(INTER-)ACTION REFINEMENT: THE EASY WAY[1]

Manfred Broy
Institut für Informatik
Technische Universität München
Postfach 20 24 20, 8 München 2, Germany

Abstract

We outline and illustrate a formal concept for the specification and refinement of networks of interactive components. We describe systems by modular, functional specification techniques. We distinguish between black box and glass box views of interactive system components as well as refinements of their black box and glass box views. We identify and discuss several classes of refinements such as *behaviour refinement*, *communication history refinement*, *interface interaction refinement*, *state space refinement*, *distribution refinement*, and others. In particular, we demonstrate how these concepts of refinement and their verification are supported by functional specification techniques leading to a general formal refinement calculus. It can be used as the basis for a method for the development of distributed interactive systems.

[1]This work was supported by the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Architekturen"

1. Introduction

It is well accepted by now that the development of distributed systems is most adequately carried out by going through a sequence of development phases (often called *levels of abstraction*). Through these phases an envisaged system or system component is described in more and more detail until a sufficiently detailed description or even an efficient implementation of the system is obtained. The individual steps of such a proceeding can be captured by appropriate notions of *refinement*. In a refinement step parts or aspects of a system description are made more complete or more detailed. Often refinement steps include a considerable restructuring of system descriptions.

Throughout this study we consider interactive system components (sometimes called *open systems*). Interactive systems are connected to their outside world, their environment, by communication *channels* (often also called *ports*).

For the classification of refinements it is helpful to distinguish between the following views of a system:

- *black box view:* in the black box view of a system component only its interaction with its environment is described. The black box view of a system component consists of its *syntactic interface* (input and output channels and the sort of messages for each channel) and its *semantic interface* (sometimes called its *behaviour*, that is the causal relationship between input messages and output messages).
- *glass box view:* in the glass box view details of the internal system structure are described. This includes a description of the internal state space of the system and/or the internal distribution of the system into subsystems. The subsystems may again be described in different levels of detail (in a black box view or a glass box view).

Clearly the black box view is more abstract. It does not take into account how a system is presented or realized but characterizes its behaviour with respect to the surrounding environment. The black box view is captured by an *interface specification.*

A glass box description does also, however more implicitly, define a black box view. Therefore, often a black box view of a system is given implicitly in terms of a glass box description. Accordingly we have to distinguish between glass box descriptions that are just auxiliary constructions for giving a black box view and glass box views

that describe the actual internal structure (in terms of states or distribution) of the envisaged system realization.

Throughout the specification and development of a distributed system and its components there is a variety of aspects to be described and refined. These aspects include

- for the *black box view*: the *syntactic* (the channels by which system components are connected to the outside world and the sort of messages for each channel) and *semantic interface* (the system's observable behaviour);

- for the *glass box view*: the *internal representation* of the system: the system may be represented by a *state transition system* or by a *distributed system*. In a state transition system representation the representation of the states and the state transition function is included. A distributed system consists of a network of subcomponents forming subsystems and the syntactic and semantic properties of these subcomponents (their black box and perhaps also glass box views), their internal connections by channels, and the control and data flow within and between the components.

Clearly, these two views are not independent. The internal structure and the properties of the subcomponents induce also an external behaviour. The two complementary views give rise to the notion of correctness. Given an interface description (black box view) and an internal description (glass box view) the glass box view is called *correct* with respect to the black box view, if the properties of the system forming the black box view are logically implied by the properties of the system expressed by the glass box view.

Both views include the involved data sorts and computation structures describing in the black box view the sorts of messages and in the glass box view in addition the internal data, actions and states of the system and its subcomponents.

The behaviour of a system may be described by logical formulae in a very abstract, property-oriented way. Since a description of the glass box view implicitly also includes the description of a behaviour, the behaviour of a system may be described in terms of a state machine with input and output, or it may be described by programming language like notations in a constructive ("algorithmic") way. Also this syntactic form of a system description is a target for refinement steps.

Refinement steps for the black box as well as for the glass box view always are expressed via a transformation of the syntactic description of the system. Obviously, some of these refinement steps do not change the semantics of a system description, but just the way it is presented.

Based on the above classification of system descriptions into levels of abstraction and syntactic description forms, we may distinguish between the following types of refinement:

- black box refinement:
  - behaviour refinement (leaving the syntactic interface unchanged),
  - communication history and interface interaction refinement (even changing the syntactic interface such as the number of channels and their sorts);
- glass box refinement:
  - refinement of the state space (data structure refinement),
  - distribution (architectural) refinement (refining a component into a network),
  - refinement of subcomponents of a distributed system,
  - refinement of the interaction between the subcomponents;
- refinement by rewriting and reformulating the syntactic form of a system specification without changing its meaning.

Although algebraic and functional specification techniques provide a common formal logical framework in which all these different concepts of refinement can be captured, different refinement concepts use and need quite different development, specification and verification styles and formats. For all the mentioned concepts of refinement it is helpful to provide an appropriate syntactic and semantic refinement relation that formalizes the syntactic and semantic relationship between the original and the refined system. Since refinement is the decisive concept in system development, obviously the usefulness of a development formalism strongly depends on its flexibility with respect to incorporating refinement notions.

A well worked out formal approach to program refinement is that of *program transformation* (cf. [CIP 84]). A program transformation can be viewed as a rule for obtaining from a given specification or program a more refined semantically equivalent one provided the rule is applicable. Working just with schematic transformation rules may sometimes be too narrow. Often refinement steps cannot be captured by simple schematic rules but need more elaborate descriptions (see [Hoare 72], [Jones 86], [Nipkow 86]).

In the following we study various notions of refinement within a functional axiomatic formalism. We start by defining a functional system model. Then we introduce functional system specification techniques and a number of composition operators such as sequential and parallel composition as well as feedback. With the help of the composition operators we can form networks from given components. We use logical implication as the basic notion of refinement for system specifications. More specific forms of refinement are then defined that allow us to change the number of channels and the granularity of the messages of a system component. Finally, refinements of the glass box views are studied. All notions are demonstrated with the help of a simple running example.

2. System Model and Specification

In this section we introduce functional models of interactive systems and system components. We define the basic mathematical structures and concepts for the specification of components.

2.1 Basic Structures

In the following interactive systems are supposed to communicate asynchronously via unbounded FIFO channels. Streams are used to denote histories of communications on channels. Given a set M of messages, a *stream* over M is a finite or infinite sequence of elements from M. By M^* we denote the finite sequences over M. M^* includes the empty stream which is denoted by ‹›.

By M^∞ we denote the infinite streams over the set M. M^∞ can be represented by the total mappings from the natural numbers $\mathbb{N}$ into M. We denote the set of all streams over the set M by M^ω. Formally we have

$$M^\omega =_{def} M^* \cup M^\infty.$$

We introduce a number of functions on streams that are useful in system descriptions.

A classical operation on streams is the *concatenation* which we denote by ^. The concatenation is a function that takes two streams (say s and t) and produces a stream s^t as result, starting with the stream s and continuing with the stream t. Formally the concatenation has the following functionality:

$$.\hat{}. : M^\omega \times M^\omega \rightarrow M^\omega .$$

If the stream s is infinite, then concatenating the stream s with a stream t yields the stream s again:

$$s \in M^\infty \Rightarrow s\hat{}t = s .$$

Concatenation is associative and has the empty stream ‹› as its neutral element:

$$r\hat{}(s\hat{}t) = (r\hat{}s)\hat{}t, \qquad \langle\rangle\hat{}s = s = s\hat{}\langle\rangle .$$

For any message $m \in M$ we denote by ‹m› the one element stream consisting of the element m.

On the set M^ω of streams we define a *prefix ordering* $\sqsubseteq$. We write $s \sqsubseteq t$ for streams s and t if s is a *prefix* of t. Formally we have

$$s \sqsubseteq t \quad \text{iff} \quad \exists\, r \in M^\omega : s\hat{}r = t .$$

The prefix ordering defines a partial ordering on the set M^ω of streams. If $s \sqsubseteq t$, then we also say that s is an *approximation* of t. The set of streams ordered by $\sqsubseteq$ is complete in the sense that every directed set $S \subseteq M^\omega$ of streams has a *least upper bound* denoted by lub S. A nonempty subset S of a partially ordered set is called *directed,* if

$$\forall\, x, y \in S: \exists\, z \in S: x \sqsubseteq z \wedge y \sqsubseteq z\ .$$

By least upper bounds of directed sets of finite streams we may describe infinite streams. Infinite streams are also of interest as (and can also be described by) fixpoints of prefix monotonic functions. The streams associated with feedback loops in interactive systems correspond to such fixpoints.

