Advance Access publication on February 12, 2010
doi:10.1093/comjnl/bxq005

A Logical Basis for Component-Oriented Software and Systems Engineering

MANFRED BROY*

Institut für Informatik, Technische Universität München, D-80290 München, Germany
**Corresponding author: broy@in.tum.de*

A theory for the systematic development of distributed interactive software systems constructed in terms of components requires a basic system model and description techniques supporting specific views and abstractions of systems. Typical system views are the interface, the distribution, or the state transition view. We show how to represent these views by mathematics and logics. The development of systems consists in working out these views leading step by step to implementations in terms of sets of distributed, concurrent, interacting state machines. For large systems, the development is carried out by refinement through several levels of abstraction. We formalize the typical steps of the development process and express and justify them directly in logic. In particular, we treat three types of refinement steps: horizontal refinement which stays within one level of abstraction, vertical refinement addressing the transition from one level of abstraction to another, and implementation by glass box refinement. We introduce refinement relations to capture these three dimensions of the development space. We derive verification rules for the refinement steps and show the modularity of the approach.

Keywords: systems engineering, interactive software systems, systems development

Received 6 January 2010; revised 6 January 2010
Handling editor: Erol Gelenbe

1. INTRODUCTION: BASICS OF A LOGIC OF SYSTEMS

For the development of large software systems a large variety of skills is essential including decisions about technical platforms and issues of management. In the following we concentrate exclusively on formal modeling.

Most of the work on the formalization of program and system development and its logical foundations is directed towards what is called 'programming in the small'. Typical examples are assertion techniques where the effect of statements is captured by logic. Less was achieved and published, however, on a formalization of 'development in the large'. In this paper we outline a fundamental approach to 'development in the large'; we rather speak of *system development*. Although we only describe a purist's system modeling technique here, the concept called FOCUS has been extended towards typical system development approaches used in practice (see [1]).

For the development of large systems, it is essential that parts of a system can be developed independently. We speak of *modular system development.* The parts a composed system is built of are called its *components*. Each component is a system by itself. A composed system is therefore a system of systems.

A discipline of component-based modular system development requires a tractable mathematical theory. Such a theory has to provide a clear notion of a component and ways to specify, to manipulate and to compose components. In this paper, we introduce such a theory and a mathematical model of a system with the following characteristics:

(i) A system is *interactive*.
(ii) A system *encapsulates* a state that cannot be accessed from the outside directly. It interacts with its environment exclusively by its *interface* formed by named and typed *channels*. Channels are communication links for asynchronous, buffered message exchange.
(iii) It receives *input messages* from its environment on its *input* channels and generates *output messages* to its environment on its *output* channels.

(iv) A system can be *underspecified* and thus *nondeterministic*. This means that for a given input history there may exist several output histories representing possible reactions of the system.
(v) The interaction between the system and its environment takes place *concurrently* in a *global time* frame. In the model, there is a global notion of discrete time that applies both to the system and its environment.

Throughout this paper we work exclusively with discrete time. Discrete time is a satisfactory model for most of the typical applications for information processing systems. For an extension of the model to continuous time see [2].

Systems are specified by logical expressions relating the input and output communication histories on their channels. State machines (see [3]) describe the input and output histories by operational means in terms of state transitions with input and output.

Based on the ideas of an interactive system, we define forms of composition. We basically introduce only one powerful composition operator, namely *parallel composition with feedback*. This composition operator allows us to model *concurrent execution* and *interaction* of systems within a network. We briefly show that other forms of composition can be introduced as special cases of parallel composition with feedback.

For the systematic stepwise development of systems we introduce the concept of *refinement*. We study three refinement relations namely *property refinement*, *glass box refinement* and *interaction refinement*. These notions of refinement typically occur in a systematic top-down system development.

Finally, we prove the *compositionality* of FOCUS. Compositionality guarantees that refinement steps for the systems of a composed system realize a refinement step for the composed system. As a consequence, global reasoning about the system can be structured into local reasoning about the systems. Compositionality is the basis for the notion of *modularity* in systems engineering.

The main contribution of this paper is a presentation of the relational version of the stream processing approach as developed at the Technische Universität München under the keyword FOCUS (see [4–6]). The paper aims at a self-contained presentation of recent results and extensions of this approach.

We begin with the informal introduction of the concept of an interactive system. This concept is based on communication histories called streams that are introduced in Section 3. Then a mathematical notion of a system is introduced in Section 4 and illustrated by a number of simple examples. We show how to specify systems by state machines. Section 5 treats operators for composing systems into distributed systems. In Section 6 we introduce three patterns of refinement to develop systems and show the compositionality of these patterns. Again all described concepts are illustrated by simple examples. Section 7 discusses the results and compares the proposed approach with approaches in the literature.

2. CENTRAL NOTION: SYSTEM

The notion of system is essential both in systems engineering and software engineering. Especially in software engineering a lot of work is devoted to the concepts of *software architecture* (see [7–9]) as a principle for structuring a system into a family of sub-systems called components (see [10, 11]) for a development method where software systems are composed from prefabricated components. The idea is that main parts of systems can be obtained from appropriate new configurations of reusable software solutions. Thus they do not have to be re-implemented over and over again. Key issues for such an approach are precisely specified and well-designed *interfaces* and *software architectures*. Software architectures mainly can be described by the structure of distributed systems, composed of components. For handling them, a clean and clear concept using a mathematical model of a system is indispensable.

In software engineering literature the following informal definition of a component is found:

> A system is a physical encapsulation of related services according to a published specification.

According to this definition we work with the idea of a system that encapsulates either a local state or a set of sub-systems forming a distributed architecture and providing certain services via its *interface*. We suggest a logical way to write a specification of system services. We relate these notions to glass box views, where systems are described by state machines, to the derived interface abstractions and to system specifications.

We introduce the mathematical notion of a system and on this basis a concept of system specification. A system specification is a description of its syntactic interface and a logical formula that relates input to output histories.

A powerful semantic concept of a system interface is an essential foundation for the key issues in system development such as well-structured system (software and hardware) construction, clear interfaces of systems, proper system architecture and systematic systems engineering.

In the following we introduce a minimalist's mathematical concept of a system and a syntactic form to describe it by using logic. We show how basic notions of software development such as specification and refinement of systems can be based on this concept.

3. STREAMS

A *stream* is a finite or infinite sequence of elements of a given set. In interactive systems streams are built over sets of messages or actions. Streams are used that way to represent communication histories for channels or histories of activities.

Let M be a given set of messages.

> A stream over the set M is a finite or an infinite sequence of elements from M.

We use the following notation:

- M^* denotes the set of finite sequences over M including the *empty* sequence $\langle\,\rangle$,
- M^∞ denotes the set of infinite sequences over M (that are represented by the total mappings $\mathbb{N}\backslash\{0\} \to M$).

A stream is an element of the set M^ω that is defined by the equation

$$M^\omega = M^* \cup M^\infty.$$

We introduce the *prefix ordering* $\sqsubseteq$, which is a partial order on streams specified for streams $x, y \in M^\omega$ by the formula

$$x \sqsubseteq y \equiv \exists z \in M^\omega : x\hat{}z = y.$$

Here $x\hat{}z$ denotes the well-known concatenation of sequences; the stream z is appended to the stream x. To cover the concatenation of infinite streams we define as follows: if x is infinite then $x\hat{}z = x$.

Throughout this paper we do not work with the simple concept of a stream as introduced but find it more appropriate to use so-called *timed streams*. An infinite timed stream represents an infinite history of communications over a channel or an infinite history of activities that are carried out sequentially in a discrete time scale. The discrete time frame represents time as an infinite chain of time intervals of finite equal duration. In each time interval a finite number of messages is communicated or a finite number of actions is executed. Hence we represent a communication history of a system model with such a discrete time frame by an infinite sequence of finite sequences of messages or actions. By

$$(M^*)^\infty,$$

we denote the set of timed streams. Note that the elements of $(M^*)^\infty$ are infinite sequences of finite sequences.

A timed stream over a set M is an infinite sequence of finite sequences of elements from M.

For a function f we often write $f.z$ for function applications instead of $f(z)$ to avoid unnecessary brackets.

The tth sequence $s.t$ in a timed stream $s \in (M^*)^\infty$ represents the sequence of messages appearing on a channel in the tth time interval or, if the stream represents a sequence of actions, the sequence of actions executed in the tth time interval.

Throughout this paper we work with a couple of simple basic operators and notations for streams that are summarized below:

- $\langle\,\rangle$ empty sequence or empty stream,
- $\langle m\rangle$ one-element sequence containing m as its only element,
- $x.t$ tth element of the stream x,
- $\#x$ length of the stream x,
- $x\hat{}z$ concatenation of the sequence x to the sequence or stream z,
- $x \downarrow t$ prefix of length t of the stream x,
- $S©x$ stream obtained from x by deleting all its messages that are not elements of the set S,
- $\bar{x}$ finite or infinite stream that is the result of concatenating all sequences in the timed stream x. Note that $\bar{x}$ is finite if x carries only a finite number of nonempty sequences.

In a timed stream $x \in (M^*)^\infty$ we express at which times which messages are transmitted. As long as the timing is not relevant for a system it does not matter if a message is transmitted a bit late (scheduling messages earlier may make a difference with respect to causality – see the following). To take care of this we introduce a *delay closure*. For a timed stream $s \in (M^*)^\infty$ we define the set $x\uparrow$ of timed streams that carry the same stream of messages but perhaps with some additional delay as follows:

$$x\uparrow = \{x' \in (M^*)^\infty : \forall t \in \mathbb{N} : \overline{x'\downarrow t} \sqsubseteq \overline{x\downarrow t} \wedge \bar{x} = \bar{x'}\}.$$

Obviously $x \in x\uparrow$ and for every $x' \in x\uparrow$ we have $x'\uparrow \subseteq x\uparrow$ and $\bar{x} = \bar{x'}$. The set $x\uparrow$ is called the *delay closure* for the stream s. The delay closure operator is easily extended to sets of streams by (let $S \subseteq (M^*)^\infty$)

$$S\uparrow = \bigcup_{s\in S} s\uparrow.$$

We may also consider timed streams of states to model the traces of state-based system models (see [12]). In the following, we restrict ourselves to message-passing systems and therefore to streams of messages.

Throughout this paper, we use streams exclusively to model the communication histories of sequential communication media called *channels*. In general, in a system several communication streams occur. Therefore we work with channels to refer to individual communication streams. Accordingly, in FOCUS, a channel is simply an identifier in a system that evaluates to a stream in every execution of the system.

4. SYNTACTIC AND SEMANTIC SYSTEM INTERFACES

In this section we introduce a mathematical notion of systems in terms of their interface behavior. Systems interact with their environment via channels. A channel is a unidirectional communication link and uniquely identified by a channel identifier.

4.1. Interface behaviors

Types or *sorts* are useful concepts to describe interfaces. We work with a simple notion of types where each type is a set of data elements. These data elements are used as messages or as

values of state attributes. Let a set S of types be given. Let

$$M = \bigcup_{s \in S} s$$

be the set of all data messages.

In FOCUS a typed channel is an identifier for a sequential directed communication link for messages of that type. By C we denote a set of typed channel names. We assume that a type assignment for the channels in the set C is given as follows:

$$\text{type} : C \to S$$

Given a set C of typed channels, a *channel valuation* is an element of the set $\vec{C}$ defined as follows (let all types be included in the 'universe' M):

$$\vec{C} = \{x : C \to (M^*)^\infty : \forall c \in C : x.c \in (\text{type}(c)^*)^\infty\}.$$

A channel valuation $x \in \vec{C}$ associates a stream of elements of type type(c) with each channel $c \in C$. This way the channel valuation x defines a communication history for the channels in the set C.

The operators on streams induce operators on channel valuations and, furthermore, on sets of streams as well as sets of channel valuations by pointwise application. This way all introduced operators on streams generalize to histories.

Given a set of typed input channels I and a set of typed output channels O, we introduce the notion of a *syntactic interface* of a system:

(I, O) syntactic interface,
I set of typed input channels and,
O set of typed output channels.

A graphical representation of a system and its interface as a data flow node is shown in Fig. 1. We do not require that the sets I and O be disjoint.

For a component-oriented method to system development in addition to the syntactic interface a concept for describing and specifying the *interface behavior* of a system is mandatory. We work with a simple and straightforward notion of interface behavior.

Relations between input histories and output histories represent interface behaviors of systems.

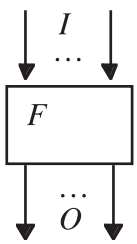

FIGURE 1. Graphical representation of a system F with input channels I and output channels O.

Input histories are represented by valuations of the input channels and output histories are represented by the valuations of the output channels.

In FOCUS we represent the *black box* or *interface behavior* of a system by a set-valued function (we also speak of the *semantic interface* of the system):

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

As is well known a set-valued function is isomorphic to a relation. We prefer set-valued functions in the case of system behaviors to emphasize the different roles of input and output. We call the function F an *interface behavior*. Given the input history $x \in \vec{I}$, $F.x$ denotes the set of all output histories that a system with behavior F may exhibit on the input x.

An interface behavior F is called *deterministic*, if $F.x$ is a one-element set for every history $x \in \vec{I}$. A deterministic interface behavior represents simply a function $\vec{I} \to \vec{O}$.

It is well known that modeling the behavior of systems by relations leads to the so-called merge anomaly (also called Brock–Ackermann anomaly, see [13]). In FOCUS this anomaly is overcome by the notion of causality (see later).

4.2. Specification of interface behaviors

An interface behavior represents the black box behavior of a system. Using logical means, an interface behavior F can be described by a logical formula Φ, called *interface assertion*, relating the streams on the input channels to the streams on the output channels. In interface assertions channel identifiers occur syntactically as identifiers (variables) for streams of the respective type. The interface assertions are interpreted in the standard way of typed higher order logic (see [14]).

