to Definition 10, the function `AND'` is a restricted sub-function of the function $F_{AND/OR}$ with the set $R \subseteq \text{dom}(F_{AND/OR})$ defined as $R = \{x \in \text{dom}(F_{AND/OR}) : \forall t : x(\text{switch}).t = \wedge\}$.

□

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

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

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*

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

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 $F_{\text{AND/OR}}$:

$$\text{AND} \oplus \text{OR} = \text{AND/OR}.$$

In Figure 4 on Page 10 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⁺09]).

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⁺09]). 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 channels). 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 $F_{AND/OR}$ has the same syntactic interface as `AND`. However, $F_{AND/OR}$ 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`, $F_{AND/OR}$ defines the output also in these cases (disjunction of the inputs). Thus, $F_{AND/OR}$ is a property refinement of `AND`: $AND \rightsquigarrow F_{AND/OR}$.

□

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

$$\begin{aligned} A &: \mathbb{H}(I' \cup O') \rightarrow \mathcal{P}(\mathbb{H}(I \cup O)) \\ R &: \mathbb{H}(I \cup O) \rightarrow \mathcal{P}(\mathbb{H}(I' \cup O')) \end{aligned}$$

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

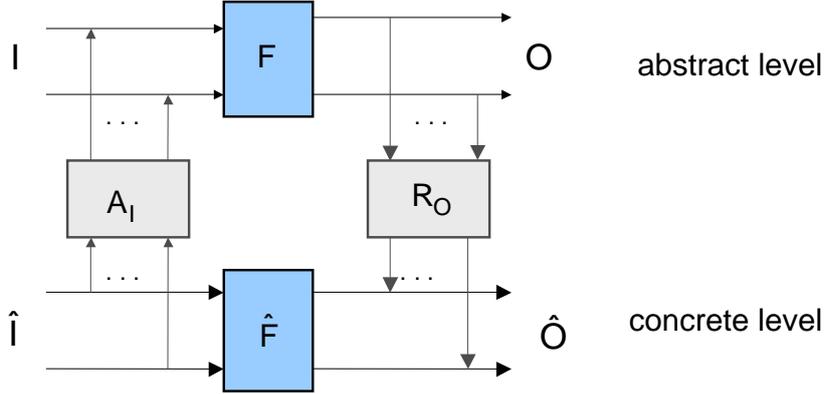


Figure 11: Interface Refinement of a Stream Processing Function

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:

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

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*

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

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 **result_∧** and **result_∨**. The types of both channels are defined as $Type(\mathbf{result}_\wedge) = Type(\mathbf{result}_\vee) = \mathbb{B} \cup \{\epsilon\} = \{0, 1, \epsilon\}$. The type of the channel **switch** is extended by the message ϵ : $Type(\mathbf{switch}) = \{0, 1, \epsilon\}$. The types of the other channels are not modified. The resulting syntactic interface of $F'_{AND/OR}$ is depicted in Figure 12.

$F'_{AND/OR}$ 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 $switch$. $F'_{AND/OR}$ 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 $switch$. $F'_{AND/OR}$ outputs an ϵ through both ports if it receives an ϵ through $switch$.

To show, that $F'_{AND/OR}$ is an interaction refinement of $F_{AND/OR}$, we define an abstraction A and a 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 $F'_{AND/OR}$ are mapped to the same message on the same channels in $F_{AND/OR}$. The message ϵ on **switch** in $F'_{AND/OR}$ is mapped to \wedge or \vee in $F_{AND/OR}$ 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 $F_{AND/OR}$ is non-deterministically mapped to the same message or an ϵ on both output ports **result_∧** and **result_∨** in $F'_{AND/OR}$. With the so-defined abstractions and representation, the condition for an interaction refinement is fulfilled:

$$F'_{AND/OR}(x') \subseteq (R_O \circ F_{AND/OR} \circ A_I)(x').$$

□

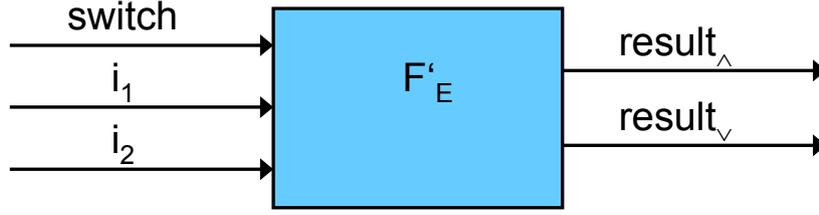


Figure 12: Interaction Refinement of the Evaluator

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*

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

For each state $\sigma \in \Sigma$ and each valuation $u : I \rightarrow M$ 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 $s : O \rightarrow M$ 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

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

induces a function

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

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 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

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

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

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

by the following definition

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

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

Note that for $\langle b \rangle^y \in G(\langle a \rangle^x)$ 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 simply defined as their parallel composition.

