Compositional Refinement of Interactive Systems Modelled by Relations*

Manfred Broy

Fakultät für Informatik, Technische Universität München, D-80290 München
e-mail: broy@informatik.tu-muenchen.de

Abstract. We introduce a mathematical model of components that can be used for the description of both hardware and software units forming distributed interactive systems. As part of a distributed system a component interacts with its environment by exchanging messages in a time frame. The interaction is performed by accepting input and by producing output messages on named channels. We describe forms of *composition* and three forms of *refinement*, namely *property refinement*, *glass box refinement*, and *interaction refinement*. Finally, we prove the compositionality of the mathematical model with respect to the introduced refinement relations.

1. Introduction

For a discipline of system development firmly based on a scientific theory we need a clear notion of components and ways to manipulate and to compose them. In this paper, we introduce a mathematical model of a component with the following characteristics:

- A component is *interactive*.
- It is connected with its environments by named and typed *channels*.
- It receives *input messages* from its environment on its *input* channels and generates *output messages* to its environment on its *output* channels.
- A component can be *nondeterministic*. This means that for a given input history there may exist several output histories that the component may produce.
- The interaction between the component and its environment takes place in a *global time* frame.

Throughout this paper we work with discrete time. Discrete time is a sufficient model for most of the typical applications. For an extension of our model to continuous time see [16].

Based on the ideas of an interactive component we can define forms of composition. We basically introduce only one form of composition, *namely parallel composition with feedback*. This form of composition allows us to model *concurrent*

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

execution and *interaction*. We will show that other forms of composition can be introduced as special cases of parallel composition with feedback.

For the systematic stepwise development of components we introduce the concept of *refinement*. By refinement we can develop a given component in a stepwise manner. We study three refinement relations namely *property refinement*, *glass box refinement*, and *interaction refinement*. We claim that these notions of refinement are all what we need for a systematic top down system development.

Finally, we prove that our approach is *compositional*. This means that a refinement step for a composed system is obtained by refinement steps for its components. As a consequence, global reasoning can be structured into local reasoning on the components. Compositionality relates to *modularity* in systems engineering.

The new contribution of this paper the relational version of the stream processing approach as developed at the Technische Universität München (under the keyword FOCUS, see [11], [12]). Moreover, the paper aims at a survey over this approach.

We begin with the informal introduction of the concept of interactive components. This concept is based on communication histories called streams that are introduced in section 3. Then a mathematical notion of a component is introduced in section 4 and illustrated by simple examples. Section 5 treats operators for composing components to distributed systems. In section 6 we introduce three notions of refinements to develop systems and show the compositionality of these notions. All concepts are illustrated by simple examples.

2. Central Notion: Component

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

The notion of component is essential in systems engineering and software engineering. Especially in software engineering a lot of work is devoted to the concept of *software architecture* and to the idea of *componentware*. Componentware is a catchword in software engineering (see [15]) for a development method where software systems are composed from given components such that main parts of the systems do not have to be reprogrammed every time again but can be obtained by a new configuration of existing software solutions. A key for this approach are well designed *software architectures*. Software architectures mainly can be described as specifically structured systems, composed of components. In both cases a clean and clear concept of a component is needed.

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

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

According to this definition we work with the idea of a component which encapsulates a local state or a distributed architecture. We provide a logical way to write a specification of component services. We will relate these notions to glass box views, to the derived black box views, and to component specifications.

A powerful semantic concept of a component interface is an essential ingredient for the following key issues in system development:

- modular program construction,
- software architecture,
- systems engineering.

In the following we introduce a mathematical concept of a component. We show how basic notions of development such as specification and refinement can be based on this concept.

3. Streams

A *stream* is a finite or infinite sequence of messages or of actions. Streams are used to represent communication histories or histories of activities. Let M be a given set of messages.

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

We use the following notation:

M^*	denotes the finite sequences over M with the *empty* sequence ‹›,
M^∞	denotes the infinite sequences over M.

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

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

we denote the set of timed streams. The k-th sequence in a timed stream represents the sequence of messages exchanged on the channel in the k-th time interval.