An abstract specification of a system provides the following information:

(i) its syntactic interface, describing how the system interacts with its environment via its input and output channels,
(ii) its behavior by an interface assertion Φ relating input and output channel valuations.

This leads to a specification technique for systems (see [15] for lots of examples). In FOCUS we specify systems by a scheme of the following form:

‹name›

in ‹input channels›
out ‹output channels›
‹interface assertion›

The shape of the scheme is inspired by well-known specification approaches like Z (see [16]).

EXAMPLE (TRANSMISSION, MERGE AND FORK). As simple but quite fundamental examples of systems we specify a merge

component MRG, a transmission component TMC and a fork component FRK. In the examples let $T1$, $T2$ and $T3$ be types (recall that in our case types are simply sets) where $T1$ and $T2$ are assumed to be disjoint and $T3$ is the union of the sets of elements of type $T1$ and $T2$. The specification of the merge component MRG (actually the specification relies on the assumption that $T1$ and $T2$ are disjoint, which should be made explicit in the specification in a more sophisticated specification approach) reads as follows:

MRG

in x: $T1$, y: $T2$
out z: $T3$
$\overline{x} = T1 \copyright \overline{z} \wedge \overline{y} = T2 \copyright \overline{z}$

In this specification there is nothing specified about the time flow and therefore the specification refers only to the time abstractions of the involved streams. The causality of the time flow is considered in detail in the following subsection.

We specify interface assertion $x \sim y$ for timed streams x and y of arbitrary type T; $x \sim y$ is true if the messages in x are a permutation of the messages in y. Formally we define it by the following logical equivalence:

$$x \sim y \equiv (\forall m \in T : \{m\}\copyright\bar{x} = \{m\}\copyright\bar{y}).$$

Based on this definition we specify the component TMC.

Often it is helpful to use certain channel identifiers both for input channels and for output channels. These are then two different channels, which might have different types. To distinguish these channels in interface assertions, we use a well-known notational trick. In an interface specification, we write for a channel c that occurs both as input and as output channel simply c to denote the stream on the input channel c and c' to denote the stream on the output channel c. Accordingly in the following specification z is the external name of the output channel z and z' is its internal name.

TMC

in z: $T3$
out z: $T3$
$z \sim z'$

This simple specification says that every input message occurs eventually also as output message, and vice versa. Nothing is specified about the timing of the messages. In particular, messages may be arbitrarily delayed and overtake each other. The specification does not exclude the output messages might even be produced earlier than they are received. This paradox is excluded by causality as introduced in the following section.

The following component FRK is just the 'inversion' of the component MRG.

FRK

in z: $T3$
out x: $T1$, y: $T2$
$\overline{x} = T1 \copyright \overline{z}$ $\overline{y} = T2 \copyright \overline{z}$

Note that the merge component MRG as well as the TMC component and the fork component FRK as they are specified here are 'fair'. Every input is eventually processed and reproduced as output. □

Based on the interface assertion given in a specification of an interface behavior F, we may prove properties about the function F.

4.3. Causality in Interface behaviors

For input/output information processing devices there is a crucial notion of causality. Certain output depends on certain input. Causality indicates dependencies between the actions of a system. Thus far interface behaviors are nothing but relations represented by set-valued functions. In the following we introduce and discuss the notion of causality for interface behaviors.

Interface behaviors generate their output and consume their input in a time frame. This time frame is useful to characterize causality between input and output. Output that depends causally on certain input cannot be generated before this input has been received.

Let an interface behavior

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

be given. In the following we define a couple of fundamental notions to characterize specific properties of interface behavior F that relate to causality and the timing of the input and output messages.

DEFINITION (PROPER TIME FLOW). *An interface behavior F is called* (weakly) causal *if, for all times $t \in \mathbb{N}$, we have*

$$x \downarrow t = z \downarrow t \Longrightarrow (F.x) \downarrow t = (F.z) \downarrow t.$$

Here F is causal if the output in the tth time interval does not depend on input that is received after time t. This ensures that there is a proper time flow for the system modeled by F.

If F is not causal there exists a time t and input histories x and x' such that $x \downarrow t = x' \downarrow t$ but $(F.x) \downarrow t \neq (F.x') \downarrow t$. A difference between x and x' occurs only after time t but at time t the reactions of F in terms of output messages are already different.

Nevertheless, proper timing permits an instantaneous reaction [17]: the output at time t may depend on the input at time t. This may lead to problems with causality if we consider in addition delay-free feedback loops. To avoid such problems we strengthen the concept of proper time flow to the notion of causality.

DEFINITION (STRONG CAUSALITY/CAUSALITY). *An interface behavior F is called* ***strongly causal*** *(or* time guarded*) if, for all times $t \in \mathbb{N}$, we have*

$$x \downarrow t = z \downarrow t \Longrightarrow (F.x) \downarrow t+1 = (F.z) \downarrow t+1.$$

If F is strongly causal, then the output in the tth time interval does not depend on input that is received after the $(t-1)$th time interval. Then F is causal and in addition reacts to input received in the $(t-1)$th time interval not before the tth time interval. This way causality between input and output is guaranteed.

A function $f : \vec{I} \to \vec{O}$ is called causal and strongly causal, respectively, if the deterministic interface behavior $F : \vec{I} \to \wp(\vec{O})$ with $F.x = \{f.x\}$ for all $x \in \vec{I}$ has the specified properties.

By $[F]$ we denote the set of strongly causal total functions $f : \vec{I} \to \vec{O}$ with $f.x \in F.x$ for all input histories $x \in \vec{I}$.

A nondeterministic specification F defines the set $[F]$ of total deterministic behaviors. A specification is only meaningful if the set $[F]$ is not empty. This idea leads to the following definition.

DEFINITION (REALIZABILITY). *An interface behavior F is called* realizable*, if there exists a strongly causal total function $f : \vec{I} \to \vec{O}$ such that*

$$\forall x \in \vec{I} : f.x \in F.x.$$

A strongly causal function $f : \vec{I} \to \vec{O}$ provides a deterministic strategy to calculate for every input history a particular output history that is correct with respect to F. Every input $x \downarrow t$ till time point t fixes the output till time point $t+1$ and, in particular, the output at time $t+1$. Actually f essentially defines a deterministic automaton, called Mealy machine, with input and output. There are sophisticated examples of behaviors that are not realizable. For instance consider the following example of a behavior $F : \vec{I} \to \wp(\vec{I})$ that is not realizable (the proof of this fact is left to the reader, a proof is given in [15]):

$$F.x = \{x' : x \neq x'\}.$$

Note that $F.x$ is strongly causal.

DEFINITION (FULL REALIZABILITY). *An interface behavior F is called fully realizable if it is realizable and if, for all input histories $x \in \vec{I}$*

$$F.x = \{f.x : f \in [F]\}$$

holds.

Full realizability guarantees that, for all output histories, there is a strategy (a deterministic implementation) that computes this output history. In fact, nondeterministic state machines are not more powerful than sets of deterministic state machines (see [18]*).*

All the properties of interface behavior defined thus far are closely related to time. To characterize whether the timing of the messages is essential for a system, we introduce notions of time dependencies of systems. Time independence expresses that the timing of the input histories does not restrict the choice of the messages but at most their timing in the output histories. We give a precise definition of this notion as follows.

DEFINITION (TIME INDEPENDENCE). *An I/O-function F is called time independent if for all its input histories $x, x' \in \vec{I}$,*

$$\bar{x} = \bar{x'} \Longrightarrow \overline{F.x} = \overline{F.x'}$$

holds.

Time independence means that the timing of the input histories does not influence the streams of messages produced as output but only their timing. We use this notion also for the functions

$$f : \vec{I} \to \vec{O}.$$

By analogy, f is time independent if, for all its input histories $x, x' \in \vec{I}$,

$$\bar{x} = \bar{x'} \Longrightarrow \overline{f.x} = \overline{f.x'}.$$

holds.

DEFINITION (TIME INDEPENDENT REALIZABILITY). *An interface behavior F is called time independently realizable, if there exists a time independent, strongly causal total function $f : \vec{I} \to \vec{O}$ such that*

$$\forall x \in \vec{I} : f.x \in F.x.$$

By $[F]_{ti}$ we denote the set of strongly causal, time independent total functions $f : \vec{I} \to \vec{O}$ where $f.x \in F.x$ for all input histories $x \in \vec{I}$.

DEFINITION (FULL TIME INDEPENDENT REALIZABILITY). *An interface behavior F is called* fully time independently realizable*, if it is time independent and time independently realizable and if, for all input histories $x \in \vec{I}$, we have*

$$F.x = \{f.x : f \in [F]_{ti}\}.$$

Full time independent realizability guarantees that for all output histories there is a strategy that computes this output history and that does not use the timing of the input.

The FOCUS system model has a built-in notion of time. This has the advantage that we can explicitly specify timing properties. However, what if we want to deal with systems where the timing is not relevant? In that case we use a special subclass of specifications and systems called time permissive.

DEFINITION (TIME PERMISSIVITY). *An interface behavior F is called* time permissive *if, for all input histories $x \in \vec{I}$, we have*

$$F.x = (F.x) \uparrow .$$

This means that, for every output history $y \in F.x$, any delay is tolerated but not acceleration since this may lead to conflicts with causality. Note that time independency and time permissiveness are independent notions. There are interface behaviors that are time independent but not time permissive and vice versa. For an extensive treatment of changing the time granularity in interface behaviors, see [19].

If we want to specify a system for an application that is not time critical, the interface behavior should be fully time independently realizable and time permissive. This means that

(i) the timing of the input does not influence the timing of the output,
(ii) the timing of the output is only restricted by causality, but apart from that any timing is feasible.

This way we specify systems for which time is only relevant with respect to causality.

4.4. Inducing properties on specifications

An interface assertion for systems with the set of input channels I and the set of output channels O defines a predicate

$$p : \vec{I} \times \vec{O} \longrightarrow \mathbb{B}.$$

This predicate defines an I/O-behavior

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

by the equation (for $x \in \vec{I}$)

$$F.x = \{y \in \vec{O} : p(x, y)\}.$$

For a system specification, we also may carefully formulate the interface assertion such that the specified interface behavior fulfills certain of the properties introduced above. Another option is to add these properties, if wanted, as schematic requirements to interface assertions. This is done with the help of closures for specified interface behavior F. By closures with a given interface behavior either the inclusion greatest or the inclusion least interface behavior is associated that has the required property and is included in the interface behavior F or includes the interface behavior F, respectively. We demonstrate this idea for causality.

4.4.1. *Imposing causality*

Adding strong causality as a requirement on top of the given interface assertion p specifying the interface behavior F should lead to a function F' that is strongly causal. Here F' should guarantee all the restrictions expressed by the specifying predicate p and by strong causality but not more. Following this idea F' is defined as the inclusion greatest function F', where $F'.x \subseteq F.x$ for all input histories x such that F' is strongly causal and $y \in F'.x$ implies $p(x, y)$. This characterization leads to the following recursive definition for the function F' written in the classical way that is commonly used to define a closure.

DEFINITION (CAUSALITY RESTRICTION). *Given an interface behavior F the* causality restriction *F' is the inclusion greatest function such that the following equation holds:*

$$F'.x = \{y \in \vec{O} : p(x, y) \wedge \forall x' \in \vec{I}, t \in \mathbb{N} : x \downarrow t = x' \downarrow t \Longrightarrow \exists y' \in \vec{O} : y \downarrow t + 1 = y' \downarrow t + 1 \wedge y' \in F'.x'\}$$

Since the right-hand side of this equation is inclusion monotonic in F' this definition is proper.

Obviously, the behavior F' is included in F, since $y \in F'.x$ implies $p(x, y)$ and thus $y \in F.x$. Since the formula to the right of this equation is inclusion monotonic in F' such a function exists and is uniquely determined.

THEOREM (CAUSALITY RESTRICTION IS STRONGLY CAUSAL). *For every interface behavior F its causality restriction F' is strongly causal.*

Proof. Given

$$y \in F'.x \wedge x \downarrow t = x' \downarrow t$$

we conclude by the definition of F' that

$$\exists y' \in \vec{O} : y' \downarrow t + 1 = y \downarrow t + 1 \wedge y' \in F'.x'.$$

Thus we obtain

$$(F'.x) \downarrow t + 1 \subseteq (F'.x') \downarrow t + 1.$$

Conversely if

$$y'' \in F'.x'$$

then by $x \downarrow t = x' \downarrow t$ we get

$$\exists y' \in \vec{O} : y'' \downarrow t + 1 = y' \downarrow t + 1 \wedge y' \in F'.x.$$

Thus we obtain

$$(F'.x') \downarrow t + 1 \subseteq (F'.x) \downarrow t + 1.$$

Hence

$$(F'.x) \downarrow t + 1 = (F'.x') \downarrow t + 1$$

which shows that F' is strongly causal. □

Note that the causality restriction F' may be the trivial function $F'.x = \emptyset$ for all $x \in \vec{I}$, if there is a contradiction between strong causality and the specifying predicate p. We abbreviate for a given function F its causality restriction by TG[F].

EXAMPLE (CONFLICT WITH STRONG CAUSALITY). Consider the specification

CTG

in x: $T1$ **out** y: $T1$
$\forall t \in$ IN: $x.t+1 = y.t$

The system CTG is required to show at time t always as output what it receives as input at time $t + 1$. This specification is obviously in conflict with strong causality.

