Towards a Mathematical Concept of a Component and Its Use*

Keynote Speech at the Components' Users Conference CUC'96
Revised and Extended Version

Manfred Broy
Fakultät für Informatik
Technische Universität München
80290 München

Abstract

We deal with the concept of a component considered as a black box that is a physical encapsulation of related services according to a published specification. These services can only be accessed through a consistent and published interface that includes an interaction standard. Such a notion of a component needs a carefully chosen semantic concept of a syntactic and a semantic interface that allows us to provide a precise unambiguous published specification. We present a mathematical model for the interface of components. We demonstrate its use for the modeling of software architectures, interoperability between components, and the process of incremental development.

1. Introduction

A descriptive functional semantic model of distributed systems of interacting components is of major interest in many research areas as well as applications of computing science and systems engineering. For the modular, systematic development of systems and a proper method for for the design of software following the idea of componentware we need precise and readable interface descriptions of system components. We require that such interface descriptions contain all the informations about the syntactic and semantic properties of a component needed in order to use it properly. In the interface specification, we also have to be describe the time dependencies of a component if relevant for its use.

*) This work was partially sponsored by the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen", the BMBF-project ARCUS and the industrial research project SysLab sponsored by Siemens Nixdorf and by the DFG under the Leibniz program.

We are interested in the description of components that react interactively to input by output. Both input and output takes place within a frame of time. The modular specification of the observable behaviour of interactive systems is an important technique in system and software development. We speak of *black box specifications* or *interface specifications*. An adequate concept of interface specification does not only depend on a simple notion of observability, but also on the operators that we apply to compose components into systems.

Although ignored in theoretical computer science for a while, the incorporation of time and its formal representation is of essential interest for system models and system specification. In time dependent systems, the timing and the data values of the output depend upon the timing and the data values of the input. Of course, for certain components the timing of the input does not influence the data values of the output, but only their timing. Then we can describe the input/output behaviour of a component without explicit reference to time.

In the following, we introduce a semantic model of an interface behaviour for components and study composition operators. On the basis of this semantic model we introduce specification techniques for the description of reactive components. In a first section, we introduce the mathematical basis for modelling the behaviour of components. Then we show how to describe the syntactic interfaces and the dynamic behaviours of interactive systems. To demonstrate the generality of our component concept we show that sequential object-oriented systems or systems with synchronous message exchange can be incorporated as special instances of our concept of components. We treat refinement and composition operators. Finally we demonstrate the use of our concept to deal with software architectures and with incremental system development.

2. A Semantic Concept of a Component Interface

The notion of a component is essential in modular program construction, in software architecture, and in systems engineering. In this section we define a mathematical concept of a component.

2.1 Streams as Interaction Histories

In this section we introduce the basic mathematical concepts for the description of system components by functional and relational techniques. We concentrate on a black box view of a component. We define the concept of streams over a given set of messages to model the communication over channels. We introduce a number of useful functions on this set. Finally we define the notion of stream processing function that models the black box behaviour of a component.

We model components as interactive systems that communicate asynchronously through channels. We use streams to denote histories of communications on channels. Given a set M of messages, a stream over the set M is a finite or infinite sequence of elements from M. By M^* we denote the finite sequences over the set M. The set M^* includes the empty sequence that we denote by ‹›.

By the set M^∞ we denote the infinite sequences over the set M. The set M^∞ can be understood to be represented by the total mappings from the natural numbers $\mathbb{N}$ into M. Formally we define the set of timed streams by (we write S^∞ for the function space $\mathbb{N}^+ \to S$ and $\mathbb{N}^+$ for $\mathbb{N} \setminus \{0\}$)

$$M^\aleph =_{def} (M^*)^\infty .$$

Another possibility to represent timed streams is to add a special symbol $\surd$ (called time tick) to the set M. The set $M^\aleph$ is isomorphic to the set $(M \cup \{\surd\})^\infty$ of streams over the set $M \cup \{\surd\}$ with an infinite number of time ticks (here we assume, of course, that $\surd \notin M$). We denote for $i \in \mathbb{N}$, $x \in M^\aleph$ by

$$x{\downarrow}i$$

the sequence of the first i sequences in the stream x it represents the history to the first i time intervals. By

$$\overline{x}$$

we denote the finite or infinite stream that is the result of concatenating all the sequences in x. Here $\overline{x}$ denotes the sequence of data values in x without representing its timing. We use both notations also for tuples and sets of timed streams by applying them pointwise.

We work with channels that are represented by a set C. Channels are assumed to have assigned a sort from a given set of sorts S. The idea of a sort (also called a type) is used here as common in programming languages or algebraic specifications. A sort is a name for a set of data elements. We assume a function

$$sort: C \to S$$

which assigns sorts to the channels in S.

For a sort $s \in S$ we denote by $(\!|s|\!)$ the data set (often called carrier set in algebraic specifications) associated with the sort s. Channels get assigned histories represented by streams. We define the set of valuations for the sorted channels in C by

$$\vec{C} = \{x: C \to M^\aleph: \forall\, c \in C: x.c \in (\!|sort(c)|\!)^\aleph\}$$

$\vec{C}$ denotes the sets of the families of streams that can be associated with the set of channels C and that have a proper sort as associated with the channels in C.

2.2 Syntactic and Semantic Interfaces

The syntactic interface of a component is given by a set I of input channels and a set O of output channels. A graphical representation of a component F as a data flow node is given in Fig. 1.

