Innovations Syst Softw Eng (2007) 3:75–102
DOI 10.1007/s11334-006-0011-y

ORIGINAL PAPER

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

Manfred Broy

Received: 14 April 2006 / Accepted: 18 August 2006 / Published online: 15 November 2006

Abstract Today, in general, embedded software is distributed onto networks and structured into logical components that interact asynchronously by exchanging messages. The software system is connected to sensors, actuators, human machine interfaces and networks. In this paper we study fundamental models of composed embedded software systems and their properties, identify and describe various basic views, and show how they are related. We consider, in particular, models of data, states, interfaces, functionality, hierarchically composed systems, and processes. We study relationships by abstraction and refinement as well as forms of composition and modularity. In particular, we introduce a comprehensive mathematical model and a corresponding mathematical theory for composed systems, its essential views and their relationships. We introduce two methodologically essential, complementary and orthogonal concepts for the structured modeling of multifunctional embedded systems in software and systems engineering and their scientific foundation. One approach addresses mainly tasks in requirements engineering and the specification of the comprehensive user functionality of multifunctional systems in terms of their functions, features and services. The other approach essentially addresses the design phase with its task to develop logical architectures formed by networks of interactive components that are specified by their interface behavior.

M. Broy (✉)
Institut für Informatik, Technische Universität München, 80290 München, Germany
e-mail: broy@in.tum.de
URL: http://wwwbroy.informatik.tu-muenchen.de

1 Motivation

Development of software in general, and in particular of embedded software, is today one of the most complex but at the same time most effective tasks in the engineering of innovative applications. Software drives innovation in many application domains. Modern software systems are typically embedded in technical or organizational processes, distributed, dynamic, and accessed concurrently by a variety of independent user interfaces. Just by constructing appropriate software we can provide engineering solutions that can calculate results, communicate messages, control devices, and illustrate and animate all kinds of information.

Making embedded systems and their development more accessible and reducing their complexity in terms of development, operation, and maintenance we use classical concepts from engineering, namely abstraction, the separation of issues, and appropriate ways of structuring software. Well-chosen models and their theories support such concepts. Also software development can be based on models of system behavior and, since well-chosen models are a successful way to understand software and hardware development, modeling is an essential and crucial issue in software construction (Fig. 1).

In the development of large, complex software systems it is simply impossible to provide one comprehensive model for the system in only one step. Rather we:

- Specify a system or subsystem first in terms of its functionality modeled by its interface, given in a structured way in terms of a function hierarchy
- Add stepwise details by property refinement

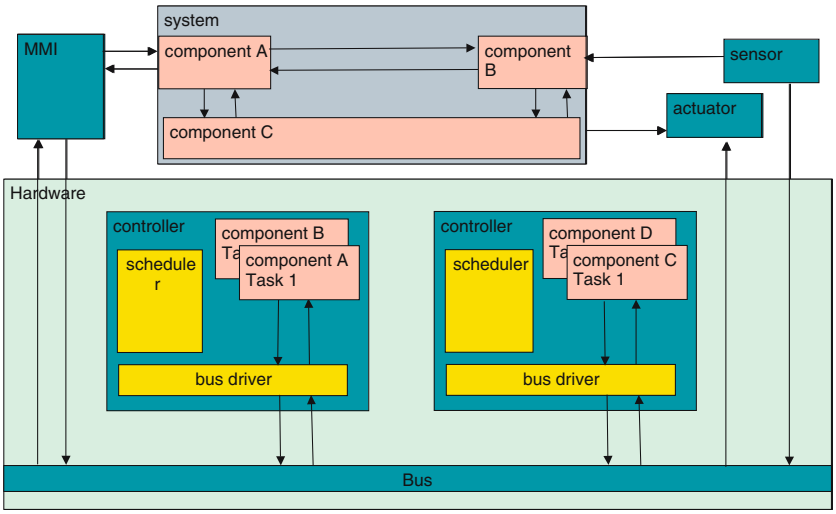

Fig. 1 Schematic architecture of an embedded system. *Top*: an abstract view; *bottom*: a concrete view

- Provide several views of the system in terms of states, architecture and processes
- Decompose the system hierarchically into components
- Construct a sequence of models on different levels of abstraction

Each step in these activities introduces models, refines them, verifies or integrates them. Figure 2 shows an idealized process based on modeling. The most remarkable issue is the independence of the architecture verification from its implementation, the modularity (mutual independency) of the component implementation, and the guaranteed correctness of the integration, which follows as a theorem from the correctness of the architecture and the correctness of each of the realizations of the components.

2 Comprehensive system modeling theory

In this section we introduce a comprehensive system model. It is aimed at distributed systems that interact by asynchronous message exchange in a time frame. It introduces the following views of systems:

- The *data* view, which introduces the fundamental data types
- The *syntactic interface* view, which shows through which events a system may interact with its environment
- The *state* view, where systems are modeled by state machines with input and output
- The *semantic interface* view, where systems are modeled in terms of their interaction with their environment
- The *architecture* view, where systems are modeled in terms of their structuring into components
- The *service* view, where the overall functionality of a large system is structured into a hierarchy of services and sub-services

We introduce mathematical models to represent these views. In the following chapters we show how these views are related.

This is to be seen as the basis of a theory of modeling for the engineering of software-intensive systems. Figure 3 gives an overview of these views in terms of the models that represent them.

2.1 The data model: algebras

We believe, like many others, that data types (typing) are a very helpful concept in structured modeling of application domains and software structures (for details, see [12]). From a mathematical point of view, a data model consists of a *heterogeneous algebra*. Such algebras are given by families of *carrier sets* and families of *functions* (including predicates and thus relations). More technically, we assume a set TYPE of types (sometimes also called *sorts*).

Given types we consider a set FUNCT of function symbols with a predefined functionality (TYPE* stands for the set of finite sequences over the set of types TYPE)

$$\mathbf{fct} : \mathrm{FUNCT} \rightarrow (\mathrm{TYPE}^* \times \mathrm{TYPE})$$

The function **fct** associates with every function symbol in FUNCT its domain types and its range type. Both the sets TYPE and FUNCT provide only names for sets and functions. The pair (TYPE, FUNCT) together with the type assignment **fct** is often called the *signature* of the algebra. The signature is the *static* (also called

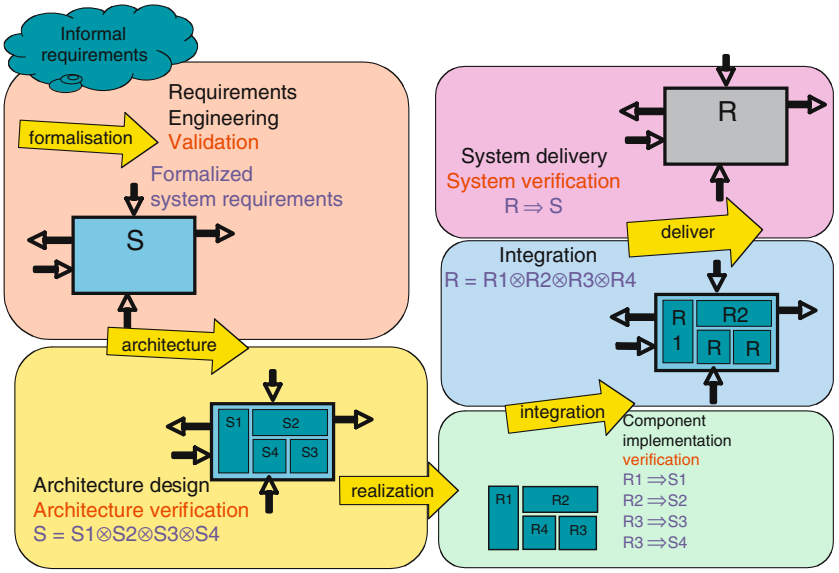

Fig. 2 Idealized modular development

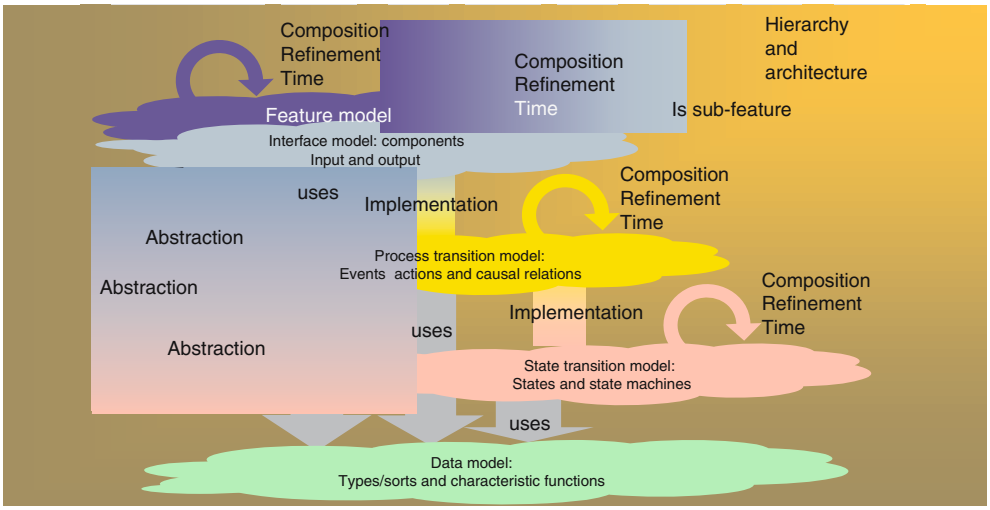

Fig. 3 The structure of modeling elements

syntactic) part of a data model. Every algebra with a signature (TYPE, FUNCT) provides a carrier set (a set of data elements) for every type and a function of the requested functionality for every function symbol. For each type $T \in$ TYPE we denote by CAR(T) its carrier set. There are many ways to describe data models such as algebraic specifications, E/R diagrams (see [29]) or class diagrams.

2.2 Syntactic interfaces of systems and their components

A system and also a system component is an active information-processing unit that encapsulates a state and communicates asynchronously with its environment through its interface, syntactically characterized by a set of input and output channels. This communication takes place within a global (discrete) time frame. In this section we introduce the notion of a syntactic interface of systems and system components. The syntactic interface models by which communication lines, which we call channels, the system or a system component is connected to the environment and which messages are communicated over the channels. We distinguish between input and output channels.

The channels and their messages determine the interaction events that are possible for a system or a system component. In the following sections we introduce several views such as state machines, semantic interfaces and architectures that all fit the syntactic interface view. As we will see, each system can be used as a component in a larger system and each component of a system

is a system by itself. As a result there is no difference between the notion of a system and that of a system component.

2.2.1 Typed channels

In this section we introduce the concept of a typed channel. A typed channel is a directed communication line over which messages of its specific type are communicated.

A *typed channel* c is a pair $c = (e, T)$ consisting of an identifier e, called the channel identifier, and the type T, called the channel type.

Let CID be a set of identifiers for channels. For a set

$$C \subseteq \text{CID} \times \text{TYPE}$$

of typed channels we denote by $\text{SET}(C) \subseteq \text{CID}$ its set of channel identifiers:

$$\text{SET}(C) = \{e \in \text{CID} : \exists T \in \text{TYPE} : (e, T) \in C\}$$

A set $C \subseteq \text{CID} \times \text{TYPE}$ of typed channels is called a *typed channel set*, if every channel identifier $e \in \text{SET}(C)$ has a unique type in C (in other words, we do not allow in typed channel sets the same channel identifier to occur twice with different types). Formally, we assume for a typed channel set:

$$(c, T1) \in C \wedge (c, T2) \in C \Rightarrow T1 = T2$$

By $\text{Type}_C(c)$ we denote for $c \in C$ with $c = (e, T)$ the type T.

A typed channel set $C1$ is called a *subtype* of a typed channel set $C2$ if the following formula holds:

$$(c, T1) \in C1 \Rightarrow \exists T2 \in \text{TYPE} : (c, T2) \in C2 \wedge \text{CAR}(T1) \subseteq \text{CAR}(T2)$$

Then $\text{SET}(C1) \subseteq \text{SET}(C2)$ holds. We write then

$C1$ **subtype** $C2$

Thus, a subtype $C1$ of a set of typed channels carries only a subset of the channel identifiers and each of the remaining channels only a subset of the messages. The idea of subtypes is essential for relating services (see later).

The **subtype** relation is a partial order, since it is obviously reflexive, transitive, and antisymmetric. In fact, it is a complete partial order on the set of typed channel sets. For two sets of typed channels $C1$ and $C2$ there is always a least set of typed channels C such that both $C1$ **subtype** C and $C2$ **subtype** C hold. C is called the *least super-type* of $C1$ and $C2$. Similarly there exists always a *greatest subtype* for two sets $C1$ and $C2$ of typed channels that is the greatest set of typed channels that is a subtype both of $C1$ and $C2$. Actually the set of the sets of typed channels forms a lattice. We denote by $\text{glb}\{C1, C2\}$ the greatest subtype and by $\text{lub}\{C1, C2\}$ the least upper bound (the least super-type) of the sets $C1$ and $C2$ of typed channels.

2.2.2 Syntactic interfaces

A syntactic interface of a system is defined by the set of messages that can occur as input and output. Since we assume that those messages are communication over channels, we introduce a syntactic interface of systems that consists of typed channels.

Let I be the set of input channels and O be the set of output channels of the system F. With every channel in the set $I \cup O$ we associate a data type indicating the type of messages sent along that channel. Then by $(I \blacktriangleright O)$ the *syntactic interface* of a system is denoted. A graphical representation of a system with its syntactic interface and individual channel types is shown in Fig. 4. It has the syntactic interface

$$(\{x_1 : S_1, \ldots, x_n : S_n\} \blacktriangleright \{y_1 : T_1, \ldots, y_m : T_m\})$$

By $(I \blacktriangleright O)$ and $\mathbb{SF}[I \blacktriangleright O]$ we denote the syntactic interface with input channels I and output channels O. By $\mathbb{SF}$ we denote the set of all syntactic interfaces for arbitrary channel sets I and O.

In the following we give three versions of representations of systems with such syntactic interfaces, namely state machines, stream functions, and architectures.

2.3 State view: state machines

One common way to model a system and its behavior is to describe it by a state machine in terms of a *state space* and its *state transitions*. This leads to a *state view* of systems.

2.3.1 State machine model: state transitions

Often systems can be modeled in an easily understandable way by a state transition machine with input and output. A state transition is one step of a system execution leading from a given state to a new state.