A *stream processing function* is a function

$$f: M^\omega \rightarrow N^\omega$$

that is *prefix monotonic* and *continuous.* The function f is called *prefix monotonic,* if for all streams s and t we have

$$s \sqsubseteq t \Rightarrow f.s \sqsubseteq f.t\ .$$

For better readability we often write for the function application f.x instead of f(x). A prefix monotonic function f is called *prefix continuous,* if for all directed sets $S \subseteq M^\omega$ of streams we have

$$f.\text{lub } S = \text{lub } \{f.s: s \in S\}\ .$$

If a function is prefix continuous, then its results for infinite input can be already determined from its results on all finite approximations of the input.

By $\perp$ we denote the pseudo element which represents the result of diverging computations. We write $M^\perp$ for $M \cup \{\perp\}$. Here we assume that $\perp$ is not an element of M. On $M^\perp$ we define also a simple partial ordering called the flat ordering as follows:

$$x \sqsubseteq y \quad \text{iff} \quad x = y \vee x = \perp$$

We use the following functions on streams

$$ft: M^\omega \rightarrow M^\perp,$$

$$rt: M^\omega \rightarrow M^\omega.$$

The function ft selects the first element of a nonempty stream. The function rt deletes the first element of a nonempty stream.

For keeping our notation simple we extend concatenation ^ also to elements of the message set M (treating them like one element sequences) and to tuples of streams (by concatenating the streams elementwise). For the special element $\perp$ we specify $\perp\hat{}s = \langle\rangle$. This equation reflects the fact that there cannot be any further message on a channel

after trying to send a message that is to be generated by a diverging (and therefore never ending) computation.

The properties of the introduced functions can be expressed by the following equations (let $m \in M$, $s \in M^{\omega}$):

$$ft.\langle\rangle = \bot, \qquad rt.\langle\rangle = \langle\rangle, \qquad ft(m\hat{}s) = m, \qquad rt(m\hat{}s) = s.$$

All the introduced concepts and functions such as the prefix ordering and the concatenation carry over to tuples of streams by pointwise application. Similarly the prefix ordering induces a partial ordering on functions with streams and tuples of streams as range.

We denote the function space of (n,m)-ary prefix continuous stream processing functions by:

$$[(M^{\omega})^n \rightarrow (M^{\omega})^m]$$

The operations ft and rt are prefix monotonic and continuous, whereas concatenation $\hat{}$ as defined above is prefix monotonic and continuous only in its second argument.

2.2 Specification

In functional system modelling as used in the sequel the observable behaviours of a system component are described by giving a *syntactic interface* consisting of

- a number of input channels with sort information for each channel (specifying the sort of the messages received on each channel),
- a number of output channels with sort information,

and a *semantic interface specification*.

More technically, an input (or output) interface is syntactically determined by the number n of channels and by sets M_i, $1 \leq i \leq n$, of messages for each channel. We write

$$M_i;^n$$

to denote the set

$$M_1;^{\omega} \times ... \times M_n;^{\omega}.$$

By

$$[M_i;^n \rightarrow N_i;^m]$$

we denote the set of prefix continuous stream processing functions of functionality

$$M_1;^{\omega} \times ... \times M_n;^{\omega} \rightarrow N_1;^{\omega} \times ... \times N_m;^{\omega}.$$

The semantic interface (the behaviour) of a system component is represented by a set of functions characterized by a predicate

$$P: [M_i;^n \rightarrow N_i;^m] \rightarrow \mathbb{B} .$$

The predicate P specifies the set of possible behaviours of an interactive component. P is called a *behavioural* interface specification.

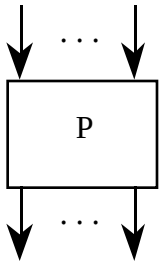

Fig.1 Graphical Representation of Component P

The *behaviour* of a deterministic system is represented by a prefix continuous function

$$f \in [M_i;^n \rightarrow N_i;^m] .$$

The function f associates with every tuple of input streams $x \in M_i;^n$ a tuple of output streams $f.x \in N_i;^m$. Nondeterministic systems are described by predicates that characterize a set of functions representing their set of possible behaviours.

We illustrate our specification technique by a simple example.

Example: Interactive Queue
An interactive queue is a system that receives a stream of data elements from a given set D and request signals represented by the symbol ¿, and produces a stream of data in reply. It can be specified by a predicate

$$QU: [M^{\omega} \rightarrow M^{\omega}] \rightarrow \mathbb{B}$$

where the set M of messages is given by

$$M = D \cup \{¿\} .$$

The specification QU is given by the following formula:

QU.f ≡	$f \in [M^{\omega} \to M^{\omega}] \wedge \forall\, x \in D^*, d \in D, y \in M^{\omega}$:
	$f.x = \langle\rangle$,
	$f(d^\frown x^\frown ¿^\frown y) = d^\frown f(x^\frown y)$

For simplicity, the request signal ¿ is included also in the sort of the messages in the output stream. This will be used later in the further development of the example.

The predicate QU does not characterize a function f uniquely. If the first input is a request signal, then nothing is specified about the interactive queue's behaviour. The predicate QU specifies a set of functions. □

In our setting we do not distinguish between *nondeterminism* (as an operational notion of choices made during a computation) and *underspecification* (as a property of a description allowing several functions to fulfil a specifying predicate). This unifying view of nondeterminism and underspecification is justified by the conception that freedom of choice in a specification (underspecification) can be resolved on the one hand during the *design* by design decisions narrowing (or completely removing) underspecification or on the other hand by nondeterministic choices during *execution* (nondeterminism). It is considered to be one of the strong properties of the presented approach that no distinction between underspecification and nondeterminism (introducing a more operational view) has to be made (for the difficulties with refinement concepts for more sophisticated concepts of nondeterminism see [Aceto, Hennessy 91], [Janssen et al. 91]).

Sets of functions provide a very convenient way for representing the behaviour of interactive systems. Sets of functions can be specified by logical formulas. A logical formula that formalizes a property of a function f is generally valid not just for one particular function (as in the case that a specification characterizes f uniquely), but for a set of functions. If this set is empty the specification is called *inconsistent*.

2.3 Forms of Composition

For composing stream processing functions and thereby forming networks we use the three classical forms of composition, namely *sequential* and *parallel* composition and *feedback*.

Let $g \in [M_i^{\,n} \to N_i^{\,m}]$ and $h \in [M\tilde{}_i^{\,n'} \to N\tilde{}_i^{\,m'}]$; we denote the *parallel composition* of g and h by g||h where

$g||h \in [M_i^{\,n} \times M\tilde{}_i^{\,n'} \to N_i^{\,m} \times N\tilde{}_i^{\,m'}]$.

Parallel composition can be visualized by the diagram given in Fig. 2.

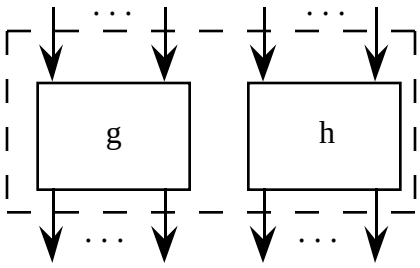

Fig. 2 Parallel Composition

We define for input histories $x \in M_i;^n$, $y \in M;\tilde{}_i;^{n'}$ the function obtained by parallel composition of the functions g and h by the equation

$$(g||h)(x \oplus y) = g(x) \oplus h(y)$$

where $\oplus$ denotes the concatenation of tuples of streams.

Let the function $g \in [M_i;^n \rightarrow N_i;^m]$ and the function $h \in [N_i;^m \rightarrow N;\tilde{}_i;^{m'}]$ be given; we denote the function obtained by the *sequential composition* of of the function g and the function h by g;h where

$$g;h \in [M_i;^n \rightarrow N;\tilde{}_i;^{m'}].$$

Sequential composition can be visualized by the diagram given in Fig. 3.

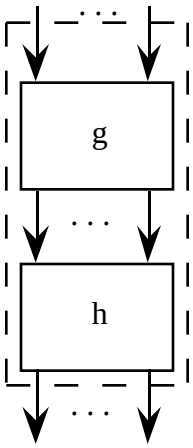

Fig. 3 Sequential Composition

We define for the input history $x \in M_i;^n$ the function obtained by sequential composition of the functions g and h by the equation

$(g;h).x = h.g.x$.

We define the feedback operator μ^k for a function $f \in [M_i;^n \times K_i;^k \rightarrow N_i;^m \times K_i;^k]$. If we apply the fixed-point operator to a function f we obtain the function

$\mu^k f \in [M_i;^n \rightarrow N_i;^m \times K_i;^k]$

It is the function derived from f by k feedback loops. The feedback operator can be visualized by the diagram given in Fig. 4.

