Specification and Refinement of Networks of Asynchronously Communicating Agents Using the Assumption/Commitment Paradigm

Ketil Stølen, Frank Dederichs, Rainer Weber

Fakultät für Informatik, Technische Universität München
Arcisstraße 21, Postfach 20 24 20, D-80290 München, Germany

Keywords: Specification; Assumption/Commitment; Rely/Guarantee; Decomposition; Refinement; Calculus; Streams; Dataflow

Abstract. This paper presents an assumption/commitment specification technique and a refinement calculus for networks of agents communicating asynchronously via unbounded FIFO channels in the tradition of Kahn.

- We define two types of assumption/commitment specifications, namely simple and general specifications.
- It is shown that semantically, any deterministic agent can be uniquely characterized by a simple specification, and any nondeterministic agent can be uniquely characterized by a general specification.
- We define two sets of refinement rules, one for simple specifications and one for general specifications. The rules are Hoare-logic inspired. In particular the feedback rules employ invariants in the style of a traditional while-rule.
- Both sets of rules have been proved to be sound and also (semantic) relative complete.
- Conversion rules allow the two logics to be combined. This means that general specifications and the rules for general specifications have to be introduced only at the point in a system development where they are really needed.

Correspondence and offprint requests to: Ketil Stølen, Fakultät für Informatik, Arcisstrasse 21, Postfach 20 24 20, D-80290 München, Germany (email:stoelen@informatik.tu-muenchen.de)

1. Introduction

Ever since formal system design became a major research direction some 20 years ago, it has been common to write specifications in an *assumption/commitment* form. The assumption characterizes the essential properties of the *environment* in which the specified program, from now on referred to as the *agent*, is supposed to run, while the commitment is a requirement which must be fulfilled by the agent whenever it is executed in an environment which satisfies the assumption.

For example, in Hoare-logic [Hoa69] the post-condition characterizes the states in which the agent is allowed to terminate when executed in an initial state which satisfies the pre-condition. Thus, the pre-condition makes an assumption about the environment, while the post-condition states a commitment which must be fulfilled by the agent.

In general, the popularity of the assumption/commitment paradigm is due to the fact that an agent is normally not supposed to work in an arbitrary environment, in which case specifications and agent designs can be simplified by "restricting" the environment in terms of assumptions.

There are many different techniques for writing assumption/commitment specifications. Roughly speaking, they can be split into two main categories: those which require an *explicit* assumption/commitment form, and those which are content with an *implicit* assumption/commitment form.

In the first category, the assumption is clearly separated from the commitment. A specification can be thought of as a pair $[A, C]$, where A is the assumption about the environment, and C is the commitment to the agent. The pre/post specifications of Hoare-logic belong to this category, so does Jones' rely/guarantee method [Jon83], the Misra/Chandy technique [MC81] for hierarchical decomposition of networks, and a number of other contributions like [Pnu85], [Sta85], [Pan90], [AL90], [Stø91], [PJ91].

In the second category, specifications still make assumptions about the environment and state commitments to the agent, but the assumptions and the commitments are mixed together and stated more implicitly. Examples of such methods are [BKP84], [CM88].

The motivation for insisting on an explicit assumption/commitment form varies from approach to approach. In some methods like [Jon83] and [MC81] this structure is mainly employed to ensure *compositionality* ([dR85], [Zwi89]) of the design rules, namely that the specification of an agent always can be verified on the basis of the specifications of its subagents, without knowledge of the interior construction of those subagents. For example, in the Owicki/Gries method [OG76] environment assumptions can only be made about the initial state, and as a consequence the rule for parallel composition is not compositional.

In other methods, with a richer assertion language, an explicit assumption/commitment form is not needed in order to ensure compositionality. Nevertheless, an explicit assumption/commitment form is still favored by many researchers. Abadi/Lamport [AL90], for example, argue as below[1]:

- "Why write a specification of the form $[A, C]$ when we can simply write C? The answer lies in the practical matter of what the specification looks like. If we eliminate the explicit environment assumption, then that assumption appears implicitly in the properties C describing the system. Instead of C

---

[1] In the quotation $[A, C]$, A and C have been substituted for $E \Rightarrow M$, E and M, respectively.

describing only the behavior of the system when the environment behaves correctly, C must also allow arbitrary behavior when the environment behaves incorrectly. Eliminating A makes C too complicated, and it is not a practical alternative to writing specifications in the form $[A, C]$."

The objective of this paper is to investigate the assumption/commitment paradigm in the context of (possibly) nondeterministic Kahn-networks [Kah74], [KM77], [Kel78]. Agents are modeled as sets of stream processing functions as explained in [Kel78], [Bro89], [BDD+93]. They communicate via unbounded FIFO channels. We distinguish between two types of explicit assumption/commitment specifications, namely simple and general specifications. A simple specification can be used to specify any deterministic agent, while any nondeterministic agent can be specified by a general specification. Refinement rules are formulated and proved sound and (semantic) relative complete.

The basic notation and the semantic model are introduced in Section 2. In Section 3 agents and networks of agents are defined. Section 4 introduces simple specifications and the corresponding refinement calculus. What we call symmetric specifications is discussed and rejected in Section 5, while general specifications and their refinement rules are the topics of Section 6. Section 7 gives a brief summary and relates our approach to other proposals known from the literature. Finally, there is an appendix containing soundness and completeness proofs.

2. Basic Concepts and Notation

Let $\mathbb{N}$ denote the set of positive natural numbers, and let $\mathbb{B}$ denote the set $\{true, false\}$. A *stream* is a finite or infinite sequence of data. It models the history of a communication channel, i.e. it represents the sequence of messages sent along the channel. $\langle \rangle$ stands for the empty stream, and $\langle d_1, d_2, \ldots, d_n \rangle$ stands for a finite stream whose first element is d_1, and whose n'th and last element is d_n. Given a set of data D, D^* denotes the set of all finite streams generated from D; D^∞ denotes the set of all infinite streams generated from D, and D^ω denotes $D^* \cup D^\infty$.

This notation is overloaded to tuples of data sets in a straightforward way: $\langle \rangle$ denotes any empty stream tuple; moreover, if $T = (D_1, D_2, \ldots, D_n)$ then T^* denotes $(D_1^* \times D_2^* \times \ldots \times D_n^*)$, T^∞ denotes $(D_1^\infty \times D_2^\infty \times \ldots \times D_n^\infty)$, and T^ω denotes $(D_1^\omega \times D_2^\omega \times \ldots \times D_n^\omega)$.

There are a number of standard operators on streams and stream tuples. If $d \in D$, $r \in D^\omega$, $s, t \in T^\omega$, $A \subseteq D$, and j is a natural number then:

- $s \frown t$ denotes the pointwise concatenation of s and t, i.e. the j'th component of $s \frown t$ is equal to the result of prefixing the j'th component t_j of t with the j'th component s_j of s if s_j is finite, and is equal to s_j otherwise;
- $s \sqsubseteq t$ denotes that s is a prefix of t, i.e. $\exists p \in T^\omega . s \frown p = t$;
- $A \copyright r$ denotes the projection of r on A, data not occurring in A are deleted, e.g. $\{0, 1\} \copyright \langle 0, 1, 2, 0, 4 \rangle = \langle 0, 1, 0 \rangle$;
- $\#r$ denotes the length of r, i.e. the number of elements in r if $r \in D^*$, and ∞ otherwise;
- $r|_j$ denotes the prefix of r consisting of its j first elements if $j \leq \#r$;
- r_j denotes the j'th element of r if $1 \leq j \leq \#r$;

- $\mathsf{dom}(r)$ denotes the set of indices corresponding to r, i.e. $\mathsf{dom}(r) = \{j \mid 1 \leq j \leq \#r\}$;
- $\mathsf{rng}(r)$ denotes the set of elements in r, i.e. $\mathsf{rng}(r) = \{r_j \mid j \in \mathsf{dom}(r)\}$.

A *chain* $\hat{c}$ is an infinite sequence of stream tuples $\hat{c}_1, \hat{c}_2, \ldots$ such that for all $j \in \mathsf{N}$, $\hat{c}_j \sqsubseteq \hat{c}_{j+1}$. For any chain $\hat{c}$, $\sqcup\hat{c}$ denotes its least upper bound. Since streams may be infinite such least upper bounds always exist. $Ch(X)$, where X is a set of streams, denotes the set of all chains over X. Variables denoting chains are always decorated with a ˆ to distinguish them from other variables.