Given a state space Σ a state machine (Δ, Λ) with input and output according to the syntactic interface $(I \blacktriangleright O)$ is given by a set $\Lambda \subseteq \Sigma$ of initial states as well as a state transition function

$$\Delta : (\Sigma \times (I \to M^*)) \to \wp(\Sigma \times (O \to M^*))$$

For each state $\sigma \in \Sigma$ and each valuation $u : I \to M^*$ of the input channels in I by sequences we obtain by every pair $(\sigma', s) \in \Delta(\sigma, u)$ a successor state σ' and a

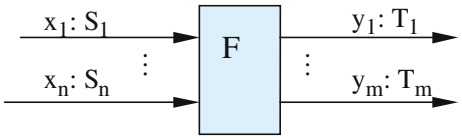

Fig. 4 Graphical representation of a system as a data flow node with input channels $x_1, \ldots, x_n$ and output channels $y_1, \ldots, y_m$ and their respective types

valuation $s : O \to M^*$ of the output channels consisting of the sequences produced by the state transition. If the output depends only on the state we speak of a Moore machine.

By SM[$I \blacktriangleright O$] we denote the set of all Moore machines with input channels I and output channels O. By SM we denote the set of all Moore machines.

2.3.2 Computations of state machines

In this section we define computations for state machines with input and output.

A computation for a state machine (Δ, Λ) and an input $\{x.t + 1 \in I \to M^* : t \in \mathbb{N}\}$ is given by a sequence of states

$$\{\sigma_t : t \in \mathbb{N}\}$$

and an output history $\{y.t + 1 \in O \to M^* : t \in \mathbb{N}\}$ such that for all $t \in \mathbb{N}$ we have:

$$(\sigma_{t+1}, y.t + 1) \in \Delta(\sigma_t, x.t + 1) \quad \text{and } \sigma_0 \in \Lambda$$

The history y is called an *output* of the computation of the state machine (Δ, Λ) for input x and initial state σ_0. We also say that the machine computes the output y for the input x and initial state σ_0.

The introduced state machines communicate a finite sequence of messages on each of their channels in each state transition. Thus a state transition comprises a set of events. We therefore call a state transition a macro-step. We assume that each macro-step is carried within a fixed time interval. It subsumes all events of interaction in this time interval and produces the state that is the result of these events. Thus, state machines model concurrency in the time intervals.

2.4 The interface model

In this section we introduce an interface model for systems. It describes the behavior of a system in the most abstract way by the relation between its streams of input messages and output messages.

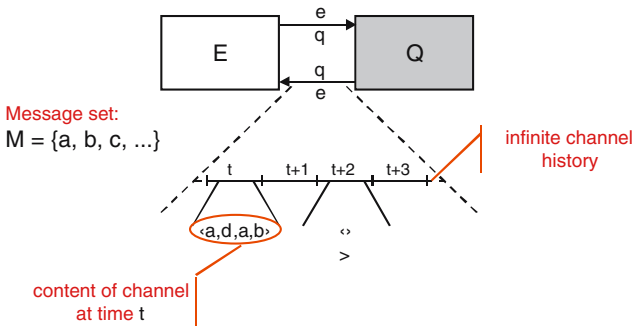

Fig. 5 Streams as models of the interaction between systems

2.4.1 Streams

Let M be a set of messages, for instance the carrier set of a given type. By M^* we denote the finite sequences of elements from M. By M^∞ we denote the set of infinite streams of elements of set M, which can be represented by functions $\mathbb{N}_+ \to M$ where $\mathbb{N}_+ = \mathbb{N}\backslash\{0\}$. By $M^\omega = M^* \cup M^\infty$ we denote the set of streams of elements from the set M, which are finite or infinite sequences of elements from M. A stream represents the sequence of messages sent over a channel during the lifetime of a system.

In fact, in concrete systems communication takes place in a time frame; we often find it convenient to be able to refer to this time. Therefore we work with *timed streams*.

Our model of time is extremely simple. We assume that time is represented by an infinite sequence of time intervals of equal length. In each interval on each channel a finite, possibly empty sequence of messages is transmitted. By $(M^*)^\infty$ we denote the set of infinite streams of sequences of elements of set M. Mathematically, a timed stream in $(M^*)^\infty$ can also be understood as a function $\mathbb{N}_+ \to M^*$.

On each channel that connects two systems there flows a stream of messages in a time frame as shown in Fig. 5.

Throughout this paper we work with a few simple notations for streams. We use, in particular, the following notations for a timed stream x:

$z^\frown x$: concatenation of a sequence or stream z to a stream x
$x.k$: the k-th sequence in the stream x
$x \downarrow k$: prefix of the first k sequences in the timed stream x
$M©x$: the stream obtained from the stream x by keeping only the messages in the set M (and deleting all messages not in M)

$\bar{x}$: the finite or infinite (untimed) stream that is the result of concatenating all sequences in x

Let $r \in (M^*)^\infty$; $\bar{r}$ is called the *time abstraction* of the timed stream r.

One of the advantages of timed streams is that we can easily define a merge operator. Merging two streams is a very basic concept. First, we introduce a function to merge sequences

$$\text{merge} : M^* \times M^* \to \wp(M^*)$$

where (for $s, s1, s2 \in M^*, a1, a2 \in M$):

$$\text{merge}(\langle\rangle, s) = \textit{merge}(s, \langle\rangle) = \{s\}$$
$$\text{merge}(\langle a1\rangle^\frown s1, \langle a2\rangle^\frown s2)$$
$$= \{\langle a1\rangle^\frown s : s \in \text{merge}(s1, \langle a2\rangle^\frown s2)\}$$
$$\cup \{\langle a2\rangle^\frown s : s \in \text{merge}(\langle a1\rangle^\frown s1, s2)\}$$

This function is easily extended to timed streams $x, y \in (M^*)^\infty$ as follows (for $t \in \mathbb{N}$)

$$\text{merge}(x, y).t = \text{merge}(x.t, y.t)$$

2.4.2 Channel valuations

A typed channel c is given by a channel type T and a channel identifier e, which denotes a stream with stream type Stream T. A channel is basically a name for a stream. Let C be a set of typed channels. A channel valuation is a mapping

$$x : C \to (M^*)^\infty$$

that associates a timed stream $x(c)$ with each channel $c \in C$, where the timed stream $x(c)$ carries only messages of the type of c. The set of valuations of the channels in C is denoted by $\vec{C}$.

The operators introduced for streams easily generalize to sets of streams and valuations. Thus we denote for a channel valuation $x \in \vec{C}$ by $\bar{x}$, its time abstraction, defined for each channel $c \in C$ by the equation

$$\bar{x}(c) = \overline{x(c)}$$

Note that $\bar{x}$ defines a time abstraction for the channel valuation x.

We generalize the idea of merging streams to valuations of channels by timed streams to the *direct sum* of two histories.

Definition (*Direct sum of histories*) *Given two sets C and C' of typed channels with consistent types (i.e., for joint channel identifiers their types coincide) and histories $z \in \mathbb{H}(C)$ and $z' \in \mathbb{H}(C')$ we define the* direct sum *of the histories z and z' by $(z \oplus z') \subseteq \mathbb{H}(C \cup C')$. It is specified as follows:*

$$\{y.c : y \in (z \oplus z')\} = \{z.c\} \Leftarrow c \in \text{SET}(C) \backslash \text{SET}(C'),$$
$$\{y.c : y \in (z \oplus z')\} = \{z'.c\} \Leftarrow c \in \text{SET}(C') \backslash \text{SET}(C)$$
$$(z \oplus z').c = \text{merge}(z.c, z'.c) \Leftarrow c \in \text{SET}(C) \cap \text{SET}(C')$$

This definition expresses that each history in the set $z \oplus z'$ carries all the messages the streams z and z' carry in the same time intervals and the same order.

The sum operator $\oplus$ is commutative and associative. The proof of these properties is rather straightforward.

Based on the direct sum we can introduce the notion of a *sub-history ordering*. It expresses that a history contains only a selection of the messages of a given history.

Definition (*Sub-history ordering*) *Given two histories $z \in \mathbb{H}(C)$ and $z' \in \mathbb{H}(C')$ where C subtype C' holds, we define the* sub-history ordering $\leq_{\text{sub}}$ *as follows:*

$$z \leq_{\text{sub}} z' \textit{ iff } \exists z'' : z \in z' \oplus z''$$

In fact, the sub-history ordering relation between histories is a partial ordering on the set of channel histories. Again the proof is rather straightforward. The empty stream is the least element in this ordering.

2.4.3 Interface behavior

In the following let I and O be sets of typed channels.

We describe the *interface behavior* of a system by an input/output (I/O) function that defines a relation between the input streams and output streams of a system. An I/O function is represented by a set-valued function on valuations of the input channels by timed streams.

$$F : \vec{I} \to \wp(\vec{O})$$

The function yields a set of valuations for the output channels for each valuation of the input channels.

If F is called *strongly causal* it fulfills the following *timing property*, which axiomatizes the time flow (for $x, z \in \vec{I}, y \in \vec{O}, t \in \mathbb{N}$) :

$$x \downarrow t = z \downarrow t \Rightarrow \{y \downarrow t + 1 : y \in F(x)\}$$
$$= \{y \downarrow t + 1 : y \in F(z)\}$$

Here $x \downarrow t$ denotes the stream that is the prefix of the stream x and contains t finite sequences. In other words, $x \downarrow t$ denotes the communication histories in the channel valuation x until time interval t. The timing property expresses that the set of possible output histories for the first $t + 1$ time intervals only depends on the input histories for the first t time intervals. In other words, the processing of messages in a system takes at least one time tick.

Strong causality implies that for every interface behavior F either for all input histories x the sets $F(x)$ are not empty or that for all input histories x the sets $F(x)$ are empty. This is easily proved by choosing $t = 0$ in the formula defining strong causality. In the case where for all input histories x the sets $F(x)$ are empty we speak of a *paradoxical* interface behavior.

If F is called *weakly causal* it fulfills the following *timing property*, which axiomatizes the time flow (for $x, z \in \vec{I}, y \in \vec{O}, t \in \mathbb{N}$) :

$$x \downarrow t = z \downarrow t \Rightarrow \{y \downarrow t : y \in F(x)\} = \{y \downarrow t : y \in F(z)\}$$

Weakly causal functions also show a proper time flow. A reaction to input must not occur before the input arrives. For them, however, there is no time delay required for output that reacts to input. As a consequence, there is no causality required within a time interval for the input and output. This may lead to causal loops. In the following we rather insist on strong causality (which is the counterpart to Moore machines as introduced above). However, many of our concepts and theorems carry over to the case of weak causality.

By $\mathbb{CF}[I \blacktriangleright O]$ we denote the set of all I/O functions with input channels I and output channels O. By $\mathbb{CF}$ we denote the set of all I/O functions for arbitrary channel sets I and O. For any $F \in \mathbb{CF}$ we denote by $\mathrm{In}(F)$ its set of input channels and by $\mathrm{Out}(F)$ its set of output channels.

2.5 The distributed system model: composed systems

An interactive composed system consists of a family of interacting subsystems called *components* (in some approaches also called *agents* or *objects*). These components interact by exchanging messages via their channels, which connect them. A network of communicating components gives a structural system view, also called the *system architecture*. Architectures can nicely be represented graphically by directed graphs. Their nodes represent components and their arcs communication lines (channels) on which streams of messages are sent. The nodes represent components, also called *data flow nodes*. The graph represents a net, also called a *data flow net*.

2.5.1 Uninterpreted architectures

We model architectures by data flow nets. Let $\mathbb{K}$ be a set of identifiers for components and I and O be sets of input and output channels, respectively. An uninterpreted architecture (composed system, distributed system) (ν, O) with syntactic interface $(I \blacktriangleright O)$ is represented by the mapping

$$\nu : \mathbb{K} \to \mathbb{SF}$$

that associates with every node a syntactic interface. O denotes the output channels of the system.

As a well-formedness condition for forming a net from a set of component identifiers $\mathbb{K}$, we require that for all component identifiers $k, j \in \mathbb{K}$ (with $k \neq j$) the sets of output channels of the components $\nu(k)$ and $\nu(j)$ are disjoint. This is formally guaranteed by the condition

$$k \neq j \Rightarrow \mathrm{SET}(\mathrm{Out}(\nu(k))) \cap \mathrm{SET}(\mathrm{Out}(\nu(j))) = \emptyset$$

In other words, each channel has a uniquely specified component as its source.[1] We denote the set of all (internal and external) channels of the net by the equation

$$\mathrm{Chan}((\nu, O)) = \{c \in \mathrm{In}(\nu(k)) : k \in \mathbb{K}\} \cup \{c \in \mathrm{Out}(\nu(k)) : k \in \mathbb{K}\}$$

The channel set O determines which of the channels occur as output. We assume that $O \subseteq \{\mathrm{c} \in \mathrm{Out}(\nu(k)) : k \in \mathbb{K}\}$.

The set

$$I = \mathrm{Chan}((\nu, O)) \backslash \{c \in \mathrm{Out}(\nu(k)) : k \in \mathbb{K}\}$$

denotes the set of input channels of the net. The channels in the set

$$\mathrm{Intern}((\nu, O)) = \{c \in \mathrm{Out}(\nu(k)) : k \in \mathbb{K}\} \backslash O$$

are called *internal*. By $\mathrm{CUS}[I \blacktriangleright O]$ we denote the set of all uninterpreted architectures with input channels I and output channels O. CUS denotes the set of all uninterpreted architectures.

2.5.2 Interpreted architectures

Given an uninterpreted architecture (ν, O) represented by the mapping

$$\nu : \mathbb{K} \to \mathbb{SF}$$

we get an interpreted architecture by a mapping

$$\eta : \mathbb{K} \to \mathbb{CF} \cup \mathrm{SM}$$

where $\eta(k) \in \mathrm{SM}[I' \blacktriangleright O']$ or $\eta(k) \in \mathbb{CF}[I' \blacktriangleright O']$ if $\nu(k) = (I' \blacktriangleright O')$. We write also (η, O) for the interpreted architecture.

By $\mathrm{CIS}[I \blacktriangleright O]$ we denote the set of all interpreted architectures with input channels I and output channels O. CIS denotes the set of all interpreted architectures.

[1] Channels that occur as input channels but not as output channels have the environment as their source.