Adding strong causality as a requirement to CTG, for every input history x and every output history y, we derive

$$[\forall t \in \mathbb{N} : x.t + 1 = y.t] \wedge \forall t \in \mathbb{N}, x' \in (T1^*)^\infty : x \downarrow t = x' \downarrow t \\ \Longrightarrow \exists y' : y \downarrow t + 1 = y' \downarrow t + 1 \wedge \forall t \in \mathbb{N} : x'.t + 1 = y'.t.$$

If we choose $x.t + 1 \neq x'.t + 1$ (assuming $T1 \neq \emptyset$), by the formula we get

$$x.t + 1 = y.t = y'.t = x'.t + 1,$$

which is a contradiction to the assumption $x.t + 1 \neq x'.t + 1$. Thus there does not exist any output history for TG[CTG]. Assuming causality CTG gets inconsistent.

If an interface behavior F is strongly causal, then obviously $F = \mathrm{TG}[F]$. Nevertheless, also in some other cases TG[F] can be easily identified. If a function F'' defined as:

$$F".x = \{y : p(x, y) \wedge \forall x' \in \vec{I}, t \in \mathbb{N} : x \downarrow t = x' \downarrow t \\ \Longrightarrow \exists y' \in \vec{O} : y \downarrow t + 1 = y' \downarrow t + 1 \wedge p(x', y')\}$$

fulfills the defining equation for TG[F], then $F"$ is the required function, that is $F" = TG$[F]; otherwise, TG[F].$x \subseteq F".x$ for all x.

EXAMPLE (TRANSMISSION SYSTEM). Consider the transmission component TMC given in the example above. In this case we have $p(x, y) = (x.z \sim y.z)$, where z is the only channel for the histories x and y and $x.z$ and $y.z$ are the streams for channel z. Adding strong causality to TMC, we get the function (with $I = \{z\}$)

$$\mathrm{TG[TMC]}.x = \{y : p(x, y) \wedge \forall t \in \mathbb{N}, x' \in \vec{I} : x \downarrow t = x' \downarrow t \\ \Longrightarrow \exists y' : y \downarrow t + 1 = y' \downarrow t + 1 \wedge p(x', y')\}.$$

From this we easily prove the formula

$$y \in \mathrm{TG[TMC]}.x \\ \Longrightarrow \forall m \in T3, t \in \mathbb{N} : \#\{m\}©\overline{x.z \downarrow t} \geq \#\{m\}©\overline{y.z \downarrow t + 1},$$

which expresses that, at every point in time t, the number of messages in y at time $t + 1$ is less than or equal to the number of messages m in x at time t. This formula is a simple consequence of the fact that, for each input history x and each time t, we can find an input history x' such that

$$\overline{x.z \downarrow t} = \overline{x'.z \downarrow t}$$

and

$$\overline{x'.z \downarrow t} = \overline{x'.z},$$

where $\overline{x'.z}$ is the finite sequence of messages in $\overline{x'.z \downarrow t}$. For all $y' \in \mathrm{TG[TMC]}.x'$ we have $y' \sim x'$. Moreover, for $y \in \mathrm{TG[TMC]}.x$ there exists $y' \in \mathrm{TG[TMC]}.x'$ with $y \downarrow t + 1 = y' \downarrow t + 1$. For all messages $m \in T3$, we get

$$\#\{m\}©\overline{y.z \downarrow t + 1} = \#\{m\}©\overline{y'.z \downarrow t + 1} \leq \#\{m\}©\overline{y'.z} \\ = \#\{m\}©\overline{x'.z} = \#\{m\}©\overline{x.z \downarrow t}.$$

Strong causality is an essential property both for conceptual modeling aspects and for the verification of properties of specifications.

Strong causality models the time dependencies and in this way the asymmetry between input and output.

For time permissive, strongly causal systems there is a direct relationship to *prefix monotonicity* of functions over nontimed streams (see [4] for a functional approach to system modeling without notions of time where prefix ordering is the central notion for defining least fixpoints). This is shown in detail in Appendix 2. By causality we also rule out the merge anomaly (see [13]).

4.4.2. *A short discussion of time and causality*

As pointed out above, notions like time independence and time permissiveness, and strong causality are logical properties that can either be added as properties to specifications explicitly or proved for certain specifications. It is easy to show, for instance, that MRG, TMC and FRK are time permissive. If we add strong causality as a requirement then all three specified interface behaviors are fully realizable.

We may add also other properties of interface behaviors in a schematic way to specifications. For instance, adding time permissiveness can be interpreted as a weakening of the specification by ignoring any restrictions with respect to timing. We define for an interface behavior F a time permissive function F' by the equation

$$F'.x = (F.x) \uparrow .$$

As pointed out, we do not require that interface behaviors described by specifications always posses all the properties introduced above. We are more liberal and allow for more flexibility. We may add specific properties to specifications freely (using key words, see [6]) whenever appropriate and therefore deal in a schematic way with all kinds of specifications of interface behaviors and timing properties.

4.5. State transition specifications

Often a system can be described in a well-understandable way by a state transition machine with input and output. In FOCUS we describe the data state of a transition machine by a set of typed attributes V that can be seen as *program variables*. A data state is given by the mapping

$$\eta : V \longrightarrow \bigcup_{v \in V} \mathrm{type}(v)$$

It is a valuation of the attributes in the set V by values of the corresponding type. By $\hat{V}$ is denoted the set of valuations of the

attributes in V. In addition, we may use a finite set K of control states. Then each state of the system is a pair (k, η) consisting of a *control state* k and a *data state* η. By Σ is denoted the set of all states.

A state machine with input and output (see [3]) is given by a set $\Lambda \subseteq \Sigma$ of initial states $\sigma_0 \in \Sigma$ as well as a state transition function

$$\Delta{:}(\Sigma \times (I \longrightarrow M^*)) \longrightarrow \wp(\Sigma \times (O \longrightarrow M^*)).$$

Given a state $\sigma \in \Sigma$ and a valuation $u : I \to M^*$ of the input channels by sequences every pair $(\sigma', r) \in \Delta(\sigma, u)$ represents a successor state σ' and a valuation $r : O \to M^*$ of the output channels representing the sequences produced by the state transition.

The state transition function Δ induces a function

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

Here B_Δ provides the interface abstraction onto the state transition function Δ. For each state $\sigma' \in \Sigma$, and each input channel valuation $x \in \vec{I}$, the interface abstraction function B_Δ is specified by

$$B_\Delta(\sigma').x = \{y \in \vec{O} : \exists \sigma \in \mathbb{N} \longrightarrow \Sigma : \sigma(0) = \sigma' \\ \wedge\, t \in \mathbb{N} : (\sigma(t+1), y.t+1) \in \Delta(\sigma(t), x.t+1)\}.$$

Here $B_\Delta(\sigma)$ defines an interface behavior for the state σ, which represents the behavior of the system described by the state machine Δ if initialized by the state σ. Note that $B_\Delta(\sigma)$ is always fully realizable. We can prove this by introducing oracles into the states leading to a deterministic behavior for each state.

The guardedness of the recursion guarantees strong causality of the interface behavior $B_\Delta(\sigma)$. Here B_Δ generalizes to sets Λ of pairs of states and initial output sequences as follows:

$$B_\Delta(\Lambda).x = \{y \in B_\Delta(\sigma) : \sigma \in \Lambda\}.$$

Based on these definitions we relate state machines and interface behavior (see also [3]).

> Given a state transition function Δ and a set Λ of initial states, $B_\Delta(\Lambda)$ provides the interface abstraction on the behavior of the state transition machine Δ for the set Λ of pairs of states and initial output sequences.

The concept of state machines with input and output as introduced is a generalization of Mealy machines to infinite state spaces and infinite input and output sets. The interface abstraction $B_\Delta(\Lambda)$ is always causal (as a simple proof shows). For Moore machines, where in state transitions the output does not depend on the input but only on the states the interface abstraction is strongly causal. More precisely, $B_\Delta(\Lambda)$ is strongly causal if (Δ, Λ) is a Moore machine.

We describe state machines often by state transition diagrams. A state transition diagram consists of a number of nodes representing control states and a number of transition rules represented by labeled arcs between the control states.

EXAMPLE (STATE TRANSITION SPECIFICATION). The simple component SWT (switching transmission) receives two input streams one of which has priority until in one time interval its input is empty, and then the priority changes to the other channel. It has only one attribute val of sort $T3^*$. The specification of SWT is given in a graphical style by Fig. 2. A short explanation of the notation is found below and in [20].

Here the arrow starting from the dot indicates the initial state of the machine. The component SWT always forwards the input of one of its input channels until it gets empty. Then it switches to the transmission of the input on the other channel.

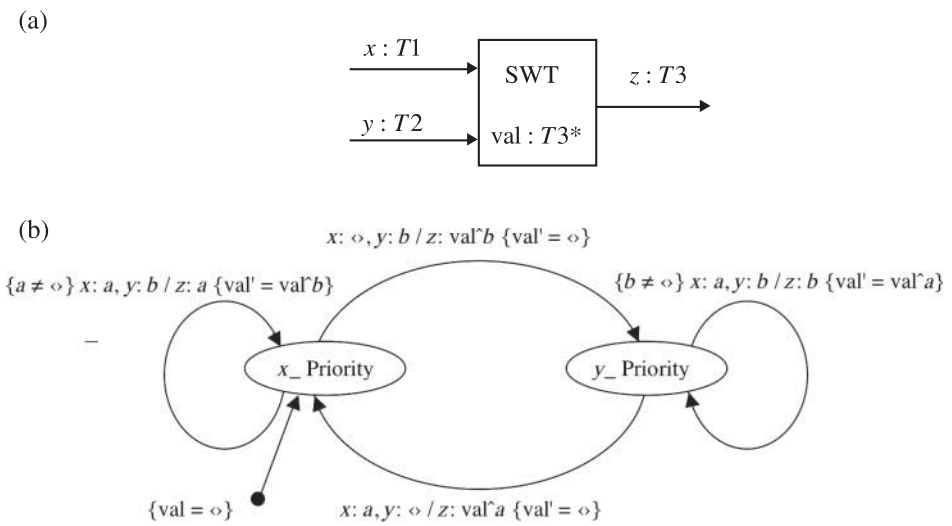

FIGURE 2. (a) SWT as data flow node. (b) State transition diagram for SWT.

The meaning of a state transition diagram is defined as follows. Given a finite set K of control states (which correspond to the nodes in the state transition diagram) and a set V of typed attributes our state space Σ is defined by

$$\Sigma = K \times \hat{V}.$$

For each control state $k \in K$ we define a set of transition rules. Each transition rule leads from k to a control state $k' \in K$ and is labeled by a transition expression

$$\{P\}x_1 : a_1, \ldots, x_n : a_n / y_1 : b_1, \ldots, y_m : b_m\{Q\},$$

where P is a logical expression called the *guard* that contains only the attributes from V as logical variables and some auxiliary variables that are bound variables to the transition rule. The $x_1, \ldots, x_n$ are the input channels (pairwise distinct) and the $y_1, \ldots, y_m$ are the output channels (pairwise distinct). The $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$ are terms denoting sequences of messages of the types of the respective channels. By Q is denoted a logical expression called the post condition that contains besides the local variables of the transition rule the attributes in V as logical variables, also in a primed form v'. The primed variables refer to the values of the attributes in the state of the transition.

The transition rule can only fire, if the machine is in control state k, if the guard evaluates to true and if all the input sequences $a_1, \ldots, a_n$ are available on the channels $x_1, \ldots, x_n$ and if there exist sequences $b_1, \ldots, b_m$ such that Q holds. For more details see [20].

4.6. Properties of specifications

The logical formulas used for specifications are a powerful and flexible mechanism for describing system behaviors. Every formula uniquely characterizes an interface behavior. Like interface behaviors we also classify specifications, or more precisely, their interface assertions. All notions introduced for interface behaviors carry over to their specifications. Hence we speak for instance of a strongly causal or of a time independent specification.

4.6.1. Safety and liveness

In this section we introduce another helpful classification for properties of interface behaviors, namely *safety* and *liveness*.

> A property is called a *safety property* if its violation can always be observed within finite observation intervals; a property is called a *liveness property* if its violation can only be observed by an infinite observation.

Let I be a set of typed input channels, O a set of typed output channels and $p(x, y)$ an interface assertion. The proposition $p(x, y)$ is called a *safety property* if (for all $x \in \vec{I}$, $y \in \vec{O}$) the following formula holds:

$$p(x, y) \equiv (\forall t \in \mathbb{N} : \exists y' \in \vec{O} : y' \downarrow t = y \downarrow t \wedge p(x, y')).$$

The proposition $p(x, y)$ is called a *liveness property* if the following formula holds:

$$\forall x \in \vec{I}, y \in \vec{O}, t \in \mathbb{N} : \exists y' \in \vec{O} : y' \downarrow t = y \downarrow t \wedge p(x, y').$$

In general, an interface assertion p(x, y) includes safety as well as liveness properties in a straightforward way since it talks about infinite streams. In our running example of the component TMC, the formula p(x, y)

$$x \sim y \wedge \forall x', t \in \mathbb{N} : x \downarrow t = x' \downarrow t \\ \Longrightarrow \exists y' : y \downarrow t + 1 = y' \downarrow t + 1 \wedge x' \sim y'$$

implies the safety property

$$p_S(x, y) = \forall m \in T3, t \in \mathbb{N} : \#\{m\}\copyright x \downarrow t \geq \#\{m\}\copyright y \downarrow t + 1.$$

Moreover, it implies the liveness property

$$p_L(x, y) \equiv (p_S(x, y) \Longrightarrow p(x, y)).$$

As is well known, each interface assertion p over the input channels I and the output channels O can be decomposed canonically into a pure safety and a pure liveness property.

