89

# Compositional Refinement of Interactive Systems

**Manfred Broy**

**July 15, 1992**

# Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in l984—their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

# Compositional Refinement of Interactive Systems

Manfred Broy

July 15, 1992

Manfred H. B. Broy is at the Institut für Informatik, Technische Universität München, Postfach 20 24 20, 8 München 2, Germany

E-mail: broy@informatik.tu-muenchen.de

The author was partially supported by the German Ministry of Research and Technology (BMFT) as part of the compound project "KORSO - Korrekte Software" and by the German Research Community (DFG) project SPECTRUM

**©Digital Equipment Corporation 1992**

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the Systems Research Center of Digital Equipment Corporation in Palo Alto, California; an acknowledgment of the authors and individual contributors to the work; and all applicable portions of the copyright notice. Copying, reproducing, or republishing for any other purpose shall require a license with payment of fee to the Systems Research Center. All rights reserved.

**Abstract**

We use functional specification techniques to describe systems and their components. We define the notions of property refinement and interaction refinement for interactive systems and their components. Interaction refinement allows changes to the syntactic interface (the number of channels and the sorts of messages on the channels) as well as the semantic interface (causality flow between messages and interaction granularity). We prove that these notions of refinement are compositional with respect to sequential and parallel composition, communication feedback, and recursive declarations of system components. These proofs demonstrate that refinements of networks can be accomplished in a modular way by refining their components. We generalize the notions of refinement to refining contexts. Finally, we define full abstraction for specifications and show compositionality with respect to this abstraction as well.

# Contents

# 1 Introduction

A distributed interactive system consists of a family of interacting components. To reduce complexity, they can be developed by a number of successive steps. In each step, the system is described in more detail and closer to an implementation level. We speak of levels of abstraction and of stepwise refinement in system development.

Logical implication provides a simple concept of stepwise refinement when logical specifications are used to describe the behavior of system components. A system component specification is a refinement of another specification if it exhibits all specified properties and possibly more. Refinement allows the replacement of system specifications by more refined ones exhibiting more specific properties.

More sophisticated notions of refinement allow the refinement of a system component to one exhibiting quite different properties than the original one. In this case, however, we need a concept relating the behaviors of the refined system component to behaviors of the original one such that behaviors of the refined system component can be understood to represent behaviors of the original. The behavior of interactive system components is basically given by their interaction with their environment. Therefore the refinement of system components basically has to deal with the refinement of their interaction. We will introduce such a notion of interaction refinement.

Concepts of refinement for software systems have been investigated since the early 1970s. Data structure refinement is treated in Hoare's pioneering paper [Hoare 72]. These ideas were further explored and developed (see, for instance, [Jones 86], [Broy et al. 86], [Sannella 88], see [Coenen et al. 91] for a survey). The idea of refining interacting systems has also been treated in numerous papers (see, for instance, [Lamport 83], [Abadi, Lamport 90], and [Back 90]).

Typically, distributed interactive systems are composed of a number of components that interact, for example, by exchanging messages or by updating shared memory. Various forms of composition allow the construction of systems from smaller ones. Parallel and sequential composition, communication feedback, and recursion are basic forms of composition for systems.

A method for specifying system components is called *compositional* (or *modular*) for a set of forms of composition if the specifications of composed systems can be derived from the specifications of the constituent components. We call a refinement concept *compositional*, if refinements of a composed system are obtained by giving refinements for the components. Traditionally, compositional notions of specification and refinement for concurrent systems are considered hard to obtain. For instance, the elegant approach of [Chandy, Misra 88] is not compositional with respect to liveness properties and does not provide a compositional notion of refinement.

Note that it only makes sense to talk about compositionality with respect to a set of forms of composition. Forms of composition of system components define an algebra of systems, also called a *process algebra*. Not all approaches to system specifications emphasize forms of composition for systems. For instance, in state machine oriented specifications, systems are modelled by state transitions. No particular forms of composition of system components are used. As a consequence compositionality is less significant there. Approaches that favor describing systems using forms of composition are called "algebraic". A discussion of the advantages and disadvantages of algebraic versus nonalgebraic approaches can be found, for instance, in [Janssen et al. 91].

Finding compositional specification methods and compositional interaction refinement concepts is difficult. Compositional refinement seems especially difficult to achieve for programming languages with tightly coupled parallelism, such as the "rendezvous" concept in CCS and CSP. In tightly coupled parallelism, the actions are used directly for the synchronization of parallel activities. Therefore the granularity of the actions cannot be refined, in general, without changing the synchronization structure (see, for instance, [Aceto, Hennessy 91] and [Vogler 91]).

The following sections present a compositional notion of refinement where the granularity of interaction can be refined. We use functional, purely descriptive, "nonoperational" specification techniques. The behavior of distributed systems interacting by communication over channels is represented by functions processing streams of messages. Streams of messages represent communication histories on channels. System component specifications are predicates characterizing sets of stream processing functions. System components described that way can be composed and decomposed using the above mentioned forms of composition such as sequential and parallel composition as well as communication feedback. With these forms of composition all kinds of finite data processing nets can be described. Allowing in addition recursive declarations even infinite data processing nets can be described.

In the following, concepts of refinement for interactive system components are defined that allow one to change both the number of channels of a component as well as the granularity of the messages sent by it. In particular, basic theorems are proved that show that our notion of refinement is compositional for the basic compositional forms as well as for recursive declarations. Accordingly for an arbitrary net of interacting components a refinement is schematically obtained by giving refinements for its components. The correctness of such a refinement follows according to the proved theorems schematically from the correctness proofs for the refinements of the components.

We give examples for illustrating the compositionality of refinement. We have deliberately chosen very simple examples to keep their specifications small such that we can concentrate on the refinement aspects. The simplicity of these examples does not mean that much more complex examples cannot be treated.

Finally we generalize our notion of refinement to refining contexts. Refining contexts allow refinements of components where the refined presentation of the input history may depend on the output history. In particular, this allows unreliable components to be understood as refinements of reliable components, as long as the refining context takes care of the unreliability. Refining contexts are represented by predicate transformers with special properties. We give examples for refining contexts.

An appendix treats full abstraction of functional specifications for these composing forms.

# 2 Specification

In this section we introduce the basic notions for functional system models and functional system specifications. In the following we study system components that exchange messages asynchronously via channels. A *stream* represents a communication history for a channel. A stream of messages over a given message set $M$ is a finite or infinite sequence of messages. We define

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

We briefly repeat the basic concepts from the theory of streams that we shall use later. More comprehensive explanations can be found in [Broy 90].

- By $x\frown y$ we denote the result of concatenating two streams $x$ and $y$. We assume that $x\frown y = x$, if $x$ is infinite.
- By $\langle\rangle$ we denote the empty stream.
- If a stream $x$ is a *prefix* of a stream $y$, we write $x \sqsubseteq y$. The relation $\sqsubseteq$ is called *prefix order*. It is formally specified by

$$x \sqsubseteq y \equiv_{df} \exists z \in M^{\omega} : x\frown z = y$$

- By $(M^{\omega})^{n}$ we denote *tuples* of $n$ streams. The prefix ordering on streams as well as the concatenation of streams is extended to tuples of streams by elementwise application.

A tuple of finite streams represents a partial communication history for a tuple of channels. A tuple of infinite streams represents a total communication history for a tuple of channels.

The behavior of deterministic interactive systems with n input channels and m output channels is modeled by $(n, m)$*-ary stream processing functions*

$$f : (M^{\omega})^{n} \to (M^{\omega})^{m}$$

A stream processing function determines the output history for a given communication history for the input channels in terms of tuples of streams.

**Example 1** *Stream processing function*

Let a set $D$ of data elements be given and let the set of messages $M$ be specified by:

$$M = D \cup \{?\}$$

Here the symbol ? is a signal representing a request. For data elements $d \in D$ a stream processing function

$$(c.d) : M^\omega \to M^\omega$$

is specified by

$$\begin{array}{rl} \forall e \in D, x \in M^\omega : & (c.d)(?\frown x) = d\frown ?\frown (c.d)(x) \\ \wedge & (c.d)(e\frown x) = e\frown (c.e)(x) \end{array}$$

The function $(c.d)$ describes the behavior of a simple storage cell that can store exactly one data element. Initially $d$ is stored. The behavior of the component modeled by $(c.d)$ can be illustrated by an example input

$$\begin{array}{rl} (c.d)( & ?\frown ?\frown d_1\frown ?\ \frown d_2\frown ?\ \frown d_3\frown d_4\frown ?\ \frown d_5\frown x) = \\ & d\frown d\frown d_1\frown d_1\frown d_2\frown d_2\frown d_3\frown d_4\frown d_4\frown d_5\frown (c.d_5).x \end{array}$$

The function $(c.d)$ is a simple example of a stream processing function where every input message triggers exactly one output message.

**End of example**

In the following we use some notions from domain and fixed point theory that are briefly listed:

- A stream processing function is called *prefix monotonic*, if for all tuples of streams $x, y \in (M^\omega)^n$ we have
$$x \sqsubseteq y \Rightarrow f.x \sqsubseteq f.y$$
We denote the function application $f(x)$ by $f.x$ to avoid brackets.
- By $\sqcup S$ we denote a least upper bound of a set $S$, if it exists.
- A set $S$ is called *directed*, if for any pair of elements $x$ and $y$ in $S$ there exists an upper bound of $x$ and $y$ in $S$.
- A partially ordered set is called *complete*, if every directed subset has a least upper bound.
- A stream processing function $f$ is called *prefix continuous*, if $f$ is prefix monotonic and for every directed set $S \subseteq M^\omega$ we have:
$$f. \sqcup S = \sqcup\{f.x : x \in S\}$$

The set of streams as well as the set of tuples of streams are complete. For every directed set of streams there exists a least upper bound.

We model the behavior of interactive system components by sets of continuous (and therefore by definition also monotonic) stream processing functions. Monotonicity models causality between input and output. Continuity models the fact that for every behavior the system's reaction to infinite input can be predicted from the component's reactions to all finite prefixes of this input[1]. Monotonicity takes care of the fact that in an interactive system output already produced cannot be changed when further input arrives. The empty stream is to be seen as representing the information "further communication unspecified". Note, in the example above by the preimposed monotonicity of the function $(c.d)$ we conclude $(c.d)(\langle\rangle) = \langle\rangle$; otherwise, we could construct a contradiction.

A specification describes a set of stream processing functions that represent the behaviors of the specified systems. If this set is empty, the specification is called *inconsistent*, otherwise it is called *consistent*. If the set contains exactly one element, then the specification is called *determined*. If this set has more then one element, then the specification is called *underdetermined* and we also speak of *underspecification*. As we shall see, an underdetermined specification may be refined into a determined one. An underdetermined specification can also be used to describe hardware or software units that are *nondeterministic*. An executable system is called *nondeterministic*, if it is underdetermined. Then the underspecification in the description of the behaviors of a nondeterministic system allows nondeterministic choices carried out during the execution of the system. In the descriptive modeling of interactive systems there is no difference in principle between underspecification und the operational notion of nondeterminism. In particular, it does not make any difference in such a framework, whether these nondeterministic choices are taken before the execution starts or step by step during the execution.

The set of all (n,m)-ary prefix continuous stream processing functions is denoted by

$$SPF_m^n$$

The number and sorts of input channels as well as output channels of a specification are called the component's *syntactic interface*. The behavior, represented by the set of functions that fulfill a specification, is called the component's *semantic interface*. The semantic interface includes in particular the granularity of the interaction and the causality between input and output. For simplicity we do not consider specific sort information for the individual channels of components in the following and just assume $M$ to be a set of messages. However, all our results carry over straightforwardly to stream processing functions where more specific sorts are attached to the individual channels.

[1] This does not exclude the specification of more elaborate liveness properties including fairness. Note, fairness is, in general, a property that has to do with "fair" choices between an infinite number of behaviors.

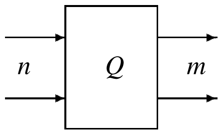


Figure 1: Graphical representation of a component $Q$

A *specification* of a possibly underdetermined interactive system component with n input channels and $m$ output channels is modeled by a predicate

$$Q : SPF_m^n \to Bool$$

characterizing prefix continuous stream processing functions. $Q$ is called an $(n, m)$-ary system's specification. A graphical representation of an $(n, m)$-ary system component $Q$ is given in Figure 1. The set of specifications of this form is denoted by

$$SPEC_m^n$$

**Example 2** *Specification*

A component called $C$ (for storage *Cell*) with just one input channel and one output channel is specified by the predicate $C$. The component $C$ can be seen as a simple store that can store exactly one data element. $C$ specifies functions $f$ of the functionality:

$$f : M^\omega \to M^\omega$$

Let the sets $D$ and $M$ be specified as in example 1. If $C$ receives a data element it sends a copy on its output channels. If it receives a request represented by the signal ?, it repeats its last data output followed by the signal ? to indicate that this is repeated output. The signal ? is used this way for indicating a "read storage content request". The signal ? triggers the *read* operation. A data element in the input stream changes the content of the store. The message $d$ triggers the *write* operation. Initially the cell carries an arbitrary data element. This behavior is formalized by the following specification for $C$:

$$C.f \equiv \exists d \in D : f = (c.d)$$

where the auxiliary function $(c.d)$ is specified as in example 1. Notice that the data element stored initially is not specified and thus component $C$ is underdetermined.