A fully interpreted hierarchical system is defined iteratively as follows: a hierarchical system of level 0 and syntactic interface $(I \blacktriangleright O)$ is denoted by $\mathrm{HS}_0[I \blacktriangleright O]$ and defined by

$$\mathrm{HS}_0[I \blacktriangleright O] = \mathrm{SM}[I \blacktriangleright O] \cup \mathrm{C}\mathbb{F}[I \blacktriangleright O] \cup \mathrm{CIS}[I \blacktriangleright O]$$

By HS_0 we denote the set of all hierarchical systems of level 0 with arbitrary syntactic interfaces. A hierarchical system of level $j+1$ and syntactic interface $(I \blacktriangleright O)$ is denoted by $\mathrm{HS}_{j+1}[I \blacktriangleright O]$ and defined by an uninterpreted architecture represented by the mapping

$$\nu : \mathbb{K} \to \mathrm{S}\mathbb{F}$$

a mapping

$$\eta : \mathbb{K} \to \mathrm{HS}_j$$

where $\eta(k) \in \mathrm{SM}[I' \blacktriangleright O'] \cup \mathrm{C}\mathbb{F}[I' \blacktriangleright O'] \cup \mathrm{HS}_j[I' \blacktriangleright O']$ if $\nu(k) = (I' \blacktriangleright O')$.

By $\mathrm{HS}[I \blacktriangleright O]$ we denote the set of all hierarchical systems of arbitrary level with syntactic interface $(I \blacktriangleright O)$. By HS we denote the set of all hierarchical system of arbitrary level with arbitrary syntactic interfaces.

3 Structuring interfaces

Modern software systems offer their users large varieties of different functions, in our terminology called *services* or *features*. We speak of *multifunctional distributed interactive systems*.

3.1 Structuring multifunctional systems

In this section we concentrate on concepts supporting the structuring of the functionality of multifunctional systems. Our goal, in essence, is a scientific foundation for the structured presentation of system functions, organized in abstraction layers, forming hierarchies of functions and subfunctions (we speak of services, sometimes also of features).

Our vision addresses a service-oriented software engineering theory and methodology where services are basic system building blocks. In particular, services address *aspects* of interface behaviors and interface specifications. In fact, services induce also an *aspect-oriented view* of the functionality of systems and their architectures (see [23]).

Our approach aims at modeling two fundamental, complementary views onto multifunctional systems. These views are addressing the two most significant, in principle independent, dimensions of a structured modeling of systems in the analysis and design phases of software and systems development:

- User functionality and feature hierarchy: structured modeling of the user functionality of systems as done in requirements engineering and specification:
  - A multifunction system offers many different functions (*features* or *services* often captured by *use cases*); we are interested in a structured view of the family of these functions and their logical dependencies.
  - The main result is structured interface specifications of systems.
- Logical hierarchical component architecture: decomposition of systems into components as done in the design of the architecture:
  - A system is decomposed into a family of components that mutually cooperate to generate the behavior that implements the specified user functionality of the system.
  - The main result is the logical component architecture of the system.

These two complementary tasks of structuring systems and their functionality are crucial in the early phases of software and systems development. The architectural view was introduced at the end of the previous section. The functional hierarchy view is introduced in this section.

3.1.1 Services

The notion of a service is the generalization of the notion of an interface behavior of a system. It has a syntactic interface just like a system. Its behavior, however, is "*partial*", in contrast to the totality of a non-paradoxical system interface. Partiality here means that a service is defined (has a nonempty set of output histories) only for a subset of its input histories. This subset is called the *service domain*.

Definition (*Service interface*) *A service interface with the syntactic interface* $(I \blacktriangleright O)$ *is modeled by a function*

$$F : \vec{I} \to \wp(\vec{O})$$

that fulfills the strong causality property *only for the input histories with a nonempty output set* (*for* $x, z \in \vec{I}, y \in \vec{O}, t \in \mathbb{N}$) :

$$F(x) \neq \emptyset \neq F(z) \wedge x \downarrow t = z \downarrow t \Rightarrow \{y \downarrow t+1 : y \in F(x)\} = \{y \downarrow t+1 : y \in F(z)\}$$

Such a partial behavior function that fulfills this property is called strongly causal *too. The set*

$$\mathrm{dom}(F) = \{x : F(x) \neq \emptyset\}$$

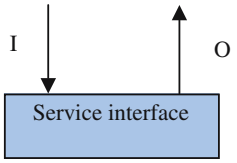

Fig. 6 General representation of a service interface

is called the service domain *of F. The set*

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

is called the service range *of F. By*

$$\mathbb{F}[I \blacktriangleright O]$$

we denote the set of all service interfaces with input channels I and output channels O. By $\mathbb{F}$ *we denote the set of all interfaces for arbitrary channel sets I and O.*

Obviously we have $\mathbb{CF} \subseteq \mathbb{F}$ and $\mathbb{CF}[I \blacktriangleright O] \subseteq \mathbb{F}[I \blacktriangleright O]$. In contrast to a system, where the causality property implies that for a system F either all output sets $F(x)$ are empty for all input histories x or none is, a service is a partial function, in general, with a nontrivial service domain.

To get access to a service, typically, certain access conventions have to be observed. We speak of a *service protocol*. Input histories x that are not in the service domain do not fulfill the service access assumptions. This gives a clear view: a non-paradoxical system is total, while a service may be partial. In other words, a non-paradoxical system is a total service. For a non-paradoxical system there exist nonempty sets of possible behaviors for every input history.

From a methodological point of view, a service is closely related to the idea of a use case as found in object-oriented analysis. It can be seen as a formalization of this idea (Fig. 6).

3.1.2 Structuring functionality into hierarchies of services

Given a service, we can refine it by extending its domain (provided it is not total). This means that we allow for more input histories with specified output histories and thus enlarge its service domain. Extending service domains is an essential step in service development for instance in making services error tolerant against unexpected input (streams). It is also a step from services to systems if the behaviors are finally made total.

We are, however, not only interested in specifying services and their extensions in isolation, but also interested in being able to specify such extensions in a structured way on top of already specified services. We are, in particular, looking for helpful relationships between the different services, looking for relations that are methodologically useful such as a refinement relation for services. These relations form the arcs of the functional hierarchy.

3.1.3 Structuring the functionality of multifunctional systems

In the following, we give a formal definition of the concept of services. This formalization is guided by our basic philosophy concerning services, which is briefly outlined as follows:

- Services are formalized and specified by patterns of interactions.
- Services are partial sub-behaviors of (interface) behaviors of systems.
- Systems realize families of services.
- Services address aspects of user functionality.

This philosophy of services is taken as a guideline for our way to build up a theory of services. We work out a formal approach to services. We specify services in terms of relations on interactions represented by streams. Consequences of our formal definitions are, in particular, as follows:

- A system is described by a total behavior.
- In contrast, a service is, in general, described by a partial behavior.
- A system is the special case of a total service.
- Services are related to systems; systems offer services.
- A multifunctional system/component is defined in terms of its family of services.

Our theory captures and covers all these notions. Open questions that remain are mainly of methodological and practical nature.

Multifunctional systems can incorporate large families of different, in principle, independent functions, which in our terminology offer different forms of services. So they can be seen as formal models of different use cases of a system.

Our basic methodological idea is that we should be able to reduce the complexity of the user functionality of a system, by describing the functionality of systems as follows:

- First we describe each of its single use cases independently by simple services

- Then we relate these services into a service hierarchy and
- Specify relationships between these individual services that show how the services influence or depend on each other.

Typically some of the services are completely independent and are just grouped together into a system that offers a collection of functionalities. In contrast, some services may restrict others. There is quite a variety of relations between the individual services of a system. While some services may just have small, often not very essential side effects on others, other services may rely heavily on other services that influence their behaviors in very essential ways.

3.1.4 Structuring systems into a hierarchy of services

A system or a combined service may actually implement or offer many independent services. In fact, we can structure the overall functionality of a multifunctional system into the hierarchy of its sub-services. We may decompose each system into a family of its sub-services and each of these services again and again into families of their sub-services.

Understanding the user functionality of a system requires the understanding of the single services, but also understanding of how they are related and mutually dependent. Our vision here is that we can introduce a number of characteristic relations between the services of a system such that in the end we describe a system structure by just specifying which services are available and how they are related. Each of the individual services is then described in isolation. This can reduce the complexity of the task of specifying a multifunctional system considerably.

Today's information processing systems offer many services as part of their overall functionality. We speak of *multifunctional systems*. We illustrate this idea by an informal description of a simple example of a multifunctional system.

Example (Communication unit) We look at the example of a simple communication network. It has three sub-interfaces and offers the following global services

- User identification, authentication and access control
- Communication
- User administration

At a finer-grained service level we may distinguish sub-services such as

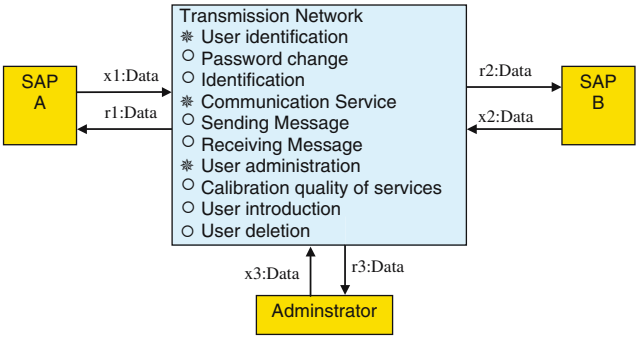

Fig. 7 Service structure and interface

- User login and identification
- Password change
- Sending message
- Receiving message
- Calibration of the quality of services
- User introduction
- User deletion
- Change of user rights

All these extended services can be described in isolation by specification techniques such as that introduced and demonstrated above. However, there are a number of dependencies between these services.

To obtain a comprehensive view of the hierarchy of services, we introduce a notion of user roles, such as that shown in our example:

- SAP A and B (SAP = service access point)
- Administrator

The set of services offered by a system can be informally described as shown in Fig. 7. In fact, it can be structured into a hierarchy of services and sub-services as shown in Fig. 8. By such a service hierarchy we obtain a structuring of a system into a hierarchy of services that are in the sub-service relation.

In such a service hierarchy we can relate services by introducing appropriate more specific relations between services, converting the service hierarchy into a directed graph.

3.2 Relating services

In this section we deal with a number of characteristic relations between services. The most significant one for requirements engineering is that of a sub-service, which will be formally introduced later. There are, of course, many more practically interesting relations between two services A and B besides the sub-service relation as

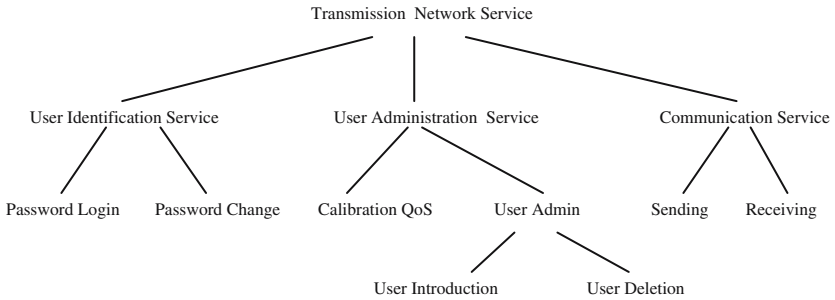

Fig. 8 Service hierarchy; the *lines* represent the sub-service relations

introduced in the previous section. Informal examples of such relations for services A and B are:

- A and B are mutually independent,
- A affects B,
- A controls B,
- A includes B.

Just characterizing the relations by well-chosen names is helpful, but in the end is not sufficient, since it does, of course, not fix their precise meaning. It is not difficult to dream up a number of further methodologically useful relations. Typical further characteristic relations between services are listed in the following:

- Uses
- Enables
- Changes
- Interferes (feature interaction)
- Is combined of
- Is sequentially composed of

Of course, just giving names to these relations is not sufficient. Since we have a formal notion of service, we are actually able to give formal definitions for such concepts of relations between services. Later, in the section on refinement, we introduce a few further examples for precise characterizations of relations between services.

4 Relating system views

In the previous chapter we introduced three system views: state machines, interfaces, and architectures. In this chapter we show how to relate one view to the other. Our basic goals are ways to go from one system view to the other. So a state machine defines an interface behavior and vice versa. An interpreted architecture defines a state machine as well as an interface behavior. This finally gives the flexibility of a truly hierarchical modeling method: every system – whether it is represented by a state machine, an interpreted architecture, or directly by an interface behavior – defines an interface behavior and can in turn be used as a component of an interpreted architecture defining a larger system. In short, every system is a component, which is itself a system.

4.1 From state machines to interfaces and back

In this section we study relationships between the state view and the interface view. 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.

4.1.1 From state machines to interfaces

In this section we define the interface abstraction for state machines. Each state transition function

$$\Delta : (\Sigma \times (I \to M^*)) \to \wp(\Sigma \times (O \to M^*))$$

induces a function

$$B_\Delta : \wp(\Sigma) \to (\vec{I} \to \wp(\vec{O})).$$

B_Δ provides the interface abstraction for the state transition function Δ. For each state set $\Lambda \subseteq \Sigma$ and each input channel valuation $x \in \vec{I}$, we specify the set $B_\Delta(\Lambda)(x)$ for the given state set Λ by set of all output histories generated by computations of Δ for the input history x starting with a state in Λ.

Based on these definitions we relate state machines to their interface abstractions.

> Given a state transition function Δ and $\Lambda \subseteq \Sigma, B_\Delta(\Lambda)$ provides the interface abstraction of the behavior of the state transition machine Δ for the initial states from Λ.

In this way we define a function

$$\text{abs}_{\text{SM}/\mathbb{F}} : \text{SM}[I \blacktriangleright O] \to \mathbb{F}[I \blacktriangleright O]$$

mapping state machines onto their interface behavior by

$$\text{abs}_{\text{SM}/\mathbb{F}}((\Delta, \Lambda)) = B_\Delta(\Lambda)$$

Note that we get the interface abstraction $B_\Delta(\Lambda)$ of a state machine (Δ, Λ) in this way.

4.1.2 From interfaces to state machines

In this section we show that an interface abstraction defines itself an abstract state machine. Given an interface

$$F : \vec{I} \to \wp(\vec{O})$$

we define the state space by

$$\Sigma = \mathbb{F}[I \blacktriangleright O]$$

We get a state machine

$$\Delta_F : (\Sigma \times (I \to M^*)) \to \wp(\Sigma \times (O \to M^*))$$

by the following definition (for $G \in \mathbb{F}[I \blacktriangleright O]$, $a \in (I \to M^*)$)

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

Note that for $\langle b \rangle^\frown y \in G(\langle a \rangle^\frown x)$ the value b does not depend on x according to the 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 initial states of the state machine are defined by the set $\{F\}$.