We define the canonical safety property p_S for the interface assertion p by

$$p_S(x, y) \equiv \forall t \in \mathbb{N} : \exists y' \in \vec{O} : y' \downarrow t = y \downarrow t \wedge p(x, y').$$

We define the canonical liveness property p_L by

$$p_L(x, y) \equiv (p_S(x, y) \Longrightarrow p(x, y)).$$

Clearly we have

$$p \equiv p_S \wedge p_L.$$

The construction of p_S essentially corresponds to a prefix closure of p. The following example shows a system with a weakened liveness property.

EXAMPLE (UNFAIR MERGE). An unfair merge component UFM is similar to the merge component, but it guarantees the throughput of all messages only for one of its input channels. We use the following specification:

UFM

in x: $T1$, y: T2
out z: $T3$
$(\bar{x} = T1 \copyright \bar{z} \wedge \bar{y} \sqsubseteq T2 \copyright \bar{z})$
$\vee\ (\bar{x} \sqsubseteq T1 \copyright \bar{z} \wedge \bar{y} = T2 \copyright \bar{z})$

We find that UFM and MRG only differ with respect to liveness properties.

4.6.2. Proofs of interface assertions about state machines

A system with a set of input channels I and a set of output channels O is described by a logical formula $p(x, y)$ that contains identifiers of I and O as free variables for streams of the respective type. The formula defines a relation between valuations for the input and output channels. Thus far we have worked with infinite streams. We may also consider formulas $q(x, y)$ with valuations x and y associating finite streams with channels. Since a valuation with finite streams can be understood as a state, we can interpret $q(x, y)$ as a formula characterizing states.

A state transition system as introduced in the previous section owns certain invariants. An invariant is a logical formula that refers to the state of a system. These states are composed of the control state, the state attributes and the streams associated with the input and output channels. Invariants provide an effective method for proving safety properties for systems described by state machines. Proofs of certain liveness properties require assumptions about the fairness of state transitions and something like a leads_to logic as in Unity (see [21]); note the close relationship between interface assertions and invariants of state machines.

In fact, invariants characterize safety properties. By q is denoted an invariant for a system specified by the predicate p if for all input histories $x \in \vec{I}$ and all output histories $y \in \vec{O}$,

$$p(x, y) \Longrightarrow \forall t, j \in \mathbb{N} : t \geq j \Longrightarrow q(x \downarrow t, y \downarrow j).$$

From a given predicate p we deduce both safety and liveness properties for the specified system as demonstrated above.

For systems specified by state machines we work with invariants that refer to the data states and to the control states. Control states are referred to by the special variable c of type K. In invariants channels stand for finite sequences of finite sequences of messages. Note that thus far finite as well as infinite streams can interpret the channels in formulas.

Now we briefly illustrate how to prove invariants for state machines. Given a system with one attribute v, one input channel x and one output channel y we have transition rules,

$$\{P\}x : a/y : b\{Q\}.$$

Let R be a logical formula that contains the input channels in x, the output channels in y, the attribute v and the control state identifier as free variables. We find that R is an invariant if R holds for the initial states and initial output histories (where the input channels are empty) and for each transition rule leading from control state k to control state k', we prove that

$$c = k \wedge P \wedge R \wedge Q \Longrightarrow R[x\hat{}\langle a\rangle/x, y\hat{}\langle b\rangle/y, v'/v, k'/c].$$

Here $R[\ldots E/v \ldots]$ denotes the substitution of the identifier v by the expression E in the formula R. The verification rule easily generalizes to systems and rules with multiple input and output channels.

EXAMPLE (PROVING INVARIANTS FOR STATE MACHINES). Consider the example of the system SWT specified by the state machine above. We consider the following predicate as an invariant:

$$\begin{aligned} & T1©(\bar{z}\hat{}\mathrm{val}) = \bar{x} \\ \wedge\ & T2©(\bar{z}\hat{}\mathrm{val}) = \bar{y} \\ \wedge\ & (c = x\_\mathrm{Priority} \Longrightarrow T1©\mathrm{val} = \langle\,\rangle) \\ \wedge\ & (c = y\_\mathrm{Priority} \Longrightarrow T2©\mathrm{val} = \langle\,\rangle). \end{aligned}$$

The proof is quite straightforward. Recall that $T1 \cup T2 = T3$, $T1 \cap T2 = \emptyset$. For instance, since the transition rule on the left of Fig. 2b leads to the verification condition

$$\begin{aligned} & c = x\_\mathrm{Priority} \\ \wedge\ & a \neq \langle\,\rangle \\ \wedge\ & T1©(\bar{z}\hat{}\mathrm{val}) = \bar{x} \\ \wedge\ & T2©(\bar{z}\hat{}\mathrm{val}) = \bar{y} \\ \wedge\ & \mathrm{val}' = \mathrm{val}\hat{}b \\ \wedge\ & (c = x\_\mathrm{Priority} \Longrightarrow T1©\mathrm{val} = \langle\,\rangle) \\ \wedge\ & (c = y\_\mathrm{Priority} \Longrightarrow T2©\mathrm{val} = \langle\,\rangle) \\ \Longrightarrow\ & \\ & T1©(\overline{z\hat{}\langle a\rangle}\hat{}\mathrm{val}') = \overline{x\hat{}\langle a\rangle} \\ \wedge\ & T2©(\overline{z\hat{}\langle a\rangle}\hat{}\mathrm{val}') = \overline{y\hat{}\langle b\rangle} \\ \wedge\ & (x\_\mathrm{Priority} = x\_\mathrm{Priority} \Longrightarrow T1©\mathrm{val} = \langle\,\rangle) \\ \wedge\ & (x\_\mathrm{Priority} = y\_\mathrm{Priority} \Longrightarrow T2©\mathrm{val} = \langle\,\rangle). \end{aligned}$$

The proof is straightforward.

From invariants of state machines we prove interface assertions. For a more comprehensive treatment of this issue, see [12].

5. COMPOSITION

In this section we introduce an operator for the *composition* of systems. We prefer to introduce only one general composition operator and later define a number of other operators as special cases.

Given interface behaviors with disjoint sets of output

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

channels where the sets of output channels are disjoint $O_1 \cap O_2 = \emptyset$ we define the parallel composition with feedback, as illustrated by Fig. 3, by the interface behavior

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

where the syntactic interface is specified by the equations

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

The resulting function is specified by the following equation (here $y \in \vec{C}$ where the set of channels C is given by

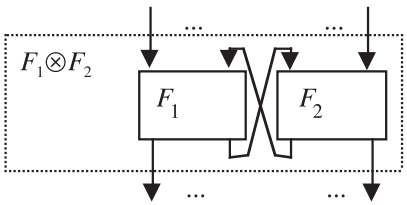

FIGURE 3. Parallel composition with feedback.

$C = I_1 \cup I_2 \cup O_1 \cup O_2$):

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

Here y denotes a valuation of all the channels in C of F_1 and F_2. By $y|C'$ is denoted the restriction of the valuation y to the channels in $C' \subseteq C$. The formula essentially says that all the streams on output channels of the systems F_1 and F_2 are feasible output streams of these systems.

Let Φ_1 and Φ_2 be the interface assertions for the functions F_1 and F_2, respectively; the interface assertion of $F_1 \otimes F_2$ reads as follows:

$$\exists z_1, \ldots, z_k : \Phi_1 \wedge \Phi_2,$$

where $\{z_1, \ldots, z_k\} = (I_1 \cup I_2) \cap (O_1 \cup O_2)$ are the internal channels of the system. We underline the following principle:

> Composition corresponds to logical conjunction, channel hiding to existential quantification.

For the composition operator $\otimes$ we show the following facts by rather straightforward proofs:

(i) if the F_i are *realizable* for $i = 1, 2$, then so is $F_1 \otimes F_2$,
(ii) if the F_i are *fully realizable* for $i = 1, 2$, then so is $F_1 \otimes F_2$,
(iii) if the F_i are *fully time independently realizable* for $i = 1, 2$, then so is $F_1 \otimes F_2$.

For proofs see Appendix 1. If all interface behaviors F_i are total and properly timed for $i = 1, 2$, we cannot conclude, however, that the function $F_1 \otimes F_2$ is total. This shows that the composition operator works in a modular way only for well-chosen subclasses of specifications.

Some further forms of composition that can be defined as special cases of the operator $\otimes$ are listed in the following (we do not give formal definitions for most of them, since these are quite straightforward):

(i) feedback without hiding: μ F

let $F : \vec{I} \to \wp(\vec{O})$; then we define: $\mu F : \vec{J} \to \wp(\vec{O})$, where $J = I \backslash O$ by the equation (here we assume that $z \in \vec{C}$, where $C = I \cup O$):

$$(\mu F).x = \{z|O : z|J = x \wedge z|O \in F(z|I)\}$$

(ii) parallel composition: $F_1||F_2$ (without feedback)

It is only defined if $O_1 \cap O_2 = \emptyset (F_1||F_2).x = \{y : y|O_1 \in F_1(x|I_1) \wedge y|O_2 \in F_2(x|I_2)\}$
if $(I_1 \cup I_2) \cap (O_1 \cup O_2) = \emptyset : F_1||F_2 = F_1 \otimes F_2$

(iii) hiding of a channel $c : F \backslash \{c\}$

Hiding an input channel means to assign the empty stream of messages to it. Hiding an output channel simply means that this output is hidden from the environment.

(iv) renaming of channels: $F[c/c']$
(v) sequential composition: $F_1 \circ F_2$

Sequential composition is essentially functional composition (or more precisely relational product). Sequential composition (also called pipelining) of the systems F_1 and F_2 requires that $O_1 = I_2$ and we have

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

Note the order of the operands. In the special case where $O_1 = I_2$ and $I_1 \cap O_1 = \emptyset$ and $I_2 \cap O_2 = \emptyset$ and $I_1 \cap O_2 = \emptyset$ we reduce sequential composition to parallel composition with feedback along the lines illustrated in Fig. 4 as follows:

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

A simple example of sequential composition (where $O_1 = I_2$) is the composed system MRG $\circ$ FRK as well as FRK $\circ$ MRG.

In addition to the composition operators introduced above, there exist the classical logical connectors for interface behaviors with the same syntactic interface as follows:

(vi) logical connectors: $F_1 \cup F_2$, $F_1 \cap F_2$, $\neg F$

Here $\neg F$ denotes the set complement. Given a system specification S, we define by $S[c/c']$ the renaming of the channel c in the system S to c'.

All the mentioned forms of composition can be defined formally for the FOCUS concept of systems and in principle reduced to parallel composition with feedback. Conversely, parallel composition with feedback can be defined in terms of

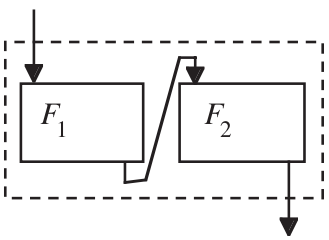

FIGURE 4. Sequential composition as a special case of the composition operator $\otimes$.

parallel and sequential composition as well as feedback with hiding.

Note the significance of uniform properties such as strong causality for a proper concept of composition. This is demonstrated by the following example.

EXAMPLE (FEEDBACK FOR THE SYSTEM TMC). In this example we study feedback and its dependency on the additional requirement of strong causality. Without strong causality the specification μ TMC boils down to the interface assertion

$$z \sim z,$$

which is obviously equivalent to true. Assuming strong causality the property

$$\forall t \in \mathbb{N} : \overline{z \downarrow t} \sim \overline{z \downarrow t+1}$$

holds, from which by the fact

$$z \downarrow 0 = \langle\,\rangle$$

and by straightforward induction, we can conclude that $\bar{z} = \langle\,\rangle$.

This example demonstrates that strong causality is an essential property when dealing with feedback loops to guarantee the causality between input and output according to the operational data flow principle. This way all fixpoints correspond to what are called least fixpoints in the partial order approach.

6. SYSTEM DEVELOPMENT BY REFINEMENT

In requirements engineering and in the design 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 (see [8, 9]). These activities are connected with development steps. Refinement relations (see [1]) are the medium to formalize development steps (see [22]) and in this way the development process.

In FOCUS we formalize the following basic ideas of refinement:

(i) *property refinement* — enhancing requirements — allows us to add properties to a specification,
(ii) *glass box refinement* — designing implementations — allows us to decompose a system into a distributed system or to give a state transition description for a system specification,
(iii) *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 (see [1]).

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

6.1. Property refinement

Property refinement is a well-known concept in structured programming. It allows us to replace an interface behavior with one having additional properties. This way a behavior is replaced by a more restricted one. In FOCUS an interface behavior

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

is refined by a behavior

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

if

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

where 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 therefore reflexive, asymmetric, and transitive. Moreover, the inconsistent specification logically described by false refines everything.

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 more and more sophisticated properties for systems until a desired behavior is specified, in general, does this.

EXAMPLE. A specification of a system that transmits its input from its two input channels to its two output channels (but does not necessarily observe the order) is specified as follows.

TM2

in x: $T1$, y: $T2$ **out** x: $T1$, y: $T2$
$x' \sim x \wedge y' \sim y$

We refine this specification to the simple specification of the time permissive identity TII that reads as follows:

TII

in x: $T1$, y: $T2$ **out** x: $T1$, y: $T2$
$\overline{x}' = \overline{x} \wedge \overline{y}' = \overline{y}$

TII is a property refinement of TM2, formally expressed as

$$\text{TM2} \approx\!> \text{TII}.$$

A proof of this relation is straightforward (see below).