Predicates will be expressed in first order predicate logic. As usual, $\Rightarrow$ binds weaker than $\wedge$, $\vee$, $\neg$ which again bind weaker that all other function symbols. $P[^a_t]$ denotes the result of substituting t for all occurrences of the variable a in P. Predicates are sometimes classified as safety and liveness predicates. These concepts are defined as in [AS85].

P is *admissible* iff for all chains $\hat{c}$, $(\forall j \in \mathsf{N}. P(\hat{c}_j)) \Rightarrow P(\sqcup\hat{c})$. An admissible predicate holds for the least upper bound of a chain $\hat{c}$ if it holds for all members of $\hat{c}$. All safety predicates are admissible. However, there are admissible predicates which are not safety predicates. For example $\#i \bmod 2 = 0 \vee \#i = \infty$ is an admissible predicate, but not a safety predicate. $adm(P)$ holds iff P is admissible. There are well-known techniques for proving that a predicate is admissible [Man74].

A function $f \in I^\omega \to O^\omega$, where I^ω, O^ω are stream tuples, is called a *stream processing function* iff it is *prefix monotonic*:

$$\forall s, t \in I^\omega. s \sqsubseteq t \Rightarrow f(s) \sqsubseteq f(t),$$

and *prefix continuous*:

$$\forall \hat{c} \in Ch(I^\omega). f(\sqcup\hat{c}) = \sqcup\{f(\hat{c}_j) \mid j \in \mathsf{N}\}.$$

More informally, that a function is prefix monotonic basically means that if it is given more input then the output already produced cannot be changed, i.e. the output may at most be increased. Moreover, prefix continuity implies that the function's behavior for infinite inputs is completely determined by its behavior for finite inputs. The set of all stream processing functions in $I^\omega \to O^\omega$ is denoted by $I^\omega \xrightarrow{c} O^\omega$.

3. Agents

An *agent* $F : I^\omega \to O^\omega$ receives messages through a finite number of input channels of type I^ω and sends messages through a finite number of output channels of type O^ω.

The *denotation* of an agent F, written $[\![F]\!]$, is a set of type correct stream processing functions. Hence, from the declaration above it follows that $[\![F]\!] \subseteq I^\omega \xrightarrow{c} O^\omega$.

Agents can be nondeterministic. This is reflected by the fact that sets of functions are used as denotations. Any function $f \in [\![F]\!]$ represents a possible behavior of F. The agent may "choose" freely among these functions. Obviously, if there is no choice, the agent is deterministic. Hence, we call F *deterministic* if its denotation is a unary set and *nondeterministic*, otherwise.
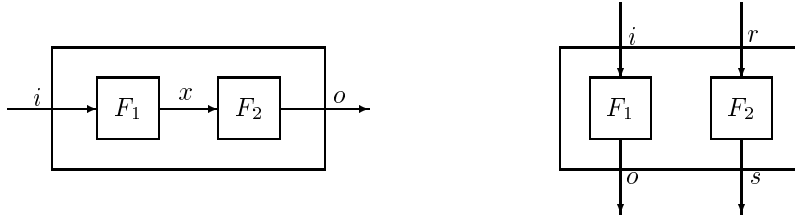
Fig. 1. Sequential and Parallel Composition.

Given two agents $F_1 : I^\omega \to X^\omega$ and $F_2 : X^\omega \to O^\omega$, then $F_1 \circ F_2$ is of type $I^\omega \to O^\omega$ and represents the *sequential composition* of F_1 and F_2. Its denotation is

$$[\![\, F_1 \circ F_2 \,]\!] \stackrel{\text{def}}{=} \{f_1 \circ f_2 \mid f_1 \in [\![\, F_1 \,]\!] \wedge f_2 \in [\![\, F_2 \,]\!]\},$$

where $f_1 \circ f_2(i) \stackrel{\text{def}}{=} f_2(f_1(i))$. Figure 1 shows the situation. Each arrow stands for a finite number of channels. In contrast to e.g. CSP-programs or sequential programs, F_1 need not terminate before F_2 starts to compute. Instead F_1 and F_2 work in a pipelined manner.

Given two agents $F_1 : I^\omega \to O^\omega$ and $F_2 : R^\omega \to S^\omega$, then $F_1 \parallel F_2$ is of type $I^\omega \times R^\omega \to O^\omega \times S^\omega$ and represents the *parallel composition* of F_1 and F_2. Its denotation is

$$[\![\, F_1 \parallel F_2 \,]\!] \stackrel{\text{def}}{=} \{f_1 \parallel f_2 \mid f_1 \in [\![\, F_1 \,]\!] \wedge f_2 \in [\![\, F_2 \,]\!]\},$$

where $f_1 \parallel f_2(i, r) \stackrel{\text{def}}{=} (f_1(i), f_2(r))$. Parallel composition is also shown in Figure 1. F_1 and F_2 are simply put side by side and work independently without any mutual communication.

Let $F : I^\omega \times Y^\omega \to O^\omega \times Y^\omega$ be an agent, where Y is a data set (and not a tuple). Then the last input channel of F has the same type as the last output channel, and they can be connected as depicted in Figure 2. This is called *feedback*. The resulting construct μF is of type $I^\omega \to O^\omega \times Y^\omega$, and its denotation is

$$[\![\, \mu F \,]\!] \stackrel{\text{def}}{=} \{\mu f \mid f \in [\![\, F \,]\!]\},$$

where $\mu f(i) \stackrel{\text{def}}{=} (o, y)$ iff

- $f(i, y) = (o, y)$,
- $\forall o' \in O^\omega . \forall y' \in Y^\omega . f(i, y') = (o', y') \Rightarrow (o, y) \sqsubseteq (o', y')$.

(o, y) is called the *least fixpoint* of f with respect to i. The continuity of stream processing functions ensures that there is a least fixpoint.

Let $F_1 : I^\omega \times X^\omega \to O^\omega \times Y^\omega$ and $F_2 : Y^\omega \times R^\omega \to X^\omega \times S^\omega$ be agents, where X and Y are data sets. They can then be connected as in Figure 2. This is called *mutual feedback*. The resulting construct $F_1 \otimes F_2$ is of type $I^\omega \times R^\omega \to O^\omega \times Y^\omega \times X^\omega \times S^\omega$, and its denotation is
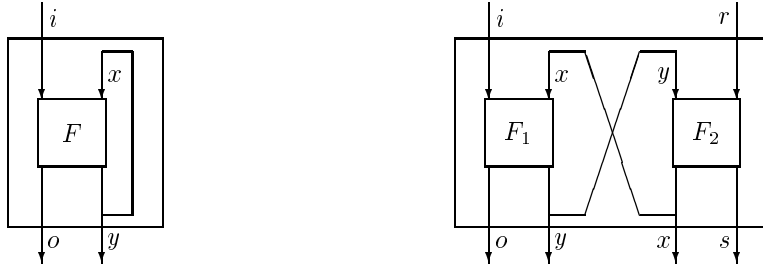
Fig. 2. Feedback and Mutual Feedback.

$$[\![\, F_1 \otimes F_2 \,]\!] \overset{\text{def}}{=} \{f_1 \otimes f_2 \,|\, f_1 \in [\![\, F_1 \,]\!] \wedge f_2 \in [\![\, F_2 \,]\!]\},$$

where $f_1 \otimes f_2(i, r) \overset{\text{def}}{=} (o, y, x, s)$ iff

- $f_1(i, x) = (o, y)$,
- $f_2(y, r) = (x, s)$,
- $\forall o' \in O^\omega. \forall x' \in X^\omega. \forall y' \in Y^\omega. \forall s' \in S^\omega.$
  $f_1(i, x') = (o', y') \wedge f_2(y', r) = (x', s') \Rightarrow (o, y, x, s) \sqsubseteq (o', y', x', s').$

It is easy to generalize the μ and $\otimes$ operators to enable feedback of stream tuples. The rules presented below remain valid (see [SDW93]).

The denotation of any network generated from some given *basic agents* using the operators $\circ$, $\|$, μ and $\otimes$ is a set of stream processing functions, and if all constituents of a network are deterministic agents, the denotation of the network is a singleton set. This is a well-known result, which dates back to [Kah74]. It makes it possible to replace an agent by a network of simpler agents that has the same denotation. This is the key concept that enables modular top-down development.

In this paper we distinguish between agents which are *syntactic* entities and their *semantic* representation as sets of stream processing functions. The four operators $\circ$, $\|$, μ and $\otimes$ can be thought of as constructs in a programming language. Thus, given some notation for characterizing the *basic agents* of a network, i.e. the "atomic" building blocks, networks can be represented in a program-like notation (see [Ded92]).