We define the function

$$\mathrm{emb}_{\mathrm{SM}/\mathbb{F}} : \mathbb{F}[I \blacktriangleright O] \to \mathrm{SM}[I \blacktriangleright O]$$

by

$$\mathrm{emb}_{\mathrm{SM}/\mathbb{F}}(F) = (\Delta_F, \{F\})$$

Furthermore, we easily define a function

$$\mathrm{abs}_{\mathrm{SM}/\mathrm{SM}} : \mathrm{SM}[I \blacktriangleright O] \to \mathrm{SM}[I \blacktriangleright O]$$

for state machines by

$$\mathrm{abs}_{\mathrm{SM}/\mathrm{SM}}((\Delta, \Lambda)) = (\Delta_F, \{B_\Delta(\sigma) : \sigma \in \Lambda\})$$

In fact, in this way we get the most abstract refinement of a state machine. We come back to this in the section on property refinement.

4.2 From interpreted architectures to state machines

In this section we study the relationship between the state view and the architecture view. We show how we may derive a state machine for an interpreted architecture.

Let an interpretation for the architecture (ν, O) be given. It is represented by a function

$$\eta : \mathbb{K} \to \mathrm{C}\mathbb{F} \cup \mathrm{SM}$$

we refine this function into a function

$$\eta' : \mathbb{K} \to \mathrm{C}\mathbb{F} \cup \mathrm{SM}$$

where $\eta'(k) = \eta(k)$ if $\eta(k) \in \mathrm{SM}$ and $\eta'(k) = \mathrm{emb}_{\mathrm{SM}/\mathbb{F}}(\eta(k))$ if $\eta(k) \in \mathrm{C}\mathbb{F}$. As a result $\eta'(k)$ is a state machine for all $k \in \mathbb{K}$. Now we construct a large state machine for the architecture. A state of this machine is defined by the mapping

$$\beta : \mathbb{K} \to \mathrm{USTATE}$$

where USTATE is the universe of all state spaces of state machines and for each $k \in \mathbb{K}$ the state $\beta(k)$ is a member of the state space of state machine $\eta'(k)$. Let STATE denote the set of all states of the architecture:

$$\mathrm{STATE} = \{\beta : \mathbb{K} \to \mathrm{USTATE}\}$$

The set of initial states is given by these functions, where for each $k \in \mathbb{K}$ the state $\beta(k)$ is a member of the set of initial states of state machine $\eta'(k)$. The state transition function

$$\Delta : (\mathrm{STATE} \times (I \to M^*)) \to \wp(\mathrm{STATE} \times (O \to M^*))$$

is given by the following condition (let $\beta, \beta' \in \mathrm{STATE}, a : I \to M^*, b : O \to M^*$)

$$(\beta', b) \in \Delta(\beta, a)$$

if and only if there exists a valuation of all channels by sequences of messages $z : \mathrm{Chan}((\nu, O)) \to M^*$

$$(\beta'(k), z|_{\mathrm{Out}(\nu(k))}) \in \Delta_k(\beta(k), z|_{\mathrm{In}(\nu(k))})$$

and

$$b = z|_O \quad a = z|_I$$

Here by $z|_C$ we denote the restriction of the channel set C. Formally if z is the evaluation of the channels in C' where $C \subseteq C'$ we define $z|\mathrm{C} \in \vec{C}$ by (for each channel $c \in C$)

$$z|_C(c) = z(c)$$

Since all machines $\beta(k)$ are Moore machines, this definition is consistent.

We can generalize this construction to hierarchical systems by an inductive definition over the hierarchy.

4.3 From interpreted architectures to interfaces

In this section we study the relationship between the interface view and the architecture view. We show how we may derive an interface abstraction for an interpreted architecture.

Let an interpretation for the uninterpreted architecture (ν, O) be given. It is represented by a function

$$\eta : \mathbb{K} \to \mathrm{C}\mathbb{F} \cup \mathrm{SM}$$

We refine this function to a function

$$\eta' : \mathbb{K} \to \mathrm{C}\mathbb{F} \cup \mathrm{SM}$$

where $\eta'(k) = \eta(k)$ if $\eta(k) \in \mathrm{C}\mathbb{F}$ and $\eta'(k) = \mathrm{abs}_{\mathrm{SM}/\mathbb{F}}(\eta(k))$ if $\eta(k) \in \mathrm{SM}$. As a result $\eta'(k)$ is an interface behavior for all $k \in \mathbb{K}$.

Each data flow net describes an I/O function. This I/O function is called the *interface abstraction* of the composed system described by the data flow net. We get an abstraction of a composed system to its interface by mapping it to an interface behavior in $\mathbb{F}[I \blacktriangleright O]$ where I denotes the set of input channels and O denotes the set of output channels of the data flow net. This interface view is represented by the system behavior $F \in \mathbb{F}[I \blacktriangleright O]$ specified by the following formula (note that $y \in \vec{C}$, where $C \equiv \mathrm{Chan}((\nu, O))$ as defined above):

$$F_{(\eta,O)}(x) = \{y|_O : y|_I = x \wedge \forall k \in \mathbb{K} : y|_{\mathrm{Out}(\nu(k))} \in \eta'(k)(y|_{\mathrm{In}(\nu(k))})\}$$

The formula essentially expresses that the output history of a data flow net is the restriction of a fixpoint for all the net equations for the output channels of all the components of the architecture.

We define a function

$$\mathrm{abs}_{\mathrm{CIS}/\mathbb{F}} : \mathrm{CIS} \to \mathbb{F}$$

by

$$\mathrm{abs}_{\mathrm{CIS}/\mathbb{F}}((\eta, O)) = F_{(\eta,O)}$$

This defines the interface abstraction for the interpreted architecture.

We can generalize this construction to hierarchical systems by an inductive definition over the hierarchy. This yields the interface abstraction $\mathrm{abs}_{\mathrm{HS}/\mathbb{F}} : \mathrm{HS} \to \mathbb{F}$.

4.4 Design: from interfaces to state machines and architectures

In the previous section we have demonstrated how to go from an interpreted architecture to an interface behavior. This defines an interface abstraction for an interpreted architecture. In system design we go in the other direction: given the specification of an interface behavior as the result of requirements engineering, in a design step we aim at the decomposition of the system into an interpreted architecture or the implementation of the interface behavior by a state machine. This is a step of creative engineering.

For designing an implementation of the interface behavior by a state machine the schematic way of defining a state machine for an interface behavior as shown above does not help at all; it is of rather theoretical interest. For an implementation a well-chosen representation of the state space is needed, on top of which the state transitions are defined.

For the design of an interpreted architecture or an implementation of the interface behavior by a state machine the definitions above that define interface abstractions are crucial, however. The design is correct, if the interface abstraction of the interpreted architecture or the state machine coincides with the specified interface behavior. Actually, the interface abstraction needs not be exactly the specified interface behavior, but only has to be a refinement of it. What refinement precisely means is defined in the following section.

5 Refinement

In requirements engineering, in the design and implementation phase of system development many issues have to be addressed such as requirements elicitation, conflict identification and resolution, information management as well as the selection of a favorable software architecture. These activities are connected with development steps. Refinement relations are the medium to formalize development steps and in this way the development process.

We formalize the following basic ideas of refinement of interfaces:

- *Property refinement* – enhancing requirements – allows us to add properties to a specification,
- *Glass-box refinement, implementation refinement* – designing implementations – allow us to decompose a system into a composed system or to give a state transition description for a system specification,
- *Interaction refinement* – relating *levels of abstraction* – 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 system.

In fact, these notions of refinement describe the steps needed in an idealistic view of a strictly hierarchical top-down system development. The three refinement concepts mentioned are formally defined and explained in detail in the following.

5.1 Property refinement

Property refinement is a classical concept in program development. It is based – as is deduction in logics – on logical implication. In our model this relates to set inclusion.

5.1.1 Property refinement of interfaces

Property refinement allows us to replace an interface behavior or an interface with one having additional properties. This way interface behaviors are replaced by more restricted ones. An interface

$$F : \vec{I} \to \wp(\vec{O})$$

is refined by a behavior

$$\hat{F} : \vec{I} \to \wp(\vec{O})$$

if

$$F \approx\!>_{\mathbb{F}} \hat{F}.$$

This relation stands for the proposition

$$\forall x \in \vec{I} : \hat{F}(x) \subseteq F(x)$$

Obviously, property refinement is a partial order and is therefore reflexive, asymmetric, and transitive. Note that the paradoxical system logically is a refinement for every system with the same syntactic interface.

A property refinement is a basic refinement step adding requirements as it is done step by step in requirements engineering. In the process of requirements engineering, typically the overall services of a system are specified. Requiring increasingly sophisticated properties for systems until a desired behavior is specified, in general, does this.

5.1.2 Property refinement of state machines

Property refinement can easily be extended to state machines by referring to their interface abstractions. Given two state machines $M_1 = (\Delta_1, \Lambda_1)$ and $M_2 = (\Delta_2, \Lambda_2)$ we call M_2 a *property refinement* of M_1 and write

$$M_1 \approx\!>_{\mathrm{SM/SM}} M_2$$

if

$$\mathrm{abs}_{\mathrm{SM}/\mathbb{F}}(M_1) \approx\!>_{\mathbb{F}} \mathrm{abs}_{\mathrm{SM}/\mathbb{F}}(M_2).$$

In this way we require only that the interface abstractions of both machines are in the property refinement relation. The two machines may have completely different state spaces.

Two state machines are called *equivalent* if for each input history their sets of output histories coincide. A state machine is called *equivalent to a behavior F*, if for each input history x the state machine computes exactly the output histories in $F(x)$.

A state machine (Δ_2, Λ_2) with transition function

$$\Delta_2 : (\Sigma_2 \times (I \to M^*)) \to \wp(\Sigma_2 \times (O \to M^*))$$

is called a *transition refinement* or a *simulation* of a state machine (Δ_1, Λ_1) with the transition function

$$\Delta_1 : (\Sigma_1 \times (I \to M^*)) \to \wp(\Sigma_1 \times (O \to M^*))$$

if there is a mapping $\rho : \Sigma_2 \to \Sigma_1$ such that for all states $\sigma \in \Sigma_2$, and all input $\alpha \in I \to M^*$ we have:

$$\{(\rho(\sigma'), \beta) : (\sigma', \beta) \in \Delta_2(\sigma, \alpha)\}$$
$$\subseteq \Delta_1(\rho(\sigma), \alpha), \quad \{\rho(\sigma) : \sigma \in \Lambda_2\} \subseteq \Lambda_1$$

A special case is given if $\wp$ is the identity; then the equation simplifies to:

$$\Delta_2(\sigma, \alpha) \subseteq \Delta_1(\sigma, \alpha) \wedge \Lambda_2 \subseteq \Lambda_1.$$

In this way we define refinement directly on the state space.

5.1.3 Property refinement of composed systems

As for state machines, property refinement can easily be extended to interpreted architectures by referring to their interface abstractions. Given two interpreted architectures S_1 and S_2, we write

$$S_1 \approx\!>_{\mathrm{CIS}} S_2$$

if

$$\mathrm{abs}_{\mathrm{CIS}/\mathbb{F}}(S_1) \approx\!>_{\mathbb{F}} \mathrm{abs}_{\mathrm{CIS}/\mathbb{F}}(S_2)$$

Here we require only that the interface abstractions of both systems are in the property refinement relation. The two systems may have completely different composition structures, however.

If the uninterpreted architectures of S_1 and S_2 coincide, then we can define a more specific refinement by requiring that each component of S_2 is a refinement of the respective component of S_1.

5.1.4 General property refinement of hierarchical systems

It is the idea of property refinement that we do not take care of the internal structure of systems, but only compare their interface abstractions. Given two hierarchically composed systems S_1 and S_2 in HS, we write

$$S_1 \approx\!> S_2$$

if

$$\mathrm{abs}_{\mathbb{F}}(S_1) \approx\!>_{\mathbb{F}} \mathrm{abs}_{\mathbb{F}}(S_2)$$

Property refinement is a straightforward concept. Basically we can always see property refinement as combined with interface abstraction.

If the uninterpreted architectures of S_1 and S_2 coincide, then we can define a more specific refinement by requiring that each component of S_2 is a refinement of the respective component of S_1.

5.2 Compositionality of property refinement

In our case, the proof of the compositionality of property refinement is straightforward. This is a consequence of the simple definition of composition. Let (ν, O) be an uninterpreted architecture with interpretations η and η' with the set $\mathbb{K}$ of components given by their interfaces. The rule of compositional property refinement reads as follows:

$$\frac{\forall K \in \mathbb{K} : \eta(k) \approx\!> \eta'(k)}{F_{(\eta,O)} \approx\!> F_{(\eta',O)}}.$$

Compositionality is often called *modularity* in system development. Modularity allows for a separate development of components.

> Modularity guarantees that separate refinements of the components of a system lead to a refinement of the composed system.

The property refinement of the components of composed systems leads to a property refinement for the composed system independently of the question whether the components are described by interfaces, state machines or processes.

5.3 Implementation refinement

Implementation refinement is formally only a special case of property refinement. While we may change many aspects of a system in property refinement, such as the uninterpreted architecture or the structure of the state space, implementation refinement has two restrictions: it maintains certain implementation aspects and it supports steps towards a design or implementation.

5.3.1 Design: implementation refinement of interfaces

Implementation refinement of interfaces is given by the replacement of an interface behavior, described by a refined function on streams, by a state machine or an interpreted architecture. Thus implementation refinement represents a design or implementation step. For an interface behavior

$$F : \vec{I} \to \wp(\vec{O})$$

a system S is an implementation refinement of F if

- S is an interface behavior and

  $$F \approx\!>_{\mathbb{F}} S$$

- S is a state machine and

  $$F \approx\!>_{\mathbb{F}} \mathrm{abs}_{\mathrm{SM}/\mathbb{F}}(S)$$

- or S is a composed system and

  $$F \approx\!>_{\mathbb{F}} \mathrm{abs}_{\mathrm{HS}/\mathbb{F}}(S).$$

We then write

$$F \approx\!>> S$$

We get an abstract implementation (for the formal definition of the relation $\mathrm{emb}_{\mathrm{SM}/\mathbb{F}}$ see Sect. 4.1.2)

$$F \approx\!>>_{\mathbb{F}/\mathrm{SM}} \mathrm{emb}_{\mathrm{SM}/\mathbb{F}}(F)$$

Again, implementation refinement is a partial order and therefore reflexive, asymmetric, and transitive. Moreover, the inconsistent specification logically described by false refines everything.