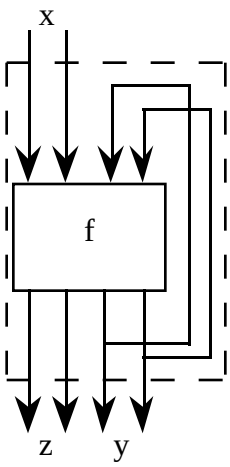

Fig. 4 Graphical Illustration of Feedback

We define for the input history $x \in M_i;^n$ the function μ^k f that is obtained by k feedback loops from the function f by the equation

$(\mu^k f).x =$ **fix** λ z, y: f(x, y)

where $z \in N_i;^m$ and $y \in K_i;^k$. By **fix** we denote the least fixed-point operator. In this equation the fixed-point operator is applied to the function λ z, y: f(x, y) which for every given value of x denotes a function which does not depend on its parameter z (z is a parameter that does not influence the result of λ z, y: f(x, y)).

In a more readable version, we may specify the μ-operator as follows: we have

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

if $z \oplus y$ is the least solution (fixed-point) of the following equation

$(z \oplus y) = f(x \oplus y).$

Formally $(\mu^k f).x$ denotes a fixed-point relative to x, such that for every given tuple x of streams we obtain a tuple of streams $(z \oplus y)$ by a fixed-point construction.

The least fixed-point concept reflects the characteristics (the causality) of feedback in communications in an appropriate manner. The output of the component with

behaviour f on its feedback lines is sent back to its own input channels. If further output requires more input than available, no further output is generated. This is properly modelled by the assumption that the chosen fixed-point is the least one.

The introduced operators carry over to specifications in a straightforward way. Let Q and R be specifications of functions with appropriate functionality; we extend the three introduced operators to specifications as follows:

$(Q \parallel R).f = \exists\, g, h: Q.g \wedge R.h \wedge f = g \| h$

$(Q\ ;\ R).f = \exists\, g, h: Q.g \wedge R.h \wedge f = g;h$

$(\mu^k\, Q).f = \exists\, g: Q.g \wedge f = \mu^k g$

The composition of specifications is understood as the pointwise composition of the specified functions.

2.4 Networks of Communicating Systems

Informally speaking a distributed system consists of a number of components interacting via channels. Some of these channels may be connected to the system's environment. The channels connected to the system's environment define the system's syntactic interface $[M_i;^n \rightarrow N_i;^m]$.

The system's *internal syntactic (static) structure* (visible in the glass box view) of a network is mathematically represented by an expression formed by sequential composition, parallel composition and feedback from basic components including those for permuting, copying or deleting their input lines.

As can be easily seen, all kinds of finite networks can be represented that way. In addition infinite networks can be defined by recursive equations for networks.

3. Notions of Refinement

In the following we treat several concepts of refinement of interactive systems. There are many related notions of refinement useful in system development. We mainly concentrate on behaviour refinement, communication history refinement, and interface refinement, but only briefly touch and discuss other types of refinement.

There are many different proposals for formalizing the notion of refinement. We choose here the most simple and most basic logical notion of refinement of

specifications, namely logical implication: a behaviour specification Q is called a *behaviour refinement* of the behaviour specification P if both P and Q have the same syntactic interface and in addition we have

$$Q.f \Rightarrow P.f$$

for all functions f; we then write $Q \Rightarrow P$. Accordingly a behaviour refinement never introduces new observable interactions, but just restricts the behaviour by adding properties. An inconsistent specification is a refinement for every specification with the same syntactic interface. It is, however, not a very useful refinement, since it cannot be refined into an implementation.

We will understand all other classes of refinements considered in the following as special forms of behaviour refinements where Q and P in addition are in a more specific syntactic or semantic relationship.

Concepts of refinement for data structures and their characteristic operations are well-known and well-understood in the framework of algebraic specification (see, for instance, [Broy et al. 86]). In the modelling of distributed interactive systems data structures are used to represent

- the messages passed between the components,
- the histories of interactions between components (streams of messages),
- the states of the system.

In all three cases we may use the very general notion of data structure refinement. As it will be demonstrated in the sequel, several concepts of system refinement can be obtained by variations of data structure refinement.

3.1 Refining Black Box Views: Behaviour Refinement

We consider two versions of refinement of the black box view: refinement of the syntactic interface (by changing the number and the names as well as the sorts of the channels) of a system and refinement of the behaviour of a system. If the syntactic interface is refined then a concept is needed for relating the behaviours of the original and the refined system. This can be done by appropriate mappings (for another approach to refinement, see [Back 88a] and [Back 88b]).

3.1.1 Behaviour Refinement

A behaviour refinement is obtained by sharpening the requirements formalized in the specification. If we have the specifying predicate

$$P: [M_i;^n \rightarrow N_i;^m] \rightarrow \mathbb{B}$$

a behaviour refinement is any predicate

$$P;^{\wedge} : [M_i;^n \rightarrow N_i;^m] \rightarrow \mathbb{B}$$

where

$$P;^{\wedge} \Rightarrow P$$

(or more precisely $\forall$ f: P;$^{\wedge}$.f $\Rightarrow$ P.f). Of course, a refinement is only practically helpful if P;$^{\wedge}$ (and in consequence P) is consistent, more formally, if we have

$$\exists \text{ f: } P;^{\wedge}.f \text{ .}$$

From a methodological point of view there are many different reasons for performing a behaviour refinement. A typical example is the sharpening of specifications for the inclusion of exceptional cases.

Example: Inclusion of Exceptional Cases
In the queue specification QU as shown in the first example nothing is said about the exceptional case where a request signal ¿ is received by the queue before a data message arrives. A behaviour refinement QU' for the specification QU is simply obtained by adding a requirement for that case.

QU'.f ≡ f ∈ [$M^{\omega} \rightarrow M^{\omega}$] ∧ QU.f ∧ ∀ y ∈ M^{ω}:
f(¿^y) = ¿^f.y

This specification requires that a request signal ¿ received when the queue is empty is processed by just reproducing the request in the output stream. □

Often behaviour specifications are written in the *assumption/commitment format* (also called *rely/guarantee format*). A simple functional assumption/commitment specification has the following form

$$P.f \equiv \forall \text{ x: } A.x \Rightarrow C(f, x).$$

Here A is called *input assumption* and C is called *commitment specification*. This simple scheme of specification immediately suggests straightforward concepts of behaviour refinement:

- weakening the input assumptions A,
- strengthening the commitment specification C.

This is especially useful for refinements making incompletely specified systems more robust by taking into account exceptional input situations and specifying the required behaviour in reaction to these.

3.1.2 Communication History Refinement

In this section we treat refinements that change the syntactic interface of a component, that is the number of input and output channels as well as their message sorts. These are often refinements between different levels of abstractions. We study refinement steps from an abstract level to a (more) concrete level. Then the behaviours of the refined component have to be related to the behaviours of the original component. This can be achieved by translating communication histories for the channels on the abstract level to communication histories for the channels on the concrete level.

In our setting a syntactic interface is given by a set of n (input or output) channels with their corresponding sorts M_i of messages. Communication histories for these channels are given by the elements of the set

$$M_i;^n .$$

In a communication history refinement a tuple of streams representing a communication history is replaced by another tuple of streams. For doing that we specify a translation. In our framework such a translation is defined by specifications R ("*representation specification*") and A ("*abstraction specification*") defining translations between the communication histories of the abstract level and the concrete level. This translation leads to commuting diagrams of the form illustrated in Fig 5.

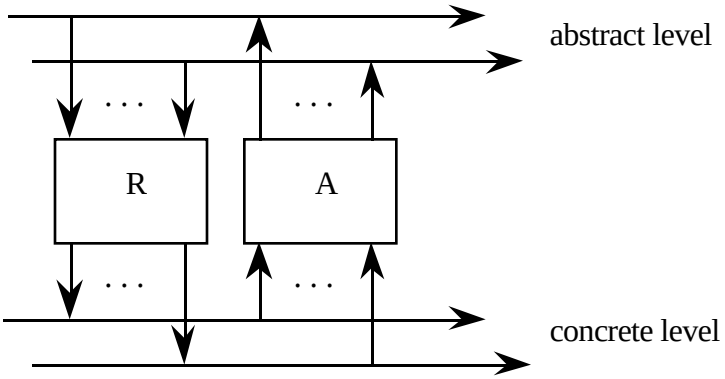

Fig. 5 Communication History Refinement

A pair of predicates specifying interactive system components

$R: [M_i;^n \rightarrow M;\tilde{}_i;^{n'}] \rightarrow \mathbb{B}$ *representation specification*

$A: [M;\tilde{}_i;^{n'} \rightarrow M_i;^n] \rightarrow \mathbb{B}$ *abstraction specification*

is called *communication history refinement pair* for the communication histories from $M_i;^n$ with representation elements in $M;\tilde{}_i;^{n'}$ if the abstraction specification A and representation specification R are consistent, that is the formula

$$\exists\, \alpha: A.\alpha \wedge \exists\, \rho: R.\rho$$

holds and for all functions

$$\alpha : [M;\tilde{}_i;^{n'} \rightarrow M_i;^n] , \qquad \rho : [M_i;^n \rightarrow M;\tilde{}_i;^{n'}]$$

we have that

$$A.\alpha \wedge R.\rho \Rightarrow \rho\ ;\ \alpha = I$$

or, for short,

$$R\ ;\ A = I\ ,$$

where I denotes the identity function (or more precisely the predicate that characterizes the identity function). Then A is called *abstraction specification* and R is called *representation specification.*

By the functions ρ with $R.\rho$ a communication history $x \in M_i;^n$ can be rewritten into a communication history $\rho.x \in M;\tilde{}_i;^{n'}$. From $\rho.x$ we can obtain x again by applying any function α with $A.\alpha$ since $x = \alpha.\rho.x$. Therefore $\rho.x$ can be viewed as a (possibly "less abstract") representation of the communication history x.