Definition 17 (Composition of State Machines) Let $SM_1 = (\Delta_1, \Lambda_1)$ and $SM_2 = (\Delta_2, \Lambda_2)$ be two state machines according to Definition 15, i. e., their state transition functions are as follows

$$\Delta_k : (\Sigma_k \times (I_k \rightarrow M)) \rightarrow \mathcal{P}(\Sigma_k \times (O_k \rightarrow M)) \quad k = 1, 2.$$

The composed state machine $SM = SM_1 || SM_2$ is defined over the syntactic interface $(I \blacktriangleright O)$ with

$$I = I_1 \cup I_2 \quad \text{and} \quad O = O_1 \cup O_2.$$

and given as follows: $SM = (\Delta, \Lambda)$ with

$$\begin{aligned}\Sigma &= \Sigma_1 \times \Sigma_2, \\ \Lambda &= \Lambda_1 \times \Lambda_2 \text{ and} \\ \Delta &= \Delta_1 \parallel \Delta_2.\end{aligned}$$

Thereby, the state transition function $\Delta : \Sigma \times (I \rightarrow M) \rightarrow \mathcal{P}(\Sigma \times (O \rightarrow M))$ is defined as

$$\begin{aligned}\forall x \in I, (s_1, s_2) \in \Sigma : \\ \Delta((s_1, s_2), x) &= \{((s'_1, s'_2), z|O) : x = z|I \wedge \\ &\quad (s'_1, z|O_1) \in \Delta_1(s_1, z|I_1) \wedge \\ &\quad (s'_2, z|O_2) \in \Delta_2(s_2, z|I_2)\}.\end{aligned}$$

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:

$$B_{\Delta_1 \parallel \Delta_2}(\Lambda_1 \times \Lambda_2) \equiv B_{\Delta_1}(\Lambda_1) \otimes B_{\Delta_2}(\Lambda_2)$$

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.

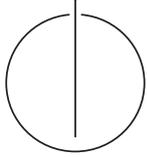
Acknowledgments We are grateful to Manfred Broy, Maria Victoria Cengarle, and David Cruz for their advice on early versions of the paper.

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. Salzmänn. 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.
- [Bro06] Manfred Broy. Challenges in automotive software engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 33–42, New York, NY, USA, 2006. ACM.
- [Bro07] Manfred Broy. Two Sides of Structuring Multi-Functional Software Systems: Function Hierarchy and Component Architecture. pages 3–12, 2007.

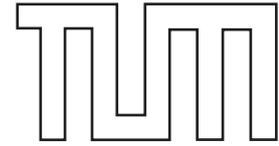
- [BS01] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.
- [FHH⁺09] Martin Feilkas, Alexander Harhurin, Judith Hartmann, Daniel Ratiu, and Wolfgang Schwitzer. SPES 2020: Motivation and Introduction of a System of Abstraction Layers for Embedded Systems. Technical report, Technische Universität München, 2009.
- [Kru] Heiko Krumm. Website "Algorithmenspezifikationsweise und Modellierung". <http://www4.cs.uni-dortmund.de/MA/hk/OrdnerVertAlgo/Specification.html>. Accessed at 08/08/2009.

B Deliverable D1.1.A-2



TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Software & Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy



SPES 2020 Deliverable D1.1.A-2

Towards an Integrated Modeling Theory

A First Step in the Integration of the Semantic Foundations of

FOCUS, Rich Components, and Mechatronic UML

Author: Alexander Harhurin (TUM-SSE)
Daniel Ratiu (TUM-SSE)
Judith Thyssen (TUM-SSE)
Jörg Holtmann (UPB)
Hary Hungar (OFFIS)
Eike Thaden (OFFIS)

Version: 1.1

Date: January 11, 2010

Status: DRAFT

About the Document

The aim of this paper is to compare three modeling approaches that are relevant within the SPES context, namely the FOCUS approach from TU-München, the Rich Components approach of OFFIS, and the Mechatronic UML of Uni Paderborn. Hereby, we focus on the comparison of the underlying modeling theories, as an essential prerequisite for the integration of the modeling approaches.

In this paper, we motivate the use of an integrated modeling theory, shortly introduce the three modeling approaches, compare them with respect to their underlying modeling theories, and sketch the next steps towards their integration into a comprehensive modeling theory that can be used as semantic basis for the whole SPES project.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 4 |
| 2 | Focus | 5 |
| 3 | Heterogeneous Rich Components | 6 |
| 4 | Mechatronic UML | 8 |
| 5 | Towards an Integrated Modeling Theory | 10 |
| | References | 11 |

1 Introduction

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. By *modeling theory* we mean the semantic foundation of a modeling language.