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

In general, in a system several communication streams occur. Therefore we work with *channels* to identify the individual communication streams. Hence, in our approach, a channel is nothing than an identifier in a system that is related to a stream in every execution of the system.

Throughout this paper we work with some simple forms of notation for streams that are listed in the following. We use the following notation for timed streams x:

$z^\frown x$	concatenation of a sequence z to a stream x,
$x\downarrow i$	sequence of the first i sequences in the stream x,
$S \copyright x$	stream obtained from x by deleting all messages that are not elements of the set S,
$\overline{x}$	finite or infinite stream that is the result of concatenating all sequences in x.

We can also consider timed streams of states to model the traces of state-based system models. In the following, we restrict ourselves to message passing systems, however.

4. Syntactic and Semantic Interfaces of Components

In this section we introduce a mathematical notion of components. We work with typed channels. Let a set S of *sorts* or *types* be given. By

$$C$$

we denote the set of typed channels. We assume that we have given a type assignment for the channels:

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

Given a set C of typed channels we now can introduce what we call a *channel valuation* (let M be the set of all messages, by $(\!|s|\!)$ we denote for a type its set of elements):

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

A channel valuation $x \in \vec{C}$ associates a stream of elements of type type(c) with each channel $c \in C$.

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 component:

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

In addition to the syntactic interface we need a concept for describing the *behaviour* of a component. We work with a very simple and straightforward notion of a behaviour. A behaviour is a *relation between input histories and output histories*. Input histories are represented by valuations of the input channels and output histories are represented by the valuations of output channels. To express that a component maps input onto output we do not describe a component by a relation but by a set valued function. Therefore we represent the semantic interface of a component F as follows:

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

Given $x \in \vec{I}$, by F.x we denote the set of all output histories a component with behaviour F may produce on the input x.

Of course, a set valued function, as well known, is isomorphic to a relational definition. We call the function F an *I/O-function*.

Using logical means, such a function can be described by a formula relating input channels with output channels. Syntactically therefore such a formula uses channels as identifiers for streams.

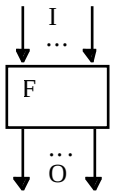

Fig 1 Graphical Representation of a Component F with Input Channels I and Output Channels O

A specification of a component defines:

- its syntactic interface,
- its behaviour by a specifying formula relating input and output channel valuations.

This way we obtain a specification technique that gives us a very powerful way to describe components.

Example. As examples of components we specify a merge component MRG and a fork component FRK as follows:

MRG

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

Here let T1, T2, T3 be types (in our case we can see types simply as sets) where T1 and T2 are assumed to be disjoint and T3 is the union of T1 and T2.

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 as specified here is fair. Every input is finally

processed. ❑

We use the following notation for a component F to refer to the constituents of its syntactic interface:

In(F)	the set of input channels I,
Out(F)	the set of output channels O.

I/O-functions can be classified by the following notions. These notions can be either added as properties to specifications explicitly or proved for certain specifications.

An I/O-function F: $\vec{I} \to \wp(\vec{O})$ is called

- *properly timed*, if for all time points $i \in \mathbb{N}$ we have

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

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

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

- *partial*, if $F(x) = \varnothing$ for some $x \in \vec{I}$.

- *realisable*, if for a time guarded function $f: \vec{I} \to \vec{O}$, for all x: $f.x \in F.x$.

- *fully realisable*, if for all x: $F.x = \{f.x: f \in [\![F]\!]\}$
  Here $[\![F]\!]$ denotes the set of time guarded functions $f: \vec{I} \to \vec{O}$, where $f.x \in F.x$ for all x.

- *time independent* (see [9]), if $\bar{x} = \bar{z} \Rightarrow \overline{F.x} = \overline{F.z}$

It is easy to show that both MRG and FRK are time independent. If we add time guardedness as a requirement then both are fully realisable.

We do not require that an I/O-function described by a specification has all the properties introduced above. We are much more liberal. We may add such properties to specifications freely whenever appropriate and therefore deal with all kinds of specifications of I/O-functions that do not have these properties.