However, since we are concerned with agents which are embedded in environments, a basic agent is not always a program. It may also be a specification representing some sort of physical device, like, for instance, an unreliable wire connecting two computers, or even a human being working in front of a terminal. Of course, such agents do not always correspond to computable functions, and it is not the task of the program designer to develop such agents. However, in a program design it is often useful to be able to specify agents of this type.

4. Simple Specifications

In our approach an agent communicates with its environment via unbounded FIFO channels. Hence, at least in the case of deterministic agents, it seems

natural to define the environment assumption as a predicate on the history of the input channels, i.e. on the input streams, and the commitment as a predicate on the history of the input and output channels, i.e. as a relation between the input and output streams. The result is what we call a simple specification.

More formally, a *simple specification* is a pair of predicates $[A, C]$, where $A \in I^\omega \to \mathsf{B}$ and $C \in I^\omega \times O^\omega \to \mathsf{B}$. Its *denotation* $[\![\, [A, C] \,]\!]$ is the set of all type correct stream processing functions which satisfies the specification. More formally:

$$[\![\, [A, C] \,]\!] \stackrel{\mathrm{def}}{=} \{ f \in I^\omega \stackrel{c}{\to} O^\omega \mid \forall i \in I^\omega.\, A(i) \Rightarrow C(i, f(i)) \}.$$

In other words, the denotation is the set of all type correct stream processing functions f such that whenever the input i of f fulfills the assumption A, the output $f(i)$ is related to i in accordance with the commitment C.

Example 1. One Element Buffer:
As a first example, consider the task of specifying a buffer capable of storing exactly one data element. The environment may either send a data element to be stored or a request for the data element currently stored. The environment is assumed to be such that no data element is sent when the buffer is full, and no request is sent when the buffer is empty. The buffer, on the other hand, is required to store any received data element and to output the stored data element and become empty after receiving a request.

Let D be the set of data, and let ? represent a request, then it is enough to require the buffer to satisfy the specification RB, where

$$A_{\mathrm{RB}}(i) \stackrel{\mathrm{def}}{=} \forall i' \in (D \cup \{?\})^*.\; i' \sqsubseteq i \Rightarrow \#\{?\}\copyright i' \le \#D\copyright i' \le \#\{?\}\copyright i' + 1,$$
$$C_{\mathrm{RB}}(i, o) \stackrel{\mathrm{def}}{=} o \sqsubseteq D\copyright i \wedge \#o = \#\{?\}\copyright i.$$

The assumption states that no request is sent to an empty buffer (first inequality), and that no data element is sent to a full buffer (second inequality). The commitment requires that the buffer transmits data elements in the order they are received (first conjunct), and moreover that the buffer always eventually responds to a request (second conjunct). □

The operators $\circ$, $\|$, μ and $\otimes$ can be used to compose specifications, and also specifications and agents in a straightforward way. By a *mixed specification* we mean an agent, a simple specification or any network built from agents and simple specifications using the four composition operators. Since simple specifications denote sets of stream processing functions, the denotation of a mixed specification is defined in exactly the same way as for networks of agents.

During program development it is important that the specifications which are to be implemented remain *implementable*, i.e. that they remain fulfillable by computer programs. From a practical point of view, it is generally accepted that it does not make much sense to formally check the implementability of a specification. The reason is that to prove implementability it is often necessary to construct a program which fulfills the specification, and that is of course the goal of the whole program design exercise.

A weaker and more easily provable constraint is what we call feasibility. A simple specification $[A, C]$ is *feasible* iff its denotation is nonempty, i.e. iff $[\![\, [A, C] \,]\!] \ne \emptyset$.

Feasibility corresponds to what is called feasibility in [Mor88], satisfiability in VDM [Jon90] and realizability in [AL90]. A non-feasible specification is inconsistent and can therefore not be fulfilled by any agent. On the other hand, there are stream processing functions that cannot be expressed in any algorithmic language. Thus, that a specification is feasible does not guarantee that it is implementable. See [Bro94] for a detailed discussion of feasibility and techniques for proving that a specification is feasible.

Example 2. Non-Feasible Specification:
An example of a non-feasible specification is $[A, C]$ where

$$A(r) \stackrel{\mathsf{def}}{=} \mathsf{true},$$
$$C(r, s) \stackrel{\mathsf{def}}{=} \#r = \infty \Leftrightarrow \#s < \infty.$$

To see that this specification is not feasible, assume the opposite. This means it is satisfied by at least one stream processing function f. f is continuous which implies that for any strictly increasing chain $\hat{r}$ we have:

$$f(\sqcup \hat{r}) = \sqcup \{f(\hat{r}_j) \mid j \in \mathsf{N}\}.$$

Since $\hat{r}$ is strictly increasing, it follows for all $j \geq 1$, $\#\hat{r}_j < \infty$, and therefore also $\#f(\hat{r}_j) = \infty$. Hence:

$$\#f(\sqcup \hat{r}) = \# \sqcup \{f(\hat{r}_j) \mid j \in \mathsf{N}\} = \infty.$$

On the other hand, since $\hat{r}$ is strictly increasing we have $\#(\sqcup \hat{r}) = \infty$ which implies $\#f(\sqcup \hat{r}) < \infty$. This is a contradiction. Thus the specification is not feasible. $\square$

A simple specification $[A_2, C_2]$ is said to *refine* a simple specification $[A_1, C_1]$, written $[A_1, C_1] \rightsquigarrow [A_2, C_2]$, iff the denotation of the former is contained in or equal to the denotation of the latter, i.e. iff $[\![\, [A_2, C_2]\,]\!] \subseteq [\![\, [A_1, C_1]\,]\!]$.

This relation can again be generalized to mixed specifications. Given a requirement specification $[A, C]$, the goal of a system design is to construct an agent F such that $[A, C] \rightsquigarrow F$ holds. The refinement relation $\rightsquigarrow$ is reflexive, transitive and a congruence with respect to the composition operators. Hence, $\rightsquigarrow$ admits compositional system development: once a specification is decomposed into a network of subspecifications, each of these subspecifications can be further refined in isolation.

We will now present the refinement rules for simple specifications. The first refinement rule states that a specification's assumption can be weakened and its commitment can be strengthened:

Rule 1 :
$$A_1 \Rightarrow A_2$$
$$\frac{A_1 \wedge C_2 \Rightarrow C_1}{[A_1, C_1] \rightsquigarrow [A_2, C_2]}$$

To see that Rule 1 is sound, observe that if f is a stream processing function such that $f \in [\![\, [A_2, C_2]\,]\!]$, then since the first premise implies that the new assumption A_2 is weaker than the old assumption A_1, and the second premise

implies that the new commitment C_2 is stronger than the old commitment C_1 for any input which satisfies A_1, it is clear that $f \in [\![\, [A_1, C_1]\,]\!]$.

That $\rightsquigarrow$ is transitive and a congruence with respect to the composition operators can of course also be stated as refinement rules:

Rule 2 :
$$\frac{Spec_1 \rightsquigarrow Spec_2 \qquad Spec_2 \rightsquigarrow Spec_3}{Spec_1 \rightsquigarrow Spec_3}$$

Rule 3 :
$$\frac{Spec_1 \rightsquigarrow Spec_2}{Spec \rightsquigarrow Spec(Spec_2/Spec_1)}$$

$Spec_1$, $Spec_2$ and $Spec_3$ denote mixed specifications. In Rule 3 $Spec(Spec_2/Spec_1)$ denotes some mixed specification which can be obtained from the mixed specification $Spec$ by substituting $Spec_2$ for one occurrence of $Spec_1$.

Since stream processing functions are monotonic and continuous it is not necessary to state monotonicity and continuity constraints explicitly in the specifications. For example, in Example 1 it is not possible to deduce that an implementation must behave continuously from the predicate C_{RB} alone. However, when reasoning formally about specifications, it is often necessary to state these implicit constraints explicitly, and for this purpose the following rule is useful:

Rule 4 :
$$\frac{C_2 \wedge (\forall \hat{c}.\ \sqcup \hat{c} = i \wedge (\forall j \in \mathsf{N}.\ A_2[^{i}_{\hat{c}_j}]) \Rightarrow \exists \hat{p}.\ \sqcup \hat{p} = o \wedge \forall j \in \mathsf{N}.\ C_2[^{i}_{\hat{c}_j}\ {}^{o}_{\hat{p}_j}]) \Rightarrow C_1}{[A, C_1] \rightsquigarrow [A, C_2]}$$

$\hat{c}, \hat{p}$ are chains. The soundness of Rule 4 follows from the continuity of stream processing functions. Rule 4 is a so-called adaptation rule. There are of course a number of other adaptation rules that may be helpful. For example, Rule 4 only states that a correct implementation must behave continuously with respect to any prefix of the input i. It does not state explicitly that the behavior also must be continuous for any further input, i.e. extension of i. This implicit constraint can of course also be captured in terms of an adaptation rule. However since Rule 4 is the only adaptation rule needed below, all other adaptation rules are left out.