Many critical issues arise due to a missing, conflicting or inappropriate semantic foundation of the modeling languages used today:

- Many times, the semantics of a modeling language is only specified as prose (if at all) which leads to ambiguities in the interpretation of the created models and to inconsistencies between different implementations of the language in different tools.
- For other languages there exist several different semantic foundations. This results in different dialects of the same modeling technique which prevents an uniform treatment of the built models as well as the translation of models written in one formalism to another formalism.
- Moreover, different models applied in different phases of the development process are based on separated and unrelated modeling theories, which makes the transition from one model to another unclear and error-prone.

Within the SPES project, there exists different modeling techniques based on different modeling theories. As described before, this fragmentation and isolation of the semantic foundations prevent the integration of models written in the different modeling languages, the verification of their consistency, and the (automatic) translation of one model to another. Thus, the integration of the different semantic foundations is an indispensable prerequisite for a deep integration of the different modeling worlds and, consequently, for an integrated model-based development approach as overall result of the whole SPES project.

The integration of the semantic foundations aims at an *unified modeling theory* giving a consistent semantic foundation for all systems modeling approaches developed within SPES-project. Thereby, the modeling theory must be strong and expressive enough to support the whole development process starting from initial requirements down to a running system and to model all relevant aspects of hardware and software architectures of a system such as structuring software deployment, description of tasks and threads, as well as modeling behavior aspects of hardware.

In [HHR09] we introduced the FOCUS approach for the development of reactive systems. The aim of the current paper is to shortly compare the FOCUS approach to other approaches that are relevant within the SPES context, namely the Heterogenous Rich Components (HRC) approach of OFFIS, and the Mechatronic UML of University of Paderborn. Hereby, we focus on the comparison of the underlying modeling theories, since their integration is the essential basis for an integrated modeling approach desired in SPES.

It is important to remark that the modeling approaches can not be compared without respect to the purpose that they were designed for. Depending on the purpose an approach has to serve, a different mathematical foundation can be adequate. For example, if the models serve only as documentation and communication means, a strong mathematical foundation is less

important. However, to use the full power of model-based development including verification and code generation, a thorough mathematical foundation is essential. Moreover, the decision for a certain modeling theory also depends on the system to be developed. If the focus lies on the software that should run on discrete controllers on a discrete timing model is sufficient. However, if the theory aims at modeling the environmental/physical relations then a continuous timing model is needed.

In this paper, we sketch the differences between the different modeling approaches and show how they could be integrated. Therefore, we shortly recapitulate the different modeling approaches in Sections 2, 3, and 4. Thereafter, in Section 5, we compare the approaches with respect to their underlying modeling theory and sketch the next steps towards their integration into an unified modeling theory for the whole SPES project.

2 Focus

The FOCUS modeling approach introduced in [BS01, Bro05] aims at modeling two fundamental, complementary views onto multi-functional systems:

- **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*. 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].
- **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*. 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.

In the following we put this 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*.

FOCUS 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.

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.

FOCUS assumes a simple model of discrete time, where time advances in ticks. Therefore, events within a single tick can not be differentiated. 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. FOCUS assumes a global clock.

For more details about the issues presented above please refer to the SPES deliverable D1.1.A [HHR09].

3 Heterogeneous Rich Components

In addition to traditional static interfaces of UML-components that only define the interaction points of components, richer information is exposed on the boundaries of (Heterogeneous) Rich Components (HRCs) [DVM⁺05], in terms of contracts. Attached to a component, contracts abstract dynamics constraints on the component in terms of assumption-promise pairs, with the meaning that a promise offered by the component is guaranteed only if the corresponding assumption is fulfilled in the system context. The concept of HRC allows to combine components modeled in different front-end tools (Rhapsody, Matlab Simulink, etc.) into one abstract model with one well-defined semantics and to verify their behavioral requirements against each other.

According to [JMM08] RCs comprise the features presented below.

Different layers of abstraction. Different layers are identified in correspondence to different architectural abstractions of an embedded system. Examples of such architectural layers are: the functional layer that abstracts the functionality of the system; the ECU layer that abstracts the system as a network of tasks and signals between them on the one hand and a network of ECUs and buses on the other hand. RCs serve as the basic syntactical units of construction for all layers.