5.3.2 Implementation refinement of state machines

Given two state machines $M_1 = (\Delta_1, \Lambda_1)$ and $M_2 = (\Delta_2, \Lambda_2)$ over the same state space we call M_2 an implementation refinement of M_1 and write

$$M_1 \approx\!>> M_2$$

if

$$\Lambda_1 \subseteq \Lambda_2$$

and

$$\Delta_2(\sigma, x) \subseteq \Delta_1(\sigma, x)$$

for all $\sigma \in \Sigma, x \in (I \to M^*)$. In this case we maintain the structure of the state spaces and only reduce the sets of states in the transitions.

5.3.3 Implementation refinement of composed systems

Given two interpreted architectures $S1$ and $S2$ in CIS $[I \blacktriangleright O]$ with $S1 = (\eta_1, O)$ and $S_2 = (\eta_2, O)$ we write

$S_1 \approx>> S_2$

if S_1 and S_2 have the same set $\mathbb{K}$ of components and for all $k \in \mathbb{K}$

$\eta_1(k) \approx>> \eta_2(k)$

Each component in S_2 is an implementation refinement of the respective component in S_1. This is the relation of stepwise hierarchical decomposition if extended to hierarchical composed systems.

5.4 Granularity refinement: changing levels of abstraction

In this section we show how to change the levels of abstractions by refinements of the interfaces, state machines and processes. Changing the granularity of interaction and thus the level of abstraction is a classic technique in software system development.

5.4.1 Refinement of interfaces

Interaction refinement is the refinement notion for modeling development steps between levels of abstraction. Interaction refinement allows us to change for a component

- The number and names of its input and output channels
- The types of the messages on its channels determining the granularity of the communication

An *interaction refinement* is described by a pair of functions

$A : \vec{C}' \to \wp(\vec{C}) \quad R : \vec{C} \to \wp(\vec{C}')$

that relate the interaction on an abstract level with corresponding interaction on the more concrete level. This pair specifies a development step that is leading from one level of abstraction to the other one as illustrated by Fig. 9. Given an abstract history $x \in \vec{C}$ each $y \in R(x)$ denotes a concrete history representing x. Calculating a representation for a given abstract history and then its abstraction yields the old abstract history again. Using sequential composition, this is expressed by the requirement:

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

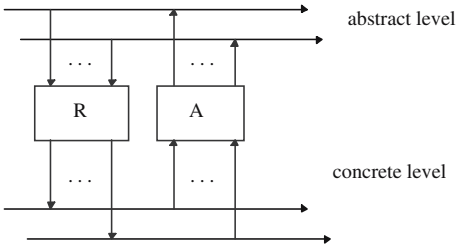

Fig. 9 Communication history refinement

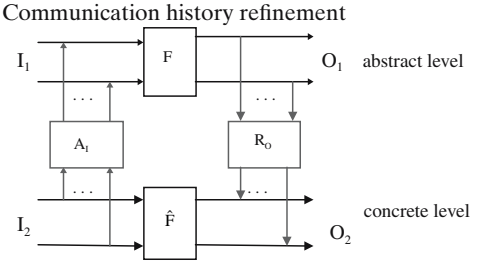

Fig. 10 Interaction refinement (U^{-1} simulation)

Let Id denote the identity relation and '$\circ$' the sequential composition defined as follows:

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

A is called the *abstraction* and R is called the *representation*. R and A are called a *refinement pair*. For untimed systems we weaken this requirement by requiring $R \circ A$ to be a property refinement of the untimed identity, formally expressed by the following equation:

$\overline{(R \circ A)(x)} = \{\bar{x}\}.$

This defines an identity under time abstraction.

Interaction refinement allows us to refine systems, given appropriate refinement pairs for their input and output channels. The idea of an interaction refinement is visualized in Fig. 10 for the so-called U^{-1} simulation. Note that here the components (boxes) A_I and A_o are no longer definitional in the sense of specifications, but rather methodological, since they relate two levels of abstraction.

Given the refinement pairs

$A_I : \vec{I}_2 \to \wp(\vec{I}_1) \quad R_I : \vec{I}_1 \to \wp(\vec{I}_2)$
$A_O : \vec{O}_2 \to \wp(\vec{O}_1) \quad R_O : \vec{O}_1 \to \wp(\vec{O}_2)$

for the input and output channels we are able to relate abstract to concrete channels for the input and for the output. We call the interface

$\hat{F} : \vec{I}_2 \to \wp(\vec{O}_2)$

an *interaction refinement* of the I/O behavior

$F : \vec{I}_1 \to \wp(\vec{O}_1)$

if the following proposition holds: $\approx>$

$A_I \circ F \circ R_o \approx> \vec{F} \quad U^{-1} -$ simulation

This formula essentially expresses that $\hat{F}$ is a property refinement of the system $A_I \circ F \circ R_O$. Thus for every concrete input history $\hat{x} \in I_2$ every concrete output $\vec{y} \in O_2$ can be also obtained by translating $\hat{x}$ onto an abstract input history $x \in A_I \cdot \hat{x}$ such that we can choose an abstract output history $y \in F(x)$ such that $\hat{y} \in R_o(y)$.

5.4.2 Granularity refinement of state machines

Also for state machines we can define a refinement both of the input and output channels and of the states. Given two state machines (Δ_k, Λ_k) for $k := 1,2$ where Δ_k is a state transition function

$$\Delta_k : (\Sigma_k \times (I_k \to M^*)) \to \wp(\Sigma_k \times (O_k \to M^*))$$

we call Δ_2 a granularity (or vertical) refinement of Δ_1 if there exists a mapping

$$\text{abs} : \Sigma_2 \to \Sigma_1$$

such that

$$\{\text{abs}(\sigma) : \sigma \in \Lambda_2\} \subseteq \Lambda_1$$

and we have for all $\Theta \subseteq \Sigma_2$

$$A_I \circ B_{\Delta_1}(\{\text{abs}(\sigma) : \sigma \in \Theta\}) \circ R_O \approx> B_{\Delta_2}(\Theta)$$

This formula expresses that for every state σ the interface abstraction is a refinement of the interface abstraction of $\text{abs}(\sigma)$. Using a state machine refinement the state space, the types of messages and the granularity of interaction can be refined.

If this holds we write:

$$(\Delta_k, \Lambda_k) \sim>_{\mathbb{F}} (\Delta_k, \Lambda_k)$$

In this way we establish the relation of vertical refinement in terms of interfaces of the state machines.

5.4.3 Granularity refinement of composed systems

Composed systems are refined in levels of abstraction by refining their components. Of course, we have to make sure, that all channels are refined consistently. Since a composed system can be seen either as one big interface or one big state machine the refinement notions introduced carry over in a straightforward manner.

Given two composed systems S_1 and S_2, we write

$$S_1 \sim>_{\text{CIS}} S_2$$

if

$$\text{abs}_{\text{CIS}/\mathbb{F}}(S_1) \sim>_{\mathbb{F}} \text{abs}_{\text{CIS}/\mathbb{F}}(S_2)$$

Again this is a straightforward concept.

5.4.4 General refinement of systems through levels of abstractions

It is the idea of granularity refinement that we do not take care of the internal structure of systems, but only compare their interface abstractions in terms of refinement pairs. Given two systems S_1 and S_2 in HS, we write

$$S_1 \sim> S_2$$

if

$$\text{abs}_{\mathbb{F}}(S_1) \sim>_{\mathbb{F}} \text{abs}_{\mathbb{F}}(S_2)$$

Again this is a straightforward concept. Basically we can always see granularity refinement in terms of interface abstraction.

The idea of granularity refinement is of particular interest from a methodological point of view. It formalizes the idea of levels of abstraction as found in many approaches to software and systems engineering. This includes the classic International Standard Organization's open systems interconnection (ISO/OSI) protocol models (see [28]) as well as the idea of layered architectures (see [16]).

5.5 Formalizing relationships between services

In this section we study refinement relations and more general relations between services. Our approach, in fact, offers the possibility to give a precise formal definition of relations between services. In the following, we deal formally with only a few most fundamental instances of such relations to demonstrate what useful relations are and how they can be formally captured.

5.5.1 Property refinement and domain extension of services

An essential notion to relate services and also systems is that of *refinement*. Note that a system is a special case of a service (which is either a total function or paradoxical) and therefore all the following concepts of property refinement work both for services and systems. Refinement was already introduced for systems in the beginning of the section on refinement.

Definition (*Property refinement*) *Given two service interfaces* $F_1, F_2 \in \mathbb{F}[I \blacktriangleright O]$ *the service* F_2 *is called a* property refinement *of service* F_1 *if the following formula holds*:

$$\forall x \in \vec{I} : F_2(x) \subseteq F_1(x)$$

Property refinement is a straightforward notion and corresponds to (is guaranteed by) logical implication

between the specifying assertions of the services F_2 and F_1.

As a relation between services to be used in requirements engineering, property refinement as defined above appears to be both too restrictive and too liberal. On the other hand, the property refinement relation allows us to decrease the service domain. For systems, this is not a problem, since systems are either total or paradoxical. For services this situation is more intricate. We introduce therefore a more suitable relation called a sub-service relation.

5.5.2 Projection of histories and services

To be prepared to define service refinement, we introduce some auxiliary notions beforehand. First recall the idea of a subtype as introduced in Sect. 2. Based on the subtype relation between sets of typed channels we define the concept of a projection.

Definition (*History projection*) *Let C and C' be sets of typed channels where C subtype C' holds. We define for history $x \in C'$ its restriction $x|C \in \vec{C}$ to the channels in the set C and to the messages of the types of the channels in C. For channels $c \in C$ with type T we specify the restriction by the equation*:

$(x|C)(c) = \mathrm{CAR}(T)©(x(c))$

The mapping $\alpha : \vec{C'} \to \vec{C}$, where

$\alpha(x) = x|C$

is called a sub-history projection.

A sub-history is the projection of a history with respect to a subset of its channel set and their types. To obtain the sub-history $x|C$, keep those channels and types of messages in the history x that belong to the typed channels in C.

5.5.3 Service refinement and domain extension of services

Based on the concept of projection, we define the concept of the restriction of interfaces and services. It allows for the concentration on a certain sub-behavior of a given more comprehensive behavior.

Definition (*Restriction of a service*) *Given syntactic interfaces $(I_1 \blacktriangleright O_1)$ and $(I_2 \blacktriangleright O_2)$ where $(I_1 \blacktriangleright O_1)$ subtype $(I_2 \blacktriangleright O_2)$ holds, we define for $F_2 \in \mathbb{F}[I_2 \blacktriangleright O_2]$ its* restriction $F_2 \dagger (I_1 \blacktriangleright O_1) \in \mathbb{F}[I_1 \blacktriangleright O_1]$ *to the syntactic interface $(I_1 \blacktriangleright O_1)$ by the following equation (for all input histories $x \in \mathbb{H}[I_1]$)*:

$$F_2 \dagger (I_1 \blacktriangleright O_1).x = \{y|O_1 : \exists\, x' \in \mathbb{H}[I_2] : x = x'|I_1 \wedge y \in F_2.x'\}$$

This notion of restriction applies both to services and to system specifications.

A restriction $F_2 \dagger (I_1 \blacktriangleright O_1)$ may introduce nondeterminism into the behavior F_2 since we may abstract away some input messages that determine the output. On the other hand, by abstracting away certain output messages, we may eliminate nondeterminism.

In the following definition we introduce service refinement. It is defined in a way that is appropriate for the stepwise refinement of services as part of the development process.

Definition (*Service refinement*) *Given two service interfaces $F_1 \in \mathbb{F}[I_1 \blacktriangleright O_1]$ and $F_2 \in \mathbb{F}[I_2 \blacktriangleright O_2]$, where I_1 subtype I_2 and O_1 subtype O_2, we call the service F_2 a* service refinement *of F_1 if, for all input histories $x \in \vec{I_1}$,*

$F_2 \dagger (I_1 \blacktriangleright O_1)(x) \subseteq F_1(x)$.

Then we write

$F_1 \approx> F_2$.

This notion of refinement applies again both to services and to systems.

Note that this refinement notion is a generalization of the notion of property refinement as introduced in [15]. In contrast to the refinement notion in [15], where the syntactic interfaces of the two systems are required to be identical, we permit the enlargement of the number of channels and their types in a service refinement step.

Service refinement introduces a partial order on the set of services: the least element in this ordering is the service with no channels, the largest element the service with all channels as input and output channels where each channel has the largest type and each input history is in the service domain and is mapped onto all output histories.

The set of services in fact forms an ordered lattice with service refinement as its ordering. Given two services $F_1 \in \mathbb{F}[I_1 \blacktriangleright O_1], F_2 \in \mathbb{F}[I_2 \blacktriangleright O_2]$ we define its least upper bound by the service $F \in \mathbb{F}[I \blacktriangleright O]$, where I is the least upper bound of I_1 and I_2, O is the least upper bound of O_1 and O_2, and $F(x)$ is the least service in the set $\mathbb{F}[I \blacktriangleright O]$ that has both F_1 and F_2 as sub-services. Greatest lower bounds are defined analogously.

Unfortunately, this notion of refinement for services is still far too liberal. The paradoxical service, where all sets of possible outputs are empty, always defines a service refinement. In a service refinement, we may reduce the service domain. Methodologically, however, we rather insist that the service domains are not decreased by refinement. A service F_1 is implemented by a service F_2, if in F_2 all the messages relevant for the

service F_1 occur in F_2, as in F_1. The service domain must not be made smaller. For input histories $x \in \vec{I}_1$ outside of the domain of F_1, we can introduce additional output histories. This idea is formalized by the concept of a sub-service.

Definition (*Sub-service relation*) *A service* $F_1 \in \mathbb{F}[I_1 \blacktriangleright O_1]$ *is a* sub-service *of a service* $F_2 \in \mathbb{F}[I_2 \blacktriangleright O_2]$, *if* F_2 *is a service refinement of* F_1 *and we have in addition*

$$\mathrm{dom}(F_1) \subseteq \mathrm{dom}(F_2 \dagger (I_1 \blacktriangleright O_1))$$

Then we say that the service F_2 *offers the service* F_1 *or that* F_1 *is a sub-service of* F_2. *We write* $F_1 \subseteq_{\mathrm{sub}} F_2$, *then.*

The sub-service relation, in fact, implies service refinement. In other words, the sub-service relation is a specialization of service refinement.