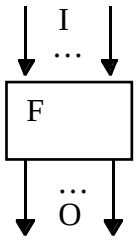

Fig. 1 Graphical Representation of a Component F with Input Channels I and Output Channels O

A function

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

maps each input history $c \in \vec{I}$ onto a set of output histories F(x). The function F models the black box behaviour of a component. For a given input history x each output history $y \in F(x)$ is an output history the component may show. Of course, since components are interactive, operationally the component gets the input history not in one step, but in many little steps of communication. And it produces its output history not in one step but in many little steps of interaction. This idea of a stepwise interaction has to be reflected by certain properties of F. Therefore we define and require a number of properties for such a function.

- A function F is called *timed* (or *causal*) if for all time points $i \in \mathbb{N}$ we have

  $$x{\downarrow}i = z{\downarrow}i \Rightarrow F(x){\downarrow}i = F(z){\downarrow}i$$

This property guarantees a consistent time flow in the behaviour modelled by F. Output at time point i does not depend on input that arrives later. The component cannot predict the future.

- A function F is called *time guarded* (or *causal*), if for all time points $i \in \mathbb{N}$ we have

  $$x{\downarrow}i = z{\downarrow}i \Rightarrow F(x){\downarrow}i+1 = F(z){\downarrow}i+1$$

Time guardedness is a strengthening of the property of being timed. Time guardedness guarantees that the component needs at least one time tick to react to input. In other word, each reaction is delayed by at least one time unit. Time guardedness is a reasonable assumption. We only have to choose the time granularity fine enough. Time guardedness has a pleasant implication for the theory, especially when dealing with feedback by fixpoint equations.

- A function F is called *realizable*, if there exists a time guarded function $f: \vec{I} \to \vec{O}$, such that for all input histories x:

  $$f.x \in F.x$$

A time guarded function f with $f.x \in F.x$ for all input histories is a deterministic behaviour that is consistent with F. Such a function f provides a deterministic strategy of interaction.

By $[\![F]\!]$ we denote the set of time guarded functions $f: \vec{I} \to \vec{O}$, such that $f.x \in F.x$ for all x.

- A function F is called *fully realizable* if for all x:

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

Full realizability guarantees that for every output there is a strategty (a deterministic consistent behaviour) that produces this output.

A timed stream processing function F models the behaviour, often also called the I/O-behaviour or the *semantic interface*, of a component. We call it therefore simply a *behaviour*.

A behaviour F is called

- *time independent*, if

$$\overline{x} = \overline{z} \Rightarrow \overline{F.x} = \overline{F.z}$$

For time independent behaviours the timing of the messages in the input streams does not influence the messages in the output streams but only their timing.

2.3 Specific Instances of Components

The notion of components that we have introduced is fairly general. It is powerful enough, in particular, to represent a number of more specific concepts of components. In the following we demonstrate the power and usefulness of our model by representing sequential object oriented components and components with synchronous message exchange.

2.3.1 Sequential Object Oriented Components

In object-orientation, a component is an object described by a class. In object-orientation with a sequential control flow, the syntactic interface of a component, is defined by the methods that can be called. A method call is considered as a message containing the following informations:

- the identifier of the callee (the object that receives the call and issues the return message),
- the identifier of the caller (the object that issues the call and receives the return message),
- the method identifier,
- the parameters.

If a method call terminates, a return message is send back to the caller that contains all the information listed above for the call, only the parameters are replaced by the results and do not consider call-by-name or call-by-reference concepts. For simplicity we assume only value parameters and value results. This is sufficient to model call-by-value-return parameter mechanisms. Let Mds be the set of method call messages and Rms be the set of return messages. Let the set M of messages be the union of both. An object (for simplicity we restrict our discussion to deterministic objects) has a behaviour

$$F: M^{\infty} \rightarrow M^{\infty}$$

with the specific properties that are captured by the following rules:

(1) assumption: the input stream for a component contains only method calls or method returns to calls issued last and not answered (stack principle) so far;

(2) commitment: every call is eventually answered by a return message provided the method terminates; no return messages are sent without the existence of a previous call.

There are two views onto a component (an object described by a class):

- the component is seen as a unit that receives method calls and issues - as far as needed - itself further method calls, awaits their result messages and, provided the call terminates, finally issues a return message; the result of the return message may depend on the restults of the subcalls,
- the component is seen as a unit that receives method calls and replies by return messages without any further interaction with the environment.

The first view is called the *open view*, the second the *closed view* onto an object. Our formal component model describes the open view that describes the object in isolation. The closed view can be derived schematically from the open view. To do this we need the interface description (open view) of all the objects together with their black box behaviour in the environment of the object that are relevant with respect to submethod calls.

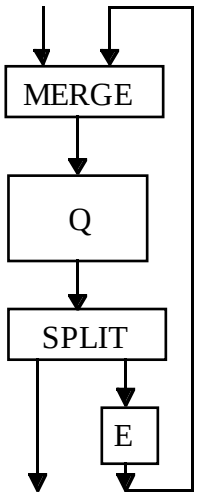

Fig. 2 The Closed View Derived from the Open View of Component Q by the local Environment E that Contains all the Objects Used by Q

In the open view we have to distinguish between the methods that are provided by a component called the *provided method* or the *in methods* and the methods that a component calls as submethod calls. These are called the *requested methods* or the *out methods*.

Note that our techniques allow us to model individual objects as well as components consisting of clusters of individual objects and also classes which can be seen as an instance of a cluster of objects. Therefore it allows us to model classes which there are seen simply as the cluster of all objects of the class.