Structure of Systems. As the basic unit of construction, an RC represents the abstraction of some functionality and its properties. More specifically, an RC consists of the following ingredients:

- a (static) interface that defines the interaction/access points to the environment

- a set of contracts that specify the dynamic functional and non-functional properties of the Rich Component
- an implementation (e.g. C code, VHDL, etc.) that realizes the functionality (optional)

Interface. The interface of a RC is a set of interaction points aggregated into ports. Both service-oriented and data-oriented interaction points are supported. An RC may either provide or require a service. In both cases, the signature of the service, namely its name, parameters and return type, are depicted in the interface. In addition to services, flows are also defined as interaction points for data exchange. A *discrete* flow defines a data exchange point where there is always a value available and the values evolve discretely. A *continuous* flow defines a data exchange point where there is always a value available and the values evolve continuously. An *event* flow defines a data exchange point where only at certain time instants there is a value available and the values may change from instant to instant.

Behavior. One can distinguish between behavior requirements and actual behavior models. Contracts are used to specify behavior requirements for Rich Components. They can belong to different viewpoints (e.g. real-time, safety, etc.) to allow separation of concerns. Dominance (implication) is the central relation between contracts.

It is possible but not necessary to specify an implementation for a Rich Component. This implementation should satisfy the contracts attached to the RC. Implementations can be given in the form of code (generated or hand-written C code, etc.). Semantically, an implementation is a set of runs which may be defined by specific extended automata.

Contracts are built using state machines, which are hybrid automaton with a well defined C-like action language to express actions on the transitions and inside locations. However, according to the agreement of a series of meetings between TUM, OFFIS and Uni Paderborn, timed rather than hybrid automata are used in the praxis.

Timing Model and Synchronicity of Time. The underlying semantics of contracts uses dense time and distinguishes discrete steps and continuous evolutions. Time is evolving during continuous evolutions while discrete changes take no time. Currently a unique global physical time is used. Clocks are modeled as ordinary continuous variables with trivial evolution (e.g. for a clock c the differential equation $\dot{z} = 1$ is always active).

Communication Paradigm. Communication in HRC is synchronous. It happens continuously during continuous evolutions, and (also) instantly at discrete events. The latter means that a transition where an event is sent, all transitions which are waiting for that event are taken at the same time.

Probabilism Though the full semantical spectrum of HRCs includes probabilism (and its relation to nondeterminism), this is not expected to become important for SPES.

4 Mechatronic UML

Mechatronic UML is an UML-based approach for modeling mechatronic systems that combine technologies from mechanical engineering, electrical engineering, and computer science. For a broad acceptance (partly extended) UML diagrams and constructs are used, but enriched with underlying formal semantics. Mechatronic systems are real-time systems and contain both discrete control modes as well as implementations of continuous feedback controllers. Moreover, they are distributed. Different parts require coordination in order to act as a whole. Thereby, each part must be reconfigurable in order to comply with different roles that result from the coordination [BGT05]. Below are described the most important features of Mechatronic UML.

Structuring the System. To avoid the state explosion problem on verification of the whole system and to allow a compositional verification of the single parts of the system, the system is divided into structure and communication.

Coordination/Communication Patterns All communication relationships used in the system are modeled by *Real-Time Coordination Patterns*. A pattern consists of roles as well as a connector between these roles. Each role is described by a protocol automaton, more precisely an extension of statecharts which allows the specification of reactive real-time behavior (see below). For the connector it is possible to specify QoS-characteristics of the underlying transportation protocol (e.g. channel delays or loss of messages) with such an extended statechart. The overall communication behavior is described by the role behaviors, which exchange messages via the connector. The role behaviors contain parts of nondeterministic behavior, which is resolved when the pattern is applied on a component [GTB⁺03].

Component Model The Mechatronic UML uses UML component diagrams to specify the structure of the system. It distinguishes between the components and their instances at runtime by providing a type concept. The internal data structure of a component is specified with a class diagram. Components possess ports describing the interaction with the components' environment with an automaton. The ports have interfaces describing the messages, which can be exchanged over the ports/automata. Each port of a component is assigned a role of a coordination pattern from the pattern catalog. The port has to refine the pattern role by resolving the nondeterminism of the role automaton. The internal behavior of the component is described by an automaton synchronizing all port automata. Components can contain further components, so a system consisting of these components is spanning a component hierarchy. Moreover, a component can be discrete or continuous. Discrete components are the ones described above, while continuous components are abstractions of continuous feedback-controllers whose behavior might be described with a differential equation, for example. These continuous components are only embedded at the leaf level in the component hierarchy. A discrete component embedding both discrete and continuous components are also called hybrid components, as we see below. After all needed components have been fully specified, different instances of each can be created to employ them in a system at runtime [BGT05, GTB⁺03].