In many approaches a relation is used instead of the representation specification R. The specification R is, however, very similar to a relation. Every function ρ with $R.\rho$ represents one choice of representations for the abstract communication histories.

In the sequel we use the following notion for *restricting* a function to a particular subdomain. Given a function

$$f: D \to E$$

for $C \subseteq D$ we denote by $f|C$ the function f restricted to arguments from C. Formally, the restriction of the function f to the set C yields the function

$$f|C: C \to E$$

where

$$(f|C).x = f.x \quad \text{for } x \in C.$$

From the definitions above we derive immediately the following properties for communication history refinement pairs A, R:

(1) all abstraction functions α with $A.\alpha$ are surjective;

(2) A is deterministic on the image of R, that is, all functions α with $A.\alpha$ behave identical on the image of R; formally expressed defining the image of R by a set $J \subseteq M;\tilde{}_i;^{n'}$, where

$$J = \{ \rho.x \in M;\tilde{}_i;^{n'} : x \in M_i;^n \wedge R.\rho \},$$

we have for all functions α and α'

$$A.\alpha \wedge A.\alpha' \Rightarrow \alpha|J = \alpha'|J ;$$

(3) all functions ρ with $R.\rho$ are injective. More generally, we have for all x and x' and all representation functions ρ and ρ':

$$R.\rho \wedge R.\rho' \wedge \rho.x = \rho'.x' \Rightarrow x = x' .$$

Property (1) immediately follows by the surjectivity of I.

Property (2) can easily be proved as follows: assume $A.\alpha$ and $A.\alpha'$; then for $y = \rho.x$ with $x \in M_i;^n$ we have

$$\alpha.y = \alpha.\rho.x = x = \alpha'.\rho.x = \alpha'.y .$$

Property (3) immediately follows by the injectivity of I.

Example: Communication History Refinement

(1) A simple example for communication history refinement is multiplexing. In multiplexing by interleaving messages one channel is used instead of several ones. We define the representation specification as follows:

$R1.\rho \equiv$	$\rho \in [\, \mathbb{N}^\omega \to \mathbb{N}^\omega \times \mathbb{N}^\omega] \wedge \forall\, x \in \mathbb{N}^\omega$:
	$\rho(x) = (ft.x,\ ft.rt.x)\ \hat{}\ \rho(rt.rt.x)$

We define the abstraction specification as follows:

$A1.\alpha \equiv$	$\alpha \in [\, \mathbb{N}^\omega \times \mathbb{N}^\omega \to \mathbb{N}^\omega] \wedge \forall\, x, y \in \mathbb{N}^\omega$:
	$\alpha(x, y) = (ft.x)\ \hat{}\ (ft.y)\ \hat{}\ \alpha(rt.x,\ rt.y)$

We immediately obtain the following theorem.

Theorem: $(R1\ ;\ A1) = I$

(2) Let the set M of messages be defined as in the example of interactive queues above. To allow data from D and request signals to be sent on separated channels we introduce a synchronizing message √. We define the sets of messages:

$TD = D \cup \{\surd\}$,

$TR = \{¿, \surd\}$.

We define the representation specification as follows:

$R2.\rho \equiv$	$\rho \in [\, M^\omega \to (TD^\omega \times TR^\omega)\,] \wedge \forall\, d \in D,\ x \in M^\omega$:
	$\rho(d\hat{}x) = (d\hat{}\surd,\ \surd)\hat{}\rho.x$, $\rho(?\hat{}x) = (\surd,\ ¿\hat{}\surd)\hat{}\rho.x$

We define the abstraction specification as follows:

$A2.\alpha \equiv \quad \alpha \in [(TD^{\omega} \times TR^{\omega}) \rightarrow M^{\omega}] \wedge \forall\, d \in D, y \in TD^{\omega}, z \in TR^{\omega}:$
$\alpha(\surd\hat{}y, \surd\hat{}z) = \alpha(y, z),$ $\alpha(d\hat{}y, \surd\hat{}z) = d\hat{}\alpha(y, \surd\hat{}z),$ $\alpha(\surd\hat{}y, ?\hat{}z) = ?\hat{}\alpha(\surd\hat{}y, z)$

The signal $\surd$ is used to separate the data messages and request signals. We immediately obtain the following theorem:

Theorem: $(R2 ; A2) = I$ □

Let us now briefly look at the specification obtained by the sequential composition (A;R), that is all functions f such that (A;R).f. The set

$$\{f: (A;R).f\}$$

denotes the sheaf of functions that map representations of communication histories onto (in the sense of the abstraction) equivalent representations.

For practical purposes we are not always interested in a communication history refinement that works correctly on the full set of communication histories. Often we are interested only in correctness for a subset of the considered input space: let

$$S: M_i;^n \rightarrow \mathbb{B}$$

be a predicate that defines a subset of communication histories. It is called a *communication history restriction*. A pair of consistent predicates is called *communication history refinement pair with respect to the restriction* S if for all tuples of streams $x \in M_i;^n$:

$$S.x \wedge A.\alpha \wedge R.\rho \;\Rightarrow\; \alpha.\rho.x = x$$

Communication history refinements can in particular be used to define refinements of interacting components. This is treated in detail in the following section.

3.1.3 Interface Interaction Refinement

We consider a specification of an interactive component

$$P: [M_i;^n \rightarrow N_i;^m] \rightarrow \mathbb{B}$$

and predicates

$$R: [M_i;^{n} \rightarrow M;\tilde{}_i;^{n'}] \rightarrow \mathbb{B}$$

$$P;\hat{}: [M;\tilde{}_i;^{n'} \rightarrow N;\tilde{}_i;^{m'}] \rightarrow \mathbb{B}$$

$$A;\tilde{}: [N;\tilde{}_i;^{m'} \rightarrow N_i;^{m}] \rightarrow \mathbb{B}$$

where R is a representation specification (with a corresponding abstraction specification A) and $A;\tilde{}$ is an abstraction specification (for a representation specification $R;\tilde{}$); the triple

$$(R, P;\hat{}, A;\tilde{})$$

of predicates is called *component interface interaction refinement* for the component specification P with representation specification R and abstraction specification $A;\tilde{}$, if the following condition is fulfilled (also called *U-simulation*):

$$R ; P;\hat{}; A;\tilde{} \Rightarrow P \qquad \textit{U-simulation}$$

This condition is graphically expressed by the commuting diagram given in Fig. 6.