**End of example**

For a deterministic specification $Q$ where for exactly one function $q$ the predicate $Q$ is fulfilled, in other words where we have

$$Q.f \Leftrightarrow f = q$$

we often write (by misuse of notation) simply $q$ instead of $Q$. This way we identify determined specifications and their behaviors.

By $I_m \in SPF_m^m$ we denote the identity function; that is we assume

$$\forall x \in (M^\omega)^m : I_m.x = x$$

We shall drop the index $m$ for $I_m$ whenever it can be avoided without confusion.

By $\Omega_m^n \in SPF_m^n$ we denote the function that produces for every input just the empty stream as output on all its output channels; that is we define

$$\forall x \in (M^\omega)^n : \Omega_m^n.x = \langle\rangle^m$$

Similarly we write $\dagger^m$ for the unique function in $SPF_0^m$; in other words the function with m input channels, but with no output channels.

By $Ł_m^n \in SPEC_m^n$ we denote the logically *weakest* specification, which is the specification that is fulfilled by all stream processing functions. It is defined by

$$\forall f \in SPF_m^n : Ł_m^n.f$$

By $\overset{n}{\Upsilon}$ we denote the function that produces two copies of its input. We have $\overset{n}{\Upsilon} \in SPF_{2n}^n$ and

$$\forall x \in (M^\omega)^n : \overset{n}{\Upsilon}.x = (x, x)$$

By $\overset{n\,m}{\chi} \in SPF_{n+m}^{n+m}$ we denote the function that permutes its input streams as follows ( let $x \in (M^\omega)^n$, $y \in (M^\omega)^m$ ):

$$\overset{n\,m}{\chi}(x, y) = (y, x)$$

Again we shall drop the index $n$ as well as $m$ in $\Omega_m^n$, $Ł_m^n$, $\dagger^n$ and $\overset{n}{\Upsilon}$ whenever it can be avoided without confusion.

# 3 Composition

In this section we introduce the basic forms of composition namely sequential composition, parallel composition and feedback. These compositional forms are introduced for functions first and then extended to component specifications.

## 3.1 Composition of Functions

Given functions

$$f \in SPF_k^n,\ g \in SPF_m^k$$

we write

$$f; g$$

for the *sequential composition* of the functions $f$ and $g$ which yields a function in $SPF_m^n$ where

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

Given functions

$$f \in SPF_{m1}^{n1},\ g \in SPF_{m2}^{n2}$$

we write

$$f \| g$$

for the *parallel composition* of the functions $f$ and $g$ which yields a function in $SPF_{m1+m2}^{n1+n2}$ where (let $x \in (M^\omega)^{n1}$, $y \in (M^\omega)^{n2}$):

$$(f\|g).(x, y) = (f.x, g.y)$$

We assume that " ; " has higher precedence than "$\|$". Given a function

$$f \in SPF_m^{n+m}$$

we write

$$\mu f$$

for the *feedback* of the output streams of function $f$ to its input channels which yields a function in $SPF_m^n$ where

$$(\mu f).x = fix.\lambda\ y : f(x, y)$$

Here $fix$ denotes the fixed point operator associating with any monotonic function $f$ its least fixed point $fix.f$. Thus $y = (\mu f).x$ means that $y$ is the least solution (with respect to the prefix ordering) of the equation $y = f(x, y)$. We assume that "$\mu$" has higher precedence than the binary operators ";" and "$\|$". A graphical representation for feedback is given in Figure 2.

We obtain a number of useful rules by the fixed point definition of $\mu f$. As a simple consequence of the fixed point characterization, we get the unfold rules:

$$\mu f = \Upsilon; (I\|\mu f); f$$

$$\mu f = \Upsilon; \mu((I\|f); f)$$

A graphical representation of the unfold rules for feedback is given in Figure 3.

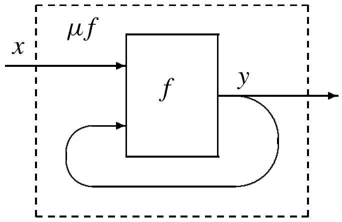


Figure 2: Graphical representation of feedback

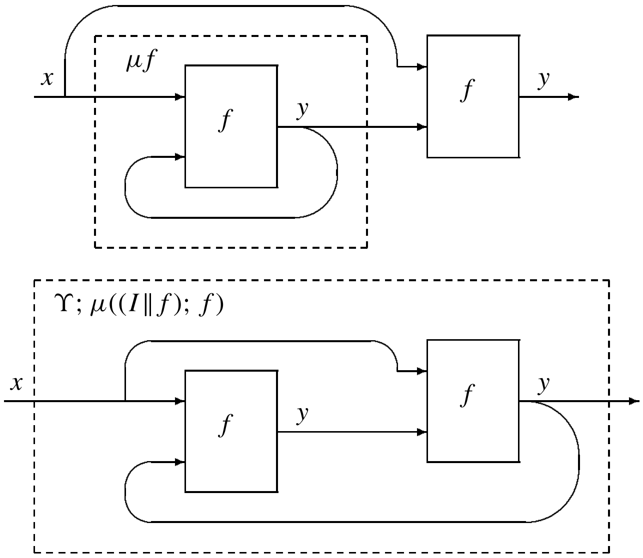


Figure 3: Graphical representation of the unfold rules for feedback

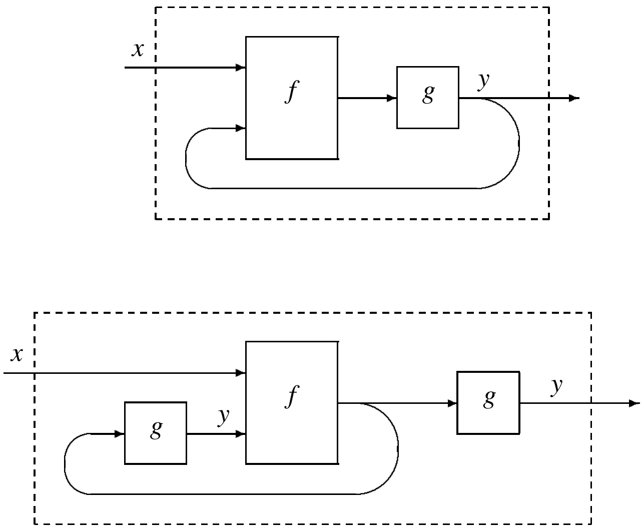


Figure 4: Graphical representation of semiunfold

A useful rule for feedback is *semiunfold* that allows one to move components outside or inside the feedback loop (let $g \in SPF_m^m$):

$$\mu(f; g) = \mu((I \| g); f); g$$

A graphical representation for semiunfold is given in Figure 4.

For reasoning about feedback loops and fixed points the following special case of semiunfold is often useful:

$$fix.\lambda\ y : m\frown f(x, y) = m\frown fix.\lambda\ y : f(x, m\frown y)$$

The rule is an instance of semiunfold with $g = \lambda\ y : m\frown y$. The correctness of this rule can also be seen by the following argument: if $y$ is the least fixed point of

$$\lambda\ y : m\frown f(x, y)$$

and $\widetilde{y}$ is the least fixed point of

$$\lambda\ \widetilde{y} : f(x, \widetilde{y})$$

then $\widetilde{y} = m\frown y$ and thus

$$y = m\frown\lambda\ y : f(x, m\frown y)$$

Semiunfold is a powerful rule when reasoning about results of feedback loops.

## 3.2 Composition of Specifications

We want to compose specifications of components to networks. Each form of composition introduced for functions can be extended to component specifications in a straightforward way. Given component specifications

$$Q \in SPEC_k^n,\ R \in SPEC_m^k$$

we write

$$Q;\ R$$

for the predicate in $SPEC_m^n$ where

$$(Q;\ R).f \Leftrightarrow \exists q, r : Q.q \wedge R.r \wedge f = q; r$$

Trivially we have for all specifications $Q \in SPEC_m^n$ the following equations:

$$Q;\ I = Q$$

$$I;\ Q = Q$$

$$Q;\ \dagger^m = \dagger^n$$

Given specifications

$$Q \in SPEC_{m1}^{n1},\ R \in SPEC_{m2}^{n2}$$

we write

$$Q \| R$$

for the predicate in $SPEC_{m1+m2}^{n1+n2}$ where

$$(Q \| R).f \Leftrightarrow \exists q, r : Q.q \wedge R.r \wedge f = q \| r$$

Given specification

$$Q \in SPEC_m^{n+m}$$

we write

$$\mu Q$$

for the predicate in $SPEC_m^n$ where

$$(\mu Q).f \Leftrightarrow \exists q : Q.q \wedge f = \mu q$$

For feedback over underdetermined specifications we get the following rules[2]:

$$\mu Q \Rightarrow \Upsilon;\ (I \| \mu Q);\ Q$$

[2]For determined system specifications $Q$ we get the stronger rules $\mu Q = \Upsilon; (I \| \mu Q); Q$ and $\mu Q = \Upsilon; \mu((I \| Q); Q)$ which do not hold for underdetermined systems, in general. The erroneous assumption that these rules are valid also for underdetermined systems is the source for the merge anomaly (see [Brock, Ackermann 81]).

$$\mu Q \Rightarrow \Upsilon; \mu((I \| Q); Q)$$

A useful rule for feedback is *fusion* that allows one to move components that are not affected by the feedback outside or inside the feedback operator application. Let $R \in SPEC_n^k$:

$$R; \mu Q = \mu((R \| I); Q)$$

$$\mu((Q \| \dagger^m); (I \| R)) = \mu(Q); (I \| R)$$

With the help of the basic functions and the forms of composition introduced so far we can represent all kinds of finite networks of systems (data flow nets)[3]. Our composing forms lead to an algebra of system descriptions.

# 4 Refinement, Representation, Abstraction

In this section we introduce concepts of refinement for system components both with respect to the properties of their behaviors as well as with respect to their syntactic interface and granularity of interaction.

We start by defining a straightforward notion of property refinement for system component specifications. Then we introduce a notion of refinement for communication histories. Based on this notion we define the concept of interaction refinement for interactive components. This notion allows refining a component by changing the number of input and output channels as well as the granularity of the exchanged messages.

## 4.1 Property Refinement

Specifications are predicates characterizing functions. This leads to a simple notion of refinement of component specifications by adding logical properties.
Given specifications

$$Q, \widetilde{Q} \in SPEC_m^n$$

*$\widetilde{Q}$ is called a (property) refinement of $Q$*
if for all $f \in SPF_m^n$:

$$\widetilde{Q}.f \Rightarrow Q.f$$

Then we write

$$\widetilde{Q} \Rightarrow Q$$

If $\widetilde{Q}$ is a property refinement for $Q$, then $\widetilde{Q}$ has all the properties $Q$ has and may be some more. Every behavior that $\widetilde{Q}$ shows is also a possible behavior of $Q$.

[3]Of course, our combinatorial style for defining networks is not always very useful, in practice, since the combinatorial formulas are hard to read. However, we prefer throughout this report to work with these combinatorial formulas, since this puts emphasis on the compositional forms and the structure of composition. For practical purposes a notation with named channels is often more adequate.

All considered composing forms are monotonic for the refinement relation as indicated by the following theorem.

**Theorem 1 (Compositionality of Refinement)**

$$(\widetilde{Q}_1 \Rightarrow Q_1) \wedge (\widetilde{Q}_2 \Rightarrow Q_2) \Rightarrow (\widetilde{Q}_1 ; \widetilde{Q}_2 \Rightarrow Q_1 ; Q_2)$$

$$(\widetilde{Q}_1 \Rightarrow Q_1) \wedge (\widetilde{Q}_2 \Rightarrow Q_2) \Rightarrow (\widetilde{Q}_1 \| \widetilde{Q}_2 \Rightarrow Q_1 \| Q_2)$$

$$(\widetilde{Q} \Rightarrow Q) \Rightarrow (\mu \widetilde{Q} \Rightarrow \mu Q)$$

**Proof**: Straightforward, since all operators for specifications are defined pointwise on the sets of functions that are specified.