A special case of I/O-functions are partial functions which are functions that for certain input histories may have an empty set of output histories. An extreme case is a function that maps every input history onto an empty set. Such functions are not very interesting when used for modelling the requirements for the implementation, since an implementation shows at least one output for each input. However, partial functions may be interesting as intermediate steps in the specification process, since based on these functions we can construct other functions that are more interesting for composition and implementation.

5. Composition Operators

In this section we introduce a notion of *composition* for components. We prefer to introduce a very general form and later define a number of special cases for it.

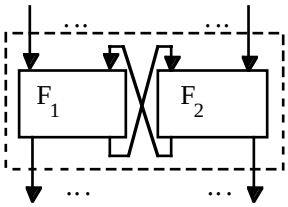

Fig 2 Parallel Composition with Feedback

Given two disjoint sets of channels C_1 and C_2 we define a join operation $\oplus$ for the valuations $x \in \vec{C}_1$, $y \in \vec{C}_2$ by the following equations:

$(x \oplus y).c = x.c$ if $c \in C_1$ and
$(x \oplus y).c = y.c$ if $c \in C_2$

Given I/O-functions with disjoint sets of input channels (where $O_1 \cap O_2 = \varnothing$)

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

we define the parallel composition with feedback by the I/O-function

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

where $I = (I_1 \cup I_2) \backslash (O_1 \cup O_2)$, $O = (O_1 \cup O_2) \backslash (I_1 \cup I_2)$. The resulting function is specified by the equation (here $y \in \vec{C}$ where $C = I_1 \cup I_2 \cup O_1 \cup O_2$):

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

By x|C we denote the restriction of the valuation x to the channels in C.

For this form of composition we can prove the following facts by rather simple straighforward proofs:

(1) if the F_i are *time guarded* for i = 1, 2, so is $F_1 \otimes F_2$,

(2) if the F_i are *realisable* for i = 1, 2, so is $F_1 \otimes F_2$,

(3) if the F_i are *fully realisable* for i = 1, 2, so is $F_1 \otimes F_2$,

(4) if the F_i are *time independent* for i = 1, 2, so is $F_1 \otimes F_2$.

If the F_i are total and properly timed for i = 1, 2, we cannot conclude that $F_1 \otimes F_2$ is total. This shows that the composition works only in a modular way for well-chosen

subclasses of specifications.

Further forms of composition that can be defined (we do not give formal definitions for them, since these are quite straightforward):

- feedback without hiding: μF
  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 $y \in \vec{C}$ where $C = I \cup O$):

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

- parallel composition: $F_1 \parallel F_2$

  if $(I_1 \cup I_2) \cap (O_1 \cup O_2) = \varnothing$ we have $F_1 \parallel F_2 = F_1 \otimes F_2$

- logical connectors: $F_1 \cup F_2$

- hiding: $F \setminus \{c\}$

- renaming of channels: $F[c/c']$

Given a component specification S we define by S[c/c'] the renaming of the channel c in S to c'. Finally, we also can work with input and output operations on components:

- input transition: $F \leftarrow c:m$

- output transition: $c:m \leftarrow F$

For a careful treatment of the last three operators see [14]. All the forms of compositions can be defined formally for our concept of components and in principle reduced to parallel composition with feedback.

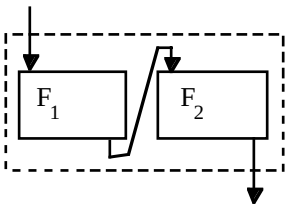

Fig 3 Sequential Composition as a Special Case of Composition

Sequential composition of the components F_1 and F_2 is denoted by

$$F_1 ; F_2$$

In the special case where $O_1 = I_2 = (O_1 \cup O_2) \cap (I_1 \cup I_2)$ we can reduce sequential composition to parallel composition with feedback along the lines illustrated in Fig. 3 as follows:

$$F_1 ; F_2 = F_1 \otimes F_2$$

A simple example of sequential composition (where $O_1 = I_2$) is the composed component MRG;FRK as well as FRK[x/x´, y/y´];MRG.