Given that the input/output variables are named in accordance with Figure 1 on Page 5, then the rule for sequential composition can be formulated as follows:

Rule 5 :
$$\frac{\begin{array}{l} A \Rightarrow A_1 \\ A \wedge C_1 \Rightarrow A_2 \\ A \wedge C_1 \wedge C_2 \Rightarrow C \end{array}}{[A, C] \rightsquigarrow [A_1, C_1] \circ [A_2, C_2]}$$

This rule states that in any environment, a specification can be replaced by the sequential composition of two component specifications provided the three premises hold.

Observe that all stream variables occurring in a premise are local with respect to that premise. This means that Rule 5 is a short-hand for the following rule:

$$\frac{\begin{array}{l}\forall i \in I^\omega.\, A(i) \Rightarrow A_1(i)\\[2pt] \forall i \in I^\omega.\, \forall x \in X^\omega.\, A(i) \wedge C_1(i,x) \Rightarrow A_2(x)\\[2pt] \forall i \in I^\omega.\, \forall o \in O^\omega.\, \forall x \in X^\omega.\, A(i) \wedge C_1(i,x) \wedge C_2(x,o) \Rightarrow C(i,o)\end{array}}{[A,C] \rightsquigarrow [A_1,C_1] \circ [A_2,C_2]}$$

Throughout this paper, all free variables occurring in the premises of refinement rules are universally quantified in this way.

To prove soundness it is necessary to show that for any pair of stream processing functions f_1 and f_2 in the denotations of the first and second component specification, respectively, their sequential composition satisfies the overall specification. To see that this is the case, firstly observe that the assumption A is at least as restrictive as A_1, the assumption of f_1. Since f_1 satisfies $[A_1,C_1]$, this ensures that whenever $A(i)$ holds, f_1's output x is such that $C_1(i,x)$. Now, the second premise implies that any such x also meets the assumption A_2 of f_2. Since f_2 satisfies $[A_2,C_2]$, it follows that the output o of f_2 is such that $C_2(x,o)$. Thus we have shown that $C_1(i,x) \wedge C_2(x,o)$ characterizes the overall effect of $f_1 \circ f_2$ when the overall input stream satisfies A, in which case the desired result follows from premise three.

If the input and output variables are named in accordance with Figure 1 on Page 5, i.e. the input variables are disjoint from the output variables, and the variables of the left-hand side component are disjoint from the variables of the right-hand side component, the parallel rule

Rule 6 :
$$\frac{\begin{array}{l}A \Rightarrow A_1 \wedge A_2\\[2pt] A \wedge C_1 \wedge C_2 \Rightarrow C\end{array}}{[A,C] \rightsquigarrow [A_1,C_1] \parallel [A_2,C_2]}$$

is almost trivial. Since the overall assumption A implies the component assumptions A_1 and A_2, and moreover the component commitments C_1 and C_2, together with the overall assumption imply the overall commitment C, the overall specification can be replaced by the parallel composition of the two component specifications.

Also in the case of the feedback rule the variable lists are implicitly given — this time with respect to Figure 2 on Page 6. This means that the component specification $[A_1,C_1]$ has $(i,x)/(o,y)$ as input/output variables, and that the overall specification $[A,C]$ has $(i)/(o,y)$ as input/output variables.

Rule 7 :
$$\frac{\begin{array}{l}A \Rightarrow adm(\lambda x.A_1)\\[2pt] A \Rightarrow A_1[^x_{\langle\rangle}]\\[2pt] A \wedge A_1[^x_y] \wedge C_1[^x_y] \Rightarrow C\\[2pt] A \wedge A_1 \wedge C_1 \Rightarrow A_1[^x_y]\end{array}}{[A,C] \rightsquigarrow \mu\,[A_1,C_1]}$$

The rule is based on the stepwise computation of the feedback streams formally characterized by Kleene's theorem [Kle52], i.e. the generation of the so-called Kleene chain. Initially the feedback streams are empty. Then the agent starts to work consuming input and producing output in a stepwise manner. Output on

the feedback channels becomes input again, triggering the agent to produce additional output. This process goes on until a "stable situation" is reached (which implies that it may go on forever). Formally a "stable situation" corresponds to the least fixpoint of the recursive equation in the feedback definition on Page 5.

The feedback rule has a close similarity to the while-rule of Hoare logic. A_1 can be thought of as the invariant. The invariant holds initially (second premise), and is maintained by each computation step (fourth premise), in which case it also holds after infinitely many computation steps (first premise). The conclusion is then a consequence of premise three.

The mutual feedback rule may be formulated in a similar way:

Rule 8 :
$$A \Rightarrow adm(\lambda x.A_1) \vee adm(\lambda y.A_2)$$
$$A \Rightarrow A_1[\substack{x \\ \langle\rangle}] \vee A_2[\substack{y \\ \langle\rangle}]$$
$$A \wedge A_1 \wedge C_1 \wedge A_2 \wedge C_2 \Rightarrow C$$
$$A \wedge A_1 \wedge C_1 \Rightarrow A_2$$
$$\frac{A \wedge A_2 \wedge C_2 \Rightarrow A_1}{[A,C] \rightsquigarrow [\exists r.\, A_1, C_1] \otimes [\exists i.\, A_2, C_2]}$$

In accordance with Figure 2 on Page 6, the component specifications have respectively $(i,x)/(o,y)$ and $(y,r)/(x,s)$ as input/output variables, and the overall specification has $(i,r)/(o,y,x,s)$ as input/output variables. In some sense, this rule can be seen as a "generalization" of Rule 7. Due to the continuity constraint on stream processing functions, it is enough if one of the agents "kicks off". This means that we may use $A_1 \vee A_2$ as invariant instead of $A_1 \wedge A_2$. It follows from premises four and five that if one of the component assumptions holds for one element of the Kleene-chain, then the other component assumption holds for the next element of the Kleene-chain. Since the second premise implies that at least one of the component assumptions holds for the first element of the Kleene-chain it follows that that both components assumptions holds for infinitely many elements of the Kleene-chain. The first premise then implies that one of the component assumptions holds for the least upper bound of the Kleene-chain, in which case premises four and five imply that both component assumptions hold for the least upper bound of the Kleene-chain. The conclusion is then a consequence of premise three.

Note, that without the existential quantifiers occurring in the component specifications, the rule becomes too weak. The problem is that the input received on x may depend upon the value of r, and that the input received on y may depend upon the value of i. In the above rule these dependencies can be expressed due to the fact that r may occur in A_1 and i may occur in A_2.

Example 3. Summation Agent:
The task is to design an agent which for each natural number received on its input channel, outputs the sum of all numbers received up to that point in time. The environment is assumed always eventually to send a new number. In other words, we want to design an agent which refines the specification SUM, where

$$A_{\mathrm{SUM}}(r) \stackrel{\text{def}}{=} \#r = \infty,$$
$$C_{\mathrm{SUM}}(r,o) \stackrel{\text{def}}{=} \#o = \infty \wedge \forall j \in \mathbb{N}.\, o_j = \sum_{k=1}^{j} r_k.$$
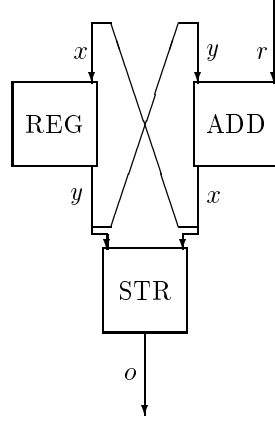
Fig. 3. Network Refining SUM.