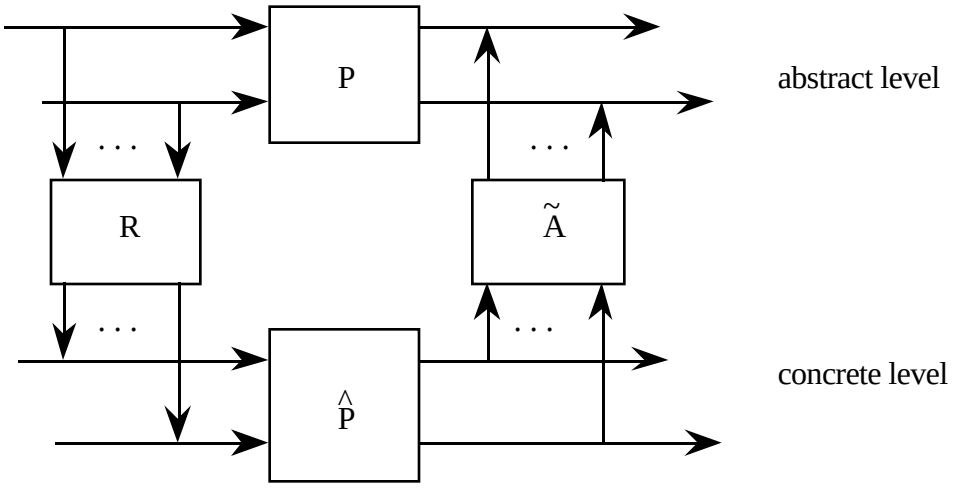

Fig. 6. Commuting Diagram of Interface Interaction Refinement (*U-simulation*)

Example: Interface Interaction Refinement

(1) Addition

Using the example of the communication history refinement given above we may refine the following specification:

ADD.f ≡ $f \in [(\mathbb{N}^\omega \times \mathbb{N}^\omega) \to \mathbb{N}^\omega] \wedge \forall x, y \in \mathbb{N}^\omega$:
$f(x, y) = (ft.x + ft.y) \hat{} f(rt.x, rt.y)$

The refinement is defined by (R1, ADD', I) where ADD' is specified as follows:

ADD'.f ≡ $f \in [\mathbb{N}^\omega \to \mathbb{N}^\omega] \wedge \forall x \in \mathbb{N}^\omega$:
$f.x = (ft.x + ft.rt.x) \hat{} f(rt.rt.x)$

We obtain the following theorem

Theorem: R1 ; ADD' ; I ⇒ ADD

(2) Queues

We specify the predicate QI by a state oriented specification technique which will be treated more systematically in the following section.

QI.f ≡ (f = h.‹›) **where** h: $D^* \to [(TD^\omega \times TR^\omega) \to (TD^\omega \times TR^\omega)] \wedge$ $\forall q \in D^*, d \in D, y \in TD^\omega, z \in TR^\omega$:
$(h.q).(\surd\hat{}y, \surd\hat{}z) = (\surd, \surd)\hat{}(h.q).(y, z)$ $(h.q).(d\hat{}y, \surd\hat{}z) = h(q\hat{}\langle d\rangle).(y, \surd\hat{}z)$ $(h.\langle\rangle).(\surd\hat{}y, ¿\hat{}z) = (\langle\rangle, ¿)\hat{}(h.q).(\surd\hat{}y, z)$ $(h.d\hat{}q).(\surd\hat{}y, ¿\hat{}z) = (d, \langle\rangle)\hat{}(h.q).(\surd\hat{}y, z)$

We immediately obtain the following theorem:

Theorem: R2 ; QI ; A2 ⇒ QU' □

The component interaction refinement as introduced above basically expresses, that we may replace in a refinement step the component specified by P by a component specified by $(R;P;\hat{};A;\tilde{})$. After this refinement, instead of computing with the component P on the "abstract" level, we transform an input history given on the abstract level by R into an input history on the "concrete" level, compute with the component $P;\hat{}$ and transform the result by $A;\tilde{}$ back onto the abstract level. This form of refinement leads to a behaviour refinement of the original component specification P.

There are alternatives to this form of refinement (see also [Coenen et al. 91] for a systematic treatment). Instead of computing with the component P on the "abstract" level, we may translate an input history given on the abstract level by an appropriate representation specification R into an input history on the "concrete" level, where we compute with a component specified by $P;^{\wedge}$. We require that this form of computation is a refinement of the computation obtained for the abstract input with the component specified by P on the abstract level and a transformation of the result to the concrete level by a representation specification $R;^{\sim}$.

This concept of refinement between levels of abstraction is formalized as follows. Given representation specifications and corresponding abstraction specifications:

$$R:\ [M_i;^{n} \to M;^{\sim}_i;^{n'}] \to \mathbb{B}\ , \qquad A:\ [M;^{\sim}_i;^{n'} \to M_i;^{n}] \to \mathbb{B}\ ,$$

$$R;^{\sim}:\ [N_i;^{m} \to N;^{\sim}_i;^{m'}] \to \mathbb{B}\ , \qquad A;^{\sim}:\ [N;^{\sim}_i;^{m'} \to N_i;^{m}] \to \mathbb{B}\ ,$$

and specifications

$$P:\ [M_i;^{n} \to N_i;^{m}] \to \mathbb{B}$$

$$P;^{\wedge}:\ [M;^{\sim}_i;^{n'} \to N;^{\sim}_i;^{m'}] \to \mathbb{B}$$

then $P;^{\wedge}$ is called a *refinement of* P *under the representation specifications* R and $R;^{\sim}$ if

$$R\ ;\ P;^{\wedge}\ \Rightarrow\ P\ ;\ R;^{\sim}\ .$$

This formula is visualized by the commuting diagram in Fig. 7.

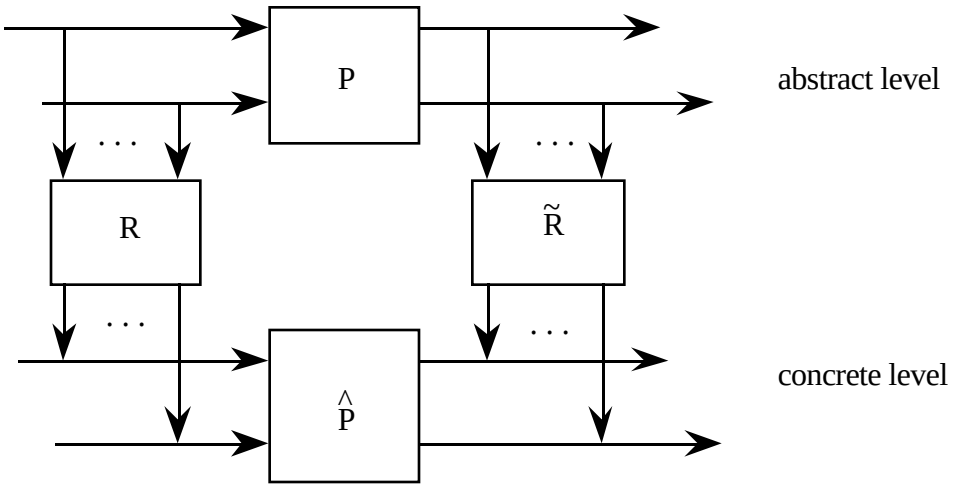

Fig. 7 Downward Simulation

Of course, with the help of the abstraction specification $A;^{\sim}$ for the representation specification $R;^{\sim}$ we obtain:

$$R\ ;\ P;^{\wedge}\ ;\ A;^{\sim}\ \Rightarrow\ P.$$

However, even if we work with specifications R and $R;^{\sim}$ for which abstractions do not exist, refinements of the form

$$R\ ;\ P;^{\wedge}\ \Rightarrow\ P\ ;\ R;^{\sim} \qquad \textit{downward simulation}$$

are often useful, since they allow to relate specifications at different levels of abstraction where the representations of the abstract communication histories are not necessarily unique. In [Broy 92b] it is shown that the condition

$$R\ ;\ P;^{\wedge}\ \Rightarrow\ P\ ;\ R;^{\sim}$$

can be used as the basic condition for a compositional refinement of networks. A compositional refinement concept allows the refinement of a network by refinements of its components (for another approach to the composition of specifications, see [Abadi, Lamport 90]).

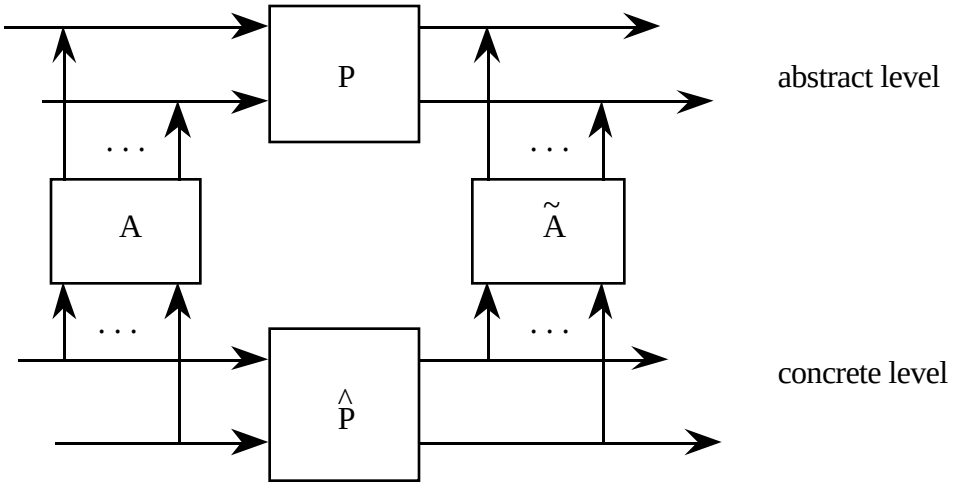

Fig. 8 Upward Simulation

Two further versions of interaction refinement are obtained by the condition

$$P;^{\wedge}\ ;\ A;^{\sim}\ \Rightarrow\ A\ ;\ P \qquad \textit{upward simulation}$$

and by the stronger condition (also called *U^{-1}-simulation*)

$$P;^{\wedge}\ \Rightarrow\ A\ ;\ P\ ;\ R;^{\sim}\ .$$

By this formula the specification $P;\hat{}$ is a behaviour refinement of the specification $(A;P;R;\tilde{})$. This last version of refinement is the strongest requirement. All other three versions of refinement can be deduced if this refinement formula is valid.