6. Refinement - the Basic Concept for System Development

Refinement relations (see [13]) are the key to formalize development steps (see [8]) and the development process. We work with the following basic ideas of refinement:

- *property refinement* - enhancing requirements - allows us to add properties to a specification,
- *glass box refinement* - designing implementations - allows us to decompose a component into a distributed system or to give a state transition description for a component specification,
- *interaction refinement* - relating levels of abstraction - allows us to change the granularity of the interaction, the number and types of the channels of a component (see [10]).

We claim that these notions of refinement are sufficient to describe all the steps needed in the idealistic view of a strict top down hierarchical system development. The three refinement concepts mentioned above are explained in detail in the following.

6.1 Property refinement

Property refinement allows us to replace an I/O-function by one with additional properties. A behaviour

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

is refined by a behaviour

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

if

$$\hat{F} \subseteq F$$

This stands for the proposition

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

A property refinement is a basic refinement step as it is needed in requirements engineering. In the process of requirement engineering, typically the overall services of a system are specified. This, in general, is done by requiring more and more

sophisticated properties for components until a desired behaviour is specified.

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

TM

in x: T1, y: T2,
out x´: T1, y´: T2,
$\forall$ m $\in$ T1: {m}© $\overline{x}$´ = {m}© $\overline{x}$
$\forall$ m $\in$ T2: {m}© $\overline{y}$´ = {m}© $\overline{y}$

We want to relate this specification to the simple specification of the time independent identity TII that reads as follows:

TII

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

Given these two specifications we immediately obtain that TII is a property refinement of TM.

$$TII \subseteq TIM$$

This relation is straightforward to prove (see below). ❑

The verification conditions for property refinement are obtained as follows. For given specifications S_1 and S_2 with specifying formulas E_1 and E_2, the specifications S_2 is a property refinement of S_1 if the syntactic interfaces of S_1 and S_2 coincide and if for the formulas E_1 and E_2 we have

$$E_1 \Leftarrow E_2$$

In our example the verification condition is easily obtained and reads as follows:

$$(\forall\ m \in T1: \{m\}© \overline{x}' = \{m\}© \overline{x}) \Leftarrow \overline{x}' = \overline{x}$$

$$\wedge\ \ (\forall\ m \in T2: \{m\}© \overline{y}' = \{m\}© \overline{y}) \Leftarrow \overline{y}' = \overline{y}$$

The proof of this condition is trivial.

Property refinement can also be used to relate composed components to given components. For instance, we obtain the refinement relation.

$$(MRG\ ;\ FRK) \subseteq TII$$

Again the proof is quite straightforward.

Property refinement is used in requirements engineering. It is also used in the

design process where decisions are taken that introduce further properties for the components.

6.2 Compositionality of Property Refinement

In our case, the compositionality of property refinement is simple. This is a consequence of the simple definition of composition. The rule of compositional property refinement reads as follows:

$$\frac{\hat{F}_1 \subseteq F_1 \qquad \hat{F}_2 \subseteq F_2}{\hat{F}_1 \otimes \hat{F}_2 \subseteq F_1 \otimes F_2}$$

The proof of the soundness of this rule is straightforward by the monotonicity of the operarator $\otimes$ with respect to set inclusion.

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 by

MRG1

in x: T1, y: T2, **out** z: T3,
z = ‹›^f(x, y) **where** f(‹s›^x, ‹t›^y) = ‹s^t›^f(x, y)

Note that this merge component MRG1 is deterministic and time dependent. According to our rule of compositionality and transitivity of refinement, it is sufficient to prove

$$\text{MRG1} \subseteq \text{MRG}$$

to conclude

$$\text{MRG1;FRK} \subseteq \text{MRG;FRK}$$

and by transitivity of the refinement relation

$$\text{MRG1;FRK} \subseteq \text{TII}$$

This shows how local refinement steps and their proofs are schematically extended to global proofs. ❑

The usage of the composition operator and the relation of property refinement leads to a design calculus for requirements engineering. 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 that we need and use in the design phase. In the design phase we typically decompose a system with a specified black box behaviour into a distributed system architecture or we represent this behaviour by a state transition machine. By this decomposition we are fixing the basic components of a system.