□

A simple example of a property refinement is obtained for the component $C$ as described in Example 2 on page 8 if we add properties about the data element initially stored in the cell. A property refinement does not allow one to change the syntactic interface of a component, however.

## 4.2 Interaction Refinement

Recall from section 2 that streams model communication histories on channels. In more sophisticated development steps for a component the number of channels and the sorts of messages on channels are changed. Such steps do not represent property refinements. Therefore we introduce a more general notion of refinement. To be able to do this we study concepts of representation of communication histories on $n$ channels modeled by a tuple of $n$ streams by communication histories on $m$ channels modeled by a tuple of $m$ streams.

Tuples of streams $y \in (M^\omega)^m$ can be seen as *representations* of tuples of streams $x \in (M^\omega)^n$, if we introduce a mapping $\rho \in SPF_m^n$ that associates with every $x$ its representation. $\rho$ is called a *representation function*. If $\rho$ is injective then it is called a *definite* representation function. Note, a mapping $\rho$ is injective, if and only if:

$$\forall x, \overline{x} : \rho.x = \rho.\overline{x} \Rightarrow x = \overline{x}$$

If a specification $R \in SPEC_m^n$ is used for the specification of a set of representation functions, $R$ is called a *representation specification*.

**Example 3** *Representation Specification*

We give a representation specification $R$ that allows streams of data elements and requests to be represented by two separate streams, one of which carries the requests and the other of which carries the data elements. The representation functions are mappings $\rho$ of the following functionality:

$$\rho : M^\omega \to \{?, \surd\}^\omega \times (D \cup \{\surd\})^\omega$$

Here $\surd$ is used as a separator signal. It can be understood as a *time tick* that separates messages. Given streams $x$ and $y$ let $[x, y]$ denote a pair of streams and $[x, y]\frown[\widetilde{x}, \widetilde{y}]$ the elementwise concatenation of pairs of streams, in other words:

$$[x, y]\frown[\widetilde{x}, \widetilde{y}] = [x\frown\widetilde{x}, y\frown\widetilde{y}]$$

Let $Ticks$ be defined by the set of pairs of streams of ticks that have equal length:

$$Ticks = \{[\surd^k, \surd^k] : k \in \mathbb{N}\}$$

We specify the representation specification $R$ explicitly as follows:

$$\begin{array}{llll} R.\rho \equiv \forall d \in D, x \in M^\omega : & & \exists t \in Ticks : \rho(?\frown x) & = t\frown[?, \langle\rangle]\frown\rho.x \\ & \wedge & \exists t \in Ticks : \rho(d\frown x) & = t\frown[\surd, d\frown\surd]\frown\rho.x \end{array}$$

Note, by the monotonicity of the specified functions:

$$R.\rho \Rightarrow \exists t \in Ticks : \rho.\langle\rangle = t$$

The computation of a representation is illustrated by the following example:

$$\begin{array}{l} \rho(?\frown?\frown d_1\frown \quad ?\frown d_2\frown \quad ?\frown d_3\frown \quad x) = \\ [\ \ ?\frown?\frown \quad \surd\frown?\frown \quad \surd\frown?\frown \quad \frown\surd\ , \\ \qquad\quad d_1\frown\surd\frown \quad d_2\frown\surd\frown \quad d_3\frown\surd]\frown\rho(x) \end{array}$$

The example demonstrates how the time ticks are used to indicate in the streams $\rho(x)$ the order of the requests relatively to the data messages in the original stream $x$.

**End of example**

The elements in the images of the functions $\rho$ with $R.\rho$ are called *representations*.

**Definition 1 (Definite representation specification)** *A representation specification $R$ is called* definite*, if*

$$\forall x, \overline{x}, \rho, \overline{\rho} : R.\rho \wedge R.\overline{\rho} \wedge \rho.x = \overline{\rho}.\overline{x} \Rightarrow x = \overline{x}$$

*In other words $R$ is definite, if different streams $x$ are always differently represented.*

Obviously, if $R$ is a definite representation specification, then all functions $\rho$ with $R.\rho$ are definite. For definite representation specifications for elements $x$ and $\overline{x}$ with $x \neq \overline{x}$

the sets of representation elements $\{\rho.x : R.\rho\}$ and $\{\rho.\overline{x} : R.\rho\}$ are disjoint. Note, the representation specification given in the example above is definite.

For every injective function, and thus for every definite representation function $\rho$, there exists a function $\alpha \in SPF_n^m$ such that:

$$\rho; \alpha = I$$

The function $\alpha$ is an inverse to $\rho$ on the image of $\rho$. The function $\alpha$ is called an *abstraction for* $\rho$. Notice that $\alpha$ is not uniquely determined if $\rho$ is not surjective. In other words, the elements in $(M^\omega)^m$ are not all used as representations of elements in $(M^\omega)^n$ there may be several functions $\alpha$ with $A.\alpha$, as defined below.

The concept of abstractions for definite representation functions can be extended to definite representation specifications.

**Definition 2 (Abstraction function)** *Let $R \in SPEC_m^n$ be a definite representation specification; a function $\alpha \in SPF_n^m$ with*

$$R; \alpha = I$$

*is called an* abstraction function for $R$.

The existence of abstractions follows from the definition of definite representation specification. Again for definite representation specifications the abstraction functions $\alpha$ are uniquely determined only on the image of $R$, that is on the union of the images of functions $\rho$ with $R.\rho$.

**Definition 3 (Abstraction for a definite representation specification)** *Let $A \in SPEC_n^m$ be the specification with*

$$A.\alpha \Leftrightarrow R; \alpha = I$$

*Then $A$ is called the* abstraction for $R$.

For consistent definite representation specifications $R$ with abstraction $A$ we have

$$R; A = I$$

If $\rho; A = I \Rightarrow R.\rho$ then $R$ contains all possible choices of representation functions for the abstraction $A$.

**Example 4** *Abstraction*

For the representation specification $R$ described in example 3 the abstraction functions $\alpha$ are mappings of the functionality:

$$\alpha : \{?, \surd\}^\omega \times (D \cup \{\surd\})^\omega \to M^\omega$$

The specification of $A$ reads as follows.

$$A.\alpha \equiv \forall d \in D, x \in \{?, \surd\}^\omega, y \in (D \cup \{\surd\})^\omega :$$

$$\begin{array}{rrl} & \alpha(?^\frown x, y) & = ?^\frown \alpha(x, y) \\ \wedge & \alpha(\surd^\frown x, \surd^\frown y) & = \alpha(x, y) \\ \wedge & \alpha(\surd^\frown x, d^\frown \surd^\frown y) & = d^\frown \alpha(x, y) \end{array}$$

It is a straightforward rewriting proof that indeed:

$$R; A = I$$

The specification $A$ shows a considerable amount of underspecification, since not all pairs of streams in $\{?, \surd\}^\omega \times (D \cup \{\surd\})^\omega$ are used as representations.

**End of example**

Parallel and sequential composition of definite representations leads to definite representations again.

**Theorem 2** *Let $R_i \in SPEC_{m_i}^{n_i}$ be definite representation specifications for $i = 1, 2$; then*

$$R_1 \| R_2$$

$$R_1; R_2$$

*(assuming $m_1 = n_2$ in the second formula) are definite representation specifications.*

**Proof**: Sequential and parallel composition of injective functions leads to injective functions.

□

Trivially we can obtain the abstractions of the composed representations by composing the abstractions.

For many applications, representation specifications are neither required to be determined nor even definite. For an indefinite representation specification sets of representation elements for different elements are not necessarily disjoint. Certain representation elements $y$ do occur in several sets of representations for elements. They ambiguously stand for ("represent") different elements. Such an element may represent the streams $x$ as well as $\overline{x}$, if $\rho.x = \overline{\rho}.\overline{x}$ for functions $\rho$ and $\overline{\rho}$ with $R.\rho$ and $R.\overline{\rho}$. For indefinite representation specifications the represented elements are not uniquely determined by the representation elements. A representation element $y$ stands for the set

$$\{x : \exists \rho : R.\rho \wedge \rho.x = y\}$$

For a definite representation specification $R$ this set contains exactly one element while for an indefinite representation specification $R$ this set may contain more than one element. In the latter case, of course, abstraction functions $\alpha$ with $R; \alpha = I$ do not exist.

However, even for certain indefinite representations we can introduce the concept of an abstraction.

**Definition 4 (Uniform representation specifications)** *A consistent specification $R \in SPEC_m^n$ is called a* uniform representation specification*, if there exists a specification $A \in SPEC_n^m$ such that for all $\rho$:*

$$R.\rho \Rightarrow \; R; A; \rho = \rho$$

*The specification $A$ is called again the* abstraction *for $R$.*

The formula expresses that $(R; A)$ is a left-neutral element for every representation function in $R$. Essentially the existence of an abstraction expresses the following property of $R$: if for different elements $x$ and $\overline{x}$ the same representations are possible, then every representation function maps these elements onto equal representations. More formally stated, if there exist functions $\widetilde{\rho}$ and $\overline{\rho}$ with $R.\widetilde{\rho}$ and $R.\overline{\rho}$ such that

$$\widetilde{\rho}.x = \overline{\rho}.\overline{x}$$

then for all functions $\rho$ with $R.\rho$:

$$\rho.x = \rho.\overline{x}$$

Thus if elements are identified by some representation functions, this identification is present in all representation functions. The same amount of information is "forgotten" by all the representations. The representation functions then are indefinite in a uniform way. Definite representations are always uniform.

A function is *injective*, if for all $x$ and $\overline{x}$ we have:

$$\rho.x = \rho.\overline{x} \Rightarrow x = \overline{x}$$

A function that is not injective $\rho$ defines a nontrivial *partition* on its domain. A representation specification is uniform if and only if all functions $\rho$ with $R.\rho$ define the same partition.

For a uniform representation specification $R$ with abstraction $A$ the product $(R; A)$ reflects the underspecification in the choices of the representations provided by $R$. If for a function $\gamma$ with $(R; A).\gamma$ we have $x = \gamma.\overline{x}$, then $x$ and $\overline{x}$ have the same representations.

**Definition 5 (Adequate representation)** *A uniform representation specification $R$ with abstraction $A$ is called* adequate *for a specification $Q$, if:*

$$Q; R; A \Rightarrow Q$$

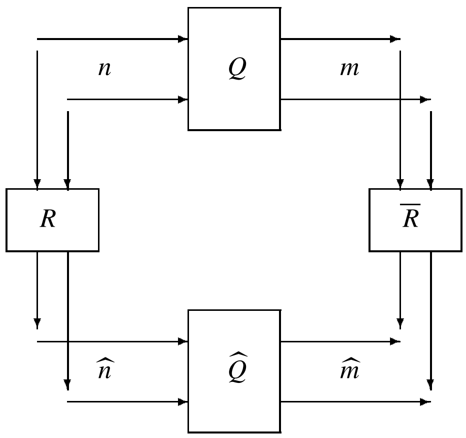


Figure 5: Commuting diagram of interaction refinement

Adequacy means that the underspecification in $(R; A)$ does not introduce more underspecification into $Q; R; A$ than was already present in $Q$. Note, definite representations are adequate for all specifications $Q$.

**Definition 6 (Interaction refinement)** *Given representations $R \in SPEC^{n}_{\hat{n}}$, $\overline{R} \in SPEC^{m}_{\hat{m}}$ and specifications $\widehat{Q} \in SPEC^{\hat{n}}_{\hat{m}}$, $Q \in SPEC^{n}_{m}$ we say that $\widehat{Q}$ is an* interaction refinement *of $Q$ for the representation specifications $R$ and $\overline{R}$, if*

$$R; \widehat{Q} \Rightarrow Q; \overline{R}$$

This definition indicates that we can replace via an interaction refinement a system of the form $Q; \overline{R}$ by a refined system of the form $R; \widehat{Q}$. We may think about the relationship between $Q$ and $\widehat{Q}$ as follows: the specification $Q$ specifies a component on a more abstract level while $\widehat{Q}$ gives a specification for the component at a more concrete level. Instead of computing at the abstract level with $Q$ and then translating the output via $\overline{R}$ onto the output representation level, we may translate the input by $R$ onto the input representation level and compute with $\widehat{Q}$. We obtain one of these famous commuting diagrams as shown in Figure 5.