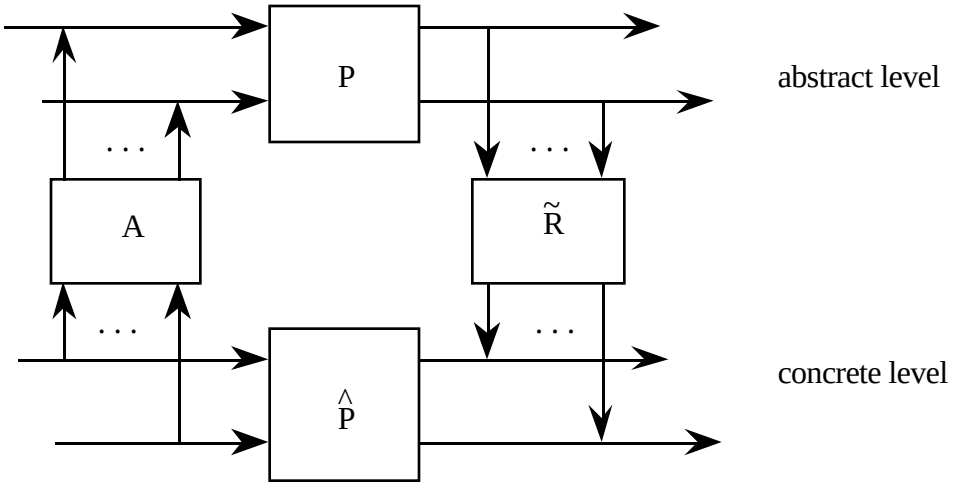

Fig. 9 U^{-1}-simulation

The introduced notion of interaction refinement is just a generalisation of notions of refinement and implementation as developed in the framework of algebraic specifications (cf. [Broy et al. 86]).

3.2 Refinement of the Glass Box View

In the glass box view we are not only interested in the observable behaviour of a system but also in its internal structure. In pure behaviour specifications nothing is said about the internal structure of a system. However, if we describe the observable behaviour of the system by

- a state-oriented description or
- a network description,

then assuming a glass box view the form of the description gives additional information about the representation and about the implementation of the system. Thus it is the *form* rather than the logical contents of a system description that represents the internal structure of a system.

3.2.1 State Oriented Specifications and their Refinement

Functional system specification techniques model the behaviour of systems in terms of system histories. Nevertheless, also such specifications can be written in a state-

oriented style. This is done by choosing an appropriate set State called the *state space*. Then we associate a specification P.σ with every state $\sigma \in$ State . In principle, any set State of sufficiently large cardinality can be used for representing the states of a system.

A straightforward state-oriented description uses the set of states to describe the behaviour of a component by a state transition function (for simplicity we treat only the case of a system with one input channel and one output channel)

$$\tau: (State \times M) \rightarrow \wp(State \times N^{\omega})$$

and a set of initial states Init $\subseteq$ State. Here M is the set of input messages and N is the set of output messages. The close relationship to the functional specification concepts can be shown as follows. Each state defines a specifying predicate

$$H: State \rightarrow ([M^{\omega} \rightarrow N^{\omega}] \rightarrow \mathbb{B})$$

by the following formula (we take the most liberal predicate H.s that fulfils the following formula)

$(H.s).f \equiv \forall\ m \in M: \exists\ s' \in State, f' \in [M^{\omega} \rightarrow N^{\omega}], y \in N^{\omega}:$
$(s', y) \in \tau(s, m) \wedge \forall\ x \in M^{\omega}: f(m\hat{}x) = y\hat{}f'(x) \wedge (H.s').f'$

This way every state is mapped onto a predicate characterizing a set of stream processing functions.

Based on this recursive specification of the predicate H we may obtain a component specification

$$P: [M^{\omega} \rightarrow N^{\omega}] \rightarrow \mathbb{B}$$

from the given state machine description by the following formula:

$$P.f = \exists\ s \in Init: (H.s).f.$$

Example: State Based Description of an Interactive Queue

A state based description of an interactive queue is given by choosing the state space State by

$$State = D^{*}$$

and the specification of the transition function τ for $\sigma \in D^{*}$, $m \in D$, as follows:

$$\tau(\sigma, m) = \{(\sigma\hat{}m, \langle\rangle)\}$$

$\tau(m\hat{}\sigma, ¿) = \{(\sigma, m)\}$

$\tau(\langle\rangle, ¿) = \{(\langle\rangle, ¿)\}$

We reformulate the specification QU' to the specification QS based on the state transition function.

QS.f ≡	(f = h.‹›) **where** h: $D^* \to [M^\omega \to M^\omega] \wedge \forall\, d \in D, q \in D^*, x \in M^\omega$:
	$(h.\langle\rangle).(¿\hat{}x) = ¿\hat{}(h.\langle\rangle).x$ $(h(d\hat{}q)).(¿\hat{}x) = d\hat{}(h.q).x$ $(h.q).(d\hat{}x) = (h(q\hat{}d)).x$

We may notice the uniform patterns of the equations for h. All equations are of the form

$$(h.\sigma).(m\hat{}x) = b\hat{}(h.\sigma').x$$

where σ denotes the given state and m denotes an input message. By b we denote the output message and by σ' the resulting state. We may represent the equations for h also by the following table:

σ	m	b	σ'
‹›	¿	¿	‹›
$d\hat{}q$	¿	d	q
q	d	‹›	$q\hat{}d$

It is straightforward to show QU' ⇔ QS. □

State-oriented descriptions of system components can be understood as a particular style of functional system specifications. State-oriented system specifications are often advocated, since an adequate choice of the state space may lead to a better understanding of the described system behaviour. On the other hand, the rather concrete, operational, single step oriented nature of state-based specifications can make the reasoning about systems more inconvenient.

The state transition functions as introduced above correspond to so-called I/O-automata (cf. [Lynch, Stark 89]). For I/O-automata a straightforward notion of equivalence (and of refinement) is obtained by observing input and output. This is not true for state transition system models without input and output. This type of machines needs a special notion of equivalence for defining refinements.

For state-oriented system specification we may also study the notion of state refinement. Technically speaking, a state refinement is given by a function

$$\rho: \text{State} \rightarrow \text{State}'$$

such that for all $\sigma \in$ State:

$$(H;^{\wedge}.\rho.\sigma).f \Rightarrow (H.\sigma).f$$

A state refinement is considered to be correct, if the corresponding component specification is a correct refinement of the given component specification. A state refinement can be used in system development to change the presentation of the states.

In [Lamport 83] the concept of stuttering is introduced. Roughly speaking, in a state-based approach by this concept two systems are considered equivalent, if they show the same state traces apart from finite repetitions of states. Stuttering is needed for an equivalence relation that is appropriate as a basis for refinements (see also [Abadi, Lamport 88]).

3.2.2 Distribution Refinement

The glass box view of a system can be described by giving a network of components that interact by exchanging messages. This way both a black box view (by abstracting from the network and just considering the external behaviour) and a glass box view (taking the network representation of the system as the description of its distributed realization) are provided.

When the system is represented by a network, of course, further *local* refinements of the subcomponents, their behaviour, their interaction, and their states are possible, leading to refinements of the system. This is due to the fact that in the functional approach the operators (sequential composition, parallel composition and feedback) are monotonic with respect to refinement: the refinement of a subcomponent always leads to a refinement of the whole system. In this sense the refinement concept is *compositional.* For a proof of the compositionality of interface interaction refinement see [Broy 92b].

A special case of distribution refinement replaces a state-oriented description of a system component by a network of subsystems that are again described in a state-oriented way, such that the collection of all states of the subcomponents can be understood as a refinement (in the sense of data structure refinement) of the initial state.

Considering the external (observable) behaviour (the black box view) there is no difference between a component that is represented by a single "sequential" program

(for instance by a state machine) and a component with its behaviour represented by a network of distributed interacting subsystems. However, there is a difference with respect to the glass box view. Replacing a component specification by a network description is called *distribution refinement.*

Example: Distribution Refinement: Queue again!
We refine the specification QU to a network described by the following recursive equation that can be understood as the recursive definition of a predicate and also as the recursive definition of a network:

$$QU = \mu^2((C \parallel QU);(I \parallel X))$$

where I is the specification of the identity function and X permutes its input lines. This formula describes a network that can be graphically represented by Fig. 10.