The model introduced for object-oriented components is sufficient for a sequential execution mode. As soon as concurrency is introduced we have to deal with concurrent threads of execution. Two extreme ways to deal with concurrent threads are listed below

- Blocking: an object is blocked as long as a method call is not completed; if an object works for a reurn message it does not process any further method calls. This is more or less the monitor concept. It excludes recursive method calls and cyclic call pathes for the objects
- Nonblocking: an object may accept method calls even in cases where it is waiting for return messages for uncompleted method calls. Then threads can be free interleaved. Moreover, it may be difficult to associate return messages to the waiting threads.

The nonblocking concept includes the sequential execution as a special case. As long as concurrent threads are not enforced by the environment no problems can arise. For the concurrent case, we need additional synchronisation primitives as provided for instance in Jave.

2.3.2 Synchronous Components

In our model, components interact by asynchronous message passing. In this section, we show how to represent components with synchronous message passing in our model. Message synchronous components communicate by rendezvous communication (also called handshake). Prominent examples for approaches based on a message synchronous communication are languages like CSP, CCS, Ada, or OCCAM. The clue for synchronous components is that their choice of interaction is determined by their readiness to communicate (see [Hoare 85]).

Like for asynchronous components we work for synchronous components with a syntactic interface consisting of a set of sorted input channels I and a set of sorted output channels O. However, now each communication act consists of two actions:

input: the component may get the offer to receive a message m on its input channel $i \in I$; the result is that the component either accepts the offer and issues an acknowledgement signal @ on its output channel i' that is associated with the input channel i or it may reject the input and issue the rejection symbol ® on its associated channel i'.

output: the component sends some output on an output channel o as an offer to its environment; for this output an accept or reject signal is received on the input channel o' associated with o. In response, the component may receive either the signal @ to indicate that the message has been accepted by some sender or the signal ® to indicate that the message has not been accepted.

We model synchronous communication along the lines explained above by asynchronous communication with the help of a simple protocol of positive and negative acknowledgements. According to this scheme we define, given the sets I and O of input and output channels, respectively, additional channels on which the acknowledgements are communicated:

$I'' = I \cup O'$ where $O' = \{o': o \in O\}$

$O'' = O \cup I'$ where $I' = \{i': i \in I\}$

With every channel c in O or I we associate a sort and a carrier set and a channel c' called the acknowledgement channel:

with $i \in I$	we associate the carrier set:	$(\!(i)\!)$
	and with $i' \in I'$ the carrier set:	$\{@, ®\}$
with $o \in O$	we associate the carrier set:	$(\!(o)\!) \cup \{®\}$
	and with $o' \in O'$ the carrier set:	$\{@\}$

A typical additional assumption for components that work with message synchronous communication is that a rejected input or a rejected output request does not change the components behaviour. The component may only send another offer whenever an output offer is rejected.

With a synchronous component we associate the behavior

$F: \vec{I}^{*} \Rightarrow \wp(\vec{O}^{*})$.

Let us look at a simple example of a synchronous component. Using the notation of CSP we may describe an interactive queue with the input channels i and r and the output channel q as follows

```
Q(s) ≡ ⌜ var q: Seq D, w: D;
         q := s;
         do q ≠ ‹›: o!ft.q   then q := rest(q)
            i?w              then q := q^‹w›
         od                                    ⌟
```

Using the techniques of an algebraic specification of compontents we specify Q(s) as follows (here we use a concept of output where output is triggered by some request and is therefore demand driven).

```
Q(s) ‹ i:d = i':@ ‹ Q(s^‹d›)
Q(s) ‹ o':@ = if s = ‹›   then o:® ‹ Q(s)
                          else o:first(s) ‹ Q(rest(s))
             fi
```

We see how easily we can specify synchronous components by our asynchronous model using the introduced protocol.

3. Software Architectures

Of course, the idea of a component is to use it side by side with other components. The structuring of a software system into components and the description how they are connected and interact is called its *software architecture*. More formally, a software architecture is built by composing a number of components in a specific way into a larger composed system. We have introduced the mathematical concept of a component. Now we introduce the notion of composition.

In the first section we show how to compose components asynchronously. Then we study the composition of sequential object oriented components and of message synchronous components.

3.1 Composition Operators for Asynchronous Components

When modelling systems and system components the composition of larger systems from smaller ones is a basic structuring technique. We consider only one basic composition operator for asynchronous interaction, namely parallel composition with feedback. It comprises *sequential* composition, *parallel* composition and *feedback* as special cases.

As well known these three composition operators suffice to formulate all kinds of networks of reactive information processing components, provided we have simple components available for permuting and copying input and output lines.

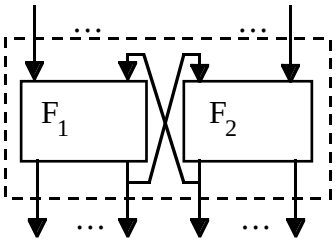

Fig. 3 Parallel Composition with Feedback

To define parallel composition with feedback we first introduce an operation that allows to form tuples (of streams) out of two tuples (of streams). Given two disjoint sets of channels C_1 and C_2 and two valuations for them

$$x \in \vec{C}_1,\ y \in \vec{C}_2$$

we construct a valuation

$$x \oplus y \in C_1 \vec{\cup} C_2$$

as follows:

$$(x \oplus y).c = x.c \text{ if } x \in \vec{C}_1 \text{ and } y.c \text{ if } c \in \vec{C}_2$$

Recall that we assume that the sets of channels C_1 and C_2 are disjoint.