The refinement relations represent partial orders on the set of services. One service may be the refinement of several quite unrelated services.

The sub-service relation is very significant from a methodological point of view. However, it is only one example of the many relations that exist and are methodologically useful between services. To give a comprehensive set of methodologically useful relations between the services of a system and to specify the precise semantics of these relations is a major piece of work. In the following we give only formal definitions for some of these relations.

5.5.4 Vertical relationship between services

The sub-service relation defines a vertical relationship between two services. We speak of a vertical relationship between a service F_2 and F_1, if F_1 is contained in some way in the services offered by F_2.

In a vertical relationship between services, we are interested in generalizations and variations of the sub-service relation. By vertical relationships we get service hierarchies. The sub-service relation $\subseteq_{\mathrm{sub}}$ introduced so far is rather straightforward. Now we study another more sophisticated vertical relationship between services.

Given services $F_1 \in \mathbb{F}[I_1 \blacktriangleright O_1]$ and $F_2 \in \mathbb{F}[I_2 \blacktriangleright O_2]$, where I_1 **subtype** I_2 and O_1 **subtype** O_2 holds and we do not necessarily (as for the sub-service relation) assume

$$F_2 \dagger (I_1 \blacktriangleright O_1) = F_1$$

there exists a subset $R \subseteq \mathrm{dom}(F_2)$ such that

$$F_1 \subseteq_{\mathrm{sub}} F_2|R$$

In other words F_1 is a sub-service only for input from a subset of the domain of F_2. F_1 is then called a *restricted sub-service* of F.

Definition (*Restricted sub-service relation*) *Given services* $F_1 \in \mathbb{F}[I_1 \blacktriangleright O_1]$ *and* $F_2 \in \mathbb{F}[I_2 \blacktriangleright O_2]$ *where* I_1 *subtype* I_2 *and* O_1 *subtype* O_2 *hold, service* F_1 *is called a* restricted sub-service *of the service* F_2, *if there exists a subset* $R \subseteq dom(F_2)$ *such that*

$$F_1 \subseteq_{\mathrm{sub}} F_2|R$$

F_2 *is then called a super-service of* F_1.

Obviously if $F_1 \subseteq_{\mathrm{sub}} F_2$ holds, then F_1 is a restricted sub-service of F_2. The reverse does not hold, in general.

The restricted sub-service relation is a much looser relation than the sub-service relation. There are many ways in which the delivery of the service F_1 as a sub-service of F_2 may be influenced by the messages outside the domains of F_1 in F_2. The key question in the restricted sub-service relation of a service is how we can get access to the service F_1 in F_2. To get such access in F_2, we do not only have to follow the patterns in $\mathrm{dom}(F_1)$ but also observe the rules in the histories in R.

5.5.5 Horizontal relationships between services

In this section we discuss horizontal relationships between services. Informally, a horizontal relationship refers to a situation where two services are offered side by side by some super-service. In some cases we can discuss their relationship independently of the super-service. Sometimes a particular super-service has to be considered to study their horizontal relationship.

In a horizontal relationship between two services F_1 and F_2 we do not deal with sub-service relations (neither F_1 is a super-service of F_2 nor vice versa) but with services that are either independent or where there is some relationship between these services (called a *feature interaction*).

Based on our definition of sub-services we define the methodologically important notions of *combinability* and *independence* of services. The definition of combinability basically uses the following idea. Two services are called combinable if there exists a system that offers both services. This does not mean that the services are independent. If combinable services share input messages we cannot select the input triggering the output for both services independently. The same holds if the services share output messages.

Definition (*Combinability of services*) *The combinability of two services* F_1 *and* F_2 *is formalized as follows. Services* F_1 *and* F_2 *are called* combinable, *if there exists a service* F *such that*:

$$F_1 \subseteq_{\mathrm{sub}} F \wedge F_2 \subseteq_{\mathrm{sub}} F$$

Otherwise we speak of feature in-combinability *or* feature inconsistency *of the services* F_1 *and* F_2.

Combinability basically means that the services do not show contradictory requirements for their output histories. Services with disjoint sets of channels or disjoint sets of input and output messages are trivially combinable. Note that combinable services may nevertheless have joint input and output messages as long as these messages are processed in a consistent manner. Feature inconsistency can be seen as a special form of feature interaction (for details, see [22]).

Often combinability can be achieved by choosing different representations and channels for the input messages. For output messages this may be different. The two services may have contradictory effects captured by particular output messages.

Combinability does not imply or guarantee the logical independency of services. Since combinable services may share some input and output messages they might not be independent. A careful investigation shows that the formalization of the notion of independency of services is not obvious. Actually there are several ways in which we may define the independency of services in a meaningful and useful way.

A very general and strict notion of independency is formalized as follows.

Definition (*Independent combinability of services*) *Let* $F_1 \in \mathbb{F}[I_1 \blacktriangleright O_1]$ *and* $F_2 \in \mathbb{F}[I_2 \blacktriangleright O_2]$ *be combinable services. The services* F_1 *and* F_2 *are called* independently combinable*, if there exists a service* F *such that* $F_1 \subseteq_{\text{sub}} F$ *and* $F_2 \subseteq_{\text{sub}} F$ *such that for all histories* $x_1 \in dom(F_1)$ *and* $x_2 \in dom(F_2)$ *we have*

$$F(x_1 \oplus x_2) = \{y_1 \oplus y_2 : y_1 \in F_1(x_1) \wedge y_2 \in F_2(x_2)\}$$

F is called the independent combination of F_1 *and* F_2.

Independent combinability of services F_1 and F_2 means that we can design a multifunctional system incorporating both the services F_1 and F_2 where the selection of the service F_1 by choosing the particular input messages is completely independent of the selection of service F_2 and vice versa. By this definition the independent subservices are accessed and offered in a concurrent or an interleaved manner and do not share any of their input or output messages. Independent combinability of service F_1 and of service F_2 means that whatever input we choose for the services F_1 and F_2 we can get any behavior for service F_1. This behavior of F_1 does not depend on the chosen input to trigger the input of the service F_2.

Independent combinability of two services $F_1 \in \mathbb{F}[I_1 \blacktriangleright O_1]$ and $F_2 \in \mathbb{F}[I_2 \blacktriangleright O_2]$ implies that the two services are mutually *independent*. This means that we do not have to refine or to change the services F_1 or F_2 to be able to combine them. This independency does not mean that the input histories of the services are disjoint. There are situations where a service is only available or can only be accessed under certain conditions.

Service independency is a very simple and general notion. However, often services are not mutually independent. Independent combinability of services is not always what is required when combining services into systems. There are many examples for this fact. If we change the password, this affects the password service, if we introduce a new user or delete a user, this may also affect the communication service or the cash machine services but not vice versa. This leads to another relation between services, namely that they may depend on each other.

5.5.6 Equivalence of services

There are many ways to introduce different concrete services that in an abstract sense offer the same functionality. The relation

"service A is a granularity refinement of service B"

as defined above induces an equivalence relation if we say "A is equivalent to B" if both "A is a granularity refinement of B" and vice versa. This relation captures equivalence of services in terms of different service messages, different service access dialogs and different ways to represent the messages in the input and the output channels.

Equivalence of services is of major interest not only from a theoretical point of view but also from a very practical and methodological point of view. When comparing multifunctional systems it is a practically relevant question, whether two systems offer the same family of services modulo the representation of the exchanged information in terms of their messages.

6 Composition and combination

In this section we study forms of composition for systems. To cope with large systems, composition should always be hierarchical. This means that we can compose systems and the composition yields systems that are units of composition again. As a result we get trees or hierarchies of subsystems. We study to operations on systems and system components: composition and combination. In principle, we have introduced both operations already, when studying

- *Functional enhancement by service combination*: the notion of a sub-service yields a notion of a

de-combination of the functionality of a larger multifunctional system into a set of sub-services; this leads again to an operation to compose – we rather say to combine systems into larger systems,
- *System composition*: the relationship between architectures and interfaces as well as state machines; this yields a form of composition of systems.

The two operations on systems are fundamental in system development. Functional enhancement addresses requirements engineering while system composition addresses architecture design.

6.1 Functional enhancement: combining services

In principle, there are several ways to combine services out of given services. First of all we can try to *combine* more elaborate services from given ones. In the simple case we just put together services, which are more or less independent within a family of services of a multifunctional system. We have introduced the concept of independency of services. In this section we are interested in a combination of services that are not necessarily independent.

Definition (*Service combination*) *The combination of the two services* $F_1 \in [I_1 \blacktriangleright O_1]$ *and* $F_2 \in [I_2 \blacktriangleright O_2]$ *is only defined if they are combinable; then we denote them, w.r.t. the subtype relation by*

$F_1 \oplus F_2 \in [I \blacktriangleright O]$

where I *is the lub of* $\{I_1, I_2\}$ *and* O *is the lub of* $\{O_1, O_2\}$. *We define* $F_1 \oplus F_2 = F$ *with the service* F *with the property*

$F(x) = \{y : y|O_1 \in F_1(x|I_1) \wedge y|O_2 \in F_2.(x|I_2)\}$

such that

$F_1 \subseteq_{\text{sub}} F \wedge F_2 \subseteq_{\text{sub}} F$

$F_1 \oplus F_2$ *is called the* service combination *of* F_1 *and* F_2.

By service combination we can build up multifunctional systems from elementary services.

A service provides a partial view of the interface behavior of a system. The characterization of the service domain can be specified and used in service specifications by formulating assumptions characterizing the input histories in the service domain.

In our interface model a system is a set-valued function on streams. As a consequence all operations on sets are available. The interface model forms a complete lattice. This way we can form the union and the disjunction of interfaces. In other words, we may join or intersect specifications.

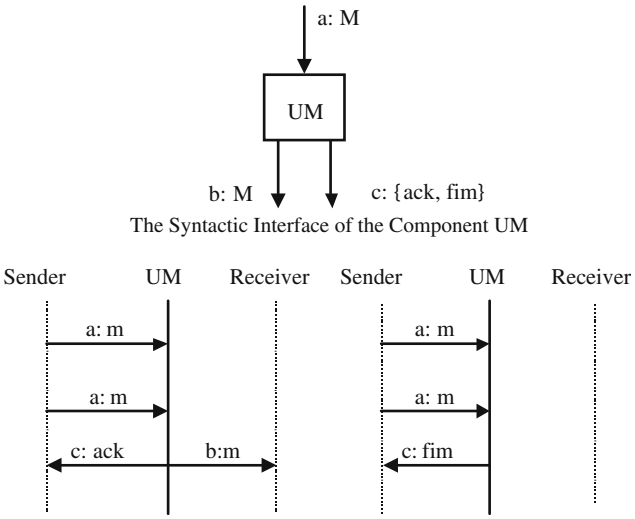

Fig. 11 MSCs for the component UM

Example (Unreliable medium) We specify an unreliable transmission component UM. It receives messages of type M on channel a and either forwards them on channel b, sending some acknowledgment to the sender via channel c, or it may forget them sending a failure indication message (fim) to the sender. The syntactic interface of the component UM is described in Fig. 11. We assume that a message has to be sent twice to be either transmitted or rejected. This is expressed by the message sequence charts (MSCs) shown in Fig. 11 (for details, including notation, see [18]).
The two MSCs translate into the specification

$$\{\langle\rangle\} = \bar{f}_{\text{UM}}(\langle a : m\rangle)$$
$$\wedge(\langle c : \text{ack}\rangle^\frown\langle b : m\rangle^\frown\bar{f}_{\text{UM}}(x) = \bar{f}_{\text{UM}}(\langle a : m\rangle^\frown\langle a : m\rangle^\frown x)$$
$$\vee\langle c : \text{fim}\rangle^\frown\bar{f}_{\text{UM}}(x) = \bar{f}_{\text{UM}}(\langle a : m\rangle^\frown\langle a : m\rangle^\frown x))$$

This is an example of the composition of two process descriptions in terms of interface abstractions.

This example shows how to translate a graphical description technique into logics in terms of our system model.

6.2 Architecture design: composition of interfaces

Architectures consist of sets of components that are connected via their channels. If these components work together in interpreted architectures according to their interface specifications architectures generate behaviors. In fact this shows that in architecture design we have two concepts of composition: the syntactic composition of syntactic interfaces into uninterpreted architectures forming data flow nets and the semantic composition of semantic interfaces in the sense of interpreted architectures into interface behaviors of the interpreted architecture.

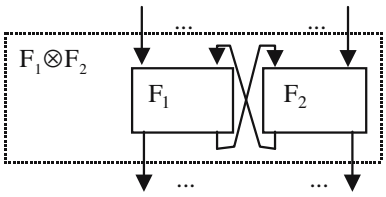

Fig. 12 Parallel composition with feedback

6.2.1 Composing interfaces

Given I/O behaviors with disjoint sets of output channels ($O_1 \cap O_2 = \emptyset$)

$$F_1 : \vec{I}_1 \to \wp(\vec{O}_1), F_2 : \vec{I}_2 \to \wp(\vec{O}_2)$$

we define the parallel composition with feedback as it is illustrated in Fig. 12 by the I/O behavior

$$F_1 \otimes F_2 : \vec{I} \to \wp(\vec{O})$$

with a syntactic interface as specified by the equations:

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

The resulting function is specified by the following equation (here we assume $y \in \vec{C}$, where the set of all channels C is given by $C = I_1 \cup I_2 \cup O_1 \cup O_2$):

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

Here for a channel set $C' \subseteq C$ we denote for $y \in \vec{C}$ by $y|C'$ the restriction of y to the channels in C'.

As long as F_1 and F_2 have disjoint sets of input and output channels the composition is simple. Given $x_1 \in \vec{I}_1$ and $x_2 \in \vec{I}_2$ we get

$$(F_1 \otimes F_2)(x_1 \oplus x_2) = \{y_1 \oplus y_2 : y_1 \in F_1(x_1) \wedge y_2 \in F_2(x_2)\}$$

Now assume

$$I_1 = O_1 \quad \text{and} \quad I_2 = O_2 = \emptyset$$

We write $\mu.F_1$ for $F_1 \otimes F_2$ since F_2 is then the system without input and output. We get $I = \emptyset$ ($\mu.F_1$ has no input channels) and

$$\mu.F_1 = \{y : y \in F_1(y)\}$$

This somewhat special construction shows that composition with feedback loops corresponds to a kind of fixpoint equation. We call $y \in F_1(y)$ a fixpoint of F_1. Note in case of a deterministic function $f_1 : \vec{O}_1 \to \vec{O}_1$ we get $y = f(y)$.