These components have to be specified and we have to prove that their composition leads to a system with the required functionality. In other words, a glass box refinement is a special case of a property refinement of the form

$$F_1 \otimes F_2 \otimes ... \otimes F_n \subseteq F \qquad \textit{design of an architecture}$$

or of the form

$$B_\Delta(\sigma_0) \subseteq F \qquad \textit{implementation by a state machine}$$

where the I/O-function $B_\Delta(\sigma_0)$ is defined by a state machine Δ (see [19]) and σ_0 is its initial state. In the case of the design of an architecture, its components $F_1,..., F_n$ can be hierarchically decomposed into a distributed architecture again, until a granularity of components is obtained which should not be further decomposed into a distributed system but realised by a state machine.

As explained, in a glass box refinement we replace a component by a design which is given by

- a network of components $F_1 \otimes F_2 \otimes ... \otimes F_n$ or
- a state machine $B_\Delta(\sigma_0)$ - let Σ be a set of states with an initial state σ_0 and a state transition function

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

  which describes a function

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

  where we define for each $\sigma \in \Sigma$, $z \in (I \to M^*)$, $x \in \vec{I}$ we specify B_Δ by the equation

  $$B_\Delta(\sigma).(\langle z \rangle \hat{} x) = \{\langle t \rangle \hat{} y: \exists\, \sigma' \in \Sigma: (\sigma', t) \in \Delta(\sigma, z) \wedge y \in B_\Delta(\sigma').x\}$$

In our approach iterated glass box refinement leads to a hierarchical, top down refinement method.

It is not in the centre of our paper to describe in detail the design steps leading to distributed systems or to a state machine. Instead, we take a very puristic point of view. Since we have introduced a notion of composition we consider a system architecture as given by a term defining a system by composing a number of components. A state machine is given by a number of transition equations that define

the transitions of the machine.

Accordingly, a glass box refinement is a special case of property refinement where the refinement component has a special syntactic form. In the case of a glass box refinement that transforms a component into a network, this form is a term composed of a number of components.

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

$$\mathrm{MRG} \otimes \mathrm{FRK} \subseteq \mathrm{TII}$$

It allows us to replace the component TII by two components. ❑

Hence, a glass box refinement works with the relation of property refinement and special terms representing the refined component.

Example. We describe a refinement of the specification TII by a state machine

$$\Delta\colon (\Sigma \times (\{x, y\} \to T3^{*})) \to (\Sigma \times (\{x', y'\} \to T3^{*}))$$

where the state space is given by the equation

$$\Sigma = T1^{*} \times T2^{*}$$

and the state transition relation Δ is specified by

$$\Delta((T1, T2), g) = \{((g(x), g(y)), h)\}$$

where h is specified by

$$h(x') = T1 \text{ and } h(y') = T2$$

This defines a most trivial state machine implementing TII by buffering its input always exactly one time unit. We obtain a glass box refinement formalised as follows

$$F_{\Delta}((\langle\rangle, \langle\rangle)) \subseteq \mathrm{TII}$$

In this case TII is refined into a state machine. ❑

Of course we may also introduce a refinement concept for state machines explicitly in terms of relations between states leading to simulations or bisimulations (see [1], [2], [5], [6], and also [3]). We do not do this here explicitly. We call a relation between state machines with initial states σ and σ' and transition function Δ and Δ' a refinement if

$$F_{\Delta'}(\sigma') \subseteq F_{\Delta}(\sigma)$$

The compositionality of glass box refinement is a straightforward consequence of the

compositionality of property refinement.

6.4 Interaction Refinement

Interaction refinement is the refinement notion that we need for modelling development steps between levels of abstraction. Interaction refinement allows us to change

- the number and names of input and output channels,
- the granularity of the messages on the channels

of a component.

An *interaction refinement* requires two functions

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

that relate the abstract with the concrete level of a development step from one level of abstraction to the next. 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. This is expressed by the requirement:

$$R ; A = Id$$

Let Id denote the identity relation. A is called the *abstraction* and R is called the *representation*. R and A are called a *refinement pair*. For untimed components it is sufficient to require for the time independent identity TII (as a generalisation of the specification TII)

$$R ; A \subseteq TII$$

Choosing the component MRG for R and FRK for A immediately gives a refinement pair for untimed components.