The verification conditions for property refinement are easily generated as follows. For given specifications S_1 and S_2 with interface assertions Φ_1 and Φ_2, the specification S_2 is a property

refinement of S_1 if the syntactic interfaces of S_1 and S_2 coincide and if for the interface assertions Φ_1 and Φ_2, the proposition

$$\Phi_1 \Longleftarrow \Phi_2$$

holds. In our example the verification condition is easily generated. It reads as follows:

$$x' \sim x \wedge y' \sim y \Longleftarrow \bar{y}' = \bar{y} \wedge \bar{x}' = \bar{x}.$$

The proof of this condition is obvious. It follows immediately from the definitions of the time abstraction $\bar{x}$ and $x' \sim x$. For an implementation of the calculus in the interactive proof assistant Isabelle (see [23]), see [24].

The property refinement relation is verified by proving the logical implication between the interface assertions.

Property refinement is useful to relate composed systems to systems specified by logical formulas (see also glass box refinement in Section 6.3). For instance, the following refinement relation

$$\text{TII} \approx\!> (\text{MRG} \circ \text{FRK})$$

holds. Again the proof is straightforward.

As demonstrated the additional assumptions of schematic properties such as strong causality or realizability is a strengthening of the specifying predicate. Therefore it is also a step in the property refinement relation.

Property refinement is a characteristic of the development steps in requirements engineering. It is also used as the baseline of the design process where decisions being made introduce further system properties.

6.2. Compositionality of property refinement

For FOCUS, the proof of the compositionality of property refinement is straightforward. This is a consequence of the simple definition of composition. The rule of compositional property refinement reads as follows:

$$\frac{F_1 \approx\!> \hat{F}_1 \qquad F_2 \approx\!> \hat{F}_2}{F_1 \otimes F_2 \approx\!> \hat{F}_1 \otimes \hat{F}_2}.$$

The proof of the soundness of this rule is straightforward due to the monotonicity of the operator $\otimes$ with respect to set inclusion. Compositionality is often called *modularity* in system development. Modularity allows for a separate development of systems.

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

EXAMPLE. For our example the application of the rule of compositionality reads as follows. Suppose we use a specific component MRG1 for merging two streams. It is defined as follows (recall that $T1$ and $T2$ form a partition of $T3$)

MRG1

in x: $T1$, y: $T2$
out z: $T3$
$z = \langle\langle\rangle\rangle\hat{}f(x, y)$
where
$\forall s \in T1^*, t \in T2^*, x \in (T1^*)^\infty, y \in (T2^*)^\infty$:
$f(\langle s\rangle\hat{}x, \langle t\rangle\hat{}y) = \langle s\hat{}t\rangle\hat{}f(x, y)$

Note that this merge component MRG1 is deterministic and not time independent. According to the FOCUS rule of compositionality and transitivity of refinement, it is sufficient to prove that

$$\text{MRG} \approx\!> \text{MRG1}$$

to conclude that

$$\text{MRG} \circ \text{FRK} \approx\!> \text{MRG1} \circ \text{FRK}$$

By the transitivity of the refinement relation

$$\text{TII} \approx\!> \text{MRG1} \circ \text{FRK}.$$

This shows how local refinement steps that are refinements of subcomponents of a composed system and their proofs are schematically extended to global proofs.

The composition operator and the relation of property refinement leads to a design calculus for requirements engineering and system design. It includes steps of decomposition and implementation that are treated more systematically in the following section.

6.3. Glass box refinement

Glass box refinement is a classical concept of refinement used in the design phase. In this phase we typically decompose a system with a specified interface behavior into a distributed system architecture or we represent (implement) it by a state transition machine. In other words, a glass box refinement is a special case of a property refinement that is of the form

$$F \approx\!> F_1 \otimes F_2 \otimes \cdots \otimes F_n \quad \textit{design of an architecture for a system with interface behavior } F$$

or of the form

$$F \approx\!> B_\Delta(\Lambda) \quad \textit{implementation of system with interface behavior } F \textit{ by a state machine,}$$

where the interface behavior $B_\Delta(\Lambda)$ is defined by a state machine Δ (see also [15]) with Λ as its initial states and outputs.

Glass box refinement means the replacement of a system F by a property refinement that is given by a design. A design is

represented by a network of systems $F_1 \otimes F_2 \otimes \cdots \otimes F_n$ or by a state machine (Δ, Λ) with interface behaviour $B_\Delta(\Lambda)$. The design is a property refinement of F provided that the interface behavior of the net or of the state machine respectively is a property refinement of the system F.

Accordingly, a glass box refinement is a special case of property refinement where the refining system has a specific syntactic form. In the case of a glass box refinement that transforms a system into a network, this form is a term shaped by the composition of a set of systems. The term describes an architecture that fixes the basic implementation structure of a system. These systems have to be specified and we have to prove that their composition leads to a system with the required functionality.

Again, a glass box refinement can be applied afterwards to each of the systems F_i in a network of systems. The systems $F_1, \ldots, F_n$ can be hierarchically decomposed again into a distributed architecture in the same way, until a granularity of systems is obtained which is not to be further decomposed into a distributed system but realized by a state machine. This form of iterated glass box refinement leads to a hierarchical top–down refinement method.

EXAMPLE. A simple instance of such a glass box refinement is already shown by the proposition

$$\text{TII} \approx> \text{MRG} \circ \text{FRK}.$$

It allows us to replace the system TII by a network of two systems.

It is not in the object of this paper to describe in detail the design steps leading from an interface specification to distributed systems or to state machines. Instead, we take a purist's point of view. Since we have introduced a notion of composition we consider a system architecture as being described by a term defining a distributed system composed of a number of systems.

A state machine is specified by a number of state transition rules that define the transitions of the machine (see Section 4.4).

EXAMPLE (GLASS BOX REFINEMENT BY STATE MACHINES). The state machine specification SWT is a glass box refinement for the system UFM. We have

$$\text{UFM} \approx> \text{SWT}.$$

The proof of this formula is a simple consequence of the invariant proved for SWT.

In fact we may also introduce a refinement concept for state machines explicitly in terms of relations between states leading to variations of simulations and bisimulations (see [25–28], and also [29]). This is useful if systems are refined by state machines. We call a relation between state machines with set of initial states Λ and Λ' and transition functions Δ and Δ', a refinement if

$$B_\Delta(\Lambda) \approx> B_{\Delta'}(\Lambda').$$

For more specific rules for a property refinement of state machines that work on the state transition structure, see [30].

Glass box refinement is a special case of property refinement. Thus it is compositional as a straightforward consequence of the compositionality of property refinement.

6.4. Interaction refinement

In FOCUS interaction refinement is the refinement notion for modeling development steps between levels of abstraction. Interaction refinement allows us to change for a system

(i) the number and names of its input and output channels,
(ii) the types of the messages on its channels determining the granularity of the messages.

A pair of two functions describes an interaction refinement

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

that relate the interaction on an abstract level with corresponding interaction on the more concrete level. This pair specifies a development step that leads from one level of abstraction to the other as illustrated by Fig. 5. 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

$$R \circ A = \text{Id}.$$

Let Id denote, the identity relation. We call A the *abstraction*, R the *representation* and (R, A) a *refinement pair*. For nontimed systems we weaken this requirement by requiring $R \circ A$ to be a property refinement of the time permissive identity TII (as a generalization of the specification TII given in Section 6.1

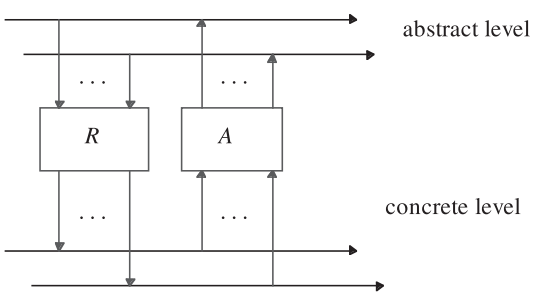

FIGURE 5. Communication history refinement.

to arbitrary sets of channels), formally expressed by (for all histories $x \in \vec{C}$)

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

Choosing the system MRG for R and FRK for A immediately gives a refinement pair for nontimed systems.

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. 6 for the so-called U^{-1}-simulation. Note that here the systems (boxes) A_I and R_O are no longer definitional in the sense of specifications, but rather methodological, since they relate two levels of abstraction. Nevertheless, we specify them as well by the specification techniques introduced thus far.

Given refinement pairs

$$A_\mathrm{I} : \vec{I}_2 \longrightarrow \wp(\vec{I}_1), \qquad R_\mathrm{I} : \vec{I}_1 \longrightarrow \wp(\vec{I}_2)$$
$$A_\mathrm{O} : \vec{O}_2 \longrightarrow \wp(\vec{O}_1), \quad R_\mathrm{O} : \vec{O}_1 \longrightarrow \wp(\vec{O}_2)$$

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 behavior

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

an *interaction refinement* of the interface behavior

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

if the following proposition holds:

$$A_\mathrm{I} \circ F \circ R_\mathrm{O} \approx\!> \hat{F} \quad U^{-1}\text{-simulation}.$$

This formula essentially expresses that $\hat{F}$ is a property refinement of the system $A_\mathrm{I} \circ F \circ R_\mathrm{O}$. Thus, for every 'concrete' input history $\hat{x} \in \vec{I}_2$, every concrete output can also be obtained by translating $\hat{x}$ onto an abstract input history $x \in A_\mathrm{I}.\hat{x}$ such that we can choose an abstract output history $y \in F.x$ such that $\hat{y} \in R_O.y$.

There are three further versions of interaction refinement obtained by replacing in Fig. 6 the upward function A_I by the downward function R_I or the upward function A_O by the downward function R_O or both:

$$F \circ R_O \approx\!> R_I \circ \hat{F} \quad \textit{Downward Simulation},$$
$$A_\mathrm{I} \circ F \approx\!> \hat{F} \circ A_O \quad \textit{Upward Simulation},$$
$$F \approx\!> R_I \circ \hat{F} \circ A_O \quad \textit{U-simulation}.$$

These are different relations to connect levels of abstractions. We prefer U^{-1}-simulation as the most restrictive, 'strongest' notion which implies the other three. This fact is easily demonstrated as follows. From

$$\mathrm{A_I} \circ F \circ R_\mathrm{O} \approx\!> \hat{F}$$

we derive by multiplication with $\mathrm{R_I}$ from the left

$$R_\mathrm{I} \circ A_\mathrm{I} \circ F \circ R_\mathrm{O} \approx\!> R_I \circ \hat{F}$$

and by $R_\mathrm{I} \circ A_\mathrm{I} = \mathrm{Id}$ we get

$$F \circ R_\mathrm{O} \approx\!> R_\mathrm{I} \circ \hat{F},$$

which is the property of downward simulation. By similar arguments we prove that a U^{-1}-simulation $\hat{F}$ is also an upward simulation and a U-simulation. Also the change of the time granularity is an instance of interaction refinement (see [19]).

A more detailed discussion of the mathematical properties of U^{-1}-simulation is given in the following section and more details are found in [1].

EXAMPLE. For the time permissive identity for messages of type $T3$, a system specification reads as follows:

TII3

in z: $T3$
out z: $T3$
$\overline{z} = \overline{z}'$

We obtain

$$\mathrm{TII} \approx\!> \mathrm{MRG} \circ \mathrm{TII3} \circ \mathrm{FRK}$$

as a simple example of interaction refinement by U-simulation. The proof is again straightforward.

Figure 7 shows a graphical description of this refinement relation.

The idea of interaction refinement is found in other approaches to system specification like TLA, as well. It is used heavily in practical system development, although it is hardly ever introduced formally there. Examples are the communication protocols in the ISO/OSI hierarchies (see [31]).

Interaction refinement formalizes the relationship between layers of abstractions in system development.

This way interaction refinement relates the layers of protocol hierarchies, the change of data representations for the messages

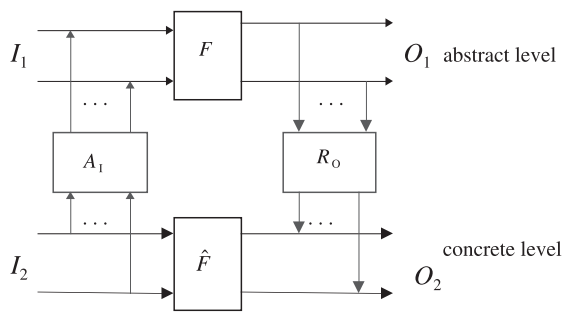

FIGURE 6. Interaction refinement (U^{-1}-*simulation*).

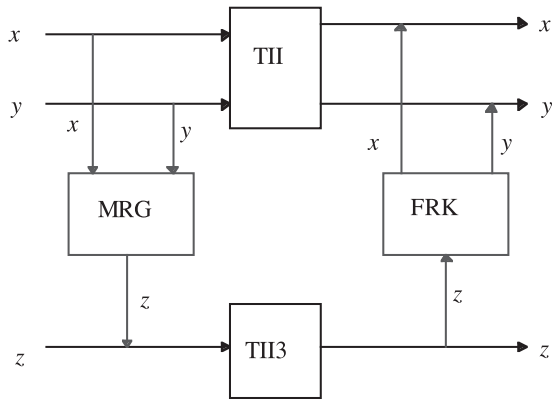

FIGURE 7. Graphical representation of an interaction refinement.

or the states as well as the introduction of time in system developments.

We show in the sequel that in FOCUS an interaction refinement in fact is a Galois connection. This indicates that interaction refinement maintains reasonable structural properties of systems. It shows in particular that under the conditions given below U-simulation and U^{-1}-simulation are in fact equivalent.