**Definition 7 (Adequate interaction refinement)** *The interaction refinement of $Q$ for the representation specifications $R$ and $\overline{R}$ is called* adequate *for a specification $Q$, if $\overline{R}$ is adequate for $Q$.*

For adequate interaction refinements using uniform representation specifications $\overline{R}$ with abstraction $\overline{A} \in SPEC^{m}_{i}$, we obtain

$$R; \widehat{Q}; \overline{A} \Rightarrow Q$$

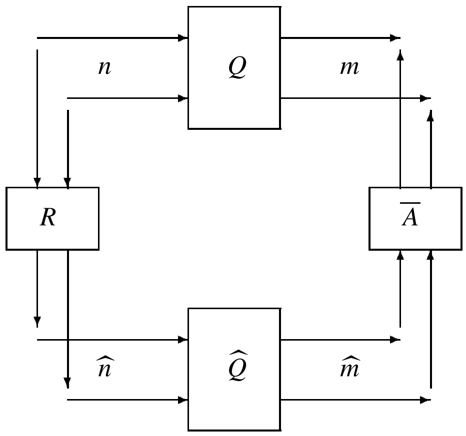


Figure 6: Commuting diagram of interaction refinement

since from the interaction refinement property we get

$$R;\ \widehat{Q};\ \overline{A} \Rightarrow\ Q;\ \overline{R};\ \overline{A}$$

and by the adequacy of $\overline{R}$ for $Q$

$$Q;\ \overline{R};\ \overline{A} \Rightarrow\ Q$$

which shows that $R;\ \widehat{Q};\ \overline{A}$ is a (property) refinement of $Q$. A graphical illustration of adequate interaction refinement is shown in Figure 6.

The following table summarizes the most important definitions so far.

| Table of definitions | |
|---|---|
| $\widetilde{Q}$ property refinement of $Q$ | $\widetilde{Q}.f \Rightarrow Q.f$ |
| $R$ consistent, definite with abstr. $A$ | $R;\ A = I$ |
| $R$ uniform with abstraction $A$ | $R.\rho \Rightarrow R;\ A;\ \rho = \rho$ |
| $R$ adequate for $Q$ with abs. $A$ | $Q;\ R;\ A \Rightarrow Q$ |
| Inter. refinement $\widehat{Q}$ of $Q$ for $R$, $\overline{R}$ | $R;\ \widehat{Q} \Rightarrow Q;\ \overline{R}$ |
| Adequate inter. refinement | $\overline{R}$ uniform and adequate for $Q$ |

The notion of interaction refinement allows one to change both the syntactic and the semantic interface. The syntactic interface is determined by the number and sorts of channels; the semantic interface is determined by the behavior of the component represented by the causality between input and output and by the granularity of the interaction.

**Example 5** *Interaction Refinement*

We refine the component $C$ as given in Example 2 into a component $\widehat{C}$ that has instead of one input and one output channel two input and two output channels. The refinement $\widehat{C}$ uses one of its channels carrying the signal ? as a read channel and one of its channels carrying data as a write channel. Let $R$ and $A$ be given as specified in the examples above

We specify the interaction refinement $\widehat{C}$ of $C$ explicitly. $\widehat{C}$ specifies functions of functionality:

$$f : \{?, \surd\}^\omega \times (D \cup \{\surd\})^\omega \to \{?, \surd\}^\omega \times (D \cup \{\surd\})^\omega$$

We specify:

$$\widehat{C}.f = \exists d \in D : f = h.d$$

where the auxiliary function $h$ is specified by:

$$h : D \to (\{?, \surd\}^\omega \times (D \cup \{\surd\})^\omega \to \{?, \surd\}^\omega \times (D \cup \{\surd\})^\omega)$$

$\forall d, e \in D, x \in \{?, \surd\}^\omega, y \in (D \cup \{\surd\})^\omega$ :

$$\begin{array}{rll} & (h.d)(?\frown x, y) & = [\surd\frown ?, d\frown\surd]\frown(h.d)(x, y) \\ \wedge & (h.d)(\surd\frown x, \surd\frown y) & = [\surd, \surd]\frown(h.d)(x, y) \\ \wedge & (h.e)(\surd\frown x, d\frown\surd\frown y) & = [\surd, d\frown\surd]\frown(h.d)(x, y) \end{array}$$

It is a straightforward proof to show:

$$R; \widehat{C} \Rightarrow C; R$$

Assume $\rho$ with $R.\rho$ and $h$ such that there exist $f$ and $d$ with $\widehat{C}.f$ and $f = h.d$; we prove by induction on the length of the stream $x$ that there exist $\widetilde{\rho}$ with $R.\widetilde{\rho}$ and $c.d$ as specified in example 1 such that:

$$(h.d).\rho.x = \widetilde{\rho}.(c.d).x$$

For $x = \langle\rangle$ we obtain: there exists $t \in Ticks$ such that:

$$\begin{array}{l} (h.d).\rho.x = \\ (h.d).t = \\ t = \\ \widetilde{\rho}.x = \\ \widetilde{\rho}.(c.d).x \end{array}$$

Now assume the hypothesis holds for $x$; there exists $t \in Ticks$:

$$\begin{array}{l}(h.d).\rho(?^\frown x) = \\ (h.d)(t^\frown[?, \langle\rangle]^\frown \rho.x) = \\ t^\frown[\surd^\frown ?, d^\frown \surd]^\frown (h.d).\rho.x = \\ \widetilde{\rho}(d^\frown (c.d).x) = \\ \widetilde{\rho}(c.d)(?^\frown x)\end{array}$$

There exists $t \in Ticks$:

$$\begin{array}{l}(h.e).\rho(d^\frown x) = \\ (h.e)(t^\frown[\surd, d^\frown \surd]^\frown \rho.x) = \\ t^\frown[\surd, d^\frown \surd]^\frown (h.d).\rho.x = \\ \widetilde{\rho}(d^\frown (c.d).x) = \\ \widetilde{\rho}(c.e)(d^\frown x)\end{array}$$

This concludes the proof for finite streams $x$. By the continuity of $h$ and $\rho$ the proof is extended to infinite $x$.

**End of example**

Continuing with the system development after an adequate interaction refinement of a component we may decide to leave $R$ and $\overline{A}$ unchanged and carry on by just further refining $\widehat{Q}$.

# 5 Compositionality of Interaction Refinement

Large nets of interacting components can be constructed by our forms of composition. When refining such large nets it is decisive for keeping the work manageable that interaction refinements of the components lead to interaction refinements of the composed system.

In the following we prove that interaction refinement is indeed compositional for sequential and parallel composition and for communication feedback.

## 5.1 Sequential and Parallel Composition

For systems composed by sequential compositions, refinements can be constructed by refining their components.

**Theorem 3 (Compositionality of refinement, seq. composition)** *Assume $\widehat{Q}_i$ is an interaction refinement of $Q_i$ for the representations $R_{i-1}$ and $R_i$ for $i = 1, 2$, then $\widehat{Q}_1; \widehat{Q}_2$ is an interaction refinement of $Q_1; Q_2$ for the representations $R_0$ and $R_2$.*

**Proof**: A straightforward derivation shows the theorem:

$$
\begin{array}{lll}
R_0;\ \widehat{Q}_1;\ \widehat{Q}_2 & \Rightarrow & \{\text{monotonicity of “;”, } \widehat{Q}_1 \text{ interaction refinement of } Q_1\} \\
Q_1;\ R_1;\ \widehat{Q}_2 & \Rightarrow & \{\text{monotonicity of “;”, } \widehat{Q}_2 \text{ interaction refinement of } Q_2\} \\
Q_1;\ Q_2;\ R_2 & &
\end{array}
$$

□

**Example 6** *Compositionality of Refinement for Sequential Composition*

Let $C$ and $\widehat{C}$ be specified as in the example above. Of course, we may compose $C$ as well as $\widehat{C}$ sequentially. We define the components $CC$ and $\widehat{CC}$ by:

$$CC =_{df} C;\ C$$

$$\widehat{CC} =_{df} \widehat{C};\ \widehat{C}$$

Note, $CC$ is a cell that repeats its last input twice on a signal ?. It is a straightforward application of our theorem of the compositionality of refinement that $\widehat{CC}$ is a refinement of $CC$ :

$$R;\ \widehat{CC} \Rightarrow CC;\ R$$

Of course, since $R;\ A = I$ we also have that $R;\ \widehat{CC};\ A$ is a property refinement of $CC$.

**End of example**

Refinement is compositional for parallel composition, too.

**Theorem 4 (Compositionality of refinement for parallel composition)** *Assume $\widehat{Q}_i$ is an interaction refinement of $Q_i$ for the representations $R_i$ and $\overline{R}_i$ for $i = 1, 2$ then $\widehat{Q}_1 \| \widehat{Q}_2$ is an interaction refinement of $Q_1 \| Q_2$ for the representations $R_1 \| R_2$ and $\overline{R}_1 \| \overline{R}_2$.*

**Proof**: A straightforward derivation shows the theorem:

$$
\begin{array}{lll}
(R_1 \| R_2);\ (\widehat{Q}_1 \| \widehat{Q}_2) & = & \{\text{rule for sequential and parallel composition}\} \\
(R_1;\ \widehat{Q}_1) \| (R_2;\ \widehat{Q}_2) & \Rightarrow & \{\widehat{Q}_i \text{ interaction refinement for } Q_i\} \\
(Q_1;\ \overline{R}_1) \| (Q_2;\ \overline{R}_2) & = & \{\text{rule for sequential and parallel composition}\} \\
(Q_1 \| Q_2);\ (\overline{R}_1 \| \overline{R}_2) & &
\end{array}
$$

□

For sequential and parallel composition compositionality of refinement is quite straightforward. This can be seen from the simplicity of the proofs.

## 5.2 Feedback

For the feedback operator, refinement is not immediately compositional. We do not obtain, in general, that $\mu\widehat{Q}$ is an interaction refinement of $\mu Q$ for the representations $R$ and $\overline{R}$ provided $\widehat{Q}$ is an interaction refinement of $Q$ for the representations $R\|\overline{R}$ and $\overline{R}$. This is true, however, if $I \Rightarrow (\overline{A}; \overline{R})$ (see below). The reason is as follows. In the feedback loops of $\mu\widehat{Q}$ we cannot be sure that only representations of streams (i.e. streams in the images of some of the functions characterized by $\overline{R}$) occur. Therefore, we have to give a slightly more complicated scheme of refinement for feedback.

**Theorem 5 (Compositionality of refinement, feedback)** *Assume $\widehat{Q}$ is an interaction refinement of $Q$ for the representation specifications $R\|\overline{R}$ and $\overline{R}$ where $\overline{R}$ is uniform; then $\mu((I\|\overline{A}; \overline{R}); \widehat{Q})$ is an interaction refinement of $\mu Q$ for the representations $R$ and $\overline{R}$.*

**Proof**: We prove:

$$(R; \mu((I\|\overline{A}; \overline{R}); \widehat{Q})).f \Rightarrow ((\mu Q); \overline{R}).f$$

From

$$(R; \mu((I\|\overline{A}; \overline{R}); \widehat{Q})).f$$

we conclude that there exist functions $\rho, \widehat{q}, \overline{\rho}$, and $\overline{\alpha}$ such that $R.\rho$, $\widehat{Q}.\widehat{q}$, $\overline{R}.\overline{\rho}$, and $\overline{A}.\overline{\alpha}$ and furthermore

$$f = \rho; \mu((I\|\overline{\alpha}; \overline{\rho}); \widehat{q})$$

Since $\widehat{Q}$ is an interaction refinement of $Q$ for the representations $R\|\overline{R}$ and $\overline{R}$ for functions $\rho$ with $R.\rho$ and $\overline{\rho}$ with $\overline{R}.\overline{\rho}$ and $\hat{q}$ with $\widehat{Q}.q$ there exist functions $q$ and $\widetilde{\rho}$ such that $Q.q$ and $\overline{R}.\widetilde{\rho}$ hold and furthermore

$$(\rho\|\overline{\rho}); \widehat{q} = q; \widetilde{\rho}$$

Given $x$, by the continuity of $\rho, \widehat{q}, \overline{\rho}$, and $\overline{\alpha}$, we may define $\mu((I\|\overline{\alpha}; \overline{\rho}); \widehat{q}).\rho.x$ by $\sqcup\widehat{y_i}$ where

$$\widehat{y}_0 = \langle\rangle^{\widehat{m}}$$

$$\widehat{y}_{i+1} = \widehat{q}(\rho.x, \overline{\rho}.\overline{\alpha}.\widehat{y}_i)$$