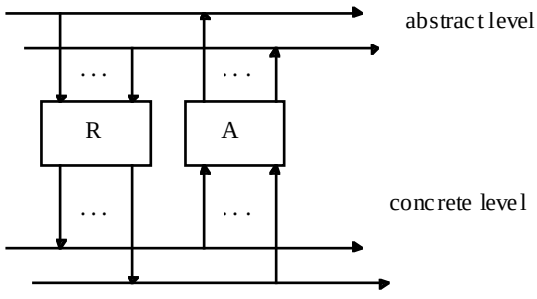

Fig 4 Communication History Refinement

Interaction refinement allows us to refine components, given appropriate refinement pairs for the input and output channels. The idea of a interaction refinement is visualised in Fig 5.

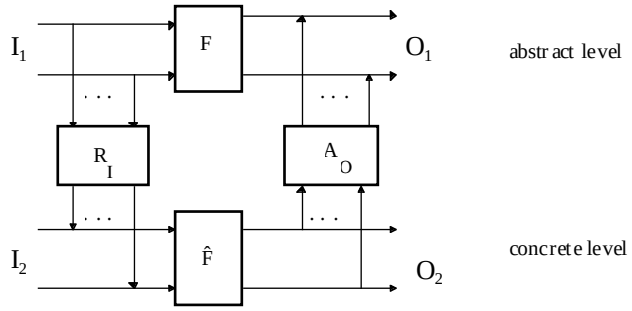

Fig 5 Interface Interaction Refinement (*U-simulation*)

Given interaction refinements

A_I: $\vec{I}_2 \rightarrow \wp(\vec{I}_1)$ $\quad$ R_I: $\vec{I}_1 \rightarrow \wp(\vec{I}_2)$

A_O: $\vec{O}_2 \rightarrow \wp(\vec{O}_1)$ $\quad$ R_O: $\vec{O}_1 \rightarrow \wp(\vec{O}_2)$

for the input and output channels we call the I/O-function

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

an *interaction refinement* of

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

if one of the following proposition holds:

$\hat{F} \subseteq A_I ; F ; R_O$	*U^{-1}-simulation*
$R_I ; \hat{F} \subseteq F ; R_O$	Downward Simulation
$\hat{F} ; A_O \subseteq A_I ; F$	Upward Simulation
$R_I ; \hat{F} ; A_O \subseteq F$	*U-simulation*

These are different versions of useful relations between levels of abstractions. A more detailed discussion is found in [13].

Example. Looking at the time independent identity for messages of type T3 we obtain the component specification as follows:

TII3

in z: T3, **out** z´: T3,
$\bar{z} = \bar{z}'$

We obtain

$$\text{MRG} ; \text{TII3} ; \text{FRK}[z'/z] \subseteq \text{TII}$$

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

6.5 Compositionality of U^{-1}-simulation

We concentrate on U^{-1}-simulation in the following and give the proof of compositionality only for that case. To keep the proof simple we do not give the proof for parallel composition with feedback but give the proof in two steps, 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:

$$\frac{\hat{F}_1 \subseteq A_I^1 ; F_1 ; R_O^1 \qquad \hat{F}_2 \subseteq A_I^2 ; F_2 ; R_O^2}{\hat{F}_1 \parallel \hat{F}_2 \subseteq (A_I^1 \parallel A_I^2); (F_1 \parallel F_2) ; (R_O^1 \parallel R_O^2)}$$

where we require the following syntactic conditions:

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

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

The proof of the soundness of this rule is straightforward since it only deals with parallel composition without interaction.