Specifying the Discrete Behavior. *Real-Time Statecharts (RTSCs)* specify the behavior of the discrete parts of the system [GB03], that is, the communication behavior of the patterns as well as the components' internal behavior. RTSCs combine constructs from timed automata like clocks, time guards, time invariants, worst time execution times, and deadlines with constructs from statecharts like hierarchical states or parallelism. Transitions make use of events, guards, time guards, resets, priorities, and synchronization. States make use of entry/exit operations at the entrance and the exit of a state, respectively, and of do operations that are periodically invoked during the residence in a state [GB03, p.24]. To continue the model-based approach, these operations are specified with a graph transformation formalism, which operates on data structures described by class diagrams and enables code generation. Since it is not trivial to determine WCETs for dynamic, object-oriented data structures, a method was developed to calculate and optimize WCETs directly on the graph transformations. Together with these model-based operations, C++ or Real-Time Java code for the whole RTSC can be synthesized. The semantics of RTSCs are specified by a mapping to Timed Automata, which also allows verification of the timed behavior by model checking.

Specifying the Hybrid Behavior. The hybrid behavior is modeled with *Hybrid Reconfiguration Charts (HRCs)*, an extension of RTSCs where each discrete state is associated with a configuration of discrete components or continuous blocks, leading to hybrid components [GBSO04]. A HRC can be applied as synchronization chart within each hybrid component. The formal semantics of HRCs are described by a mapping to Hybrid I/O Automata. To avoid a state explosion problem of the verification, these Hybrid I/O Automata are not model checked, but different verification techniques are applied. The correct embedding of component instances in a containing component instance is checked for each state syntactically by looking up the allowed configurations of the containing component type. As HRCs are extensions of RTSCs, the timing behavior is verified by model checking. After all, the continuous behavior is analyzed in the control engineering domain.

Compositional Verification. The division of the system into Real-Time Coordination Patterns and component types allows the verification by model checking the single parts of the system. First, a pattern is verified to show a correct communication behavior between the future components. Then, components with ports and interfaces are build, and the pattern roles are assigned to the ports. The refinement relation between roles and ports ensures that all patterns are employed correctly to the components. Finally, the components' internal behaviors, synchronizing all the refined roles, are verified. A system is build up only with instances of these verified components, leading to a correct system wrt. safety properties without running into scaling problems [GTB⁺03].

Timing Model. As mechatronic systems are distributed systems and thus contain asynchronous, event-based processes, in [BGHS04] a continuous timing model for the patterns was introduced to use its advantages like having an infinitely small interval between two events. A discretization can occur on several occasions and with different clock cycles for each target, for example, if a simulation for a system (part) is needed or a possible target hardware is known. In these cases, the verification properties must be preserved.

Synchronicity of Time. Mechatronic UML also has the common synchronization assumption induced by Timed Automata, that is, there are local clocks without clock drift. But the fact, that there is a local clock for each automaton, enables in turn to model and thus verify the clock drift explicitly [GHH06].

5 Towards an Integrated Modeling Theory

In Table 1, we compare the semantics of the different modeling approaches with respect to the following criteria: basic formalism used to express the behaviour, timing model, and communication paradigm, basic building blocks, compositionality, and refinement.

| Approach | Basic Formalism | Timing Model | Communication Paradigm | Basic Building Blocks | Compositionality | Notion of Refinement |
|-----------------|----------------------------------|--|--|---------------------------------------|------------------|---|
| FOCUS | Stream-processing functions | time discrete | asynchronous communication | components | yes | property refinement, interaction refinement |
| Rich Components | Timed automata (hybrid automata) | event-discrete/continuous time and continuous evolutions | synchronous communication | components | yes | dominance, satisfaction |
| Mechatronic UML | Timed automata (hybrid automata) | event-discrete/continuous time | asynchronous communication (synchronous within one RTSC) | components and communication patterns | yes | component/interface refinement(???), role/behavior refinement |

Table 1: Comparison of the modeling approaches

Scope of the SPES unified modeling theory. Even if we are aware of the fact, that for all of the presented approaches a lot research activities are currently being performed spanning over a brought variety of different aspects including hybrid systems, probabilistic systems or dynamic reconfiguration, after a series of meetings between TUM, OFFIS and Uni Paderborn, we agreed to neglect these complex aspects and to concentrate on discrete systems as basis for the first proposal of a unified modeling theory for the SPES project.

First stage of the integration. As a consequence, the next step towards an integrated modeling theory must be to find a mapping between the timed automata underlying the Rich Component and Mechatronic UML approach and the stream-based FOCUS approach. In [HHR09], we have already shown, how the stream-processing FOCUS functions can be translated into I/O-automata. Next, we have to show how timed automata and I/O-automata can be integrated.