The operator is a rather general composition operator that can be easily extended from two systems to a family of systems.

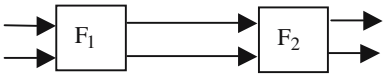

Fig. 13 Pipelining

A more specific operation is sequential composition, also called *pipelining*. It is a special case of the composition operator where $O_1 = I_2$ and the sets I_1 and O_2 are disjoint. In this case we define

$$F_1 \circ F_2 = F_1 \otimes F_2$$

where the composition is illustrated by Fig. 13.

Pipelining is the special case of composition without feedback. It can easily be generalized to the case where the channel sets I_1 and O_2 are not disjoint. The generalized definition reads as follows

$$(F_1 \circ F_2)(x) = \{z \in F_2(y) : y \in F_1(x)\}$$

This composition is also called *relational composition* if F_1 and F_2 are represented as relational or *functional composition* if F_1 and F_2 are deterministic and thus functions.

6.2.2 Composition and architecture

Each interpreted architecture (η, O) describes an interface behavior $F_{(\eta,O)} \in \mathbb{F}[I \blacktriangleright O]$. This behavior is basically the composition of all interface behaviors of the systems $F_1, F_2, F_3, \ldots$ of the architecture $F_1 \otimes F_2 \otimes F_3 \cdots$. This shows that system composition is the basis for deriving interface behaviors of architectures from the interface behaviors of the components of architectures.

Example (Giving meaning to message sequence charts (MSCs))

To show how easy it is in this model to give a precise meaning, for instance, to MSCs we look at a simple example.

Let us assume that TR is a deterministic system. Then it is modeled by a function on streams. The two MSCs in Fig. 14 are translated into the following equations for the system TR (for details, including notation, see [18]).

$$\begin{aligned} f_{\mathrm{TR}}(\langle a : m \rangle) &= \langle b : \mathrm{ready} \rangle \\ f_{\mathrm{TR}}(\langle a : m \rangle^\wedge \langle c : Y \rangle^\wedge x) &= \langle b : \mathrm{ready} \rangle^\wedge \langle d : Y \rangle^\wedge \langle b : m \rangle^\wedge f_{\mathrm{TR}}(x) \\ f_{\mathrm{TR}}(\langle a : m \rangle^\wedge \langle c : N \rangle^\wedge x) &= \langle b : \mathrm{ready} \rangle^\wedge \langle d : N \rangle^\wedge f_{\mathrm{TR}}(x) \end{aligned}$$

This translation is based on the understanding that, for every thread in an MSC, incoming arrows denote input and outgoing arrows denote output. Here we assume the system is deterministic in the sense that it reacts to every input pattern by a uniquely defined output pattern.

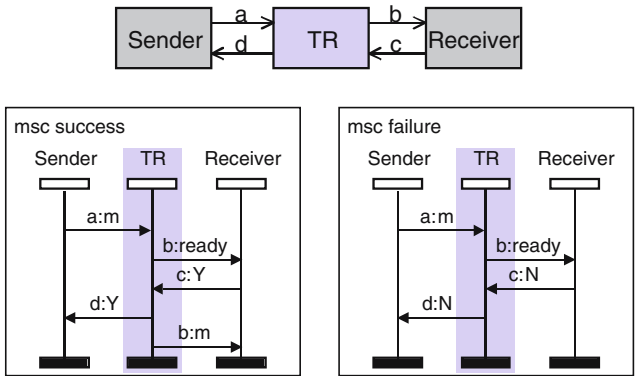

Fig. 14 System architecture and MSCs

MSCs can easily be related to system architectures as shown in Fig. 14.

This is only one example that demonstrates the flexibility of the system model to give meaning to graphical description techniques.

7 Modeling time

In this section we show how to model and work with different time granularities and time scales. We show, in particular, how we can work with different time scales for the different components within architectures. This is of interest when running several functions in multiplexing mode on one hardware node (CPU) with different sample time requirements.

7.1 Changing the time scale

We first show how to make the time scale coarser.

7.1.1 Coarsening the time scale

Let $n \in \mathbb{N}$ and C be a set of typed channels; to make the time scale coarser by the factor n for a channel history (or a stream)

$$x \in \vec{C}$$

we introduce the coarsening function

$$\mathrm{COA}(n) : \vec{C} \to \vec{C}$$

defined by (for all $t \in \mathbb{N}$):

$$\mathrm{COA}(n)(x).t + 1 = x(t * n + 1)^\wedge \ldots ^\wedge x(t * n + n)$$

$\mathrm{COA}(n)(x)$ yields a history from history x where for each stream associated with a channel n successive time intervals are concatenated (abstracted) into one. We forget about some of the time distribution. The time scale is made coarser that way.

Time coarsening obviously represents a form of abstraction. We forget some information about the timing this way. Distinct histories may be mapped onto the same history by time-scale coarsening.

It is not difficult to allow even a coarsening factor $n = \infty$ in time coarsening. Then an infinite number of time intervals is mapped into one. Timed streams are abstracted into untimed streams:

$$\mathrm{COA}(\infty)(x) = \overline{x}$$

On histories, coarsening is a function that is not injective and thus there does not exist an inverse.

We generalize the coarsening of the time scale from channel histories to behaviors. To make a behavior

$$F : \vec{I} \to \wp(\vec{O})$$

coarser by the factor n, we define the coarsening operator that maps F onto

$$\mathrm{COA}(F, n) : \vec{I} \to \wp(\vec{O})$$

which is defined as follows

$$\mathrm{COA}(F, n)(x) = \{\mathrm{COA}(n)(y) : \exists\, x' : x = \mathrm{COA}(n)(x') \wedge y \in F(x')\}$$

Coarsening maps behavior functions onto behavior functions. On one hand, coarsening may introduce some kind of further nondeterminism and underspecification into behaviors due to the coarser time scale of the input histories. Certain different input histories are mapped by the time coarsening onto the same coarser input histories. Then their sets of output histories are defined by the union of all their coarsened output histories. In this way the nondeterminism may grow.

On the other hand some nondeterminism and underspecification may be removed in behaviors by coarsening, since different output histories may be mapped by the time coarsening on the same coarsened output history.

A special case is the coarsening $\mathrm{COA}(\cdot)$, which abstracts completely away all time information. If the output of F depends on the timing of the input, then the coarsening $\mathrm{COA}(F, \cdot)$ introduces a lot of nondeterminism, in general. However, if the output produced by F does not depend on the timing of the input messages at all but only on their values and the order in which they arrive, $\mathrm{COA}(F, \cdot)$ will rather be more deterministic.

If F is weakly causal, the behavior of $\mathrm{COA}(F, n)$ is obviously weakly casual, too. However, strong causality is not maintained, in general. We will come back to more explicit rules of causality and coarsening later. Reactions

to input at later time intervals may be mapped into one time interval.

7.1.2 Making the time scale finer

We can also map a history as well as a behavior onto finer time granularities. Let $n \in \mathbb{N}$; to make the time scale finer by the factor n for a history (or a stream)

$$x \in \vec{C}$$

we use the function

$$\mathrm{FINE}(n) : \vec{C} \to \wp(\vec{C})$$

defined by the equation:

$$\mathrm{FINE}(n)(x).t + 1 = \{x' \in \vec{C} : \forall t : x.t = x'.(n^*t + 1)^\wedge \ldots ^\wedge x'.(n^*t + n)\}$$

$\mathrm{FINE}(n)(x)$ yields a set of histories where for each time interval the sequences of messages in this interval are arbitrarily subdivided into n sequences that are associated with n successive time intervals. Thus the sequence on each time interval for each channel is nondeterministically divided into n sequences. The time scale is made finer in this way.

Making the time scale finer is a form of concretization. Each history is mapped onto a number of histories by making its time scale finer. Each of these histories represents one version of the history with the finer time granularity.

Along this line of discussion another way to define the function FINE is given by the following formula

$$\mathrm{FINE}(n)(x) = \{x' : \mathrm{COA}(n)(x) = x'\}$$

This equation shows more explicitly the relationship between making the time scale coarser and making the time scale finer. They are inverse operations. Changing the time scale represents an abstraction, if we make the time scale coarser, and a concretization, if we make it finer.

The idea of making a time scale finer can also be applied to behaviors. We specify

$$\mathrm{FINE}(F, n)(x) = \{\mathrm{FINE}(n)(y) : \exists x' : x = \mathrm{FINE}(n)(x') \wedge y \in F(x')\}$$

Due to the nondeterminism in the way we make the time scale finer, there is no guarantee that we get a higher number of delays in the behaviors when moving to a finer time scale.

7.1.3 Rules for time scale refinement

Changing the time scale is an operation on histories and behaviors. In this section we study laws and rules for changing the time scale.

Our first rules of changing the time scale show that the functions $\mathrm{COA}(n)$ and $\mathrm{FINE}(n)$ form refinement pairs in the sense of granularity refinement:

$$\mathrm{COA}(n)(\mathrm{FINE}(n)(x)) = x$$
$$x \in \mathrm{FINE}(n)(\mathrm{COA}(n)(x))$$

In other words, coarsening is the inverse of making the time scale finer. We observe, in particular, the following equations

$$\mathrm{COA}(F, n) = \mathrm{FINE}(n) \circ F \circ \mathrm{COA}(n)$$
$$\mathrm{FINE}(F, n) = \mathrm{COA}(n) \circ F \circ \mathrm{FINE}(n)$$

The proof is quite straightforward. The equations show that time refinement in fact is a special case of interaction granularity refinement (see [17]).

Both abstractions and refinements by factors $n \times m$ can be seen as two consecutive refinements by the factor n followed by m or vice versa.

We get the following obvious rules:

$$\mathrm{FINE}(n * m) = \mathrm{FINE}(n) \circ \mathrm{FINE}(m)$$
$$\mathrm{COA}(n * m) = \mathrm{COA}(n) \circ \mathrm{COA}(m)$$

7.1.4 Choosing the appropriate time scale

We have seen how closely the time scale is related to the notion of causality. It depends very much on the time scale if a behavior is causal as well as on the delay properties of the systems. In a large system with many components, different time scales may be appropriate for different subsystems. In this section we therefore study the following idea of flexible timing.

A complex hierarchical system requires a flexible time model such that its time granularity can be adapted individually to the needs of its various subsystems.

This leads to an interesting idea: we establish and model different time scales for the subsystems of a composed system. Then we can choose the time scales in a flexible way, according to the following observations:

- For each system composed of strongly causal components its time delay is greater than the length of the shortest path of channels through the system of components.
- Therefore we can coarsen the interface abstraction of the system by the factor k without losing strong causality.

This leads to hierarchical system models that support local islands of finer granularity of time. A system may be composed of many subsystem with their own finer time scales. To discuss this in detail we first have to introduce a notion of composition.

7.2 Composition and the choice of the time scale

In this chapter we are interested in the question how time abstraction and composition fit together. A compositional formula should read as follows:

$$\mathrm{COA}(F_1 \otimes F_2, n) = \mathrm{COA}(F_1, n) \otimes \mathrm{COA}(F_2, n)$$

However, this formula does not hold, in general, since making a behavior coarser is an information loss that may result in the loss of strong causality and thus may introduce causal loops. This abstraction is the origin of the problems with causal loops in approaches advertised under the name "perfect synchrony" such as Esterel (see [7]). Moreover, the individual timing of the subcomponents may be highly relevant for selecting the behaviors (the output histories).

7.3 Strong causality and compositionality of coarsening

As long as $\mathrm{COA}(F_1, n)$ and $\mathrm{COA}(F_2, n)$ are still strongly causal the equation above holds.

$$\mathrm{COA}(F_1 \otimes F_2, n) = \mathrm{COA}(F_1, n) \otimes \mathrm{COA}(F_2, n)$$

We do not give the proof explicitly here. The proof uses the stepwise construction of the fixpoint and the fact that this construction is inductive as long as the behavior is causal.

However, in contrast to problems when coarsening the equation

$$\mathrm{FINE}(F_1 \otimes F_2, n) = \mathrm{FINE}(F_1, n) \otimes \mathrm{FINE}(F_2, n)$$

does always hold, if only F_1 and F_2 are time independent.

Moreover, if a system is time independent, then the following equation is valid

$$\mathrm{COA}(F, n)(\mathrm{COA}(n)(x)) = \mathrm{COA}(n)(F(x)).$$

This proposition is proved as follows: a simple computation shows that

$$\mathrm{COA}(F_1, n) \otimes \mathrm{COA}(F_2, n) \supseteq \mathrm{COA}(F_1 \otimes F_2, n).$$

The reverse holds if both $\mathrm{COA}(F_1, n)$ and $\mathrm{COA}(F_2, n)$ are strongly causal.

Let C_1 and C_2 be the sets of channels defined by

$$C_1 = O_1 \cap I_2, \quad C_2 = O_2 \cap I_1.$$

with

$$O = (O_1 \backslash C_1) \cup (O_2 \backslash C_2), \quad I = (I_1 \backslash C_2) \cup (I_2 \backslash C_1)$$

We get by definition the equation

$$\begin{aligned}&[\mathrm{COA}(F_1, n) \otimes \mathrm{COA}(F_2, n)](x)\\ &\quad = \{y : y|I = x \wedge y|O_1 \in \mathrm{COA}(F_1, n)(x|I_1) \wedge y|O_2\\ &\qquad\qquad \in \mathrm{COA}(F_2, n)(x|I_2)\}\end{aligned}$$

This proves the validity of the formula above.

A well-known effect of composition of systems is the accumulation of their delays. This can be explained nicely for pipelining.

We write $\mathrm{delay}(F, n)$ if F is a behavior with a delay by (at least) n time units. More precisely we define:

$$\begin{aligned}\mathrm{delay}(F, n) \equiv [&\forall x, z, t : x \downarrow t = z \downarrow t\\ &\Rightarrow (F(x)) \downarrow t + n = (F(z)) \downarrow t + n]\end{aligned}$$

In other words, F is (weakly) causal if $\mathrm{delay}(F, 0)$ holds and strongly causal if $\mathrm{delay}(F, 1)$ holds.

Obviously we have for all $n, m \in \mathbb{N}$:

$$n \leq m \wedge \mathrm{delay}(F, m) \Rightarrow \mathrm{delay}(F, n)$$

If $\mathrm{delay}(F, \infty)$ holds then the output does not depend on the input at all. For a weakly causal system there is always a maximal number $n \in \mathbb{N} \cup \{\infty\}$ such that $\mathrm{delay}(F, n)$ holds. This number is called the *guaranteed delay*.