SUM can be refined by a network (REG $\otimes$ ADD) $\circ$ STR as depicted in Figure 3. ADD is supposed to describe an agent which, given two input streams of natural numbers, generates an output stream where each element is the sum of the corresponding elements of the input streams, e.g. the n'th element of the output stream is equal to the sum of n'th elements of the two input streams. REG, on the other hand, is required to specify an agent which outputs its input stream prefixed with 0. Thus an agent characterized by REG can be thought of as a register which stores the last number received on its input channel. Its initial value is 0. This means that if $A_{\text{SUM}}(r)$ holds then

$$x = \langle \Sigma_{j=1}^{1} r_j \rangle \frown \langle \Sigma_{j=1}^{2} r_j \rangle \frown \ \ldots \ \frown \langle \Sigma_{j=1}^{n} r_j \rangle \frown \ \ldots \ ,$$

where x is the right-hand side output stream of (REG $\otimes$ ADD). Hence, it is enough to require STR to characterize an agent which outputs its second input stream. More formally:

$$[A_{\text{REG}}, C_{\text{REG}}],$$
$$[A_{\text{ADD}}, C_{\text{ADD}}],$$
$$[A_{\text{STR}}, C_{\text{STR}}],$$

where

$$A_{\text{REG}}(x) \stackrel{\text{def}}{=} \text{true},$$
$$C_{\text{REG}}(x, y) \stackrel{\text{def}}{=} y = \langle 0 \rangle \frown x,$$

$$A_{\text{ADD}}(y, r) \stackrel{\text{def}}{=} \#r = \infty,$$
$$C_{\text{ADD}}(y, r, x) \stackrel{\text{def}}{=} \#x = \#y \wedge \forall j \in \text{dom}(x).\, x_j = r_j + y_j,$$

$$A_{\text{STR}}(y, x) \stackrel{\text{def}}{=} \text{true},$$
$$C_{\text{STR}}(y, x, o) \stackrel{\text{def}}{=} o = x.$$

The rules introduced above can be used to formally prove that this decomposition is correct. Let

$$A'(r) \stackrel{\text{def}}{=} A_{\text{SUM}}(r),$$
$$C'(r, y, x) \stackrel{\text{def}}{=} C_{\text{SUM}}(r, x).$$

Since

$$A_{\text{SUM}} \Rightarrow A',$$
$$C' \Rightarrow A_{\text{STR}},$$
$$C' \wedge C_{\text{STR}} \Rightarrow C_{\text{SUM}},$$

it follows from Rule 5 that

$$[A_{\text{SUM}}, C_{\text{SUM}}] \rightsquigarrow [A', C'] \circ [A_{\text{STR}}, C_{\text{STR}}]. \qquad\qquad (*)$$

Moreover, since it is straightforward to prove that

$$A' \Rightarrow adm(A_{\text{REG}}) \vee adm(\lambda y.A_{\text{ADD}}),$$
$$A' \Rightarrow A_{\text{REG}}[\begin{smallmatrix}x\\\langle\rangle\end{smallmatrix}] \vee A_{\text{ADD}}[\begin{smallmatrix}y\\\langle\rangle\end{smallmatrix}],$$
$$A' \wedge A_{\text{REG}} \wedge C_{\text{REG}} \wedge A_{\text{ADD}} \wedge C_{\text{ADD}} \Rightarrow C',$$
$$A' \wedge A_{\text{ADD}} \wedge C_{\text{ADD}} \Rightarrow A_{\text{REG}},$$
$$A' \wedge A_{\text{REG}} \wedge C_{\text{REG}} \Rightarrow A_{\text{ADD}},$$

it follows from Rule 8 that

$$[A', C'] \rightsquigarrow [A_{\text{REG}}, C_{\text{REG}}] \otimes [A_{\text{ADD}}, C_{\text{ADD}}].$$

This, $(*)$ and Rules 2 and 3 imply

$$[A_{\text{SUM}}, C_{\text{SUM}}] \rightsquigarrow ([A_{\text{REG}}, C_{\text{REG}}] \otimes [A_{\text{ADD}}, C_{\text{ADD}}]) \circ [A_{\text{STR}}, C_{\text{STR}}].$$

Thus, the proposed decomposition is valid. Further refinements of the three component specifications ADD, REG and STR may now be carried out in isolation. □

In the example above we needed the agent STR because $\otimes$ does not hide the feedback channel y. It is of course straightforward to define an operator $\oplus$ which only differs from $\otimes$ in that the feedback channel represented by y is hidden. Rule 8 is also valid for $\oplus$ if C is restricted from having occurrences of y. On other occasions operators, where only the feedback channel represented by x is hidden, or where both feedback channels are hidden, are needed. Instead of introducing operators and rules for each of these situations we overload and use $\otimes$ for all four. It will always be clear from the context which version is intended.

Example 4. Recording the Maximum:
In Example 3 the verification was straightforward. Using the overloaded $\otimes$ introduced above, we will now look at a variation of Example 3, which is more complicated in the sense that it is necessary to strengthen the component assumptions with two invariants in order to use Rule 8.

The task is to develop an agent with one input channel r and one output

channel x, which for any natural number received on r outputs the maximum natural number received so far along x. More formally an agent which satisfies

$$[A_{\mathrm{MAX}}, C_{\mathrm{MAX}}],$$

where

$$A_{\mathrm{MAX}}(r) \stackrel{\mathrm{def}}{=} \mathsf{true},$$
$$C_{\mathrm{MAX}}(r, x) \stackrel{\mathrm{def}}{=} \#x = \#r \wedge \forall j \in \mathsf{dom}(x).x_j = \mathsf{max}(\mathsf{rng}(r|_j)).$$

This specification can be decomposed into two component specifications, REG and COMP, as shown in Figure 4.
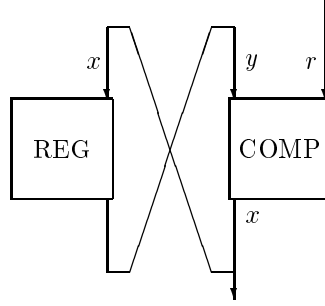
Fig. 4. Network Refining MAX.

As before, REG specifies a register which stores the last number received on x. Its initial value is still 0. However, the register is fragile in the sense that it is guaranteed to behave correctly only if the input stream is nondecreasing:

$$A_{\mathrm{REG}}(x) \stackrel{\mathrm{def}}{=} \forall j \in \mathsf{dom}(x).j \neq \#x \Rightarrow x_j \leq x_{j+1},$$
$$C_{\mathrm{REG}}(x, y) \stackrel{\mathrm{def}}{=} y = \langle 0 \rangle^\frown x.$$

COMP, on the other hand, compares any natural number received on r with the corresponding number received on y. The maximum of these two numbers is chosen and output along x.

$$A_{\mathrm{COMP}}(y, r) \stackrel{\mathrm{def}}{=} \mathsf{true},$$
$$C_{\mathrm{COMP}}(y, r, x) \stackrel{\mathrm{def}}{=} \#x = \mathsf{min}\{\#y, \#r\} \wedge \forall j \in \mathsf{dom}(x).x_j = \mathsf{max}\{y_j, r_j\}.$$

The first conjunct restricts any correct implementation to output exactly one message along the output channel for each pair of messages it receives on its two input channels. The second makes sure that the maximum is chosen.

To prove that this decomposition is correct, it must be shown that

$$[A_{\mathrm{MAX}}, C_{\mathrm{MAX}}] \rightsquigarrow [A_{\mathrm{REG}}, C_{\mathrm{REG}}] \otimes [A_{\mathrm{COMP}}, C_{\mathrm{COMP}}]. \qquad (*)$$

Unfortunately, it is not possible to use Rule 8 directly. This because

$$A_{\mathrm{MAX}} \wedge A_{\mathrm{COMP}} \wedge C_{\mathrm{COMP}} \Rightarrow A_{\mathrm{REG}}$$

does not hold. What is missing is the relationship between r and y. We will therefore strengthen the the component assumptions with two invariants:

$$I_{\mathrm{REG}}(x, r) \stackrel{\mathsf{def}}{=} \forall j \in \mathsf{dom}(x).x_j = \mathsf{max}(\mathsf{rng}(r|_j)) \wedge \#x \leq \#r,$$

$$I_{\mathrm{COMP}}(y, r) \stackrel{\mathsf{def}}{=} \forall j \in \mathsf{dom}(y).y_j = \mathsf{max}(\mathsf{rng}(r|_{j-1}) \cup \{0\}) \wedge \#y \leq \#r + 1.$$

Since Rule 1 implies that

$$[\exists r.I_{\mathrm{REG}}, C_{\mathrm{REG}}] \rightsquigarrow [A_{\mathrm{REG}}, C_{\mathrm{REG}}],$$
$$[I_{\mathrm{COMP}}, C_{\mathrm{COMP}}] \rightsquigarrow [A_{\mathrm{COMP}}, C_{\mathrm{COMP}}],$$

it follows from Rules 2 and 3 that $(*)$ holds if it can be shown that

$$[A_{\mathrm{MAX}}, C_{\mathrm{MAX}}] \rightsquigarrow [\exists r.I_{\mathrm{REG}}, C_{\mathrm{REG}}] \otimes [I_{\mathrm{COMP}}, C_{\mathrm{COMP}}].$$