Moreover, because of the continuity of $q$, we may define $\tilde{\rho}.(\mu q).x$ by $\tilde{\rho}. \sqcup y_i$ where

$$y_0 = \langle\rangle^m$$

$$y_{i+1} = q(x, y_i)$$

We prove:

$$\widetilde{\rho}. \sqcup y_i = \sqcup\widehat{y}_i$$

by computational induction. We prove by induction on $i$ the following proposition:

$$\widehat{y}_i \sqsubseteq \widetilde{\rho}.y_i \sqsubseteq \widehat{y}_{i+1}$$

If $i = 0$, we have:

$$\begin{array}{ll}
\widehat{y}_0 \sqsubseteq & \{\widehat{y}_0 \text{ is the least element}\} \\
\widetilde{\rho}.y_0 \sqsubseteq & \{y_0 \text{ is the least element}\} \\
\widetilde{\rho}.q(x, y_0) = & \{\text{refinement property}\} \\
\widehat{q}(\rho.x, \overline{\rho}.y_0) \sqsubseteq & \{y_0 \text{ is the least element}\} \\
\widehat{q}(\rho.x, \overline{\rho}.\overline{\alpha}.\widehat{y}_0) = & \{\text{definition of } \widehat{y}_1\} \\
\widehat{y}_1 &
\end{array}$$

Assume now the proposition holds for $i$; then we obtain:

$$\begin{array}{ll}
\widehat{y}_{i+1} = & \{\text{definition of } \widehat{y}_{i+1}\} \\
\widehat{q}(\rho.x, \overline{\rho}.\overline{\alpha}.\widehat{y}_i) \sqsubseteq & \{\text{induction hypothesis}\} \\
\widehat{q}(\rho.x, \overline{\rho}.\overline{\alpha}.\widetilde{\rho}.y_i) = & \{\text{uniformity of } \overline{R}\} \\
\widehat{q}(\rho.x, \overline{\rho}.y_i) = & \{\text{refinement property}\} \\
\widetilde{\rho}.q(x, y_i) = & \{\text{definition of } y_{i+1}\} \\
\widetilde{\rho}.y_{i+1} &
\end{array}$$

Furthermore we get:

$$\begin{array}{ll}
\widetilde{\rho}.y_{i+1} = & \{\text{definition of } y_{i+1}\} \\
\widetilde{\rho}.q(x, y_i) = & \{\text{refinement property}\} \\
\widehat{q}(\rho.x, \overline{\rho}.y_i) = & \{\text{uniformity of } \overline{R}\} \\
\widehat{q}(\rho.x, \overline{\rho}.\overline{\alpha}.\widetilde{\rho}.y_i) \sqsubseteq & \{\text{induction hypothesis}\} \\
\widehat{q}(\rho.x, \overline{\rho}.\overline{\alpha}.\widehat{y}_{i+1}) = & \{\text{definition of } \widehat{y}_{i+2}\} \\
\widehat{y}_{i+2} &
\end{array}$$

From this we conclude by the continuity of $\widetilde{\rho}$ that:

$$\sqcup \widehat{y}_i = \widetilde{\rho}. \sqcup y_i$$

and thus

$$(\mu((I \| \overline{\alpha}; \overline{\rho}); \widehat{q})).\rho.x = \widetilde{\rho}.\mu(q).x$$

and finally

$$(\mu(Q); \overline{R}).(\rho; \mu((I \| \overline{\alpha}; \overline{\rho}); \widehat{q}))$$

□

Assuming an adequate refinement allows us to obtain immediately the following corollary.

**Theorem 6 (Compositionality of adequate refinement, feedback)** *Assume* $\widehat{Q}$ *is an adequate interaction refinement of* $Q$ *for the representations* $R\|\overline{R}$ *and* $\overline{R}$ *with abstraction* $\overline{A}$ *then* $\mu(\widehat{Q};\,\overline{A};\,\overline{R})$ *is an interaction refinement of* $\mu Q$ *for the representations* $R$ *and* $\overline{R}$.

**Proof**: Let all the definitions be as in the proof of the previous theorem. Since the interaction refinement is assumed to be adequate there exists a function $\widetilde{q}$ with $Q.q$ such that

$$q;\,\widetilde{\rho};\,\overline{\alpha};\,\overline{\rho} = \widetilde{q};\,\overline{\rho}$$

Carrying out the proof of the previous theorem with $\widetilde{q}$ instead of $q$ and $\overline{\rho}$ instead of $\widetilde{\rho}$ we get:

$$\mu((I\|\overline{\alpha};\,\overline{\rho});\,\widehat{q}) = (\mu\widetilde{q});\,\overline{\rho}$$

By straightforward computational induction we may prove

$$\mu(\widehat{q};\,\overline{\alpha};\,\overline{\rho}) = \mu((I\|\overline{\alpha};\,\overline{\rho});\,\widehat{q})$$

This concludes the proof.

□

Assuming that $\overline{A};\,\overline{R}$ contains the identity as a refinement we can simplify the refinement of feedback loops.

**Theorem 7** *Assume* $\widehat{Q}$ *is an interaction refinement of* $Q$ *for the representations* $R\|\overline{R}$ *and* $\overline{R}$ *with abstraction* $\overline{A}$ *and assume furthermore*

$$I \Rightarrow \overline{A};\,\overline{R}$$

*then* $\mu\widehat{Q}$ *is an interaction refinement of* $\mu Q$ *for the representations* $R$ *and* $\overline{R}$.

**Proof**: Straightforward deduction shows:

$$\begin{array}{l} R;\,\mu\widehat{Q} \Rightarrow \\ R;\,\mu((I\|\overline{A};\,\overline{R});\,\widehat{Q}) \Rightarrow \\ \mu Q;\,\overline{R} \end{array}$$

□

Note, even if $I$ is not a refinement of $\overline{A};\,\overline{R}$, in other words even if $I \Rightarrow \overline{A};\,\overline{R}$ does not hold, other refinements of $\overline{A};\,\overline{R}$ may be used to simplify and refine the term $\overline{A};\,\overline{R}$ in $\mu((I\|\overline{A};\,\overline{R});\,\widehat{Q})$. By the fusion rule for feedback as introduced in section 3 we obtain:

$$R;\,\mu(\widehat{Q};\,\overline{A};\,\overline{R}) = \mu((R\|I);\,\widehat{Q};\,\overline{A};\,\overline{R})$$

This may allow further refinements for $\widehat{Q};\,\overline{A};\,\overline{R}$.

**Example 7** *Compositionality of Refinement for Feedback*

Let us introduce the component $F$ with two input channels and one output channel. It specifies functions of the following functionality:

$$f : M^\omega \times M^\omega \to M^\omega$$

$F$ is specified as follows:

$$F.f \equiv \forall x, y \in M^\omega : \exists d \in D : f(x, y) = g(x, d\frown y)$$

where the auxiliary function g is specified by

$$g : M^\omega \times M^\omega \to M^\omega$$

where $\forall d, e \in D, m \in M, x, y \in M^\omega$ :

$$\begin{array}{ll} & g(x, d\frown ?\frown y) = g(x, d\frown y) \\ \wedge & g(?\frown x, d\frown y) = d\frown ?\frown g(x, y) \\ \wedge & g(d\frown x, e\frown y) = d\frown g(x, y) \end{array}$$

It is a straightforward proof that for the specification $C$ as defined in Example 1:

$$\mu F = C$$

We carry out this proof by induction on the length of the input streams $x$. We show that $\mu f$ fulfills the defining equations for functions $c.d$ in the definition of $C$ in Example 2. Let $f$ be a function with $F.f$ and $g$ be a function as specified above in the definition of $F$. We have to consider just two cases: by the definition of $f$ there exists $g$ as defined above such that: there exists $d$:

$$\begin{array}{l} \mu(f).(?\frown x) = \\ fix.\lambda\ y : g(?\frown x, d\frown y) = \\ fix.\lambda\ y : d\frown ?\frown g(x, d\frown y) = \\ d\frown ?\frown fix.\lambda\ y : g(x, d\frown ?\frown y) = \\ d\frown ?\frown fix.\lambda\ y : g(x, d\frown y) \end{array}$$

$$\begin{array}{l} \mu(f).(e\frown x) = \\ fix.\lambda\ y : g(e\frown x, d\frown y) = \\ fix.\lambda\ y : e\frown g(x, y) = \\ e\frown fix.\lambda\ y : g(x, e\frown y) \end{array}$$

Induction on the length of $x$ and the continuity of the function $g$ conclude the proof.

The refinement $\widehat{F}$ of $F$ according to the representation specification $R$ from example 3 specifies functions of the functionality:

$$f : \{?, \surd\}^\omega \times (D \cup \{\surd\})^\omega \times \{?, \surd\}^\omega \times (D \cup \{\surd\})^\omega \to \{?, \surd\}^\omega \times (D \cup \{\surd\})^\omega$$

It reads as follows:

$$\widehat{F}.f = \forall x, \widetilde{x}, y, \widetilde{y} : \exists d \in D : f(x, \widetilde{x}, y, \widetilde{y}) = \widehat{g}(x, \widetilde{x}, \surd\frown y, d\frown\surd\frown\widetilde{y})$$

where the auxiliary function $g$ is specified by

$$\widehat{g} : \{?, \surd\}^\omega \times (D \cup \{\surd\})^\omega \times \{?, \surd\}^\omega \times (D \cup \{\surd\})^\omega \to \{?, \surd\}^\omega \times (D \cup \{\surd\})^\omega$$

$\forall d, e \in D, x, y \in \{?, \surd\}^\omega, \widetilde{x}, \widetilde{y} \in (D \cup \{\surd\})^\omega$ :

$$\begin{array}{lrl} & \widehat{g}(x, \widetilde{x}, ?\frown y, \widetilde{y}) & = \widehat{g}(x, \widetilde{x}, \surd\frown y, \widetilde{y}) \\ \wedge & \widehat{g}(?\frown x, \widetilde{x}, \surd\frown y, d\frown\surd\frown\widetilde{y}) & = [\surd\frown ?, d\frown\surd]\frown\widehat{g}(x, \widetilde{x}, \surd\frown y, \surd\frown\widetilde{y}) \\ \wedge & \widehat{g}(\surd\frown x, d\frown\surd\frown\widetilde{x}, \surd\frown y, e\frown\surd\frown\widetilde{y}) & = [\surd, d\frown\surd]\frown\widehat{g}(x, \widetilde{x}, y, \widetilde{y}) \\ \wedge & \widehat{g}(\surd\frown x, \surd\frown\widetilde{x}, y, \widetilde{y}) & = [\surd, \surd]\frown\widehat{g}(x, \widetilde{x}, y, \widetilde{y}) \\ \wedge & \widehat{g}(x, \widetilde{x}, \surd\frown y, \surd\frown\widetilde{y}) & = \widehat{g}(x, \widetilde{x}, y, \widetilde{y}) \end{array}$$

We have (again, this can be proved by a straightforward rewrite proof):

$$(R\|R); \widehat{F} = F; R$$

Moreover, we have according to Theorem 5:

$$R; \mu((I\|A; R); \widehat{F}) \Rightarrow (\mu F); R$$

and therefore

$$R; \mu((I\|A; R); \widehat{F}) \Rightarrow C; R$$

Note, the refinement is definite and therefore adequate for $F$. Therefore we may replace $\mu((I\|A; R); \widehat{F})$ by $\mu(\widehat{F}; A; R)$.

The component $\mu(\widehat{F}; A; R)$ can be further refined by refining $A; R$. Let us, therefore, look for a simplification for $A; R$. We do not have

$$I \Rightarrow A; R$$

since by the monotonicity of all $\alpha$ with $A.\alpha$ we have:

$$\alpha(\langle\rangle, d\frown\langle\rangle) = \langle\rangle$$

(otherwise we obtain a contradiction, since by monotonicity the first elements of $\alpha(x, d\frown y)$ have to coincide for all $x$ and $y$). Therefore for all $\rho$ with $R.\rho$:

$$\exists t \in ticks : \rho.\alpha(\langle\rangle, d\frown\langle\rangle) = t\frown[\langle\rangle, \langle\rangle]$$

This indicates that there are no functions $\rho$ and $\alpha$ with $R.\rho$ and $A.\alpha$ such that $\rho.\alpha.x = x$ is valid for all $x$. We therefore cannot simply refine $A; R$ into $I$.

We continue the refinement by refining $p$. We take into account properties of $\widehat{F}$. A simple rewriting proof shows:

$$(R\|I); \widehat{F} \Rightarrow (R\|I); \widehat{F}; A; R$$

Summarizing our refinements we obtain:

$$\begin{array}{l} R;\ \mu\widehat{F} \Rightarrow \\ \mu((R\|I);\ \widehat{F})) \Rightarrow \\ \mu((R\|I);\ \widehat{F};\ A;\ R)) \Rightarrow \\ R;\ \mu(\widehat{F};\ A;\ R)) \Rightarrow \\ R;\ \mu((I\|A;\ R);\ \widehat{F}) \end{array}$$

This concludes our example of refinement for feedback.

**End of example**

Recall that every finite network can be represented by an expression that is built by our forms of composition. The theorems show that a network can be refined by defining representation specifications for the channels and by refining all its components. This provides a modular method of refinement for networks.