Next stages of the integration. As mentioned before, we agreed not to consider continuous or hybrid systems in the first step. 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,

if necessary. In [Bro97], for example, it is already sketched that and how the FOCUS theory can be extended to more complex models of real or continuous time. Also, the integration of different, not necessarily orthogonal viewpoint specifications needs to be considered.

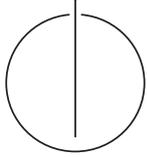
References

- [BGHS04] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling. Incremental design and formal verification with UML/RT in the FUJABA real-time tool suite. In *Proc. of the International Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004*, pages 1–20, October 2004.
- [BGT05] Sven Burmester, Holger Giese, and Matthias Tichy. Model-driven development of reconfigurable mechatronic systems with mechatronic uml. In Mehmet Aksit Uwe Assmann, Arend Rensink, editor, *Model Driven Architecture: Foundations and Applications*, volume 3599 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2005.
- [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.
- [Bro05] Manfred Broy. *Service-Oriented Systems Engineering: Specification and Design of Services and Layered Architectures - The Janus Approach*, volume 195, pages 47–81. Springer Verlag, July 2005.
- [BS01] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.
- [DVM⁺05] Werner Damm, Angelika Votintseva, Alexander Metzner, Bernhard Josko, Thomas Peikenkamp, and Eckard Böde. Boosting re-use of embedded automotive applications through rich components. In *Proceedings, FIT 2005 - Foundations of Interface Technologies*, aug 2005.
- [GB03] Holger Giese and Sven Burmester. Real-time statechart semantics. Technical report, University of Paderborn, 2003.
- [GBSO04] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. Modular design and verification of component-based mechatronic systems with online-reconfiguration. *SIGSOFT Softw. Eng. Notes*, 29(6):179–188, 2004.
- [GHH06] Holger Giese, Stefan Henkler, and Martin Hirsch. Analysis and modeling of real-time with mechatronic uml taking clock drift into account. In *Proc. of the International Workshop on Modeling and Analysis of Real-Time and Embedded Systems (MARTES), Satellite Event of the 9th International Conference on Model Driven*

Engineering Languages and Systems, MoDELS/UML2006, Genova, Italy, volume 343 of *Research Report*, pages 41–60, University of Oslo, October 2006.

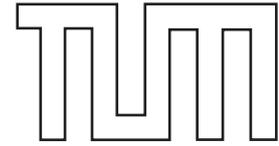
- [GTB⁺03] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan Flake. Towards the compositional verification of real-time uml designs. In *Proc. of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-11)*, pages 38–47. ACM Press, September 2003.
- [HHR09] Alexander Harhurin, Judith Hartmann, and Daniel Ratiu. Motivation and Formal Foundations of a Comprehensive Modeling Theory for Embedded Systems. Technical Report TUM-I0924, Technische Universität München, 2009.
- [JMM08] Bernhard Josko, Qin Ma, and Alexander Metzner. Designing embedded systems using heterogeneous rich components, 2008.

C Deliverable D1.1.A-3



TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Software & Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy



SPES 2020 Deliverable D1.1.A-3

Literature Survey

Identification of Relevant Literature for SPES 2020



Software Plattform Embedded Systems 2020

Author: Martin Feilkas
Alexander Harhurin
Judith Hartmann
Markus Hermannsdörfer
Florian Hölzl
Daniel Ratiu
Wolfgang Schwitzer

Version: 1.0

Date: 31.05.2009

Status: Released

Contents

| | | |
|----------|--|----------|
| 1 | Introduction | 3 |
| 2 | Relevant Literature | 3 |
| 2.1 | Seamless Model-based Development: from Isolated Tools to Integrated Model Engineering Environments | 3 |
| 2.2 | Challenges in Automotive Software Engineering | 3 |
| 2.3 | Model-driven architecture-centric engineering of (embedded) software intensive systems: modeling theories and architectural milestones | 3 |
| 2.4 | The 'Grand Challenge' in Informatics: Engineering Software-Intensive Systems | 4 |
| 2.5 | Umfassendes Architekturmodell für das Engineering eingebetteter Software-intensiver Systeme | 4 |
| 2.6 | An Architecture-Centric Approach towards the Construction of Dependable Automotive Software | 4 |
| 2.7 | A formal model of services | 4 |
| 2.8 | Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung | 5 |
| 2.9 | COLA - The COmponent LAnguage | 5 |
| 2.10 | Specification and Development of Interactive Systems | 5 |
| 2.11 | Case tools for embedded systemss | 6 |
| | References | 6 |

1 Introduction

This paper presents a survey of the literature relevant to the project “Software Plattform Embedded Systems 2020” (SPES). It focuses on literature written at the chair on Software & Systems Engineering at the Technische Universität München.