Given two behaviours (where $O_1 \cap O_2 = \emptyset$, $O_1 \cap I_1 = \emptyset$, $O_2 \cap I_2 = \emptyset$)

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

we define the parallel composition with feedback by the function

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

where

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

specified by (with the valuation $y \in \vec{C}$ where $C = O_1 \cup O_2 \cup I_1 \cup I_2$)

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

Here by y|O we denote the restriction of the valuation mapping $y \in \vec{C}$ to the channel set $O \subseteq C$. Note that according to our semantic model the component $F_1 \otimes F_2$ is realizable provided F_1 and F_2 are realizable.

3.2 Refinement

Refinement is the fundamental concept for the development of components. We describe two basic forms of refinement: *property refinement* and *interaction refinement*. Property refinement allows us to replace a behaviour by one with additional properties, in other words, by one fulfilling more requirements. A behaviour

$$F_1: \vec{I} \rightarrow \wp(\vec{O})$$

is refined by a behaviour

$$F_2: \vec{I} \rightarrow \wp(\vec{O})$$

if

$$F_2 \subseteq F_1$$

Property refinement does not allow us to change the syntactic interface of a component. This can be done by interaction refinement, however.

By interaction refinement we may change the names and numbers of input and output channels of a system but still relate the behaviours if a formal way. A *communication history refinement* requires timed functions

$$A: \vec{I} \rightarrow \wp(\vec{O})$$
$$P: \vec{O} \rightarrow \wp(\vec{I})$$

where

$$P ; A = Id$$

Let Id denote the identity relation

$$Id.x = \{x\}$$

and $F_1 ; F_2$ denote the sequential composition of the two components F_1 and F_2 defined by

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

Fig. 4 gives the “commuting diagram“ of history refinement.

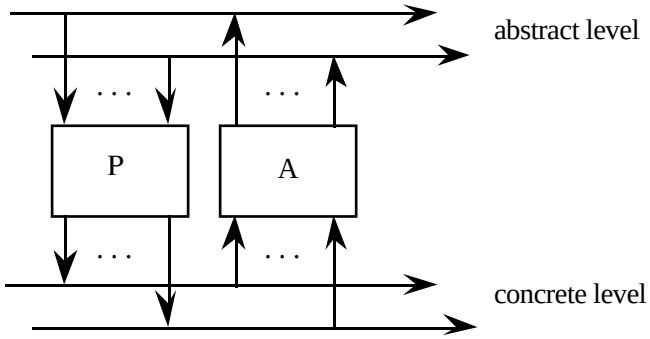

Fig. 4 Communication History Refinement

Based on the idea of a history refinement we introduce the idea of an interaction refinement for components.

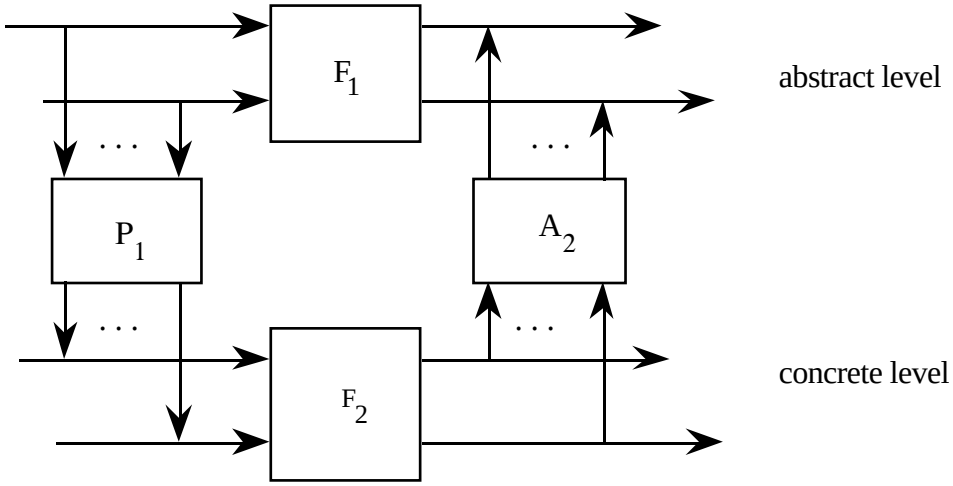

Fig. 5 Commuting Diagram of Interaction Refinement (*U-simulation*)

Given two communication history refinements

$A_1: \vec{I}_1 \rightarrow \wp(\vec{I}_2)$ $\quad P_1: \vec{I}_2 \rightarrow \wp(\vec{I}_1)$

$A_2: \vec{O}_1 \rightarrow \wp(\vec{O}_2)$ $\quad P_2: \vec{O}_2 \rightarrow \wp(\vec{O}_1)$

we call the behaviour

$F_2: \vec{I}_2 \rightarrow \wp(\vec{O}_2)$

an *interaction refinement* of the behaviour

$F_1: \vec{I}_1 \rightarrow \wp(\vec{O}_1)$

if one of the following four proposition holds

$P_1 ; F_2 ; A_2 \subseteq F_1$ *U-simulation*

$P_1 ; F_2 \subseteq F_1 ; P_2$ *downward simulation*

$F_2 ; A_2 \subseteq A_1 ; F_1$ *upward simulation*

$F_2 \subseteq A_1 ; F_1 ; P_2$ *U^{-1}-simulation*

Note that U^{-1}-simulation is the strongest condition from which all others follow. Which of these definitions are most appropriate depends very much on the method of refinement we are interested in.

3.3 Composition of Sequential Object-Oriented Components

In this section, we adapt our notion of composition to components representing objects and the way objects interact. We do this with the help of a scheduler component. Let a set K\{e} of identifiers for sequential components called objects be given. The identifier $e \in K$ is used for the channel that associates the system of objects with the environment. With every object identifier $k \in K\{e\}$ we associate the behaviour of an object given by

$F_k: \{\vec{i}_k\} \rightarrow \wp(\{\vec{o}_k\})$

K is the set of object identifiers. By i_k we denote the input channel and by o_k the output channel of the object k. The set K of identifier includes the special identifier e that denotes the environment.