According to Rule 8 the latter holds if we can prove that

$$adm(\lambda x.I_{\mathrm{REG}}) \vee adm(\lambda y.I_{\mathrm{COMP}}),$$
$$I_{\mathrm{REG}}[\begin{smallmatrix}x\\()\end{smallmatrix}] \vee I_{\mathrm{COMP}}[\begin{smallmatrix}y\\()\end{smallmatrix}],$$
$$I_{\mathrm{REG}} \wedge C_{\mathrm{REG}} \wedge I_{\mathrm{COMP}} \wedge C_{\mathrm{COMP}} \Rightarrow C_{\mathrm{MAX}},$$
$$I_{\mathrm{REG}} \wedge C_{\mathrm{REG}} \Rightarrow I_{\mathrm{COMP}},$$
$$I_{\mathrm{COMP}} \wedge C_{\mathrm{COMP}} \Rightarrow I_{\mathrm{REG}}.$$

Since I_{REG} and I_{COMP} are safety predicates with respect to x and y, respectively, it is clear that the first premise holds. The second premise is trivial. To prove premises three, four and five is also straightforward. $\square$

The example above gives a general strategy for decomposition modulo $\otimes$: strengthen the component assumptions with appropriately chosen invariants and then prove that the premises of Rule 8 hold. This closely resembles a decomposition modulo the while-construct in Hoare-logic. When conducting a decomposition modulo μ a similar strategy is often needed. The only difference is that in the case of $\otimes$ it is necessary to formulate two invariants, while one invariant is enough in the case of μ. These invariant strategies can of course be embedded in the actual refinement rules. Rules 7 and 8 could then be applied directly, i.e. without first using Rules 1,2 and 3 to strengthen the assumptions. From a formal point of view these two alternatives are equivalent — it is just a matter of taste. However, a practitioner would perhaps prefer to have the invariants explicitly in the rules.

Theorem 1. The refinement rules for simple specifications are sound.

Informal soundness proofs have been given above. More detailed proofs for Rules 7 and 8 can be found in the appendix.

In the examples above a predicate calculus related *assertion language* has been employed for writing specifications. However, in this paper no assertion language has been formally defined, nor have we formulated any *assertion logic*

for discharging the premises of our rules; we have just implicitly assumed the existence of these things. This will continue. We are just mentioning these concepts here because they play a role in the discussion below.

The logic introduced in this chapter is *(semantic) relative complete* in the following sense:

Theorem 2. If F is a deterministic agent built from basic deterministic agents using the operators for sequential composition, parallel composition, feedback and mutual feedback, and $[A, C] \rightsquigarrow F$, then F can be deduced from $[A, C]$ using Rules 1-3 and 5-8, given that

- such a deduction can always be carried out for a basic deterministic agent[2],
- any valid formula in the assertion logic is provable,
- any predicate we need can be expressed in the assertion language.

See the appendix for a detailed proof. Note that under the same expressiveness assumption as above, for any deterministic agent F, there is a simple specification $Spec$ such that $[\![\, F \,]\!] = [\![\, Spec \,]\!]$. Let $[\![\, F \,]\!] = \{f\}$ then $[\mathsf{true}, f(i) = o]$ is semantically equivalent to F.

5. Symmetric Specifications

In Section 4 it is explained what it means for an agent F, either deterministic or nondeterministic, to fulfill a simple specification $[A, C]$. Thus, simple specifications can quite naturally be used to specify nondeterministic agents, too. However, they are not expressive enough, i.e. not every nondeterministic agent can be specified by a simple specification. One problem is that for certain nondeterministic agents, the assumption cannot be formulated without some knowledge about the output. To understand the point, consider a modified version of the one element buffer:

Example 5. One Element Unreliable Buffer:
Basically the buffer should exhibit the same behavior as the one element buffer described in Example 1. In addition we now assume that it is unreliable in the sense that data communicated by the environment can be rejected. Special messages are issued to inform the environment about the outcome, namely *fail* if a data element is rejected and *ok* if it is accepted. Again the environment is assumed to send a request only if the buffer is full and a data element only if the buffer is empty. It follows from this description that the environment has to take the buffer's output into account in order to make sure that the messages it sends to the buffer are consistent with the buffer's input assumption. The example is worked out formally on page 19. □

At a first glance it seems that the weakness of simple specifications can be fixed by allowing assumptions to depend upon the output, too, i.e. by allowing specifications like $[A, C]$, with $A, C \in I^{\omega} \times O^{\omega} \rightarrow \mathsf{B}$, and

---

[2] Remember (see Section 3, Page 6) we have not given any notation (programming constructs) for expressing basic agents — thus we have to assume that there are some relative complete rules with respect to the chosen notation

$$[\![\, [A, C]\,]\!] = \{f \in I^{\omega} \xrightarrow{c} O^{\omega} \mid \forall i \in I^{\omega} . \, A(i, f(i)) \Rightarrow C(i, f(i))\}$$

We call such specifications *symmetric* since A and C are now treated symmetrically with respect to the input/output streams. Unfortunately, we may then write strange specifications like

$$[\#i \neq \infty \wedge \#i = \#o, i = o] \qquad (*)$$

which is not only satisfied by the identity agent, but also for example by any agent which for all inputs falsifies the assumption[3].

   Another argument against symmetric specifications is that in order to formulate sufficiently strong assumptions, what is needed is not really information about the agents output, but information about the nondeterministic choices taken by the agent, i.e. only information about some aspects of the output.

   A third and more serious problem is that symmetric specifications are insufficiently expressive. Consider the following example (taken from [Bro92]):

Example 6. :
Let $f_1, f_2, f_3, f_4 \in \{1\}^{\omega} \xrightarrow{c} \{1\}^{\omega}$ be such that

$$f_1(\langle\rangle) = f_2(\langle\rangle) = \langle 1 \rangle,$$
$$f_1(\langle 1 \rangle) = f_4(\langle 1 \rangle) = \langle 1, 1 \rangle,$$
$$f_3(\langle\rangle) = f_4(\langle\rangle) = \langle\rangle,$$
$$f_2(\langle 1 \rangle) = f_3(\langle 1 \rangle) = \langle 1 \rangle,$$
$$y = \langle 1, 1 \rangle \frown x \Rightarrow f_1(y) = f_2(y) = f_3(y) = f_4(y) = \langle 1, 1 \rangle.$$

Assume that F_1 and F_2 are agents such that $[\![\, F_1\,]\!] = \{f_1, f_3\}$ and $[\![\, F_2\,]\!] = \{f_2, f_4\}$. Then F_1 and F_2 determine exactly the same input/output relation. Thus for any symmetric specification $Spec$, $Spec \rightsquigarrow F_1$ iff $Spec \rightsquigarrow F_2$. In other words, there is no symmetric specification which distinguishes F_1 from F_2.

   Nevertheless, semantically the difference between F_1 and F_2 is not insignificant, because the two agents have different behaviors with respect to the feedback operator. To see this, firstly observe that $\mu f_1 = \langle 1, 1 \rangle$, $\mu f_2 = \langle 1 \rangle$ and $\mu f_3 = \mu f_4 = \langle\rangle$. Thus μF_1 may either output $\langle 1, 1 \rangle$ or $\langle\rangle$, while μF_2 may either output $\langle 1 \rangle$ or $\langle\rangle$. $\square$

   The expressiveness problem described above is basically the Brock/Ackermann [BA81] anomaly. Due to the lack of expressiveness it can be shown that for symmetric specifications no deduction system can be found that is semantically complete for nondeterministic agents in the sense explained on Page 15. Given a specification $Spec$ and an agent F, and assume we know that $Spec \rightsquigarrow \mu F$ holds. A deduction system is compositional iff the specification of an agent can always be verified on the basis of the specifications of its subagents, without knowledge of the interior construction of those subagents [Zwi89]. This means that in a complete and compositional deduction system there must be a specification

---

[3] It can be argued that the simple specification [false, P] suffers from exactly the same problem. However, there is a slight difference. [false, P] is satisfied by any agent. The same does not hold for $(*)$. As argued in [Bro94], if any assumption A of a symmetric specification is required to satisfy $\exists i. A(i, f(i))$ for any type-correct stream processing function f, and any assumption A of a simple specification is required to satisfy $\exists i. A(i)$, then this difference disappears.