# 6 Recursively defined Specifications

Often the behavior of interactive components is specified by recursion. Given a function

$$\tau : SPEC_m^n \to SPEC_m^n$$

a recursive declaration of a component specification $Q$ is given by a declaration based on $\tau$:

$$\textbf{letrec}\ Q.f = \tau[Q].f$$

Recursive specifications are restricted in the following to functions $\tau$ that exhibit certain properties.

## 6.1 Semantics of Recursively Defined Specifications

A function $\tau$ where

$$\tau : SPF_m^n \to SPF_k^j$$

is *monotonic with respect to implication*, if:

$$(Q \Rightarrow \widehat{Q}) \Rightarrow (\tau[Q] \Rightarrow \tau[\widehat{Q}])$$

A set $\{Q_i : i \in \mathbb{N}\}$ of specifications is called a *chain*, if for all $i \in \mathbb{N}$ and for all functions $f \in SPF_m^n$:

$$Q_{i+1}(f) \Rightarrow Q_i(f)$$

A function $\tau$ is *continuous* with respect to implication, if for every chain $\{Q_i : i \in I\!N\}$ and all for functions $f \in SPF_m^n$:

$$\tau[Q].f = \forall i \in I\!N : \tau[Q_i].f \quad \text{where} \quad Q.f = \forall i \in I\!N : Q_i(f)$$

Note, the set of all specifications forms a complete lattice.

**Definition 8 (Predicate transformer)** *A* predicate transformer *is a function*

$$\tau : SPEC_m^n \to SPEC_k^j$$

*that is monotonic and continuous with respect to implication (refinement).*

Note, if $\tau$ is defined by $\tau[X] = Net(X)$ where $Net(X)$ is a finite network composed of basic component specifications by our forms of composition, then $\tau$ is a predicate transformer.

A recursive declaration of a component specification $Q$ is given by a defining equation (often called the fixed point equation) based on a predicate transformer $\tau$:

$$\textbf{letrec } Q = \tau[Q]$$

A predicate $Q$ is called a *fixed point* of $\tau$ if:

$$Q = \tau[Q]$$

In general, for a function $\tau$ there exist several predicates Q that are fixed points of $\tau$ . In fixed point theory a partial order on the domain of $\tau$ is established such that every monotonic function $\tau$ has a least fixed point. This fixed point is associated with the identifier $f$ by a recursive declaration of the form $f = \tau.f$. For defining the semantics of programming languages the choice of the ordering, which determines the notion of the least fixed point, has to take into account operational considerations. There the ordering used in the fixed point construction has to reflect the stepwise approximation of a result by the execution. For specifications such operational constraints are less significant.

Therefore we choose a very liberal interpretation for recursive declarations of specifications in the following. For doing so we define the concept of an upper closure of a specification. The upper closure is again a predicate transformer:

$$\Xi : SPEC_m^n \to SPEC_m^n$$

It is defined by the following equation:

$$\Xi[Q].f = \exists g : Q.g \wedge g \sqsubseteq f$$

Notice that $\Xi$ is a classical closure operator, since it has the following characteristic properties:

$$((\widehat{Q} \Rightarrow Q) \Rightarrow (\Xi[\widehat{Q}] \Rightarrow \Xi[Q])$$

$$Q \Rightarrow \Xi[Q]$$

$$\Xi[Q] = \Xi[\Xi[Q]]$$

A predicate $Q$ is called *upward closed*, if $Q = \Xi[Q]$. Note, by $\Xi$ the least element $\Omega$ is mapped onto the specification Ł that is fulfilled by every function, that is $\Xi[\Omega] =$ Ł. From a methodological point of view it is sufficient to restrict our attention to specifications that are upward closed[4]. This methodological consideration and the considerable simplification of the formal interpretation of recursive declarations are the reasons for considering only upward closed solutions of recursive equations.

A predicate transformer $\tau$ is called *upward closed*, if for all predicates $Q$ we have:

$$\tau[Q] = \Xi[\tau[Q]]$$

By the recursive declaration

$$letrec\ Q = \tau[Q]$$

we associate with Q the predicate that fulfills the following equation:

$$Q.f = \forall i \in I\!N : Q_i.f$$

where the predicates $Q_i$ are specified by:

$$Q_0 = \text{Ł}$$

$$Q_{i+1} = \Xi[\tau[Q_i]]$$

According to this definition we associate with a recursive declaration the logically weakest[5] predicate $Q$ such that

$$Q = \Xi[\tau[Q]]$$

The predicate $Q$ is then denoted by $fix.\tau$.

## 6.2 Refinement of Recursively Specified Components

A uniform representation specification $R$ with abstraction $A$ is called adequate for the predicate transformer $\tau$, if for all predicates $X$:

$$(X;\ R;\ A \Rightarrow X) \Rightarrow (\tau[X];\ R;\ A \Rightarrow \tau[X])$$

[4]Taking the upper closure for a specification may change its safety properties. However, only safety properties for those behaviors may be changed where the further output, independent of further input, is empty. A system with such a behavior does not produce a specific message on an output channel, even, if we increase the streams of the messages on the input channels. Then what output is produced on that channel obviously is not relevant at all.

[5]True is considered weaker than false.

Adequacy implies that specifications for which $R$ is adequate are mapped by $\tau$ onto specifications by for which $R$ is adequate again.

Uniform interaction refinement is compositional for recursive definitions based on predicate transformers for which the refinement is adequate. Again definite representations are always adequate.

**Theorem 8 (Compositionality of refinement for recursion)** *Let representation specifications $R$ and $\overline{R}$ be given, where $\overline{R}$ is uniform with abstraction $\overline{A}$ and adequate for the predicate transformer*

$$\tau : SPEC_m^n \to SPEC_m^n$$

*For a predicate transformer*

$$\widehat{\tau} : SPEC_{\widehat{m}}^{\widehat{n}} \to SPEC_{\widehat{m}}^{\widehat{n}}$$

*where*

$$R; \text{Ł} \Rightarrow \text{Ł}; \overline{R}$$

*and for all predicates $X$, $\widehat{X}$:*

$$(R; \widehat{X} \Rightarrow X; \overline{R}) \Rightarrow (R; \widehat{\tau}[\widehat{X}] \Rightarrow \tau[X]; \overline{R})$$

*we have*

$$R; fix.\lambda\ X : \widehat{\tau}[X; \overline{A}; \overline{R}] \Rightarrow fix.\tau; \overline{R}$$

**Proof**: Without loss of generality assume that the predicate transformers $\tau$ and $\widehat{\tau}$ are upward closed. Define

$$Q_0 = \text{Ł}$$

$$Q_{i+1} = \tau[Q_i]$$

$$\widehat{Q}_0 = \text{Ł}$$

$$\widehat{Q}_{i+1} = \widehat{\tau}[\widehat{Q}_i; \overline{A}; \overline{R}]$$

We prove:

$$Q_i; \overline{R}; \overline{A} \Rightarrow Q_i$$

This proposition is obtained by a straightforward induction proof on i. For $i = 0$ we have to show:

$$\text{Ł}; \overline{R}; \overline{A} \Rightarrow \text{Ł}$$

which is trivially true, since Ł holds for all functions. The induction step reads as follows: from

$$Q_i; \overline{R}; \overline{A} \Rightarrow Q_i$$

we conclude by the adequacy of $\tau$:

$$\begin{array}{ll}
Q_{i+1};\overline{R};\overline{A} = & \{\text{definition of } Q_{i+1}\} \\
\tau[Q_i];\overline{R};\overline{A} \Rightarrow & \{\text{adequacy of } \tau \text{ and induction hypothesis}\} \\
\tau[Q_i] = & \{\text{definition of } Q_{i+1}\} \\
Q_{i+1} &
\end{array}$$

We prove by induction on $i$:

$$R;\widehat{Q}_i \Rightarrow Q_i;\overline{R}$$

For $i = 0$, we have to prove:

$$R;Ł \Rightarrow Ł;\overline{R}$$

This is part of our premises. Now assume the induction hypothesis holds for $i$; trivially

$$R;\widehat{Q}_i;\overline{A};\overline{R} \Rightarrow R;\widehat{Q}_i;\overline{A};\overline{R}$$

Therefore, with $X = R;\widehat{Q}_i;\overline{A}$ and $\widehat{X} = \widehat{Q}_i;\overline{A};\overline{R}$ by our premise we have:

$$R;\widehat{\tau}[\widehat{Q}_i;\overline{A};\overline{R}] \Rightarrow \tau[R;\widehat{Q}_i;\overline{A}];\overline{R}$$

By the induction hypothesis and by the fact $Q_i;\overline{R};\overline{A} \Rightarrow Q_i$ we obtain $R;\widehat{Q}_i;\overline{A} \Rightarrow Q_i$ as can be seen by the derivation

$$\begin{array}{ll}
R;\widehat{Q}_i;\overline{A} & \Rightarrow \\
Q_i;\overline{R};\overline{A} & \Rightarrow \\
Q_i &
\end{array}$$

We obtain:

$$\begin{array}{ll}
R;\widehat{Q}_{i+1} \Rightarrow & \{\text{definition of } \widehat{Q}_{i+1}\} \\
R;\widehat{\tau}[\widehat{Q}_i;\overline{A};\overline{R}] \Rightarrow & \{\text{premise for } \tau, \widehat{\tau} \text{ with } X = R;\widehat{Q}_i;\overline{A}, \widehat{X} = \widehat{Q}_i;\overline{A};\overline{R}\ \} \\
\tau[R;\widehat{Q}_i;\overline{A}];\overline{R} \Rightarrow & \{\text{uniformity of } \overline{R}\text{, see above}\} \\
\tau[Q_i];\overline{R} \Rightarrow & \{\text{definition of } Q_{i+1}\} \\
Q_{i+1};\overline{R} &
\end{array}$$

□

Note, for definite representations $R$ the premise

$$R;Ł \Rightarrow Ł;\overline{R}$$

is always valid as the following straightforward derivation shows:

$$\begin{array}{ll}
R;Ł \Rightarrow & \{\text{definition of } Ł\} \\
R;A;Ł;\overline{R} \Rightarrow & \{\text{since } R;A = I\} \\
Ł;\overline{R} &
\end{array}$$

We immediately obtain the following theorem as corollary. It can be useful for simplifying the refinement of recursion.

**Theorem 9** *Given the premisses of the theorem above and in addition*

$$I \Rightarrow \overline{A};\ \overline{R}$$

*we have*

$$R;\ fix.\hat{\tau} \Rightarrow fix.\tau;\ \overline{R}$$

**Proof**: The theorem is proved by a straightforward deduction:

$$\begin{array}{ll} R;\ fix.\hat{\tau} \Rightarrow & \{\text{premise}\} \\ R;\ fix.\lambda\ X : \hat{\tau}[X;\ \overline{A};\ \overline{R}] \Rightarrow & \{\text{theorem } 8\} \\ fix.\tau;\ \overline{R} & \end{array}$$

□

Note, even if $I$ is not a refinement of $\overline{A};\ \overline{R}$, that is even if $I \Rightarrow \overline{A};\ \overline{R}$ does not hold, other refinements of $\overline{A};\ \overline{R}$ may be used to simplify the term $\overline{A};\ \overline{R}$ in the specification.

$$fix.\lambda\ X : \hat{\tau}[X;\ \overline{A};\ \overline{R}]$$

**Example 8** *Compositionality of Refinement for Recursion*

Of course, instead of giving a feedback loop as in example 7 above we may also define an infinite network recursively by[6]:

$$\textbf{letrec}\ Q = \tau[Q]$$

where

$$\tau[X] = \Upsilon;\ (I \| X);\ F$$

Again we obtain (as a straightforward proof along the lines of the proof above for $\mu F = C$ shows):

$$Q = C$$

It is also a straightforward proof to show that

$$(R;\ \widehat{X} \Rightarrow X;\ \overline{R}) \Rightarrow (R;\ \hat{\tau}[\widehat{X}] \Rightarrow \tau[X];\ \overline{R})$$

where

$$\hat{\tau}[\widehat{X}] = \Upsilon;\ (I \| (\widehat{X};\ A;\ R));\ \widehat{F}$$

Therefore we have

$$R;\ \widehat{Q} = Q;\ R$$

where

$$\textbf{letrec}\ \widehat{Q} = \hat{\tau}[\widehat{Q}]$$

by our compositionality results. Again $A;\ R$ can be replaced by its refinement as shown above.

**End of example**

Using recursion we may define even infinite nets. The theorem above shows that a

[6]The predicate transformer $\tau$ is obtained by the unfold rule for feedback

refinement of an infinite net that is described by a recursive equation is obtained by refinement of the components of the net.