We assume that for each call m and each return message m there exists at most one recipient $rci(m) \in K$.

We define the composition of the objects with the help of a scheduling function for the set of components in K\{e} that has one input and output channels for each component in K

$sch[K] : \vec{O} \rightarrow \wp(\vec{I})$ where $I = \{i_k: k \in K\}$, $O = \{o_k: k \in K\}$

The scheduler is used to compose the components as shown in Fig. 6. We do not give a formal specification of the scheduler here. Such a description is quite straightforward. It can be found in [Klein, Rumpe, Broy 96].

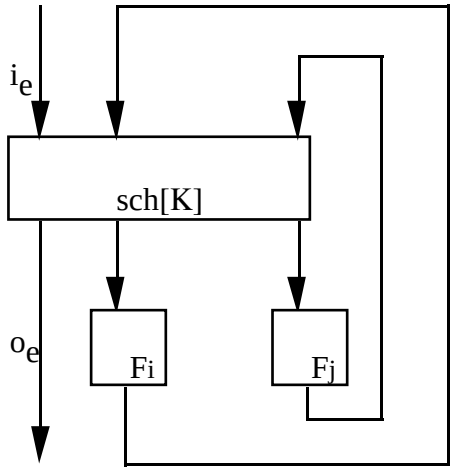

Fig. 6 The System with the Scheduler and the Components

As long as the system runs in a sequential mode, the scheduler is deterministic. Then only one component is active at a time.

Whenever the scheduler receives a call from its global environment on channel i_e, it forwards it to the addressed component. The scheduler awaits a response of the component (which is either a return message or a call) and forwards a call to the addressed component or to the environment and a return message to the environment. The scheduler together with the subcomponents forms again a (composed) component.

3.4 Composition of Message-Synchronous Components

The described technique for modelling sequential object oriented components can be carried over to concurrent messages-synchronous components. Given n components we use a parallel composition by joining all input channels with the same name and do the same for all output channels. Whenever a message is offered on an input channel of the system this offer is forwarded to one of the components having this channel as an input channel. If it is accepted by the component then the accept message @ is send to the environment, otherwise all components with that input channel are tried out in a nondeterministic order until the message gets accepted; only if none of the components accepts the message it is rejected by the composed system. Requests for output are handled in analogy.

To allow communication between components we introduce a feedback between input and output channels handled by a scheduler. To do this we connect the actual message channels as well as the reject end accept signal channels controlled by the scheduler.

3.5 Interoperability

By interoperability we mean that different types of components such as asynchronous, message synchronous or sequential object oriented components can interact within a system architecture. We have introduced these three types of components. Since all are described in the uniform framework of asynchronous components they can be easily composed by the basic operators. Since all three concepts form a concept of modular components on their own, we can build heterogeneous systems. Parts of these systems can be structured as sequential object-oriented systems, other parts as message-synchronous systems and other parts of them may be built using asynchronous components. They are connected with the help of schedulers.

Nevertheless there is a proper precise notion of behaviour for the complete system. Sometimes, it may be worthwhile to define gate-way components (such as our schedulers) to connect parts of systems written in different styles. We have the following types of connectors

- asynchronous connectors,
- sequential remote procedure call connectors (method call channels),
- synchronous message passing connectors (handshake connection).

We get hybrid components that way with asynchronous, synchronous and sequential input and output connectors. When composing a system the first two kinds of connectors can simply be connected if the types and sorts coincide. The third requires a more sophisticated notion of composition.

Given a set O of synchronous output channels of a system and a set I of synchronous input channels of a system all having the same sort we may connect them by handshake composition with the help of a component (let $O \cap I = \varnothing$)

HSC[O, I]

This component has $O \cup I'$ as input channels and $I \cup O'$ as output channels. It connects the channels as shown in Fig. 7.

The channel connection component can be seen as a little interaction protocol (a specific scheduler) that whenever it receives an offer on one of its channels in O it forwards it to one of its output channels $i \in I$. Then it awaits the reply on i' which is either @ or ®. If the reply is @ it forwards it on channel o', otherwise it tries another channel in I. Only if all channels in I lead to rejections ® a rejection is sent on channel o'.

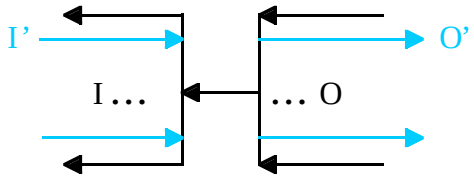

Fig. 7 Handshake Connector HSC[O, I]

Note that in the case of message synchronous communication as well as in the case of procedure or method calls we expect the validity of certain constraints on the behaviour of the components that

reflect the protocol of the interaction. The general assynchronous components that communicate with synchronous or sequential components have, of course, to respect these constraints.