$Spec_1$, such that $Spec \rightsquigarrow \mu\, Spec_1$ and $Spec_1 \rightsquigarrow F$ are provable. For symmetric specifications no such deduction system can be found. To prove this fact we may use the agents F_1, F_2 defined in Example 6, where

$$[\![\ \mu F_1\ ]\!] = \{\mu f_1, \mu f_3\} = \{\lambda.\langle 11\rangle, \lambda.\langle\rangle\}, \quad [\![\ \mu F_2\ ]\!] = \{\mu f_2, \mu f_4\} = \{\lambda.\langle 1\rangle, \lambda.\langle\rangle\}.$$

Note that $\mu F_1, \mu F_2$ have no input channels. Let $[A, C]$ with $A, C \in \{1\}^\omega \to \mathsf{B}$ be defined by

$$A(o) \stackrel{\mathrm{def}}{=} \mathsf{true}, \qquad\qquad C(o) \stackrel{\mathrm{def}}{=} o = \langle 11\rangle \vee o = \langle\rangle.$$

Obviously, $[A, C] \rightsquigarrow \mu F_1$ is valid. Now, if there is a complete compositional deduction system then there must be a symmetric specification $[A_1, C_1]$ such that

$$[A, C] \rightsquigarrow \mu\, [A_1, C_1], \qquad (*) \qquad\qquad [A_1, C_1] \rightsquigarrow F_1. \qquad (**)$$

However, because F_1 and F_2 have exactly the same input/output behavior, there is no symmetric specification that distinguishes F_1 from F_2. Thus, it follows from $(**)$ that $[A_1, C_1] \rightsquigarrow F_2$, as well as $\mu\, [A_1, C_1] \rightsquigarrow \mu F_2$. From this, $(*)$, and the transitivity of $\rightsquigarrow$ we can conclude $[A, C] \rightsquigarrow \mu F_2$, which does not hold.

6. General Specifications

As shown in the previous section, the problem with symmetric specifications is that they are not sufficiently expressive. Roughly speaking, we need a specification concept capable of distinguishing F_1 from F_2. Since as shown in Section 4, any deterministic agent can be uniquely characterized by a simple specification, we define a *general specification* as a set of simple specifications:

$$\{[A_h, C_h] \mid H(h)\}.$$

H is a predicate characterizing a set of indices, and for each index h, $[A_h, C_h]$ is a simple specification — from now on called a *simple descendant* of the above general specification.

More formally, and in a slightly simpler notation, a general specification is of the form

$$[A, C]_H,$$

where $A \in I^\omega \times T \to \mathsf{B}$, $C \in I^\omega \times T \times O^\omega \to \mathsf{B}$, and $H \in T \to \mathsf{B}$. T is the type of the indices and H, the *hypothesis predicate*, is a predicate on this type. Its denotation

$$[\![\ [A, C]_H\ ]\!] \stackrel{\mathrm{def}}{=} \bigcup\{[\![\ [A_h, C_h]\ ]\!] \mid H(h)\},$$

with $A_h(i) \stackrel{\mathrm{def}}{=} A(i, h)$ and $C_h(i, o) \stackrel{\mathrm{def}}{=} C(i, h, o)$, is the union of the denotations of the corresponding simple specifications. This definition is equivalent to:

$$[\![\,[A,C]_H\,]\!] \overset{\text{def}}{=} \{f \in I^\omega \overset{c}{\to} O^\omega \mid \exists h \in T.$$
$$H(h) \wedge (\forall i \in I^\omega.\, A(i,h) \Rightarrow C(i,h,f(i)))\}$$

Any index h can be thought of as a *hypothesis* about the agents internal behavior. It is interesting to note the close relationship between hypotheses and what are called oracles in [Kel78] and prophecy variables in [AL88]. To see how these hypotheses can be used, let us go back to the unreliable buffer of Example 5.

Example 7. One Element Unreliable Buffer, continued:

As in Example 1, let D be the set of data, and let ? represent a request. $ok, fail$ are additional output messages. The buffer outputs $fail$ if a data element is rejected and ok if a data element is accepted. Let $\{ok, fail\}^\infty$ be the hypothesis type with

$$H_{\text{UB}}(h) \overset{\text{def}}{=} \#\{ok\}\copyright h = \infty$$

as hypothesis predicate. Thus, every infinite stream over $\{ok, fail\}$, which contains infinitely many ok's, is a legal hypotheses. The idea is that the n'th data element occurring in an input stream x corresponds to the n'th element of h, which is either equal to ok or $fail$. Now, if for a particular pair of input x and hypothesis h a data element d in x corresponds to $fail$, it will be rejected, if it corresponds to ok, it will be accepted. Thus, h predicts which data elements the buffer will accept and which it will reject. We say that the buffer behaves *according* to h.

In order to describe its behavior two auxiliary functions are employed. Let

$$state \in (D \cup \{?\})^* \times \{ok, fail\}^\infty \to \{empty, full\},$$
$$accept \in (D \cup \{?\})^\omega \times \{ok, fail\}^\infty \overset{c}{\to} D^\omega,$$

be such that for all $d \in D$, $x \in (D \cup \{?\})^*$ and $h \in \{ok, fail\}^\infty$:

$$state(\langle\rangle, h) = empty,$$
$$state(x\frown\langle?\rangle, h) = empty,$$
$$h_{\#(D\copyright x)+1} = fail \Rightarrow state(x\frown\langle d\rangle, h) = state(x, h),$$
$$h_{\#(D\copyright x)+1} = ok \Rightarrow state(x\frown\langle d\rangle, h) = full,$$

$$accept(\langle\rangle, h) = \langle\rangle,$$
$$accept(\langle?\rangle\frown x, h) = accept(x, h),$$
$$accept(\langle d\rangle\frown x, \langle fail\rangle\frown h) = accept(x, h),$$
$$accept(\langle d\rangle\frown x, \langle ok\rangle\frown h) = \langle d\rangle\frown accept(x, h).$$

$state$ is used to keep track of the buffer's state. The first equation expresses that initially the buffer is empty. The others describe how the state changes when new input arrives and the buffer behaves according to hypothesis h. In the third and fourth equation $h_{\#(D\copyright x)+1}$ denotes the element of the hypothesis stream which corresponds to d in the sense explained above. For any finite input stream x and any hypothesis h, $state(x, h)$ returns the buffer's state after it has processed x according to h. Obviously, it does not make sense to define $state$ for infinite input streams, since no buffer state can be attributed to them.

accept returns the stream of accepted data for a given input and a given hypothesis. In contrast to *state*, *accept* is defined on infinite input streams although no equation is given explicitly. Since it is defined to be a continuous function, its behavior on infinite streams follows by continuity from its behavior on finite streams.

The unreliable buffer is specified by $[A_{\text{UB}}, C_{\text{UB}}]_{H_{\text{UB}}}$ where

$$A_{\text{UB}}(x,h) \stackrel{\text{def}}{=} \forall x' \in (D \cup \{?\})^*. \forall d \in D. (x' {}^\frown \langle ? \rangle \sqsubseteq x \Rightarrow state(x',h) = full) \wedge$$
$$(x' {}^\frown \langle d \rangle \sqsubseteq x \Rightarrow state(x',h) = empty),$$

$$C_{\text{UB}}(x,h,y) \stackrel{\text{def}}{=} D \copyright y \sqsubseteq accept(x,h) \wedge$$
$$\{ok, fail\} \copyright y \sqsubseteq h \wedge \#\{ok, fail\} \copyright y = \#D \copyright x \wedge$$
$$\#D \copyright y = \#\{?\} \copyright x.$$

Intuitively, the assumption states that the environment is only allowed to send a request ? when the buffer is full and a data element d when the buffer is empty.

The commitment states in its first conjunct that each data element in the output must previously have been accepted; in its second and third conjunct that the environment is properly informed about the buffer's internal decisions; and in its fourth conjunct that every request will eventually be satisfied. □

Since the denotation of a general specification is a set of type correct stream processing functions, feasibility, mixed specifications and the refinement relation can be defined in exactly the same way as for simple specifications.

Theorem 3. Given two general specifications $Spec$, $Spec'$, with respectively T, T' as hypothesis types, and H, H' as hypothesis predicates, then $Spec \leadsto Spec'$ if there is a mapping $l: T' \to T$, such that for all $h \in T'$

1. $H'(h) \Rightarrow H(l(h))$,
2. $H'(h) \Rightarrow Spec_{l(h)} \leadsto Spec'_h$.

Here $Spec_{l(h)}$ and $Spec'_h$ are the simple descendants of $Spec$ and $Spec'$ determined by h and $l(h)$, respectively.