Example. If we replace in a property refinement 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

$$\text{MRG} ; \text{TII3}' ; \text{FRK}[z'/z] \subseteq \text{TII}$$

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

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

$$\frac{\hat{F} \subseteq (A_I \parallel A) ; F ; (R_O \parallel R)}{\mu\, \hat{F} \subseteq A_I ; \mu\, F ; R_O}$$

where we require the syntactic conditions

$$\text{In}(A) = \text{In}(\hat{F}) \cap \text{Out}(\hat{F}),$$

$$In(R) = In(F) \cap Out(F),$$

For independent parallel composition the soundness proof of the compositional refinement rule is straightforward. We give only the proof of the feedback operator and only for the special case where the channels coming from the environment and leading to the environment are empty. This proof easily generalises without any difficulties to the general case. For simplicity, we consider the special case where

$$In(F) = Out(F)$$

In this special case the compositional refinement rule reads as follows:

$$\frac{\hat{F} \subseteq A \; ; F \; ; R}{\mu \; \hat{F} \subseteq A \; ; \mu \; F \; ; R}$$

The proof of the soundness of this rule is shown as follows. Here we use the classical relational notation:

xFy

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	$\hat{z} \; \hat{F} \; \hat{z}$
and by the hypothesis:	$\exists$ x, y: $\hat{z} Ax \wedge xFy \wedge yR \; \hat{z}$
then by:	$xRz \wedge zAy \Rightarrow x = y$
we obtain:	$\exists$ x, y: $\hat{z} Ax \wedge xFy \wedge yR \; \hat{z} \wedge x = y$
and thus:	$\exists$ x: $\hat{z} Ax \wedge xFx \wedge xR \; \hat{z}$
and finally	$\hat{z} \in A; \mu \; F \; ; R$ ❑

The simplicity of the proof of our result comes from the fact that we have chosen such a straightforward model of component. In our model, in particular, input and output histories are represented explicitly. This allows us to apply classical ideas (see [17], [18]) of data refinement to communication histories. Roughly speaking: communication histories are nothing than data structures that can be manipulated and refined like other data structures.

Example. To demonstrate interaction refinement let us consider the specification of two delay components.

D3

in c, z: T3, **out** c´, z´: T3,
c´ = ‹›^z z´ = ‹›^c

D

in x, c: T1, y, d: T2,
out x´, c´: T1, y´, d´:T2,
c´ = ‹›^x, x´ = ‹›^c
d´ = ‹›^y, y´ = ‹›^d

We have

$$MRG \otimes MRG[c/x, d/y, c/z] ; D3 ; FRK \otimes FRK[c´/x, d´/y, c´/z] \subseteq D$$

and in addition

$$\mu D3[c/c´] \subseteq TII3, \qquad \mu D[c/c´, d/d´] \subseteq TII$$

and so finally we obtain

$$MRG ; \mu D3[c/c´] ; FRK \subseteq \mu D[c/c´, d/d´] \subseteq TII$$

which is an instance of the compositionality rule for interaction refinement. ❑

Our refinement calculus leads to a logical calculus for "programming in the large" where we can argue about software architectures.

7. Conclusions

What we have presented in the previous chapters is a comprehensive method for a system and software development which supports all steps of a hierarchical stepwise refinement development method. It is compositional and therefore supports all the modularity requirements that are generally needed.

What we have presented is a method that provides, in particular, the following ingredients:

- a proper notion of a syntactic and semantic interface of a component,
- a formal specification notation and method,
- a proper notion of composition,
- a proper notion of refinement and development,
- a compositional development method,
- a flexible concept of software architecture,
- concepts of time and the refinement of time (see [16]).

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

- combination with tables and diagrams,
- tool support in the form of AutoFocus (see [4]).

The simplicity of our results is a direct consequence of the specific choice of our semantic model. The introduction of time makes the model robust and expressive. The

fact that communication histories are explicit allows us to avoid all kinds of complications like prophecies or stuttering and leads to an abstract relational view of systems.

Of course, what we have presented is just the scientific kernel of the method. More pragmatic ways to describe specifications are needed. These more pragmatic specifications can be found in the work done in the SysLab-Project (see [7]) 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.

Acknowledgement

It is a pleasure to thank Max Breitling for a number of comments and helpful suggestions for improvement.