THEOREM (INTERACTION REFINEMENT IS A GALOIS CONNECTION). *Let the two function spaces*

$$S_1 = (\vec{I}_1 \longrightarrow \wp(\vec{O})),$$
$$S_2 = (\vec{I}_2 \longrightarrow \wp(\vec{O}))$$

be given and the functions A_I, R_I, A_O *and* R_O *be defined as above. The condition of a Galois connection* (*see, for instance,* [32]) *then reads as follows*

$$\forall F \in S_1, \hat{F} \in S_2 : (A_I \circ F \circ R_O \approx\!> \hat{F}) \equiv (F \approx\!> R_I \circ \hat{F} \circ A_O).$$

This condition is fulfilled if

$$\begin{aligned} &\mathrm{Id} \approx\!> R_I \circ A_I, \quad A_I \circ R_I \approx\!> \mathrm{Id} \\ &\mathrm{Id} \approx\!> R_O \circ A_O, \quad A_O \circ R_O \approx\!> \mathrm{Id} \end{aligned} \qquad (*)$$

Proof. The proof for the direction from left to right reads as follows:

$$\begin{aligned} & A_I \circ F \circ R_O \approx\!> \hat{F} \\ \Longrightarrow\ & \{\text{monotonicity of '}\circ\text{' with respect to property refinement '}\approx\!>\text{'}\} \\ & R_I \circ A_I \circ F \circ R_O \circ A_O \approx\!> R_I \circ \hat{F} \circ A_O \\ \Longrightarrow\ & \{\mathrm{Id} \approx\!> R_I \circ A_I \text{ and } \mathrm{Id} \approx\!> R_O \circ A_O\} \\ & F \approx\!> R_I \circ \hat{F} \circ A_O. \end{aligned}$$

The proof for the direction from right to left reads as follows:

$$\begin{aligned} & F \approx\!> R_I \circ \hat{F} \circ A_O \\ \Longrightarrow\ & \{\text{monotonicity of '}\circ\text{' with respect to '}\approx\!>\text{'}\} \\ & A_I \circ F \circ R_O \approx\!> A_I \circ R_I \circ \hat{F} \circ A_O \circ R_O \\ \Longrightarrow\ & \{A_I \circ R_I \approx\!> \mathrm{Id} \text{ and } A_O \circ R_O \approx\!> \mathrm{Id}, \\ & \text{transitivity of '}\approx\!>\text{', monotonicity of '}\circ\text{' with respect to '}\approx\!>\text{'}\} \\ & A_I \circ F \circ R_O \approx\!> \hat{F} \end{aligned}$$

This completes the proof that an interaction refinement forms a Galois connection. □

Since it is easy to show that under the conditions (*) downward simulation implies U-simulation and also that upward simulation implies U-simulation, we get that under these conditions in fact all four notions of simulations are equivalent. Thus we speak generally of interaction refinement and refer to any of the cases.

6.5. Compositionality of U^{-1}-simulation

Interaction refinement is formulated with the help of property refinement. In fact, it can be seen as a special instance of property refinement. This guarantees that we can freely combine property refinement with interaction refinement in a compositional way.

EXAMPLE. In a property refinement, if we replace the component TII3 by a new component TII3' (for instance along the lines of the property refinement of TII into MRG ∘FRK), we get by the compositionality of property refinement

$$\mathrm{TII} \approx\!> \mathrm{MRG} \circ \mathrm{TII3}' \circ \mathrm{FRK}$$

from the fact that TII3 is an interaction refinement of TII.

We concentrate on U^{-1}-simulation in the following and give the proof of compositionality only for this special case. To keep the proof simple, we do not give the proof for parallel composition with feedback but give the proof in two steps for two special cases, first defining the compositionality for parallel composition without any interaction, which is a simple straightforward exercise, and then give a simplified proof for feedback.

For parallel composition without feedback the rule of compositional refinement reads as follows:

$$\frac{A_I^1 \circ F_1 \circ R_O^1 \approx\!> \hat{F}_1 \qquad A_I^2 \circ F_2 \circ R_O^2 \approx\!> \hat{F}_2}{(A_I^1 || A_I^2) \circ (F_1 || F_2) \circ (R_O^1 || R_O^2) \nu \approx\!> \hat{F}_1 || \hat{F}_2}$$

where we require the following syntactic conditions (let (I_k, O_k) be the syntactic interface of F_k for $k = 1, 2$):

$$O_1 \cap O_2 = \emptyset, \quad \text{and} \quad I_1 \cap I_2 = \emptyset \quad \text{and}$$
$$(I_1 \cup I_2) \cap (O_1 \cup O_2) = \emptyset$$

and analogous conditions for the channels of $\hat{F}_1$ and $\hat{F}_2$. These conditions make sure that there are no name clashes.

It remains to show the compositionality of feedback. The general case reads as follows:

$$\frac{(A_I||A)\circ F\circ(R_O||R) \approx> \hat{F}}{A_I\circ(\mu F)\circ(R_O||R) \approx> \mu\hat{F}},$$

where we require the syntactic conditions (by IN (F) we denote the set of input channels and by out (F) the set of output channels of F)

$$\mathrm{Out}(R) = \mathrm{In}(A) = \mathrm{In}(\hat{F}) \cap \mathrm{Out}(\hat{F}),$$
$$\mathrm{Out}(A) = \mathrm{In}(R) = \mathrm{In}(F) \cap \mathrm{Out}(F).$$

For independent parallel composition the soundness proof of the compositional refinement rule is straightforward. For simplicity, we consider the special case where

$$\mathrm{In}(F) = \mathrm{Out}(F)$$

In other words, we give the proof for only the feedback operator and only for the special case where the channels coming from the environment are empty. This proof generalizes without difficulties to the general case. In our special case the set I is empty and thus A_I can be dropped. For simplicity we write only R instead of R_0. The compositional refinement rule reads as follows:

$$\frac{A\circ F\circ R \approx> \hat{F}}{(\mu F)\circ R \approx> \mu\hat{F}},$$

where $R\circ A = \mathrm{Id}$. The proof of the soundness of this rule is given in the following. Here we use the classical relational notation

$$x F y$$

that stands for $y \in F.x$.

Proof. Soundness for the rule of U^{-1}-simulation:

If we have:	$\hat{z} \in \mu\hat{F}$
then by the definition of μ	$\hat{z}\hat{F}\hat{z}$
and by the hypothesis:	$\exists x, y\colon \hat{z}Ax \wedge xFy \wedge yR\hat{z}$
then by $R\circ A = \mathrm{Id}$:	$yR\hat{z} \wedge \hat{z}Ax \Rightarrow x = y$
we obtain:	$\exists x, y\colon \hat{z}Ax \wedge xFy \wedge yR\hat{z} \wedge x = y$
and thus:	$\exists x\colon \hat{z}Ax \wedge xFx \wedge xR\hat{z}$
therefore:	$x \in \mu F$
and finally:	$\hat{z} \in \mu F\circ R.$

□

The simplicity of the proof of our result comes from the fact that we have chosen such a straightforward denotational model of a system and of composition. In the FOCUS model, in particular, input and output histories are represented explicitly. This allows us to apply classical ideas (see [33, 34]) of data refinement to communication histories. Roughly speaking communication histories are nothing else but data structures that can be manipulated and refined like other data structures, too.

REMARK. Compositionality is valid for the other forms of refinement only under additional conditions (see [1]).

EXAMPLE. To demonstrate interaction refinement, let us consider the specification of two trivial delay components. They forward their input messages to their output channels with some delay.

D3

in c, z: T3
out c, z: T3
$\overline{c'} = \overline{z}$ $\overline{z'} = \overline{c}$

D

in x, c: $T1$, y, d: $T2$
out x, c: $T1$, y, d: $T2$
$\overline{c'} = \overline{x}$, $\overline{x'} = \overline{c}$ $\overline{\mathrm{d}'} = \overline{y}$, $\overline{y'} = \overline{\mathrm{d}}$

We have (see Fig. 8)

$$\mathrm{D} \approx> (\mathrm{MRG}||\mathrm{MRG})[c/x, d/y, c/z])\circ\mathrm{D3}\circ (\mathrm{FRK}||\mathrm{FRK}[c/x, d/y, c/z])$$

and in addition (here we write μ^c for a feedback only on channel c; see Fig. 9)

$$\mathrm{TII3} \approx> (\mu^c\mathrm{D3})\backslash\{c\}, \quad \mathrm{TII} \approx> (\mu^{c,d}\mathrm{D})\backslash\{c, d\}$$

and so finally we obtain ((see Fig. 10) by applying the rule of the compositionality of refinement for feedback)

$$\mathrm{TII} \approx> (\mu^{c,d}\mathrm{D})\backslash\{c, d\} \approx> \mathrm{MRG}\circ(\mu^c\mathrm{D3})\backslash\{c\}\circ\mathrm{FRK}.$$

This shows the power of compositionality for interaction refinement.

We obtain a refinement calculus, which can also be supported by a CASE tool. All the refinement rules are transformation rules. Their verification can be supported by an interactive theorem prover, their application by a transformation system.

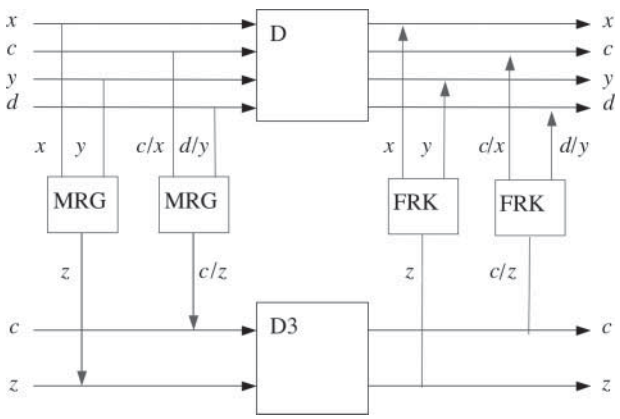

FIGURE 8. Interaction refinement.

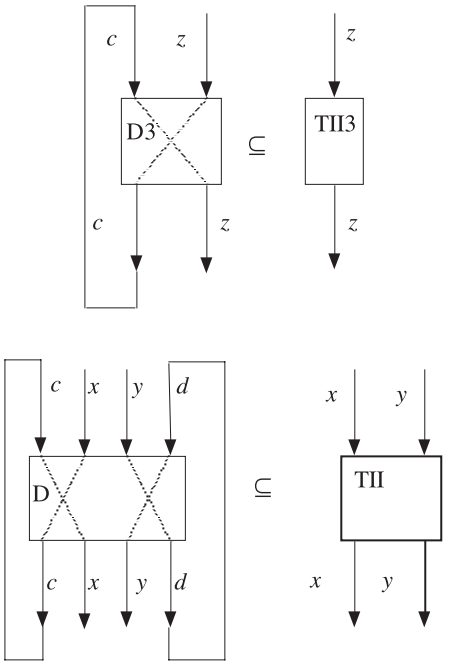

FIGURE 9. Refinement relations.

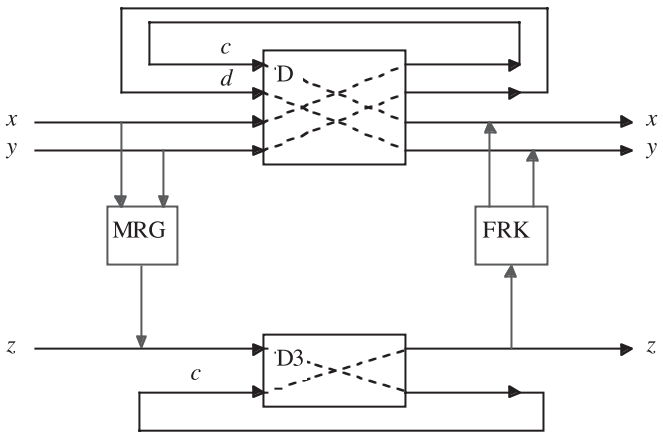

FIGURE 10. Interaction refinement.

7. DISCUSSION AND CONCLUSIONS

The previous chapters introduced a comprehensive mathematical and logical theory and method as a step towards a foundation for a component-oriented system and software development. It addresses all the steps of a hierarchical stepwise refinement development method. It is compositional and therefore supports all the modularity requirements that are generally needed. The FOCUS refinement calculus leads to a logical calculus for 'programming in the large' to argue about software architectures and their refinement (see [9]).

To keep our presentation simple and to be able to present all the theory in one paper we chose a minimalist approach and introduced just enough notation to deal with all FOCUS concepts and to be able to give simple examples. Therefore a reader interested in more practical applications might object and ask obvious questions for justification such as

(i) Is the FOCUS model expressive enough?
(ii) Are there other, perhaps for software modeling, better suited or more expressive models?
(iii) Does the theoretic approach extend to techniques and methods typically used in practice such as SDL, UML or general object-oriented approaches?
(iv) How does the logical basis relate to typical concepts of software engineering such as software architectures or design patterns?

We shortly discuss these more application-oriented questions in the following.

7.1. Expressiveness of the model

The development of large software systems includes the implicit or explicit construction of a model and its documentation. This calls for a fundamental system model of an interactive, concurrent system. In this section we discuss the FOCUS model in relation to other system models. We start with a classification of system models and compare the FOCUS model to others on these grounds.

7.1.1. Categories of system models

The system model introduced is based on the concept of *nondeterministic data flow*, also called *nondeterministic Kahn networks* (see [35]) or a generalization of the *pipes and filters* style in software architecture (see [36]). There are many more fundamental models of interacting systems. We identify at least three basic concepts of communication in distributed systems that interact by message exchange:

(i) *Asynchronous communication* (message asynchrony): a message is sent as soon as the sender is ready, independently of the question whether a receiver is ready to receive it. Sent messages are buffered (by the communication mechanism) and can be received by the receiver at any later time; if a receiver wants to receive a message but no message has been sent it has to wait. However, senders never have to wait (see [35, 37]) until receivers are ready.
(ii) *Synchronous communication* (message synchrony, rendezvous, handshake communication): a message can be sent only if both the sender and the receiver are simultaneously ready to communicate; if only one of them (receiver or sender) is ready for communication, it has to wait until a communication partner gets ready (see [38, 39]).
(iii) *Time synchronous communication* (perfect synchrony): several interaction steps (signals or atomic events)

are gathered into one time slot; this way systems are described that handle of sets of events in each time slot (see [17] as a well-known example).