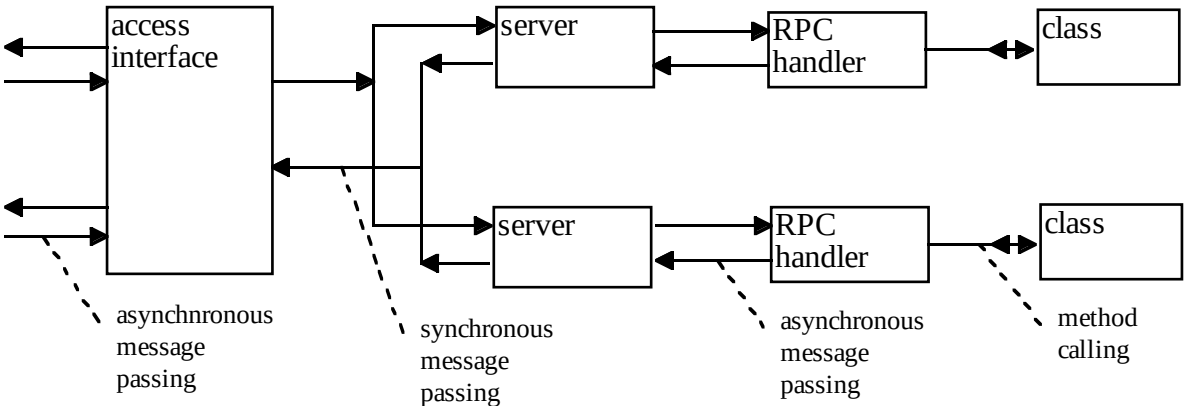

Fig. 8 Heterogeneous Architecture of Services

3.6 Heterogeneous Software Architectures

A software architecture consists of a family of components and a description of the way they are connected and interact. Since we have a proper notion of a component and of composition we can form large composed systems in a hierarchical style. Exploiting the interoperability we can form *heterogeneous software architectures.* These are architectures build of different types of components.

We may design an architecture for the asynchronous access to a family of synchronous communicating servers that have access to classes of data structures. This is sketched by Fig. 8.
We do not have enough space to handle all the details, however, it should be clear that our approach is general enough to allow us to specify the behaviour of the quite different types of components and to compose them into a system as shown in Fig. 8.

4. Incremental Development of Components

In the incremental development of a component we develop the component in a sequence of steps. In each step, we work out a new version of the component offering more services. From a semantical and a methodological point of view it is most interesting to understand the relationship between the old version of a component and its incremental successor.

In this section, we describe a software development process model for the incremental development of software systems. It supports the development of software systems, structured in components, by a number of incremental development cycles. In addition, it allows us the incremental further development of existing software systems.

4.1 The Need for Incremental Development

According to a study by the European Software Institute the most critical and unsolved problem in software development process is the problem how to achieve an adequate requirement specification. The reason for this is quite obvious. Software systems show a complex behaviour difficult to understand and assess even for an application expert. It is very difficult to imagine such a behaviour in advance only considering technical descriptions. Therefore the requirement capture process often leads to requirements which are not fully understood by the application expert. As a result, software systems are built that do not cover the users' expectations and contain features that are rarely needed.

In this situation basically only two complementing ways of proceeding can help. They can be characterized as follows:

- A careful requirement capture process is needed in which the application experts and the users can participate. This means that techniques have to be used that on one hand are suggestive such that they give the user a good understanding about the behaviour of the system under development. On the other hand the techniques have to be sufficiently precise to avoid misunderstandings also among the software engineers involved.
- A prototype of the system has to be implemented that can be used to demonstrate the behaviour of the system to the users such that they can get a feeling for its properties. Based on a careful analysis of the prototype and a carefully carried out usability study based on experiments with the users a revised specification can be worked out where then again an implementation is constructed[1)] .

Such a cycle of implementing a version of the system, analysing it, finding out that some of the properties are not as expected and constructing a revised version is a typical way of proceeding in software development today in practice, even in cases where officially a waterfall development model is used. In fact, today many companies are using incremental techniques although they often are not supported by their official design handbooks.

In the case of incremental development cycles, it is advisable not to implement in a first step a system which shows a very large and overly complex functionality. It is much better to concentrate on some essential or critical behaviours of the system, to do a careful usability analysis for that, and then enrich the functionality of the system gradually in several development steps. In this case, we speak of incremental development of the system.

The idea of incremental development and of prototyping for requirement capture and validation is of course not new. It has been and is suggested by many software engineers based of the experience that the waterfall model often leads to software systems that cannot be used or can hardly be used since they do not reflect the real needs of the users. This leads to the effect that only a small portion of the functionality of a system is actually used by a user. To avoid this problem a new version of the system has to be developed which more closely matches the users' needs. In

[1)] Of course, a prototype is useful in many ways, in addition.

practice, often an incremental development is practised anyhow although people pretend to work with a software development process model that is close to the waterfall model such as the German V-Modell [V-Modell 92].

In the meanwhile there exist many suggestions especially in the area of object oriented programming that advocate an incremental development model. Typical examples are the spiral model by Berry Boehm (see [Boehm 88]) or the macro cycles in the development as suggested by Booch (see [Booch 91]).

In the following we suggest a presentation of a development model for incremental development which captures both the concept of revisions of the features of a system and that of incremental development. Since we are interested to get a very clean terminology and sharply defined notions we work with a completely formal model of a system and of a component as introduced in the previous sections. Thus we explain a model of incremental development which can both be used in the development process of a new system, where the system is development in several incremental cycles and that can also developed for the incremental development of a system that is already in use for a while and now has to be further developed in a new version. We give a precise definition of that development process and especially study the relationship of the involved systems and system components. So we define an idealised notion of incremental refinement of a component and discuss the notion of a revised component.

4.2 Incremental Software Development

Incremental software development is not radically different to software development using the conventional waterfall model. This is especially true for the development steps involved and the produced documents. Only the organisation of the development process differs.