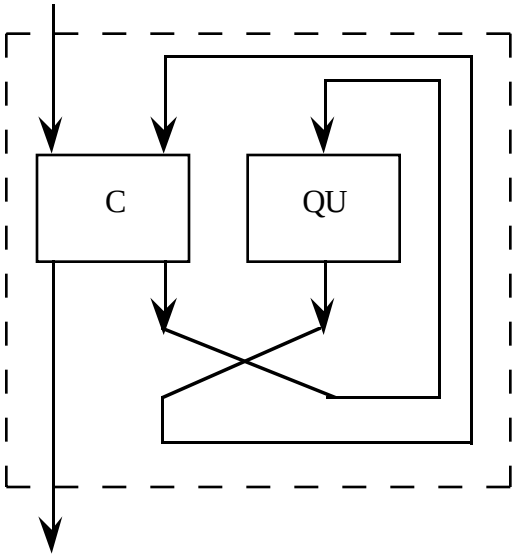

Fig. 10 Graphical Representation of the Distribution Refinement Component QU

The behaviour of C is defined as follows:

C.f ≡ (f = h.¿) **where** h: M → [(M^ω)^2 → (M^ω)^2] ∧ ∀ m, i ∈ M, x, z ∈ M^ω:

(h.m).(i^x, z) =
if i = ¿ ∧ m = ¿ **then** (¿, ‹›)^(h.m).(x, z)
▯ i ∈ D ∧ m = ¿ **then** (h.i).(x, z)
▯ i = ¿ ∧ m ∈ D **then** (m, ¿)^(h.ft.z).(x, rt.z)
▯ i ∈ D ∧ m ∈ D **then** (‹›, i)^(h.m).(x, z)
f i

In this specification the auxiliary function h is again specified in a state oriented style, namely by equations of the form

(h.m).(i^x, z) = b^(h.m').(x, z')

for which we give the following table:

m	i	b	m'	z'
¿	¿	(¿, ‹›)	¿	z
¿	d'	(‹›, ‹›)	d'	z
d	¿	(d, ¿)	ft.z	rt.z
d	d'	(‹›, d')	d	z

□

Distribution refinements are in particular difficult to prove correct, since in general complex forms of feedback have to be dealt with. A correctness proof for the distribution refinement given above can be found in [Broy 92a].

3.3 Refinement by Reformulation of the Specification

There are many different ways of describing the external (observable) behaviour (black box view) as well as the internal structure (glass box view) of system components. We may distinguish (among others) roughly the following different forms of behaviour descriptions:

- axiomatic, property-oriented descriptions,
- state-based descriptions,
- recursive predicate descriptions,
- assumption/commitment specifications,

- network descriptions.

These different ways of describing the behaviour of a system component are related with different methods for syntactically expressing refinements, and all these different methods can be formalized straightforwardly in our unifying functional framework.

As discussed in section 3.2, the form of description sometimes can be understood to give information about the glass box view of the component. Of course, one may talk more explicitly about the glass box view in descriptions of the form: "the system consists of particular subcomponents that are connected by particular channels." The more implicit way of describing the glass box view of a system by a state transition system or a network makes it necessary, however, to distinguish between following intensions when writing specifications:

- specifications that by their form describe system structures in the sense of the glass box view,
- specifications in terms of state transition systems or distributed systems that are used only for specification purposes and should not be understood as descriptions of the glass box view.

If a specification is given in terms of a set of axiomatic laws (for instance in the assumption/commitment format) and later it is reformulated by some recursive equations for the specifying predicate, this does not necessarily say anything about the glass box view. Even state-oriented specifications may be used not for implicitly describing a realization in the sense of the glass box view but rather as a specification concept. In this case we also speak of *abstract states* (cf. [Lamport 83]).

Often a component specification

$$P: [M_i;^n \rightarrow N_i;^m] \rightarrow \mathbb{B}$$

is described by a recursive equation for the predicate P (see the specification for P derived from a state machine description; for a more comprehensive treatment of recursion for specifications, see [Broy 92b]):

$$P.f \equiv \exists\, g_1, ..., g_k: \tau[g_1, ... , g_k] \sqsubseteq f \wedge P.g_1 \wedge ... \wedge P.g_k.$$

In general, we associate with such a recursive description the weakest predicate that fulfils this equation. For this form of definition we immediately obtain simple concepts of refinement.

Refinement by reformulation has to do with changing or refining the syntactic description of the behaviour of an agent. Two particularly interesting representations of the behaviour of agents are

- net representations,
- constructive representations.

Both representations can be characterized by their particular syntactic forms: a constructive representation of the behaviour of an agent

$$Q: [M_i;^n \rightarrow N_i;^m] \rightarrow \mathbb{B}$$

is of the form

(*) $Q = G$

where G is an expression formed of interactive component specifications that are again in constructive form (including possibly Q) or given as executable components. The equation (*) defines a net representation for Q, if G contains only net forming operations. Again the net representation is called constructive, if all the involved interactive component specifications are in constructive form.

4. Concluding Remarks

This study makes an attempt to classify various forms of refinements of system structures in a formal specification, refinement and verification framework. It is quite clear that in a development of a large system numerous refinement steps are needed. During the development the system will be described at several levels of abstraction such that more and more refined system versions are obtained.

We did not treat the methodological aspects of refinement at all in this study. A methodology gives advice at which levels of abstraction a system should be described during the development and which refinement steps should be tried in which order. Nevertheless, we claim that all the refinement steps that are needed in a method-oriented system development are covered by the refinement concepts described in this study (see [Broy et al. 92a]).

A flexible concept of refinement is the key to a formal method for the development of systems and their components. It is one of the decisive questions for the tractability of a specification and development method how easy it is to express and verify refinement. In this paper it has been shown how easily the different refinement concepts can be expressed and verified within the formalism of functional system specification techniques. The compositionality of the introduced concept of refinement is proved in [Broy 92b]. The correctness of most of the refinements given in this paper have been verified using the LARCH prover (cf. [Broy 92a]), which is an interactive verification system based on rewriting.

Acknowledgement

I thank my colleagues at the TU Munich for stimulating discussions. Frank Dederichs and Ketil Stølen did a careful reading of a version of the manuscript and provided helpful remarks.

Appendix A: Full Abstractness

Two specifications characterizing different sets of stream processing functions can, nevertheless, be understood to characterize the same behaviour of a component, if the difference between the two sets lies just in some superficial difference that is not observable in computations. To avoid such superficial differences we may restrict our considerations to so-called *fully abstract* predicates for specifications.

Fully abstract predicates (cf. [Broy 90]) do not present more information about an interface than necessary. A specification P is called *fully abstract*, if

$$(\forall\ x: \exists\ g: P.g \wedge f.x = g.x \wedge f\big|\{z: z \sqsubseteq x\} \sqsubseteq g\big|\{z: z \sqsubseteq x\}) \Rightarrow P.f.$$

In a fully abstract specification we do not have any representation of behavioural details in the specification that cannot be observed by appropriately chosen environments (for a justification of this choice see [Broy 90]).

The notion of fully abstract specification results from a concept of observability as induced by the basic operators for composing specifications. These operations are sequential and parallel composition as well as feedback. We introduce a predicate transformer

$$ABS: [M_i;^n \rightarrow N_i;^m] \rightarrow [M_i;^n \rightarrow N_i;^m]$$

mapping every specification $P \in [M_i;^n \rightarrow N_i;^m]$ onto its abstraction

$$(ABS.P).f \equiv \forall\ x: \exists\ g: P.g \wedge f.x = g.x \wedge f|\{z: z \sqsubseteq x\} \sqsubseteq g|\{z: z \sqsubseteq x\}$$

For abstract specifications we obtain

$$ABS.P = P$$

The use of fully abstract specifications has advantages when proving the equivalence of components' behaviours. Of course our choice of full abstractness is influenced by the operators that we consider.

Appendix B: Indefinite Representations

In communication history refinements we have considered representation specifications

$$R: [M_i;^n \rightarrow M;\tilde{}_i;^{n'}] \rightarrow \mathbb{B}$$

and abstraction specifications

$$A: [M;\tilde{}_i;^{n'} \rightarrow M_i;^n)] \rightarrow \mathbb{B}$$

such that

$$R ; A = I.$$

Let us now call such representation specifications and abstraction specifications *definite*, since they associate a unique abstraction with each representation of a communication history. However, there are cases where a representation specification may allow some loss of information in the sense that some different abstract communication histories $x, x' \in M_i;^n$ are represented by the same elements in $M;\tilde{}_i;^{n'}$. This may be appropriate, if not all the information given in x and x' is actually relevant.