In addition there is the wide class of state-based system models that interact by shared memory. Since we are favoring an approach with an encapsulated state, we do not go into a detailed discussion of system models based on global states.

7.1.2. *Examination of other system models*

We have introduced a comprehensive logical theory of component-oriented systems engineering for the model of nondeterministic data flow. Can we develop a similar approach for the other system models mentioned in the previous section? In fact, many of the notions can also be developed for the other approaches, but they may show limitations. For instance, the idea of interaction refinement does not carry over so easily to synchronous message passing such as in CSP or CCS, where it is difficult to change the granularity of messages used in interactions in a modular way. The reason is that communication and choice are combined in message synchrony (also called *external choice*). In CSP an action or a message is used to synchronize the communication agents. Replacing an action by several ones brings difficulties for the synchronization, if several different actions are replaced by a sequence of actions starting with identical actions on which then the synchronization is to be carried out.

In other candidates such as state-based systems with shared memory such as Unity (see [21]), parallel composition proves to be not modular for liveness properties, since a compositional logical theory is not available without a more complex extension such as rely/guarantee. Moreover, there is no notion of encapsulation and system interface.

7.1.3. *Justification of the model*

Finally the question arises as to how the other categories of system models compare with to the introduced model. Let us briefly discuss that matter.

Our first argument is of the methodological nature. When studying large systems in the most abstract way we often concentrate on the pure flow of information. Synchronization is of minor interest. Nevertheless, there are further, more technical arguments.

First of all, for the other models such a comprehensive framework including a mathematical model, a modular specification and refinement methodology (see also the discussion in the next section) simply does not exist, thus far. Further, for some of these aspects it is not so obvious how to introduce them into the other system models. For instance, when working with the model of message synchrony the idea of a modular interaction refinement is not so obvious (CSP, CCS, see above).

Second, the FOCUS model is quite comprehensive and expressive. In fact, it has been shown that the other system models can be mapped onto this model, such as remote procedure call and method invocation (see [40]), synchronized message passing (see [40]), and time synchrony (see [2]). This mapping is, in particular, a basis for the study of object-oriented system models (see [40]).

The approach which is certainly close to FOCUS is TLA (see [25, 26]). In TLA many of the FOCUS concepts are also available. However, the model of TLA is inherently state based and essentially uses the idea of interleaving. Thus TLA does not include explicit concurrency. Moreover, the logical framework of TLA uses temporal logic, while we are interested to deal with a more traditional straightforward logical basis. Finally, in TLA the idea of an explicit interface abstraction (in our terms of black box views) is not emphasized. In TLA the idea of an interface is defined in terms of rely/guarantee.

7.2. Relationship to practical software engineering

In fact FOCUS is rather mathematical and formal. Nevertheless how does that match with more practical pragmatic approaches to specification in software and systems engineering?

7.2.1. *Limitations and achievements*

There are many important issues in systems and software engineering that are hardly addressed directly by a sound semantic foundation as FOCUS, such as the management and control of the large quantity of information inevitably required in software development. What we can address, however, are issues of elicitation, conflict identification and resolution, for instance, in requirements engineering. The idea of system development by refinement is a highly idealistic view, which can be hardly applied in a purist way in practice. However, although being idealistic, the sound and well-founded concepts pave the way for a deeper understanding of basic development steps and their support in the system development process.

In practice, instead of logical techniques graphical methods connected with some textual annotations are advocated. A prominent example is UML (see [41]). Actually, logical techniques are also considered in this context (see CDL, the *constraint definition language*). However, UML does not have a worked out semantic theory thus far, in spite of the many attempts to provide a formalization at least of parts of UML.

We consciously did not try to give a semantic foundation for UML as it is, although we believe that FOCUS as a logical basis is powerful enough to do that (see [42]). We do not think that systems engineering modeling languages should be scientifically based by giving a semantic foundation after their design. We rather advocate a close interaction between the design of a modeling language and its semantic model and theory. The mutual relationships between the choice of the semantic model and the language are much too strong and important to develop one independent of the other. One might argue that we exactly went to the other extreme by only studying the semantic model and its logical basics without discussing the modeling and description issues. This is certainly

correct. In fact, in this paper we tried to concentrate on the logical fundamentals. Nevertheless, the presented approach was developed hand in hand with a more pragmatic method supporting a graphical modeling notation much along the lines of SDL or UML but with a strong semantic foundation.

For the introduced method a CASE tool called AutoFocus has been implemented (see [43]) which was started as a scientific experiment, and today is already used in practical application projects. AutoFocus supports a graphical notation for views (architecture, state, interaction) on a system with semantics in terms of the system model introduced above. An incorporated part of the tool are translators that generate logical formulas from the graphical descriptions that are given as input to interactive theorem provers and to model checkers. In particular, the theorem prover can make use of the logical properties of systems such as strong causality. The system model serves as the basis for the semantic integration.

AutoFocus supports also a number of simple consistency checks between the views of the system. For the concept of consistency the system model is a helpful guide.

7.2.2. *Software architecture*

In software and systems engineering, for 'development in the large' it has been recognized that the notion of software architecture is essential. Much work has been done on architectural styles (see [44]), architectural description formalism (see [8, 9]) and architectural principles including design patterns (see [45, 46]). Actually the FOCUS model is a generalization of the architectural style describing pipes and filters architectures allowing arbitrary feedback loops, buffering, nondeterminism and the encapsulated state (there is a close relationship between the model we introduced and Rapide; see [8]). It strictly concentrates on pure information flow in the context of time and causality and thus it provides an abstract view of systems. Nevertheless, it is powerful enough as a representation or embedding for other system models. In fact, embedding classical shared memory system models with multithreading and further technical issues such as locking disciplines are not easily translated into the FOCUS model. The model rather addresses highly distributed systems interacting only by message exchange.

Our paper does not contribute at all to the methodology of software architecture design. However, it contributes to the theory of integrated system and architecture modeling. In particular, it concentrates on the basic behavior issues leading to a basic system model.

Some of the work on software architecture concentrates on the idea of components and connectors. Using the FOCUS model we describe connectors again by a special class of system specifications.

Thus far, formalizations of software architectures were often based on more operational system models such as communicating sequential processes (such as that by Shaw and Garlan [36] based on CSP; see [38]) and not on logic and specification-based techniques. We see advantages, however, in a strictly descriptive approach to system modeling since this supports a style of work that allows for the flexible formalization of those properties of systems in which one is most interested. Moreover, it is certainly helpful to work with an explicit mathematical system model and not with a process algebra like CSP that leaves the semantic model implicit.

Property-oriented system specifications provide the basis for property-oriented description of software architectures.

7.3. Goals and perspectives

The presented method aims, in particular, at the following logical and mathematical foundations for software and systems engineering (see [47]):

(i) a mathematical notion of a syntactic and semantic interface of a system,
(ii) a formal interface specification notation and method (for an extension to services see [48, 49]),
(iii) a precise notion of composition,
(iv) a mathematical notion of refinement and development,
(v) a compositional development method,
(vi) a flexible concept of software architecture,
(vii) concepts of time and the refinement of time (see [2, 19]).

What we did not mention throughout the paper are concepts that are also available and helpful from a more practical point of view including

- systematic combination with tables and diagrams (see [50]),
- tool support in the form of AutoFocus (see [43]).

In fact, there may be other system models that can perhaps provide a similar fundamental framework. However, this is not obvious as indicated in the discussion above.

The simplicity of our results is a direct consequence of the specific choice of the semantic model for FOCUS. The introduction of time makes it possible to talk about causality, which makes the reasoning about feedback loops in the model robust and expressive. The fact that communication histories are explicitly included into the model allows us to avoid all kinds of complications like prophecies or stuttering (see [25, 26]).

What we have presented is just the scientific kernel and justification of method. More pragmatic ways to describe specifications are needed. These more pragmatic specifications have been worked out in the SysLab-Project (see [42]) at the Technical University of Munich. For extensive explanations of the use of state transition diagrams, data flow diagrams and message sequence charts as well as several versions of data structure diagrams we refer to this work.

An attempt to specialize the presented work also to system concepts used in practice such as object-oriented analysis, design, and programming is found in [40]. It leads there, in particular, to an abstract method for interface specifications

for classes and objects. Whether this method is of practical value is another question that can only be answered after more experimentation.

FUNDING

ACKNOWLEDGEMENT

The material presented in this paper was worked out to a large extent while working on [15] together with Ketil Stølen who has contributed substantially to the presented ideas. It is a pleasure to thank Ketil Stølen, Max Breitling, Jan Philipps, Markus Pizka, Alexander Pretschner and Bernhard Schätz for a number of comments and helpful suggestions for improvement.

REFERENCES

[1] Broy, M. (1992) Compositional Refinement of Interactive Systems. SRC Report 89, Digital Systems Research Center. Also in (1997) *J. ACM*, **44**, 850–891.

[2] Broy, M. (1997) Refinement of Time. In Bertran, M. and Rus, Th. (eds), *Transformation-Based Reactive System Development. ARTS'97*, Mallorca, 1997, pp. 44–63. Lecture Notes in Computer Science 1231. Springer 1997.

[3] Lynch, N.A. and Stark, E.W. (1989) A proof of the Kahn principle for input/output automata. *Inform. Comput.*, **82**, 81–92.

[4] Broy, M., Dederichs, F., Dendorfer, C., Fuchs, M., Gritzner, T.F. and Weber, R. (1992) The Design of Distributed Systems—An Introduction to Focus. Sonderforschungsbereich 342: Methoden und Werkzeuge für die Nutzung paralleler Architekturen TUM-I9202, Institut für Informatik, Technische Universität München.

[5] Broy, M., Dederichs, F., Dendorfer, C., Fuchs, M., Gritzner, T.F. and Weber, R. (1992) Summary of Case Studies in FOCUS—A Design Method for Distributed Systems. Sonderforschungsbereich 342: Methoden und Werkzeuge für die Nutzung paralleler Architekturen TUM-I9203, Institut für Informatik, Technische Universität München.

[6] Broy, M., Breitling, M., Schätz, B. and Spies, K. (1997) Summary of Case Studies in Focus—Part II. SFB-Bericht Nr. 342/40/97A, Institut für Informatik, Technische Universität München.

[7] Bass, L., Clements, P. and Kazman, R. (1998) *Software Architecture in Practice*. Addison-Wesley, Reading, MA.

[8] Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D. and Mann, W. (1995) Specification and analysis of system architecture using Rapide. *IEEE Trans. Softw. Eng.*, **21**, 336–355 (Special Issue on Software Architecture).

[9] Moriconi, M. Qian, X. and Riemenschneider, R.A. (1995) Correct architecture refinement. *IEEE Trans. Softw. Eng.*, **21**, 356–372 (Special Issue on Software Architecture).

[10] Leavens, G.T. and Sitaraman, M. (2000) *Foundations of Component-Based Systems*. Cambridge University Press.

[11] Szyperski, C. (1998) *Component Software—Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press.

[12] Broy, M. (2000) From States to Histories. In Bert, D., Choppy, Ch. and Mosses, P. (eds), *Recent Trends in Algebraic Development Techniques. WADT'99*, pp. 22–36. Lecture Notes in Computer Science 1827. Springer.

[13] Brock, J.D. and Ackermann, W.B. (1981) Scenarios: A Model of Non-determinate Computation. In Diaz, J. and Ramos, I. (eds), *Proc. Conf. Formal Definition of Programming Concepts,* pp. 225–259. Lecture Notes in Computer Science 107. Springer 1981.

[14] Andrews, P. (1986) *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Computer Science and Applied Mathematics. Academic Press.

[15] Broy, M. and Stølen, K. (2001) *Focus on System Development*. Springer, 2001.

[16] Spivey, J.M. (1988) *Understanding Z—A Specification Language and Its Formal Semantics*. Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press.

[17] Berry, G. and Gonthier, G. (1988) The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. Research Report 842, INRIA.

[18] Broy, M. (2007) Interaction and Realizability. In van Leeuwen, J., Italiona, G.F., van der Hoek, W., Meinel, C., Sack, H. and Plasil, F. (eds), *SOFSEM 2007: Theory and Practice of Computer Science*, pp. 29–50. Lecture Notes in Computer Science 4362. Springer.

[19] Broy, M. (2009) Relating Time and Causality in Interactive Distributed Systems. In Broy, M., Sitou, W. and Hoare, T. (eds), *Engineering Methods and Tools for Software Safety and Security*, pp. 75–130. NATO Science for Peace and Security Systems, D: Information and Communication Security 22. IOS Press.

[20] Broy, M. (1997) The Specification of System Components by State Transition Diagrams. TUM-I9729, Institut für Informatik, Technische Universität München.

[21] Chandy, K.M. and Misra, J. (1988) *Parallel Program Design: A Foundation*. Addison-Wesley.

[22] Broy, M., Möller, B., Pepper, P. and Wirsing, M. (1986) Algebraic implementations preserve program correctness. *Sci. Comput. Program.*, **8**, 1–19.

[23] Nipkow, T., Paulson, L.C. and Wenzel, M. (2002) *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science 2283. Springer.

[24] Spichkova, M. (2008) Refinement-based verification of interactive real-time systems. *Electr. Notes Theor. Comput. Sci.*, **214**, 131–157.

[25] Abadi, M. and Lamport, L. (1988) The Existence of Refinement Mappings. SRC Report 29, Digital Systems Research Center.

[26] Abadi, M. and Lamport, L. (1990) Composing Specifications. SRC Report 66, Digital Systems Research Center.