References

1. M. Abadi, L. Lamport: The Existence of Refinement Mappings. Digital Systems Research Center, SRC Report 29, August 1988
2. M. Abadi, L. Lamport: Composing Specifications. Digital Systems Research Center, SRC Report 66, October 1990
3. L. Aceto, M. Hennessy: Adding Action Refinement to a Finite Process Algebra. Proc. ICALP 91, Lecture Notes in Computer Science 510, (1991), 506-519
4. F. Huber, B. Schätz, G. Einert: Consistent Graphical Specification of Distributed Systems. In: J. Fitzgerald, C. B. Jones, P. Lucas (ed.): FME '97: 4th International Symposium of Formal Methods Europe, Lecture Notes in Computer Science 1313, 1997, 122-141
5. R.J.R. Back: Refinement Calculus, Part I: Sequential Nondeterministic Programs. REX Workshop. In: J. W. deBakker, W.-P. deRoever, G. Rozenberg (eds): Stepwise Refinement of Distributed Systems. Lecture Notes in Computer Science 430, 42-66
6. R.J.R. Back: Refinement Calculus, Part II: Parallel and Reactive Programs. REX Workshop. In: J. W. de Bakker, W.-P. de Roever, G. Rozenberg (eds): Stepwise Refinement of Distributed Systems. Lecture Notes in Computer Science 430, 67-93
7. R. Breu, R. Grosu, Franz Huber, B. Rumpe, W. Schwerin: Towards a Precise Semantics for Object-Oriented Modeling Techniques. In: H. Kilov, B. Rumpe (eds.): Proceedings ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques, 1997, Also: Technische Universität München, Institut für Informatik, TUM-I9725, 1997
8. M. Broy, B. Möller, P. Pepper, M. Wirsing: Algebraic Implementations Preserve Program Correctness. Science of Computer Programming 8 (1986), 1-19
9. M. Broy: Functional Specification of Time Sensitive Communicating Systems. REX Workshop. In: J. W. de Bakker, W.-P. de Roever, G. Rozenberg (eds): Stepwise Refinement of Distributed Systems. Lecture Notes in Computer Science 430, 153-179

10. M. Broy: Compositional Refinement of Interactive Systems. Digital Systems Research Center, SRC Report 89, July 1992, To appear in JACM
11. M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, R. Weber: The Design of Distributed Systems - An Introduction to Focus. Technische Universität München, Institut für Informatik, Sonderforschungsbereich 342: Methoden und Werkzeuge für die Nutzung paralleler Architekturen TUM-I9202, January 1992
12. M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, R. Weber: Summary of Case Studies in FOCUS - a Design Method for Distributed Systems. Technische Universität München, Institut für Informatik, Sonderforschungsbereich 342: Methoden und Werkzeuge für die Nutzung paralleler Architekturen TUM-I9203, January 1992
13. M. Broy: Interaction Refinement – The Easy Way. In: M. Broy (ed.): Program Design Calculi. Springer NATO ASI Series, Series F: Computer and System Sciences, Vol. 118, 1993
14. M. Broy: Algebraic Specification of Reactive Systems. M. Nivat, M. Wirsing (eds): Algebraic Methodology and Software Technology. 5th International Conference, AMAST '96, Lecture Notes of Computer Science 1101, Heidelberg: Springer 1996, 487-503
15. M. Broy: Towards a Mathematical Concept of a Component and its Use. First Components' User Conference, Munich 1996. Revised version in: Software-Concepts and Tools 18, 1997, 137-148
16. M. Broy: Refinement of Time. M. Bertran, Th. Rus (eds.): Transformation-Based Reactive System Development. ARTS'97, Mallorca 1997. Lecture Notes in Computer Science 1231, 1997, 44-63, To appear in TCS
17. J. Coenen, W.P. deRoever, J. Zwiers: Assertional Data Reification Proofs: Survey and Perspective. Christian-Albrechts-Universität Kiel, Institut für Informatik und praktische Mathematik, Bericht Nr. 9106, Februar 1991.
18. C.A.R. Hoare: Proofs of Correctness of Data Representations. Acta Informatica 1, 1972, 271-281
19. N. Lynch, E. Stark: A Proof of the Kahn Principle for Input/Output Automata. Information and Computation 82, 1989, 81-92