A representation specification

$$R: [M_i;^n \rightarrow M;\tilde{}_i;^{n'}] \rightarrow \mathbb{B}$$

for which an abstraction specification A with

$$R ; A = I$$

does not exist is called *indefinite* representation specification. Similarly, a specification A is called indefinite abstraction specification, if there does not exist a specification R such that the above equation holds.

Not all interface interaction refinements have to be based on definite representation and abstraction specifications. Interface interaction refinements may also be of the form $(R ; P;^{\wedge} ; A)$ where R is not a definite representation specification and A is not a definite abstraction specification.

For instance, if the component interface specification P is nondeterministic and not injective we may take advantage of the nondeterminism contained in P and choose R and A in a more liberal, indefinite way. In particular, the specification R may identify all input histories that cannot lead to distinguishable output in P. The functions ρ with $R.\rho$ need not be injective, then. For input histories $x, x' \in M_i;^n$ we may allow $\rho.x = \rho.x'$, even if $x \neq x'$, provided

(a) there exists a function p with P.p such that p.x = p.x', or more generally

(b) there exist functions p and p' with P.p and P.p' such that p.x = p'.x'.

Of course, (a) is a special case of (b). Similarly, the abstraction specification A may be nondeterminate, provided the nondeterminism in A just corresponds to the nondeterminism in the component specification P.

Along these lines there are two aspects that lead to more freedom in choosing the representation specification for a given component specification P when looking for an interface interaction refinement $P;^{\wedge}$.

(1) if P does not distinguish between certain inputs x and x' (for instance if for all functions g with P.g we have g.x = g.x') then x and x' do not have to be represented by different elements,

(2) if P produces for some input x nondeterministically results g.x and g'.x (let g and g' be functions such that both P.g and P.g' hold) then g.x and g'.x may be represented by the same element in the output of $P;^{\wedge}$.

For explaining this more general notion of interaction refinement we study neutral elements for specifications with respect to sequential composition. Trivially, the identity specification is always neutral, since

$$(P ; I) = (I ; P) = P$$

Given a component specification P specifying a (possibly) nondeterministic behaviour, a function f is called *left-neutral* for P if for all functions g:

$$P.g \Rightarrow P(f;g)$$

A left-neutral function does change the behaviour of an interactive component just in a way that nondeterministic effects are rearranged. Obviously, the identity function is always left-neutral.

Given a specification P let LN(P) denote the specification where

$$LN(P).f \equiv \forall\, g: P.g \Rightarrow P(f;g).$$

Obviously the set {f: LN(P).f} is algebraically closed under function composition since the composition of left-neutrals leads to left-neutrals again. Moreover, since

$$I \Rightarrow LN(P)$$

we have

$$P = LN(P) ; P.$$

In analogy, a function f is called *right-neutral* for P if for all functions g:

$$P.g \Rightarrow P(g;f)$$

We write RN(P).f as abbreviation for the specification where

$$RN(P).f \equiv \forall\, g:\ P.g \Rightarrow P(g;f).$$

In particular, we have

$$P = LN(P)\ ;\ P\ ;\ RN(P)$$

Now let us come back to the concept of interface interaction refinement $P;^{\wedge}$ for P. Even if for the representation specification $R;^{\sim}$ a definite abstraction does not exist but just a specification $A;^{\sim}$ such that

$$R;^{\sim}\ ;\ A;^{\sim}\ \Rightarrow\ RN(P)$$

we obtain from

$$R\ ;\ P;^{\wedge}\ \Rightarrow\ P\ ;\ R;^{\sim}$$

the deduction

$$R\ ;\ P;^{\wedge}\ ;\ A;^{\sim} \qquad \Rightarrow$$
$$P\ ;\ R;^{\sim}\ ;\ A;^{\sim} \qquad \Rightarrow$$
$$P\ ;\ RN(P) \ \Rightarrow$$
$$P$$

So even if $R;^{\sim}$ is not a definite representation specification we can use it for interaction refinements of P. However, we cannot use it for interaction refinements of arbitrary specifications with the right syntactic interface as it is possible for definite representation specifications.

References

[Abadi, Lamport 88]
M. Abadi, L. Lamport: The Existence of Refinement Mappings. Digital Systems Research Center, SRC Report 29, August 1988

[Abadi, Lamport 90]
M. Abadi, L. Lamport: Composing Specifications. Digital Systems Research Center, SRC Report 66, October 1990

[Aceto, Hennessy 91]
L. Aceto, M. Hennessy: Adding Action Refinement to a Finite Process Algebra. Proc. ICALP 91, Lecture Notes in Computer Science 510, (1991), 506-519

[Back 88a]
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

[Back 88b]
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

[Broy et al. 86]
M. Broy, B. Möller, P. Pepper, M. Wirsing: Algebraic Implementations Preserve Program Correctness. Science of Computer Programming 8 (1986), 1-19

[Broy 90]
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

[Broy 92a]
M. Broy: Experiences with Machine Supported Software and System Specification and Verification: Using the Larch Prover. Digital Systems Research Center, SRC Report 93, 1992

[Broy 92b]
M. Broy: Compositional Refinement of Interactive Systems. Digital Systems Research Center, SRC Report 89, July 1992

[Broy et al. 92a]
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

[Broy et al. 92b]
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

[CIP 84]
M. Broy: Algebraic Methods for Program Construction: The Project CIP. SOFSEM 82, see also: P. Pepper (ed.): Program Transformation and Programming Environments. NATO ASI Series. Series F: 8. Berlin-Heidelberg-New York-Tokyo: Springer 1984, 199-222

[Coenen et al. 91]
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.

[Hoare 72]
C.A.R. Hoare: Proofs of Correctness of Data Representations. Acta Informatica 1, 1972, 271-281

[Janssen et al. 91]
W. Janssen, M. Poel, J. Zwiers: Action Systems and Action Refinement in the Development of Parallel Systems - An Algebraic Approach. Unpublished Manuscript

[Jones 86]
C.B. Jones: Systematic Program Development Using VDM. Prentice Hall 1986

[Lamport 83]
L. Lamport: Specifying concurrent program modules. ACM Toplas 5:2, April 1983, 190-222

[Lynch, Stark 89]
N. Lynch, E. Stark: A Proof of the Kahn Principle for Input/Output Automata. Information and Computation 82, 1989, 81-92

[Nipkow 86]
T. Nipkow: Non-deterministic Data Types: Models and Implementations. Acta Informatica 22, 1986, 629-661

The *syntactic interface* of an interactive component is described by its number and sorts of input channels and output channels. As indicated above the number and sorts of input channels and output channels may be changed freely. However, sometimes it appropriate to add or delete input and output channels in a simple and systematic way. A well-known concept from program transformation along this line is that of *embedding*. Given a component specification

$$P: [M_i;^n \rightarrow N_i;^m] \rightarrow \mathbb{B}$$

and a specification Q with $n \leq n'$, $m \leq m'$:

$$Q: [M_i;^{n'} \rightarrow N_i;^{m'}] \rightarrow \mathbb{B}$$

then Q is called an *embedding* of P, if there exists $z \in M_{n+1} \times ... \times M_{n'}$ such that for all f and g

$$Q.f \wedge (\forall\ x: \exists\ y: f(x \oplus z) = (g(x) \oplus y) \Rightarrow P.g.$$

In an embedding the refined component works with additional input streams. Embedding is a special case of interface interaction refinement. This can be seen by defining

$$R: [M_i;^n \rightarrow M;^{\sim}_i;^{n'}] \rightarrow \mathbb{B} \qquad A: [M;^{\sim}_i;^{n'} \rightarrow M_i;^n] \rightarrow \mathbb{B}$$

$$A;^{\frown}: [N;^{\sim}_i;^{m'} \rightarrow N_i;^m] \rightarrow \mathbb{B} \qquad R;^{\frown}: [N;^{\sim}_i;^{m'} \rightarrow N_i;^m] \rightarrow \mathbb{B}$$

as follows (let y(x) be the Skolem function in the existential quantification and let x[1:n] denote the subtuple $(x_1, ..., x_n)$, if $x = (x_1, ..., x_{n+k})$)

$$A.\alpha \equiv \forall\ x: \alpha.x = x[1:n] \qquad A;^{\frown}.\alpha \equiv \forall\ x: \alpha.x = x[1:m]$$

$$R.\rho \equiv \forall\ x: \rho.x = (x \oplus z) \qquad R;^{\frown}.\rho \equiv \forall\ x: \rho.x = (x \oplus y(x))$$

It is straightforward to show that $(\rho\ ;\ \alpha) = I$ for functions ρ and α with $R.\rho$ and $A.\alpha$ and similarly for $A;^{\frown}$, $R;^{\frown}$.