Let the behaviors

$$F_1 : I_1 \to \wp(O_1), \quad F_2 : O_1 \to \wp(O_2)$$

be given. We obtain

$$\mathrm{delay}(F_1, m) \wedge \mathrm{delay}(F_2, n) \Rightarrow \mathrm{delay}(F_1 \circ F_2, m + n)$$

In the case of two strongly causal functions F_1 and F_2 we get (at least)

$$\mathrm{delay}(F_1 \circ F_2, 2)$$

On one hand this fact is very satisfactory since it leads to a useful delay calculus (see above).

On the other hand it shows an unfortunate inflexibility of the design calculus for timed systems. If we want to represent a function by two functions with pipelining we always have to accept delay by 2 if the functions are strongly guarded. In fact, if we insist on a delay less than 2, a system cannot be implemented by a system consisting of two components composed sequentially. This sounds weird and seems a reason to reject our approach. However, the operators that change the time granularity allow us to deal with this issue. If $\mathrm{delay}(F_1 \circ F_2, 2)$ holds we may replace $F_1 \circ F_2$ by $\mathrm{COA}(F_1 \circ F_2, 2)$.

8 Perspective, related work, summary and outlook

In this final section we put our modeling theory into perspective with related issues of software and systems engineering, refer to related work and give a summary and an outlook.

8.1 Perspective

In this paper we concentrate on the modeling of software intensive systems and a theory of modeling. Of course, when engineering software intensive systems we cannot work directly with the theory. We need well-chosen notations, using textual or graphical syntax or tables. In practical development it helps including notational conventions to make the presentations of the models better tractable. Furthermore, we need logical calculi to prove properties about relationships between model descriptions. And finally we need tools that support our modeling theory in terms of documentation, analysis, verification, and transformation.

Our theory is carefully designed to make such a support possible and some of what is needed is already available. There exists a well worked out proof theory (see [15]) as well as tool support (see [2,3,46]), that includes a graphical notation for the model views. There is plenty of research to apply and detail the presented theory (see [13,14,16,18,28,41]).

8.2 Related work

Modeling has been, is, and will be a key activity in software and systems engineering. Research in this area therefore has always been of major interest for informatics.

Early pioneering work in system and software modeling led to Petri nets (see [39,40]), aiming at highly abstract models for distributed concurrent systems. Early work based on Petri nets led to data flow models (see [34]). Another early line of research is denotational semantics (see [52]) aiming at mathematical models of programs. Also the work on programming logics (see, for instance, [27]) always had a flavor of modeling issues, albeit sometimes rather implicit. Also much of the early work on data structures was a search for the appropriate models (see [12]).

Very much influenced by work on programming language semantics – and also by denotational semantics – is the work on VDM (see [32]). Other work on formal specification such as Z (see [51]) and B (see [1]) has also a modeling flavor, although in this work often mathematical concepts are more explicitly emphasized than system modeling issues.

Very often in the history of software and systems engineering, notations were suggested first – of course with some ideas as to what they expressed – and only later was it found that giving a consistent meaning to these notations was much less obvious and much harder than originally thought. Examples are CCS (see [38]), CSP (see [30]), where it took several years of research to come up with a reasonable semantical model, and state charts. Recent examples along these lines are approaches such as UML (see [19]). This way plenty of interesting work on modeling was and still is triggered.

Much work in modeling and modeling theory was carried out by research on formal techniques for distributed systems. Typical examples are CSP (see [30]), CCS (see [38]), or more general process algebras (see [4]), and later Unity (see [20]) as well as TLA (see [35]). In Unity the underlying model is kept very implicit and everything is explicitly done by some abstract programming notation and a logical calculus. Nevertheless there is a state machine model behind Unity. In TLA the underlying model is explained more explicitly. All these approaches do not define system views but rather one system model. Sometimes the system model is enhanced by a specification framework, which can be understood as a complementary view. Also general work on temporal logic (see [37]) always included some modeling research. More recent developments are evolving algebras (see [25]).

Other examples are the so-called synchronous languages such as Lustre and Signal (see [5]) or Esterel (see [6,7]). They all aim at programming notations for real-time computations rather than making their underlying models explicit.

A pioneer in modeling with a more practical flavor is Jackson (see [33]). He related modeling issues with engineering methods (see also [53]).

Another line of research related to modeling is the field of architecture description languages (see [24,36]). Also there we find the discrepancy between a suggestive notation, an extensive terminology talking about components and connectors, for instance, and a poor, often rather implicit, modeling theory.

A completely different line of research aims at visual modeling notations. State charts (see [26]) were suggested as a graphical modeling language. It took a while before the difficulties of giving meaning to these state charts were recognized.

A quite different approach arose from more-pragmatic work on system modeling. Graphical notations were suggested first control flow graphs, later in SA (see [21]) and SADT (see [42,43]). Early mod-

eling languages were SDL (see [11,47–49]). Then the object-oriented approaches (see [31,45]) came, including OOD (see [8]), OADT (see [44]), ROOM (see [50]) and many others. Today UML (see [9]) is much advocated, although in many aspects it is just a graphical notation with many open problems concerning its modeling theory.

None of the approaches explicitly tried to bring together the different views and to formally relate them and to introduce, in addition, refinement as an additional relation between the views.

8.3 Summary and outlook

Why did we present this setting of mathematical models and relations between them? First of all, we wanted to show how rich and flexible the tool kit of mathematical modelling is and has to be, and how far we are in integrating and relating them. Perhaps it should be emphasized that we first presented an integrated system model that was very close to practical approaches such as SDL or UML where a system is a tree or hierarchy of components. In this tree of components the leaves are state machines. In our case the usage of streams and stream processing functions is the reason for the remarkable flexibility of our model toolkit and the simplicity of the integration.

There are many interesting directions for further research on the basis of the presented theory of modeling. One direction is to apply it to specific domains; this may lead to domain-specific modeling languages.

Another interesting issue is the relationship to programming languages and standard software infrastructure such as operating systems. In such an approach it remains to be worked out how the introduced models are mapped onto programs that are executed in an infrastructure of the operating systems and middleware. For an example in this direction, see [10].

Acknowledgments It is a pleasure to thank Markus Pizka, Leonid Kof, Ingolf Krüger, and Bernhard Schätz for stimulating discussions and helpful remarks on draft versions of the manuscript. I thank Judith Hartmann for carefully proofreading the manuscript.

References

1. Abrial JR (1996) The B-book. Cambridge University Press, Cambridge
2. Website of AutoFocus with documentation, screenshots, tutorials and download. http://autofocus.in.tum.de
3. Website AutoRAID, with documentation, screenshots and downloads http://wwwbroy.in.tum.de/~autoraid/
4. Baeten JCM, Bergstra J (1992) Process algebras with signals and conditions. In: Broy M (ed.) Programming and mathematical method. NATO ASI Series, Series F: Computer and system sciences, vol. 88. Springer, Berlin Heidelberg New York, pp 273–324
5. Benveniste A, Caspi P, Edwards S, Halbwachs N, LeGuernic P, De Simone R (2003) The synchronous languages twelve years later. Proc IEEE 91(1):64–83
6. Berry G, Gonthier G (1988) The ESTEREL synchronous programming language: design, semantics, implementation. INRIA, Research Report 842
7. Berry G (2000) The foundations of esterel. MIT Press, Cambridge
8. Booch G (1991) Object oriented design with applications. Benjamin Cummings, Redwood City
9. Booch G, Rumbaugh J, Jacobson I (1998) The unified modeling language for object-oriented development, version 1.0. RATIONAL Software Cooperation
10. Botaschanjan J, Broy M, Gruler A, Harhurin A, Knapp S, Kof L, Paul W, Spichkova M (2006) On the correctness of upper layers of automotive systems. (in press)
11. Broy M (1991) Towards a formal foundation of the specification and description language SDL. Formal Aspects Comput 3:21–57
12. Broy M, Facchi C, Hettler R, Hußmann H, Nazareth D, Regensburger F, Slotosch O, Stølen K (1993) The requirement and design specification language SPECTRUM. An informal introduction. version 1.0. Part I/II Technische Universität München, Institut für Informatik, TUM-I9311 / TUM-I9312
13. Broy M (1997) Refinement of time. In: Bertran M, Rus Th (eds) Transformation-based reactive system development. ARTS'97, Mallorca 1997. Lecture notes in computer science vol 1231, pp 44–63 (To appear in TCS)
14. Broy M, Hofmann C, Krüger I, Schmidt M (1997) A graphical description technique for communication in software architectures. Technische Universität München, Institut für Informatik, TUM-I9705, February 1997. http://www4.informatik.tu-muenchen.de/reports/TUM-I9705. Also in: Joint 1997 Asia Pacific Software Engineering Conference and International Computer Science Conference (APSEC'97/ICSC'97)
15. Broy M, Stølen K (2001) Specification and development of interactive systems: Focus on streams, interfaces, and refinement. Springer, Berlin Heidelberg New York
16. Broy M (2003) Modeling services and layered architectures. In: König H, Heiner M, Wolisz A (eds) Formal techniques for networked and distributed systems. Lecture notes in computer science, vol 2767. Springer, Berlin Heidelberg New York, pp 48–61
17. Broy M (2004) Time, abstraction, causality, and modularity in interactive systems. FESCA 2004. Workshop at ETAPS 2004, pp. 1–8
18. Broy M (2004) The semantic and methodological essence of message sequence charts. Sci Comput Program SCP 54:2–3, 213–256
19. Broy M, Cengarle MV, Rumpe B (2006) Semantics of UML. Towards a system model for UML. The structural data model. Technische Universität München, Institut für Informatik, Report TUM-IO612
20. Chandy KM, Misra J (1988) Program design: a foundation. Addison–Wesley, Reading
21. DeMarco T (1979) Structured analysis and system specification. Prentice Hall, Englewood Cliffs
22. Deubler M (2006) Dienst-orientierte Softwaresysteme: Anforderungen und Entwurf. Dissertation (To appear)
23. Filman R, Elrad T, Clarke S, Aksit M (2004) Aspect-oriented software development. Addison–Wesley, Reading

24. Garlan D, Allen R, Ockerbloom J (1995) Architectural mismatch: why reuse is so hard. IEEE Soft 12(6):17–26
25. Gurevich Y (1994) Evolving algebra. In: Pehrson B, Simson I (eds) IFIP 1994 World Computer Congress, vol. I: Technology and Foundations, Elsevier, Amsterdam, pp. 423–427
26. Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Program 8:231–274
27. Hehne ECR (1993) A practical theory of programming. Springer, Berlin Heidelberg New York
28. Herzberg D, Broy M (2005) Modeling layered distributed communication systems. Applicable formal methods, vol. 17, no. 1. Springer, Berlin Heidelberg New York
29. Hettler R (1994) Zur Übersetzung von E/R-Schemata nach Spectrum. Technischer Bericht TUM-I9409, TU München
30. Hoare CAR (1985) Communicating sequential processes. Prentice Hall, Englewood Cliffs
31. Jacobsen I (1992) Object-oriented software engineering. Addison–Wesley, ACM, Reading
32. Jones C (1986) Systematic program development using VDM. Prentice Hall, Englewood Cliffs
33. Jackson MA (1983) System development. Prentice Hall, Englewood Cliffs
34. Kahn G (1974) The semantics of a simple language for parallel processing. In: Rosenfeld JL (ed.) Information processing 74. Proceedings of the IFIP Congress 74. North Holland, Amsterdam, pp. 471–475
35. Lamport L (1994) The temporal logic of actions. ACM Trans Program Languages Syst 16(3):872–923
36. Luckham DC, Kenney JL, Augustin LM, Vera J, Bryan D, Mann W (1955) Specification and analysis of system architecture using rapide. IEEE Trans Softw Eng 21(4):336–355
37. Manna Z, Pnueli A (1992) A temporal logic of reactive systems and concurrent systems. Springer, Berlin Heidelberg New York
38. Milner R (1980) A calculus of communicating systems. Lecture notes in computer science, vol 92. Springer, Berlin Heidelberg New York
39. Petri CA (1962) Kommunikation mit Automaten.Technical Report RADCTR-65-377, Bonn, institut für Instrumentelle Mathematik
40. Petri CA (1963) Fundamentals of a theory of asynchronous information flow. In: Proceedings of IFIP Congress 62. North Holland Publishing Company, Amsterdam, pp. 386–390
41. Romberg J (2006) Synthesis of distributed systems from synchronous dataflow programs. PhD Thesis, Technische Universität München, Fakultät für Informatik
42. Ross DT (1977) Structured analysis (sa): a language for communicating ideas. IEEE Trans Softw Eng 3(1):16–34
43. Ross DT (1990) Applications and extensions of sadt. In: Glinert EP (ed) Visual programming environments: paradigms and systems. IEEE Computer Society Press, Los Alamitos, pp 147–156
44. Rumbaugh J (1991) Object-oriented modelling and design. Prentice Hall, Englewood Cliffs
45. Rumpe B (1996) Formale Methodik des Entwurfs verteilter objektorientierter Systeme. PhD Thesis, Technische Universität München, Fakultät für Informatik 1996. Published by Herbert Utz Verlag
46. Schätz B (2004) Mastering the complexity of embedded systems – the Autofocus approach. In: Fabrice Kordon F, Lemoine M (eds) Formal techniques for embedded distributed systems: from requirements to detailed design. Kluwer, Dordrecht
47. Specification and Description Language (SDL), Recommendation Z.100. Technical Report, CCITT, 1988
48. ITU-T (previously CCITT) (1993) Criteria for the use and applicability of formal description techniques. Recommendation Z. 120, Message Sequence Chart (MSC), 35p
49. ITU-T. Recommendation Z.120, Annex B: Algebraic semantics of message sequence charts. ITU-Telecommunication Standardization Sector, Geneva, Switzerland, 1995
50. Selic B, Gullekson G, Ward PT (1994) Real-time objectoriented modeling. Wiley, New York
51. Spivey M (1988) Understanding Z – a specification language and its formal semantics. Cambridge tracts in theoretical computer science 3. Cambridge University Press, Cambridge
52. Stoy JE (1997) Denotational semantics: the scott strachey approach to programming languages. MIT Press, Cambridge
53. Zave P, Jackson M (1997) Four dark corners of requirements engineering. ACM Trans Softw Eng and Methodol 6(1):1–30