[27] Back, R.J.R. (1989) Refinement Calculus, Part I: Sequential Nondeterministic Programs. REX Workshop. In de Bakker, J.W., de Roever, W.-P. and Rozenberg, G. (eds), *Stepwise Refinement of Distributed Systems*, pp. 42–66. Lecture Notes in Computer Science 430. Springer.

[28] Back, R.J.R. (1989) Refinement Calculus, Part II: Parallel and Reactive Programs. REX Workshop. In de Bakker, J.W., de Roever, W.-P. and Rozenberg, G. (eds), *Stepwise Refinement of Distributed Systems*, pp. 67–93. Lecture Notes in Computer Science 430. Springer.

[29] Aceto, L. and Hennessy, M. (1991) Adding Action Refinement to a Finite Process Algebra. *Proc. ICALP'91*, pp. 506–519. Lecture Notes in Computer Science 510. Springer.

[30] Rumpe, B. (1996) Formale Methodik des Entwurfs verteilter objektorientierter Systeme. Dissertation, Fakultät für Informatik, Technische Universität München, 1996. Published by Herbert Utz Verlag.

[31] Herzberg, D. and Broy, M. (2005) *Modeling Layered Distributed Communication Systems. Applicable Formal Methods*, Vol. 17. Springer.

[32] Möller, B. (1999) Algebraic Structures for Program Calculation. In Broy, M. and Steinbrüggen, R. (eds), *Calculational System Design.* Marktoberdorf Summer School, 1998, pp. 25–100. NATO Science Series, Series F: Computer and System Sciences 173. IOS Press, Amsterdam.

[33] Coenen, J., de Roever, W.P. and Zwiers, J. (1991) Assertional Data Reification Proofs: Survey and Perspective. Bericht Nr. 9106, Institut für Informatik und praktische Mathematik, Christian-Albrechts-Universität Kiel.

[34] Hoare, C.A.R. (1972) Proofs of correctness of data representations. *Acta Inform.*, **1**, 271–281.

[35] Kahn, G. (1974) The Semantics of a Simple Language for Parallel Processing. In Rosenfeld, J.L. (ed.), *Proc. IFIP Congress 74* (*Information Processing 74*), pp. 471–475. North-Holland, Amsterdam.

[36] Shaw, M. and Garlan, D. (1996) *Software Architecture: Perspectives of an Emerging Discipline*. Prentice-Hall, Englewood Cliffs, NJ.

[37] Specification and Description Language (SDL) (1988). Recommendation Z.100. Technical Report, CCITT.

[38] Hoare, C.A.R. (1985) *Communicating Sequential Processes*. Prentice-Hall.

[39] Milner, R. (1980) *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92. Springer.

[40] Broy, M. (1996) Towards a Mathematical Concept of a Component and its Use. *First Components' User Conf.*, Munich. Revised version in (1997) *Softw., Concepts Tools*, **18**, 137–148.

[41] Booch, G., Rumbaugh, J. and Jacobson, I. *The Unified Modeling Language for Object-Oriented Development*, Version 1.0. RATIONAL Software Cooperation.

[42] Breu, R., Grosu, R., Huber, F., Rumpe, B. and Schwerin, W. (1997) Towards a Precise Semantics for Object-Oriented Modeling Techniques. In Kilov, H. and Rumpe, B. (eds), *Proc. ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques.* Also in Technische Universität München, Institut für Informatik, TUM-I9725.

[43] Huber, F., Schätz, B. and Einert, G. (1997) Consistent Graphical Specification of Distributed Systems. In Fitzgerald, J., Jones, C.B. and Lucas, P. (eds), *4th Int. Symp. Formal Methods Europe* (*FME'97*), pp. 122–141. Lecture Notes in Computer Science 1313. Springer.

[44] Shaw, M. and Clements, P. (1997) A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. *Proc. COMPSAC*, Washington, DC, August.

[45] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996) *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, New York.

[46] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

[47] Broy, M. (2006) The 'grand challenge' in informatics: engineering software-intensive systems. *IEEE Comput.*, **39**, 72–80.

[48] Broy, M. (2004) Service-oriented Systems Engineering: Specification and Design of Services and Layered Architectures: The JANUS Approach. In Broy, M., Grünbauer, J., Harel, D. and Hoare, T (eds), *Engineering Theories of Software Intensive Systems*, Marktoberdorf, Germany, August 3–15. NATO Science Series, II. Mathematics, Physics and Chemistry 195. Springer.

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

[50] Broy, M (1995) Mathematical System Models as a Basis of Software Engineering. In van Leeuwen, J. (ed.), *Computer Science Today*. pp. 292–306. Lecture Notes of Computer Science 1000. Springer.

APPENDIX 1: STRONG CAUSALITY AND PREFIX MONOTONICITY

We look only at deterministic behaviors that can be represented by simple functions between histories. Let a strongly causal function

$$f : (M^*)^\infty \to \wp((M^*)^\infty)$$

be given such that there exists a function

$$\bar{f} : M^\omega \to M^\omega$$

that satisfies the equation

$$\{\bar{f}.\bar{x}\} = \overline{f.x},$$

where $x \in (M^*)^\infty$. By this equation, we easily prove that f is time independent: From $\bar{x} = \bar{x'}$ we derive by

$$\overline{f.x} = \{\bar{f}.\bar{x}\} = \{\bar{f}.\bar{x'}\} = \overline{f.x'}$$

the equation $\overline{f.x} = \overline{f.x'}$. Since $\overline{f.x}$ is a one element set, we write $\overline{f.x}$ in the following also to denote this unique element in the set.

Conversely, if f is time independent, then there exists always a function

$$\bar{f} : M^\omega \to \wp(M^w)$$

such that

$$\bar{f}.\bar{x} = \overline{f.x}$$

since $\overline{f.x}$ and $\overline{f.x'}$ coincide for all x and x' with $\bar{x} = \bar{x'}$.

We show that $\bar{f}$ is prefix monotonic provided that f is strongly causal. For a stream $x \in (M^*)^\infty$ where $\bar{x}$ is finite

there exists some time $t \in \mathbb{N}$ such that

$$\overline{x \downarrow t} = \bar{x} \wedge \overline{(f.x) \downarrow t+1} = \overline{f.x}$$

provided that $\overline{f.x}$ is also finite. Let us assume for $z \in (M^*)^\infty$ that

$$\bar{x} \sqsubseteq \bar{z}.$$

Then there exists a stream z' such that $\bar{z} = \bar{z'}$ and

$$x \downarrow t = z' \downarrow t.$$

By strong causality we get

$$(f.x) \downarrow t+1 = (f.z') \downarrow t+1$$

and since we have

$$\overline{f.x} = \overline{(f.z') \downarrow t+1} \sqsubseteq \overline{f.z'}$$

we obtain the proposition

$$\overline{f.x} \sqsubseteq \overline{f.z'} = \overline{f.z}.$$

If $\overline{f.x}$ is infinite although the stream $\bar{x}$ is finite, then, for all streams z with

$$\bar{x} \sqsubseteq \bar{z}$$

we prove that $\overline{f.z} = \overline{f.x}$. By strong causality we can find arbitrarily large numbers $t \in \mathbb{N}$ and streams z' such that

$$x \downarrow t = z' \downarrow t \wedge \bar{z} = \bar{z'}$$

and thus by strong causality

$$\overline{(f.x) \downarrow t+1} = \overline{(f.z') \downarrow t+1}.$$

This shows that for all times $t \in \mathbb{N}$ there exist input streams z' with $\bar{z} = \bar{z'}$ and

$$\overline{(f.x) \downarrow t+1} = \overline{(f.z') \downarrow t+1}.$$

Since, for all such z', we have $\overline{f.z} = \overline{f.z'}$, we get $\overline{f.x} = \overline{f.z}$. This shows that strong causality implies prefix monotonicity for the case of deterministic functions derived by time abstractions.

APPENDIX 2: COMPOSITIONALITY OF REALIZABILITY AND TIME INDEPENDENCE

Let $F_i : \vec{I}_i \to \wp(\vec{O}_i)$ for $i = 1, 2$, be a given specification, such that $F_1 \otimes F_2$ is well defined. We prove that:

(1) if the F_i are *realizable* for $i = 1, 2$, then so is $F_1 \otimes F_2$,
(2) if the F_i are *fully realizable* for $i = 1, 2$, then so is $F_1 \otimes F_2$,
(3) if the F_i are *fully time independently realizable* for $i = 1, 2$, then so is $F_1 \otimes F_2$.

First note that given functions $f_i : \vec{I}_i \to \vec{O}_i$ for $i = 1, 2$, with $f_i \in [[F_i]]$, then $f_1 \otimes f_2$ is a function in $[[F_1 \otimes F_2]]$; this is a simple consequence of the fact that by the strong causality of the f_i, we construct inductively a unique solution of the defining equation for $f_1 \otimes f_2$. This shows (1).

We give the proof of (2) and (3) not for the general composition operator $\otimes$ but only for the fixpoint operator. The generalization is straightforward. Let a function

$$F : \vec{I} \to \wp(\vec{O}) \quad \text{with } I = \{x, z\},\ O = \{y, z\}$$

be given. For simplicity of notation we write

$$(u, v) \in F(s, r)$$

for

$$b \in F.a$$

where $a \in \vec{I}, b \in \vec{O}$ and $a.x = s$ and $a.z = r$ as well as $b.y = u$ and $b.z = v$. We define μF as follows:

$$(\mu F).x = \{(y, z) : (y, z) \in F(x, z)\}.$$

Full realizability of the function μF is a simple consequence of full realizability of the function F: if

$$(y, z) \in (\mu F).x$$

then by definition

$$(y, z) \in F(x, z).$$

Since F is fully realizable, there exists a function $f \in [\![F]\!]$ with

$$(y, z) = f(x, z)$$

and by definition $\mu f \in [\![\mu F]\!]$. We have

$$(y, z) = (\mu f).x$$

since by the strong causality of f the fixpoint of f is unique. This proves (2).

Let F be fully time independently realizable; then for $f \in [\![F]\!]_{\text{ti}}$ we have $\mu f \in [\![\mu F]\!]_{\text{ti}}$ and $(\mu F).x = \{\mu f.x : \mu f \in [\![\mu F]\!]_{\text{ti}}\}$

It remains to prove that μF is time independent. Assume that

$$(y, z) \in (\mu F).x.$$

Then by definition there exists a function $f \in [\![F]\!]$ with $f(x, z) = (y, t)$ that is strongly causal and time independent.

As shown in Appendix 1 there exists a function $\bar{f}$ specified by

$$\bar{f}.\bar{x} = \overline{f.x}$$

such that

$$(y, z) = f(x, z)$$

and $\bar{f}$ is prefix monotonic. Therefore it has a uniquely defined fixpoint p. By induction we show that

$$(y, z) = (\mu f).x \Rightarrow \bar{z} = p.$$

We construct z as follows: we specify a sequence of timed streams y_t, z_t and q_t

$$z_0 = \langle\,\rangle^\infty,$$
$$z_{t+1} = q_t \downarrow (t+1)\hat{}\langle\,\rangle^\infty \quad \text{where } (y_t, q_t) = f(x, z_t).$$

We prove that

$$\bar{z}_t \sqsubseteq p$$

by induction: the base case with $t = 0$ is trivial, since then $= \bar{z}_t = \langle\,\rangle$. Let the induction hypothesis hold for t. We get

$$\begin{aligned}\overline{z_{t+1}} &= \overline{q_t \downarrow (t+1)} \\ &= \overline{f(x, z_t)_2 \downarrow (t+1)} \\ &\sqsubseteq \bar{f}(\bar{x}, p)_2 \\ &= p\end{aligned}$$

and also

$$z_t \downarrow t = z \downarrow t,$$
$$(\bar{y}, \bar{z}) = \bar{f}(\bar{x}, \bar{z}).$$

Since p is the least fixpoint and $\bar{z} \sqsubseteq p$ we get $\bar{z} = p$. If $\bar{x} = \bar{x'}$ holds, then by the construction with

$$(y', z') = f(x', z')$$
$$(y, z) = f(x, z)$$

we get

$$\bar{z} = \bar{z'} = p$$

and thus

$$\overline{F.x} = \overline{F.x'}$$

This shows that μF is time independent.

If F is strongly causal but not fully realizable, then μF is not guaranteed to be strongly causal. We demonstrate this with an example. Let a function

$$F : \vec{I} \to \wp(\vec{O}) \quad \text{with } I = \{x, z\},\ O = \{y, z\}$$

be given. Consider the example

$$F(z, x) = \{(y, z'): x.1 = \langle 1\rangle \Rightarrow z' \neq z\}.$$

Then for $x.1 = \langle 1\rangle$ we get

$$(\mu F).x = \emptyset$$

but for $x.1 \neq \langle 1\rangle$ we get

$$(\mu F).x = \{(y, z'): \text{true}\}$$

Thus F is not strongly causal, since μF is partial, but $(\mu F).x \neq \emptyset$ for some stream x. Even if F is strongly causal and realizable but not fully realizable, μF is not guaranteed to be strongly causal. Consider the example

$$F(z, x) = \{(y, z') : y = z \wedge ((x.1 = \langle 1\rangle \Rightarrow z' \neq z) \vee z' = \langle\langle\,\rangle\rangle^\infty)\}.$$

Then for x.1 = $\langle 1\rangle$ we get

$$(\mu F).x = \{(\langle\langle\,\rangle\rangle^\infty, \langle\langle\,\rangle\rangle^\infty)\}$$

but for $x.1 \neq \langle 1\rangle$ we get

$$(\mu F).x = \{(y, z'): \text{true}\}.$$

Obviously, μF is not strongly causal.