# 7 Predicate Transformers as Refinements

So far we have considered the refinement of components by refining on one hand their tuples of input and on the other hand their tuples of output streams. A more general notion of refinement is obtained by considering predicate transformers themselves as refinements.

**Definition 9 (Refining context)** *A predicate transformer*

$$\mathcal{R} : SPEC_m^n \to SPEC_k^i$$

*is called a* refining context*, if there exists a mapping*

$$\mathcal{A} : SPEC_k^i \to SPEC_m^n$$

*called* abstracting context *such that for all predicates $X$ we have:*

$$\mathcal{A}.\mathcal{R}.X \Rightarrow X$$

Refining contexts can be used to define a quite general notion of refinement.

**Definition 10 (Refinement by refining contexts)** *Let $\mathcal{R}$ be a refining context with abstracting context $\mathcal{A}$. A specification $\widehat{Q}$ is then called* a refinement for the abstracting context $\mathcal{A}$ *of the specification $Q$, if:*

$$\mathcal{A}.\widehat{Q} \Rightarrow Q$$

*Note, $\mathcal{R}.Q$ is a refinement of the specification $Q$ for the abstracting context $\mathcal{A}$.*

Refining contexts may be defined by the compositional forms introduced in the previous sections.

**Example 9** *Refining Contexts*

For component specifications $Y$ with one input channel and two output channels we define a predicate transformer

$$\mathcal{A} : SPEC_2^1 \to SPEC_1^1$$

by the equation:

$$\mathcal{A}.Y = \mu((P \| \dagger); Y); (\dagger \| I)$$

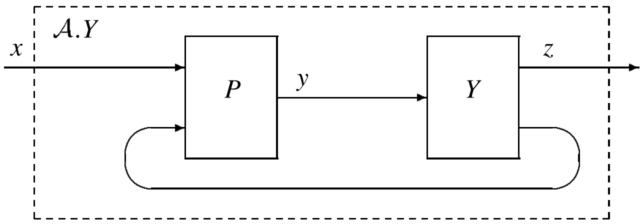


Figure 7: Graphical representation of $\mathcal{A}.Y$

where the component $P$ specifies functions

$$p : D^\omega \times \{?, \surd\}^\omega \to D^\omega$$

A graphical representation of $\mathcal{A}.Y$ is given in Figure 7. Let $P$ be specified by:

$$\begin{array}{rll} P.p \equiv \forall x \in D^\omega, y \in \{?, \surd\}^\omega : & p(m\frown x, ?\frown y) & = m\frown p(m\frown x, y) \\ \wedge & p(m\frown x, \surd\frown y) & = p(x, y) \end{array}$$

For a component specification $X$ with one input channel and one output channel we define a predicate transformer:

$$\mathcal{R} : SPEC_1^1 \to SPEC_2^1$$

where

$$\mathcal{R}.X = Q; (I \| X)$$

where the component $Q$ specifies functions

$$q : D^\omega \to \{?, \surd\}^\omega \times D^\omega$$

Let $Q$ be specified by:

$$Q.q \equiv \forall x \in D^\omega : \exists k \in I\!N : \forall i \in I\!N : i \leq k \Rightarrow$$

$$\begin{array}{rll} & q(m^i) & = [?^{i+1}, \langle\rangle] \\ \wedge & q((m^{k+1})\frown x) & = [(?^{k+1})\frown\surd, m]\frown q.x \end{array}$$

Let $m^k$ stand for the finite stream of length $k$ containing just copies of the message $m$. To show that $\mathcal{A}$ and $\mathcal{R}$ define a refining context we show that:

$$\mathcal{A}.\mathcal{R}.X = X$$

which is equivalent to showing that for all specifications $X$:

$$\mu((P\|\dagger); Q; (I\|X)); (\dagger\|I) = X$$

This is equivalent to:

$$\mu((P \| \dagger); Q); (\dagger \| I) = I$$

which is equivalent to the formula:

$$\forall p, q, x : P.p \wedge Q.q \Rightarrow x = (\mu((I \| \dagger); p; q); (\dagger \| I)).x$$

which can be shown by a proof based on the specifications of $P$ and $Q$. Let $\swarrow$ stand for $(I_2 \| \dagger)$ and $\searrow$ stand for the function $(\dagger \| I_1)$. For functions $p$ and $q$ with $P.p$ and $Q.q$ there exists $k \in I\!N$ such that $\forall i \in I\!N$ with $i \leq k$:

$$\begin{array}{l} \searrow .fix.\lambda\ y, z : q.p. \swarrow (m^\frown x, (?^i)^\frown y, z) = \\ \searrow .fix.\lambda\ y, z : q((m^i)^\frown p(m^\frown x, y)) = \\ \searrow .fix.\lambda\ y, z : [?^{i+1}, \langle\rangle]^\frown q.p(m^\frown x, y) = \\ \searrow .fix.\lambda\ y, z : q.p. \swarrow (m^\frown x, (?^{i+1})^\frown y, z) \end{array}$$

This can be shown by a straightforward proof of induction on $i$. By this we obtain for $i = k + 1$:

$$\begin{array}{l} \searrow .fix.\lambda\ y, z : q.p. \swarrow (m^\frown x, y, z) = \\ \searrow .fix.\lambda\ y, z : q.p. \swarrow (m^\frown x, (?^{k+1})^\frown y, z) \end{array}$$

Furthermore:

$$\begin{array}{l} \searrow .fix.\lambda\ y, z : q.p. \swarrow (m^\frown x, (?^{k+1})^\frown y, z) = \\ \searrow .fix.\lambda\ y, z : q((m^{k+1})^\frown p(m^\frown x, y)) = \\ \searrow .fix.\lambda\ y, z : [(?^{k+1})^\frown \surd, m]^\frown q.p(m^\frown x, y) = \\ \searrow .fix.\lambda\ y, z : q.p(m^\frown x, (?^{k+1})^\frown \surd^\frown y) = \\ \searrow .fix.\lambda\ y, z : [(?^{k+1})^\frown \surd, m]^\frown q.p(m^\frown x, y) = \\ m^\frown \searrow .fix.\lambda\ y, z : q.p(m^\frown x, y) = \\ m^\frown \searrow .fix.\lambda\ y, z : q.p. \swarrow (m^\frown x, y, z) \end{array}$$

We obtain

$$\begin{array}{l} (\mu(\swarrow; p; q); \searrow)(m^\frown x) = \\ \searrow .fix.\lambda\ y, z : q.p. \swarrow (m^\frown x, y, z) = \\ m^\frown \searrow .fix.\lambda\ y, z : q.p. \swarrow (m^\frown x, y, z) \end{array}$$

By induction on the length on $x$ and the continuity of the involved functions the proposition above is proved.

**End of example**

Context refinement is indeed a generalization of interaction refinement. Given two pairs of definite representation and abstraction specifications $R$, $A$ and $\overline{R}$, $\overline{A}$ by

$$\mathcal{A}.Y = R; Y; \overline{A}$$

$$\mathcal{R}.X = A; X; \overline{R}$$

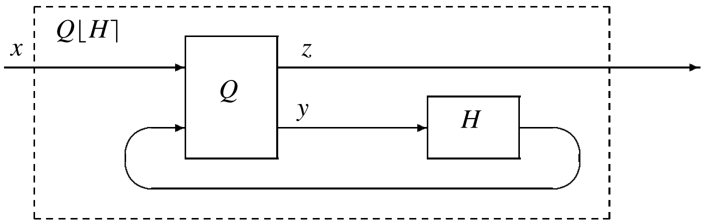


Figure 8: Graphical representation of the master/slave system

a refining context and an abstracting context is defined, since

$$\mathcal{A}.\mathcal{R}.X =$$
$$\mathcal{A}.(A;\ X;\ \overline{R}) =$$
$$R;\ (A;\ X;\ \overline{R});\ \overline{A} \Rightarrow$$
$$X$$

Refining contexts lead to a more general notion of refinement than interaction refinement. There are specifications $Q$ and $\widehat{Q}$ such that there do not exist consistent specifications $R$ and $A$ where

$$R;\ \widehat{Q};\ A \Rightarrow Q$$

but there may exist refining contexts $\mathcal{R}$ and $\mathcal{A}$ such that

$$\mathcal{A}.\widehat{Q} \Rightarrow Q$$

Refining contexts may support the usage of sophisticated feedback loops between the refined system and the refining context. This way a dependency between the representation of the input history and the output history can be achieved.

A very general form of a refining context is obtained by a special operator for forming networks called *master/slave systems*. For notational convenience we introduce a special notation for master/slave systems. A graphical representation of master/slave systems is given in Figure 8. A master/slave system is denoted by $Q\lfloor H \rceil$. It consists of two components $Q$ and $H$ called the *master* $Q \in SPEC_{k+n}^{i+m}$ and the *slave* $H \in SPEC_m^n$. Then $Q\lfloor H \rceil \in SPEC_k^i$. All the input of the slave is comes via the master and all the output of the slave goes to the master. The master/slave system is defined as follows:

$$Q\lfloor H \rceil = \mu((Q \| \dagger^k);\ (I_k \| H);\ \overset{k\,m}{X});\ (\dagger^m \| I_k)$$

or in a more readable notation:

$$(Q\lfloor H \rceil).f = \exists q, h : Q.q \wedge H.h \wedge f = q\lfloor h \rceil$$

where $\forall x, y, z$;

$$(q\lfloor h\rceil).x = z \text{ where } (z, y) = fix.\lambda\ z, y : q(x, h.y)$$

We can define a refining context and an abstracting context based on the master/slave system concept: we look for predicate transformers

$$\mathcal{R} : SPEC_m^n \to SPEC_k^i$$

with abstracting context

$$\mathcal{A} : SPEC_k^i \to SPEC_m^n$$

and for specifications $V \in SPEC_{k+n}^{i+m}$ and $W \in SPEC_{m+i}^{n+k}$ where the refining context and the abstracting context are specified as follows:

$$\mathcal{R}.X = V\lfloor X\rceil$$

$$\mathcal{A}.Y = W\lfloor Y\rceil$$

and the following requirement is fulfilled:

$$W\lfloor V\lfloor X\rceil\rceil \Rightarrow X$$

We give an analysis of this requirement based on a further form of composition called a *cooperator*. The cooperator is denoted by $\underset{m}{\overset{n}{\rightleftharpoons}}$ where $m, n \in I\!N$. For specifications $Q \in SPEC_{m+\overline{k}}^{n+k}$, $\overline{Q} \in SPEC_{\overline{m}+k}^{\overline{n}+\overline{k}}$ the cooperator is defined as follows:

$$(Q \underset{k}{\overset{\overline{k}}{\rightleftharpoons}} \overline{Q}) \in SPEC_{m+\overline{m}}^{n+\overline{n}}$$

$$(Q \underset{k}{\overset{\overline{k}}{\rightleftharpoons}} \overline{Q}).f = \exists q, \overline{q} : Q.q \wedge \overline{Q}.\overline{q} \wedge f = (q \underset{k}{\overset{\overline{k}}{\rightleftharpoons}} \overline{q})$$

where

$$(q \underset{k}{\overset{\overline{k}}{\rightleftharpoons}} \overline{q}).(x, \overline{x}) = (z, \overline{z}) \text{ where } (z, \overline{y}, y, \overline{z}) = fix.\lambda\ z, \overline{y}, y, \overline{z} : (q(x, y), \overline{q}(\overline{y}, \overline{x}))$$

A graphical presentation of the cooperator is given in Figure 9.

A straightforward rewriting shows that the cooperator is indeed a generalization of the master/slave. For $H \in SPEC_k^{\overline{k}}$:

$$Q \underset{k}{\overset{\overline{k}}{\rightleftharpoons}} H = Q\lfloor H\rceil$$

In particular we obtain:

$$W\lfloor V\lfloor X\rceil\rceil = W \underset{k}{\overset{i}{\rightleftharpoons}} (V \underset{m}{\overset{n}{\rightleftharpoons}} X) = (W \underset{k}{\overset{i}{\rightleftharpoons}} V)\lfloor X\rceil$$

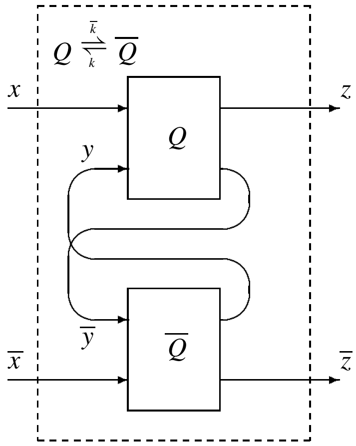


Figure 9: Graphical representation of the cooperator

and therefore the condition:

$$W\lfloor V\lfloor X\rceil\rceil \Rightarrow X$$

reads as follows:

$$(W \underset{k}{\overset{i}{\rightleftharpoons}} V)\lfloor X\rceil \Rightarrow X$$

The following theorem gives an analysis for the component $W \underset{k}{\overset{i}{\rightleftharpoons}} V$.