2 Relevant Literature

2.1 Seamless Model-based Development: from Isolated Tools to Integrated Model Engineering Environments

In [BFH⁺09] the authors envision a seamless development of models throughout the development process. The authors state that in the current practice model-based development is done in isolated tools using models are not built on a common semantic fundament. Furthermore the paper presents a vision what future modeling languages and techniques should look like. The authors call for a deep integration of the model-based development tools based in a common model repository. The different models are regarded as views onto a (potentially huge) comprehensive meta-model. To enable a very controlled execution of the development process, the workflows that are defined by the development process applied in a project should also be integrated into such a tooling environment. A mechanism to compose modeling languages is proposed to enable a domain appropriate modeling by integrating domain-specific and general-purpose modeling languages based on a building blocks principle. To implement such an approach the architecture of a common tooling platform is described. Additionally, technical as well as political barriers to the development of a common tooling platform are presented.

2.2 Challenges in Automotive Software Engineering

In [Bro06a] the author presents an overview of current and future challenges of automotive software engineering. The deficiencies of a traditionally bottom up oriented automotive software development process are pointed out. The author gives motivation to employ comprehensive model-based software development in general and specifically in the automotive domain. The characteristics of the automotive domain are described with respect to the specific demands of software production. The necessity of a seamless modeldriven development process is emphasized and an appropriate comprehensive architecture that covers all relevant levels of the system under design is introduced. The potentials of the presented approach are pointed out in contrast to possible weaknesses that modeling in the automotive sector still shows today.

2.3 Model-driven architecture-centric engineering of (embedded) software intensive systems: modeling theories and architectural milestones

[Bro07] presents a motivational overview of the Focus-like theory, the architecture, and the model based development process. This paper is quite simple and understandable by the industry.

2.4 The 'Grand Challenge' in Informatics: Engineering Software-Intensive Systems

[Bro06b] is an overview paper that presents the motivation for model-based development and its influence on more areas of software engineering. One of the key statements is that 'Developing a methodology for specifying and verifying software-intensive systems poses a Grand Challenge that a broad stream of research must address'. Furthermore the author states that *architecture* plays a dominant role and 'The architecture provides the blueprint for the development process and for integration'. The author concludes that while 'Modeling is the heart of engineering, yet the theory of modeling for software-intensive systems remains incomplete'.

2.5 Umfassendes Architekturmodell für das Engineering eingebetteter Software-intensiver Systeme

[BFG⁺08] introduces a framework of abstraction layers for the specification of software intense, embedded systems. The basic idea is to specify a system on consecutive abstraction layers, each one giving a more detailed model of the system, where the highest layer reflects a very abstract, informal description of a system, while the lowest one represents a concrete deployable implementation. The introduced framework provides the necessary basis for an systematic software engineering process and future tool support. The paper motivates the general use of different abstraction layers and includes for each layer a motivation and a short description of its basic modeling constructs.

Though describing the layered framework only from a high-level perspective, the paper represents a good starting basis for the definition of a system of abstraction layers in SPES (AP 1.2).

An english version of the paper is in preparation.

2.6 An Architecture-Centric Approach towards the Construction of Dependable Automotive Software

[WFH⁺06] contains an introduction to the automotive-specific modeling language CAR-DL (Combined Architecture Description Language) developed within the project "mobilsoft". The CAR-DL is based on a system of automotive-specific abstraction levels. This system as well as, for each abstraction level, the corresponding modeling constructs of the CAR-DL and their semantics are described in the paper.

The framework described in [BFG⁺08] was heavily inspired by this work. The paper is relevant for AP 1.2.

2.7 A formal model of services

In [BKM07], based on the FOCUS theory, a theory and formal model of services is introduced. In contrast to FOCUS components, which define total behavior, services allow for partial behavioral specifications. The paper gives a theoretical foundation of the notion of services and

shows how services can be structured and how software architectures can be composed of services and components.

The paper might be interesting for AP 1.1.

2.8 Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung

[BR07] contains the motivation and introduction of the formal foundations a FOCUS-like modelling theory.

The paper is relevant for AP 1.1.

2.9 COLA - The COmponent LAnguage

The project Base.XT developed an integrated modeling language which covers all abstraction levels identified earlier: the feature architecture, the logical architecture, and the technical architecture. The modeling theory underlying the modeling language is based on perfect synchrony (also known as weak causality), i.e. computation and communication take no time [KTB⁺07].

The *feature architecture* structures the features of a system in a hierarchy, and makes the dependencies between them explicit (feature interaction) [Rit08]. The transition from the feature architecture to the logical architecture is far from clear, although it is currently assumed that it is a combination of refactoring and refinement.