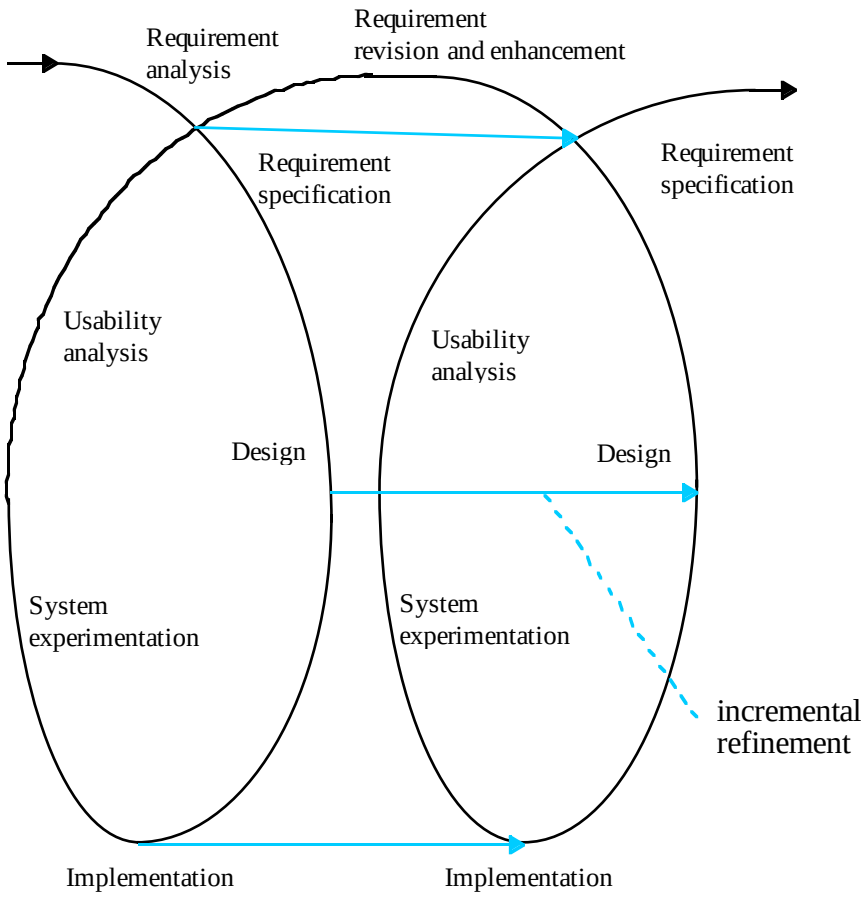

Fig. 9 Two Cycles in the Incremental Development Process

Incremental software development should go through the classical phases and steps of software development including

- analysis of the problem,
- requirement engineering and requirement specification,
- design, architecture and decomposition into components,
- component specification,
- implementation and integration,
- testing, verification, validation.

Sometimes under the catchword "rapid prototyping" it is recommended that a quick dirty development step should produce a first prototype. This is helpful, however, only in situations and applications where all experience and analytical help is missing and therefore an experimental prototype is the only way to gain insights. In all other cases, a more careful proceeding is preferable, since also the construction of a prototype is expensive and the closer a first prototype comes to the users' satisfaction the sooner the development will come to a good result. Hence, it is certainly more cost effective to analyse and capture also the requirements for a prototype carefully.

If we organise the development process in such a careful way we make sure that the documents produced in the i-th development cycle can be used as a basis for the (i+1)-th development cycle. We are, in particular, interested in the relationship between these documents.

4.3 The Documents in the Development Process

In the incremental development process we work with the following views and the corresponding mathematical models. We concentrate here on the essence of the documents to produce and not on the way they are represented:

- the *data view* given by a signature Σ and a set of Σ-algebras,
- the *interface view* (also called black box view) for the overall system given by a pair (I, O) of input and output channels with their sorts and a behaviour
  $F: \vec{I} \to \wp(\vec{O})$,
- the glass box view for the component F given by
  - the *state box view* in the form of a state space State and a state transition function
    $$\delta: \text{State} \times (I \to M^*) \to \wp(\text{State} \times (O \to M^*))$$
    or / and
  - a *structural view* (distributed system view) in the form of a data flow net and its subcomponents. For these subcomponents again black box and glass box views are to be provided.
  - the *process view* describing a set of processes for illustrating the interaction of the components with respect to their glass box and/or their black box view.

For all these views we assume refinement relations as we have introduced them above such as

- behaviour refinement,
- data representation and interface refinement.

We do not define these refinement explicitly for all views but just for the interface view. However, also for the other views such refinement concepts are available.

4.4 Refinement in the Incremental Development Process

In the incremental development process we work with *incremental refinement*. This means that we stepwise enhance the functionality of the components. This is an idealised view, of course, since often in a refinement step we carry out a revision. This means that we change parts of the behaviour in an arbitrary way only justified by new knowledge about the users' needs.

In the idealistic view the new behaviour is an incremental refinement of the old one. We give a formal definition for incremental refinement now. Let all definitions be as in the end of the previous chapter. A component with a behavior F_2 is an incremental refinement of a component with a behavior described by F_1 if:

1. Every input history to F_1 can be translated to an input history of F_2 such that the produced output history of F_2 can be translated into an output history F_1.
2. Every input history for the component F_2 can be translated into an input history for the component F_1 such that the produced output history of F_1 can be schematically translated into an output history of F_2.