**Theorem 10** *The implication*

$$(W \underset{k}{\overset{i}{\rightleftharpoons}} V)\lfloor X\rceil \Rightarrow X$$

*implies*

$$(W \underset{k}{\overset{i}{\rightleftharpoons}} V) = \overset{n\,m}{X}$$

*Recall,* $\overset{n\,m}{X}$ *just swaps its input streams.*

**Proof**: By the definition of cooperation we may conclude that for every function $\zeta$ and every function $\nu$ such that $W.\zeta$ and $V.\nu$ and for every $f$ where $X.f$ there exists a function $\widetilde{f}$ where $X.\widetilde{f}$ such that:

$$\exists \overline{z} : (z, \overline{z}) = (\zeta \underset{k}{\overset{i}{\rightleftharpoons}} \nu)(x, f.\overline{z}) \Leftrightarrow z = \widetilde{f}.x$$

Since this formula is true for all specifications $X$ and therefore also for definite specifications, the formula holds for all functions $f$ where in addition $f = \widetilde{f}$. We obtain for the constant function $f$ with $z = f.x$ for all $x$ and for all $z$:

$$\exists \overline{z} : (z, \overline{z}) = (\zeta \overset{i}{\underset{k}{\rightleftharpoons}} \nu)(x, z) \Leftrightarrow z = z$$

The equation above therefore simplifies to

$$\exists \overline{z} : (f.\overline{z}, \overline{z}) = (\zeta \overset{i}{\underset{k}{\rightleftharpoons}} \nu)(x,\ f.\overline{z}) \Leftrightarrow f.\overline{z} = f.x$$

Now we prove that from this formula we can conclude:

$$(f.\overline{z}, x) = (\zeta \overset{i}{\underset{k}{\rightleftharpoons}} \nu)(x,\ f.\overline{z})$$

We do the proof by contradiction. Assume there exists $x$ such that:

$$(f.\overline{z}, \overline{z}) = (\zeta \overset{i}{\underset{k}{\rightleftharpoons}} \nu)(x,\ f.\overline{z})$$

and $x \neq \overline{z}$. Then we can choose a function $f$ such that $f.x \neq f.\overline{z}$. This concludes the proof of the theorem.

$\square$

By the concept of refining contexts we then may consider the refined system

$$Q\lfloor W\lfloor V\lfloor H\rceil\rceil\rceil$$

The refinement of this refined network can then be continued by refining $V\lfloor H\rceil$ and leaving its environment $Q\lfloor W\lfloor ...\rceil\rceil$ as it is.

There is a remarkable relationship between master/slave systems and the system structures studied in rely/guarantee specification techniques as advocated among others in [Abadi, Lamport 90]. The master can be seen as the environment and the slave as the system. This indicates that the master/slave situation models a very general form of composition. Every net with a subnet $H$ can be understood as a master/slave system $Q\lfloor H\rceil$ where $Q$ denotes the surrounding net, the environment, of $H$. This form of networks is generalized by the cooperator as a composing form, where in contrast to master/slave systems the situation is fully symmetric.

The cooperating components $Q$ and $\overline{Q}$ in $Q \overset{\overline{k}}{\underset{k}{\rightleftharpoons}} \overline{Q}$ can be seen as their mutual environments. The concept of cooperation is the most general notion of a composing form for components. All composing forms considered so far are just special cases of cooperation; for $Q \in SPEC_m^n$, $P \in SPEC_k^i$ we obtain:

$$\begin{array}{ll} Q; P = Q \overset{m}{\underset{0}{\rightleftharpoons}} P & \text{if } m = i \\ Q \| P = Q \overset{0}{\underset{0}{\rightleftharpoons}} P & \\ \mu Q = (Q; \Upsilon) \overset{m}{\underset{m}{\rightleftharpoons}} I & \text{if } n \geq m \end{array}$$

Let a net $N$ be given with the set $\Gamma$ of components. Every partition of $\Gamma$ into two disjoint sets of components leads to a partition of the net into two disjoint subnets say $Q$ and $\overline{Q}$ such that the net is equal to $Q \underset{k}{\overset{\overline{k}}{\rightleftarrows}} \overline{Q}$ where $k$ denotes the number of channels in $N$ leading from $\overline{Q}$ to $Q$ and $\overline{k}$ denotes the number of channels leading from $Q$ to $\overline{Q}$. Then both subnets can be further refined independently.

# 8 Conclusion

The notion of compositional refinement depends on the operators, the composing forms, considered for composing a system. Compositionality is not a goal per se. It is helpful for performing global refinements by local refinements. Refining contexts, master slave systems and the cooperator are of additional help for structuring and restructuring a system for allowing local refinements.

The previous sections have demonstrated that using functional techniques a compositional notion of interaction refinement is achieved. The refinement of the components of a large net can be mechanically transformed into a refinement of the entire net.

Throughout this paper only notions of refinement have been treated that can be expressed by continuous representation and abstraction functions. This is very much along the lines of [CIP 84] and [Broy et al. 86] where it is considered as an important methodological simplification, if the abstraction and representation functions can be used at the level of specified functions. There are interesting examples of refinement, however, where the representation functions are not monotonic (see the representation functions obtained by the introduction of time in [Broy 90]). A compositional treatment of the refinement of feedback loops in these cases remains as an open problem.

**Acknowledgement**: This work has been carried out during my stay at Digital Equipment Corporation's Systems Research Center. The excellent working environment and stimulating discussions with the colleagues at SRC, in particular Jim Horning, Leslie Lamport, and Martín Abadi are gratefully acknowledged. I thank Claus Dendorfer, Leslie Lamport, and Cynthia Hibbard for their careful reading of a version of the manuscript and their most useful comments.

# A Appendix: Full Abstraction

Looking at functional specifications one may realize that sometimes they specify more properties than one might be interested in and that one may observe under the considered compositional forms. Basically we are interested in two observations for a given specification $Q$ for a function $f$ with $Q.f$ and input streams $x$. The first one is straightforward: we are interested in the output streams $y$ where

$$y = f.x$$

But, in addition, for controlling the behavior of components especially within feedback loops we are interested in causality. Given just a finite prefix. $\widetilde{x}$ of the considered input streams $x$, causality of input with respect to output determines how much output (which by monotonicity of $f$ is a prefix of $y$) is guaranteed by $f$.

More technically, we may represent the behavior of a system component by all observations about the system represented by pairs of chains of input and corresponding output streams.

A set $\{x_i \in (M^\omega)^n : i \in \mathbb{N}\}$ is called a *chain*, if for all $i \in \mathbb{N}$ we have $x_i \sqsubseteq x_{i+1}$. Given a specification $Q \in SPEC^n_m$, a pair of chains

$$(\{x_i \in (M^\omega)^n : i \in \mathbb{N}\}, \{y_i \in (M^\omega)^m : i \in \mathbb{N}\})$$

is called an *observation* about $Q$, if there exists a function $f$ with $Q.f$ such that for all $i \in \mathbb{N}$:

$$y_i \sqsubseteq f.x_i$$

and

$$\sqcup\{y_i : i \in \mathbb{N}\} = \sqcup\{f.x_i : i \in \mathbb{N}\}$$

The behavior of a system component specified by $Q$ then can be represented by all observations about $Q$. Unfortunately, there exist functional specifications which show the same set of observations, but, nevertheless, characterize different sets of functions. For an example we refer to [Broy 90].

Fortunately such functional specifications can be mapped easily onto functional specifications where the set of specified functions is exactly the one characterized by its set of observations. For this reason we introduce a predicate transformer

$$\Delta : SPEC^n_m \to SPEC^n_m$$

that maps a specification on its abstract counterpart. This predicate transformer basically constructs for a given predicate $Q$ a predicate $\Delta.Q$ that is fulfilled exactly for those continuous functions that can be obtained by a combination of the graphs of functions from the set of functions specified by $Q$. We define

$$(\Delta.Q).f \equiv \forall x : \exists \widehat{f} : Q.\widehat{f} \wedge f \sqsubseteq_x \widehat{f} \wedge \widehat{f}.x = f.x$$

where

$$f \sqsubseteq_x \widehat{f} \equiv (\forall z : z \sqsubseteq x \Rightarrow f.z \sqsubseteq \widehat{f}.z)$$

By this definition we obtain immediately the monotonicity and the closure property of the predicate transformer $\Delta$.

**Theorem 11 (Closure property of the predicate transformer $\Delta$)**

$$(Q \Rightarrow \widehat{Q}) \Rightarrow (\Delta.Q \Rightarrow \Delta.\widehat{Q})$$

$$Q \Rightarrow \Delta Q$$

$$\Delta.Q = \Delta.\Delta.Q$$

**Proof**: Straightforward, since $Q.f$ occurs positively in the definition of $\Delta.Q$, $f \sqsubseteq_x f$ and

$$\forall x : \exists \widehat{f} : (\Delta.Q).\widehat{f} \wedge f \sqsubseteq_x \widehat{f} \wedge \widehat{f}.x = f.x \equiv (\Delta.Q).f$$

□

A specification $Q$ is called *fully abstract*, if

$$Q = \Delta.Q$$

We may redefine our compositional forms such that the operators deliver always fully abstract specifications:

$$\begin{aligned} Q \,\widetilde{;}\, P &\equiv \Delta(Q; P) \\ Q \,\widetilde{\|}\, P &\equiv \Delta(Q \| P) \\ \widetilde{\mu}\ Q &\equiv \Delta(\mu\ Q) \end{aligned}$$

All the results obtained so far carry over to the abstract view by the monotonicity of $\Delta$, and by the fact that we have

$$\begin{aligned} \Delta(Q\,;\,P) &\equiv \Delta(\Delta.Q; \Delta.P) \\ \Delta(Q \,\|\, P) &\equiv \Delta(\Delta.Q \| \Delta.P) \\ \Delta(\mu Q) &\equiv \Delta(\mu\ \Delta.Q) \end{aligned}$$

Furthermore, given an upward closed predicate transformer $\tau$ we have: if Q is the least solution of

$$Q = \tau[Q]$$

then $\overline{Q} = \Delta.Q$ is the least solution of

$$\overline{Q} = \Delta.\tau[\overline{Q}]$$

The proof is straightforward. Note, by this concept of abstraction we may obtain

$$I \Rightarrow A \,\widetilde{;}\, R$$

in cases where $I \Rightarrow A; R$ does not hold. This allows additional simplifications of network refinements.

Note, full abstraction is a relative notion. It is determined by the basic concept of observability and the composing forms. In the presence of refinement it is unclear whether full abstraction as defined above is appropriate. We have:

$$(\widehat{Q} \Rightarrow Q) \Rightarrow (\Delta.\widehat{Q} \Rightarrow \Delta.Q)$$

However, if a component $Q$ is used twice in a network $\tau[Q]$, then we do not have, in general, that for (determined) refinements $\widetilde{Q}$ of $\Delta.Q$ there exist (determined) refinements $\widehat{Q}$ of $Q$ such that:

$$(\tau[\widetilde{Q}] \Rightarrow \tau[\widehat{Q}])$$

Therefore, when using more sophisticated forms of refinement our notion of full abstraction might not always be adequate.

# References


[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

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

[Back 90] R.J.R. Back: Refinement Calculus, Part I: Sequential Nondeterministic Programs. REX Workshop. In: [deBakker et al. 90], 42-66

[Back 90] R.J.R. Back: Refinement Calculus, Part II: Parallel and Reactive Programs. REX Workshop. In: [deBakker et al. 90], 67-93

[deBakker et al. 90] J. W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.): Stepwise Refinement of Distributed Systems. Lecture Notes in Computer Science 430, Springer 1990

[Brock, Ackermann 81] J.D. Brock, W.B. Ackermann: Scenarios: A Model of Nondeterminate Computation. In: J. Diaz, I. Ramos (eds.): Lecture Notes in Computer Science 107, Springer 1981, 225-259

[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: [deBakker et al. 90], 153-179

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

[Chandy, Misra 88] K.M. Chandy, J. Misra: Parallel Program Design: A Foundation. Addison Wesley 1988

[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, February 1991

[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

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

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

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

[Sannella 88] D. Sannella: A Survey of Formal Software Development Methods. University of Edinburgh, Department of Computer Science, ECS-LFCS-88-56, 1988

[Vogler 91] W. Vogler: Bisimulation and Action Refinement. Proc. STACS 91, Lecture Notes in Computer Science 480, (1991), 309-321