The *logical architecture* decomposes the system according to control flow (automata) and according to data flow (networks) into modular components that can be realized independently of each other [KTB⁺07].

The *cluster architecture* clusters the logical architecture into distributable entities (clustering), and maps them onto a hardware architecture (allocation). Both clustering and allocation can be performed without any user interaction. Finally, code can be generated from the technical architecture which can be deployed on real hardware [HTB08].

Unfortunately, the integrated modeling language is not fully integrated yet. However, the experiences gained through Base.XT might be interesting for AP 1.2.

2.10 Specification and Development of Interactive Systems

This book [BS01] provides a mathematical and logical foundation for the specification and development of interactive systems based on a model that describes systems in terms of their input- and output-behavior. Based on the model, the authors build a basic method, called FOCUS, that enables interactive systems to be described by characterizing their histories of message interaction. The book progresses from an introduction and guided tour of FOCUS through streams, specifications and their properties, and behavioral, interface, and conditional refinements.

2.11 Case tools for embedded systems

This case study [SHH⁺03] shows what modeling concepts are reasonable and can be useful in the domain of embedded systems. It also shows that general purpose object-oriented modeling is not desirable in this domain. The authors state that suitable tools have to address issues like describing structures more abstractly than on the level of class/object diagrams and the introduction of communication mechanisms more suitable than method or procedure. Furthermore, it is shown what models and description techniques are commonly accepted as essential, giving a snapshot of the CASEbased specification of embedded software. Additionally, the authors present what kind of tool support is available for the development of such specifications and what the resulting development process looks like with a focus on the design phase by comparing eight tools from different domains. A sketch of what should be expected from future tools for embedded software is given.

References

- [BFG⁺08] Manfred Broy, Martin Feilkas, Johannes Grünbauer, Alexander Gruler, Alexander Harhurin, Judith Hartmann, Birgit Penzenstadler, Bernhard Schätz, and Doris Wild. Umfassendes architekturmodell für das engineering eingebetteter software-intensiver systeme. Technical Report TUM-I0816, Technische Universität München, june 2008.
- [BFH⁺09] Manfred Broy, Martin Feilkas, Markus Herrmannsdoerfer, Stefano Merenda, and Daniel Ratiu. Seamless model-based development: from isolated tools to integrated model engineering environments. *Proceedings of the IEEE - Special Issue on Aerospace & Automotive*, to appear in 2009.
- [BKM07] Manfred Broy, Ingolf Krüger, and Michael Meisinger. A formal model of services. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 16(1), 2007.
- [BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische modellierung als grundlage der software- und systementwicklung. *Informatik Spektrum*, 30(1):3–18, 2007.
- [Bro06a] Manfred Broy. Challenges in automotive software engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 33–42, New York, NY, USA, 2006. ACM.
- [Bro06b] Manfred Broy. The 'Grand Challenge' in Informatics: Engineering Software-Intensive Systems. *Computer*, 39(10):72–80, 2006.
- [Bro07] Manfred Broy. Model-driven architecture-centric engineering of (embedded) software intensive systems: modeling theories and architectural milestones. *Innovations in Systems and Software Engineering*, 3(1):75–102, 2007.
- [BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

- [HTB08] Wolfgang Haberl, Michael Tautschnig, and Uwe Baumgarten. From COLA Models to Distributed Embedded Systems Code. *IAENG International Journal of Computer Science*, 35(3):427–437, September 2008.
- [KTB⁺07] Stefan Kugele, Michael Tautschnig, Andreas Bauer, Christian Schallhart, Stefano Merenda, Wolfgang Haberl, Christian Kühnel, Florian Müller, Zhonglei Wang, Doris Wild, Sabine Rittmann, and Martin Wechs. COLA – The component language. Technical Report TUM-I0714, Institut für Informatik, Technische Universität München, September 2007.
- [Rit08] Sabine Rittmann. *A methodology for modeling usage behavior of multi-functional systems*. PhD thesis, Technische Universität München, 2008.
- [SHH⁺03] Bernhard Schätz, Tobias Hain, Frank Houdek, Wolfgang Prenninger, Martin Rappl, Jan Romberg, Oscar Slotosch, Martin Strecker, and Alexander Wispeintner. Case tools for embedded systems. Technical Report TUM-I0309, Technische Universität München, 2003.
- [WFH⁺06] Doris Wild, Andreas Fleischmann, Judith Hartmann, Christian Pfaller, Martin Rappl, and Sabine Rittmann. An architecture-centric approach towards the construction of dependable automotive software. In *Proceedings of the SAE 2006 World Congress*, 2006.