More formally component F_2 is called an *incremental refinement* of a component F_1 if every behaviour of F_1 can be represented by a behaviour of F_2 and if every behaviour of F_2 can be interpreted as a behaviour of F_1. Given two communication history refinements

$$A_1\colon \vec{I}_1 \to \wp(\vec{I}_2) \qquad P_1\colon \vec{I}_2 \to \wp(\vec{I}_1)$$
$$A_2\colon \vec{O}_1 \to \wp(\vec{O}_2) \qquad P_2\colon \vec{O}_2 \to \wp(\vec{O}_1)$$

we call the behaviour

$$F_2\colon \vec{I}_2 \to \wp(\vec{O}_2)$$

an incremental refinement of the behaviour

$$F_1\colon \vec{I}_1 \to \wp(\vec{O}_1)$$

if the propoposition

$$F_2 \subseteq A_1 ; F_1 ; P_2 \qquad \textit{U}^{-1}\textit{-simulation}$$

holds. Note that by this definition F_2 may provide more services than F_1, and that the definition also implies

$$P_1 ; F_2 ; A_2 \subseteq F_1. \qquad \textit{U-simulation}$$

This means we can use F_2 instead of F_1 (provided we implement P_1 and A_2). A very simple instance of communication history refinement are filters for the abstraction function A_1 and that filter out the messages not considered for F_1.

5. Conclusions

Component-based software development cannot be done properly without a clear semantic notion of a component and its interface. The specification of interactive system components has to be done in a time/space frame. A specification should indicate which events (communication actions) can take place where, when and how they are causally related. Such specification techniques are an important prerequisite for the development of safety critical systems.

Modelling information processing systems appropriately is basically a matter of choosing the adequate abstractions in terms of the corresponding mathematical models. Giving operational

models that contain all the technical and computational details of interactive nondeterministic computations is relatively simple. However, for system and software engineering purposes operational models are not very helpful. Only, if we manage to find good abstractions it is possible reach a tractable basis for system specifications.

Abstraction means forgetting information. Of course, we may forget only information that is not needed. Which information is needed does not only depend upon the explicit concept of observation, but also upon the considered forms of the composition of systems.

Finding appropriate abstractions for operational models of distributed systems is a difficult but nevertheless important task. Good abstract nonoperational models are the basis of a discipline of system development.

A flexible scientific foundation for a component-based method for software development needs proper and flexible concepts of:

- a syntactic and semantic interface of a component,
- composition,
- incremental refinement and development,
- interoperability for building heterogeneous systems,
- software architecture.

These ingredients are needed for a scientific foundation and proper methodology of the attractive idea of component-based software development and componentware. We have shown how the mathematical foundation of such a scientific methodology of componentware may look like.

Acknowledgement

I am grateful to Ketil Stølen and Cornel Klein for a number of discussions that were helpful to clarify the basic concepts.

References

[Boehm 88]
B.W. Boehm: The Spiral Model of Software Development and Enhancement. IEEE Computer 21:5, 61-72, 1988

[Booch 91]
G. Booch: Object Oriented Design with Applications. The Benjamin Comings Publishing Company 1991

[Broy 83]
M. Broy: Applicative real time programming. In: Information Processing 83, IFIP World Congress, Paris 1983, North Holland Publ. Company 1983, 259-264

[Broy 85]
M. Broy: Specification and top down design of distributed systems. In: H. Ehrig et al. (eds.): Formal Methods and Software Development. Lecture Notes in Computer Science 186, Springer 1985, 4-28, Revised version in JCSS 34:2/3, 1987, 236-264

[Broy 86]
M. Broy: A theory for nondeterminism, parallelism, communication and concurrency. Habilitation, Fakultät für Mathematik und Informatik der Technischen Universität München, 1982, Revised version in: Theoretical Computer Science 45 (1986) 1-61

[Broy 87a]
M. Broy: Semantics of finite or infinite networks of communicating agents. Distributed Computing 2 (1987), 13-31

[Broy 87b]
M. Broy: Predicative specification for functional programs describing communicating networks. Information Processing Letters 25 (1987) 93-101

[Broy 93]
M. Broy: Functional Specification of Time Sensitive Communicating Systems. REX Workshop. ACM Transactions on Software Engineering and Methodology 2:1, Januar 1993, 1-46

[Broy, Stølen 94]
M. Broy, K. Stølen: Specification and Refinement of Finite Dataflow Networks – a Relational Approach. In: Langmaack, H. and de Roever, W.-P. and Vytopil, J. (eds): Proc. FTRTFT'94, Lecture Notes in Computer Science 863, 1994, 247-267

[Hoare 85]
C.A.R. Hoare: Communicating Sequential Processes. Prentice Hall, 1985

[Klein, Rumpe, Broy 96]
C. Klein, B. Rumpe, M. Broy :A stream-based mathematical model for distributed information processing systems - SysLab system model. In: E. Naijm, J.-B. Stefani (ed.): FMOODS'96 - Formal Methods for Open Object-based Distributed Systems. ENST France Telecom 1996, 323-338

[Kahn, MacQueen 77]
G. Kahn, D. MacQueen: Coroutines and networks of processes, Proc. IFIP World Congress 1977, 993-998

[Lynch, Stark 89]
N. Lynch, E. Stark: A proof of the Kahn principle for input/output automata. Information and Computation 82, 1989, 81-92

[Lynch, Tuttle 87]
N. A. Lynch, M. R. Tuttle: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing, 1987

[Park 80]
D. Park: On the semantics of fair parallelism. In: D. Björner (ed.): Abstract Software Specification. Lecture Notes in Computer Science 86, Berlin-Heidelberg-New York: Springer 1980, 504-526

[Park 83]
D. Park: The "Fairness" Problem and Nondeterministic Computing Networks. Proc. 4th Foundations of Computer Science, Mathematical Centre Tracts 159, Mathematisch Centrum Amsterdam, (1983) 133-161

[V-Modell 92]
Das Bundesministerium des Inneren: Planung und Durchführung von IT-Vorhaben - Vorgehensmodell. August 1992