To see that Theorem 3 is valid, assume that the two conditions (1, 2) hold, and let $f \in [\![Spec']\!]$. Then, by the definition of $[\![ ]\!]$, there is an hypothesis h such that $f \in [\![Spec'_h]\!]$ and $H'(h)$. It follows from the two conditions that $H(l(h)) \wedge f \in [\![Spec_{l(h)}]\!]$. Thus, again by the definition of $[\![ ]\!]$, $f \in [\![Spec]\!]$.

This statement can of course easily be generalized to the case where $Spec'$ is the result of composing several general specifications using the four basic composition operators. The proof is again straightforward.

Rules 2-3 are also valid for mixed specifications containing general specifications. The other rules for general specifications are given below:

Rule 9 :
$$\frac{H \wedge A_1 \Rightarrow A_2 \qquad H \wedge A_1 \wedge C_2 \Rightarrow C_1}{[A_1, C_1] \leadsto [A_2, C_2]_H}$$

Rule 10 :
$$\exists h. H$$
$$\frac{H \wedge A_1 \Rightarrow A_2 \qquad H \wedge A_1 \wedge C_2 \Rightarrow C_1}{[A_1, C_1]_H \leadsto [A_2, C_2]}$$

Rule 11 :
$$H_2 \Rightarrow H_1[^q_{l(h)}]$$
$$H_2 \wedge A_1[^q_{l(h)}] \Rightarrow A_2$$
$$\frac{H_2 \wedge A_1[^q_{l(h)}] \wedge C_2 \Rightarrow C_1[^q_{l(h)}]}{[A_1, C_1]_{H_1} \leadsto [A_2, C_2]_{H_2}}$$

Rule 12 :

$$\frac{C_2 \wedge (\forall \hat{c}. \sqcup \hat{c} = i \wedge \forall j \in \mathsf{N}. A_2[^i_{\hat{c}_j}] \Rightarrow \exists \hat{p}. \sqcup \hat{p} = o \wedge \forall j \in \mathsf{N}. C_2[^i_{\hat{c}_j} {}^o_{\hat{p}_j}]) \Rightarrow C_1}{[A, C_1]_H \rightsquigarrow [A, C_2]_H}$$

Rule 13 :

$$\frac{\begin{array}{l} H \wedge A \Rightarrow A_1 \\ H \wedge A \wedge C_1 \Rightarrow A_2 \\ H \wedge A \wedge C_1 \wedge C_2 \Rightarrow C \end{array}}{[A, C]_H \rightsquigarrow [A_1, C_1]_H \circ [A_2, C_2]_H}$$

Rule 14 :

$$\frac{\begin{array}{l} H \wedge A \Rightarrow A_1 \wedge A_2 \\ H \wedge A \wedge C_1 \wedge C_2 \Rightarrow C \end{array}}{[A, C]_H \rightsquigarrow [A_1, C_1]_H \parallel [A_2, C_2]_H}$$

Rule 16 :

Rule 15 :

$$\frac{\begin{array}{l} H \wedge A \Rightarrow adm(\lambda x. A_1) \\ H \wedge A \Rightarrow A_1[^x_{\langle\rangle}] \\ H \wedge A \wedge A_1[^x_y] \wedge C_1[^x_y] \Rightarrow C \\ H \wedge A \wedge A_1 \wedge C_1 \Rightarrow A_1[^x_y] \end{array}}{[A, C]_H \rightsquigarrow \mu[A_1, C_1]_H}$$

$$\frac{\begin{array}{l} H \wedge A \Rightarrow adm(\lambda x. A_1) \vee adm(\lambda y. A_2) \\ H \wedge A \Rightarrow A_1[^x_{\langle\rangle}] \vee A_2[^y_{\langle\rangle}] \\ H \wedge A \wedge A_1 \wedge C_1 \wedge A_2 \wedge C_2 \Rightarrow C \\ H \wedge A \wedge A_1 \wedge C_1 \Rightarrow A_2 \\ H \wedge A \wedge A_2 \wedge C_2 \Rightarrow A_1 \end{array}}{[A, C]_H \rightsquigarrow [\exists r. A_1, C_1]_H \otimes [\exists i. A_2, C_2]_H}$$

The close relationship between simple and general specifications is reflected by Rules 9-10. This means that the two logics can be combined. Thus, general specifications and the rules for general specifications have to be introduced only at the point in a system development where they are really needed. Rules 9-10 are trivially sound, and so is Rule 12.

Rule 1 states that a simple specification can be refined by weakening the assumption and/or strengthening the commitment. For general specifications still another aspect must be considered: two general specifications may rely on different hypothesis types T_1 and T_2 or, if T_1 and T_2 coincide, different hypothesis predicates H_1 and H_2. Rule 11 captures all these aspects. Here $l \in T_2 \rightarrow T_1$ is a mapping between the two hypothesis types, and h and q are the corresponding hypotheseses. Rule 1 can be seen as a special case of Rule 11. Simply choose $T_1 = T_2$, $H_1 = H_2 = \mathsf{true}$ and let l denote the identity function. Since the first premise implies the first condition of Theorem 3 on page 20, and premises two and three together with Rule 1 imply the second condition of Theorem 3, it follows that the rule is sound.

As in the case of simple specifications there is one rule for each of the four composition operators. As for Rule 11 their soundness follows straightforwardly from (the general version of) Theorem 3 and the corresponding rules of the previous chapter. Thus:

Theorem 4. The refinement rules for general specifications are sound.

Theorem 5. If F is an agent built from basic agents using the operators for sequential composition, parallel composition, feedback and mutual feedback, and $[A, C]_H \rightsquigarrow F$, then F can be deduced from $[A, C]_H$ using Rules 2-3, 11 and 13-16, given that

- such a deduction can always be carried out for a basic agent,
- any valid formula in the assertion logic is provable,

- any predicate we need can be expressed in the assertion language.

A proof can be found in the appendix. Under the same expressiveness assumption as above, for any nondeterministic agent F, it is now straightforward to write a general specification $Spec$ which is semantically equivalent to F. Choose $I^\omega \xrightarrow{c} O^\omega$ as the hypothesis type. Then $[\![\, Spec\,]\!] = [\![\, F\,]\!]$, if

$$H_{Spec}(h) \stackrel{\text{def}}{=} h \in F, \quad A_{Spec}(i,h) \stackrel{\text{def}}{=} \text{true}, \quad C_{Spec}(i,h,o) \stackrel{\text{def}}{=} h(i) = o.$$

Roughly speaking, our specification technique uses a set of relations in the same sense as [BDD$^+$93] employs a set of functions to get around the compositionality problems reported in [BA81].

Example 8. Decomposing the Reliable Buffer:
In this example we will refine RB of Example 1 into a network of two specifications, as pictured in Figure 5. UB of Example 7 is one of the component specifications. The other one, SRV, specifies a server which is supposed to run the unreliable buffer in such a way that its unreliability is invisible from the outside.

   This is a typical situation in interactive system design: often it is fixed in advance that certain components are to be used when a given specification is to be implemented. These components can be software modules, which already exists or which are to be implemented by other developers, as well as pieces of hardware — for instance processors, storage cells or a physical wire connecting two protocol entities. Since hardware components often are unreliable, it is not uncommon that one has to deal with strange specifications like UB.
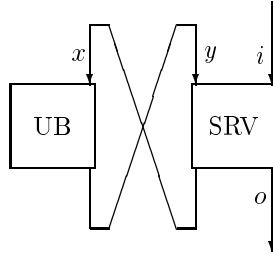
Fig. 5. Network Refining RB.

   The idea behind the server is quite simple: since the unreliable buffer may loose data elements (in which case it outputs a $fail$), the server repeatedly sends the same data element until it finally is accepted. Remember that (due to the hypotheses predicate) the specification UB guarantees that a data element is always eventually accepted provided it is sent often enough. To formally specify the server, two auxiliary functions are needed. The first one

$$ok\_in : (D \cup \{ok, fail\})^\omega \times (D \cup \{?\})^\omega \to \mathsf{B},$$

can be used to state that for any data element occurring in y (see Figure 5) there is a corresponding request in i, and that for any sequence

$$\langle fail \rangle ^\frown \langle fail \rangle ^\frown \ldots ^\frown \langle fail \rangle ^\frown \langle ok \rangle$$

occurring in y there is a corresponding data element in i.

More precisely, given that $i \in (D \cup \{?\})^\omega$, $y \in (D \cup \{ok, fail\})^\omega$, $d, d' \in D$, $x \in D \cup \{ok, fail\}$, ok_in is defined as follows:

$ok\_in(\langle\rangle, i) = \mathsf{true},$
$ok\_in(\langle x \rangle ^\frown y, \langle\rangle) = \mathsf{false},$
$y \in (D \cup \{ok, fail\})^* \Rightarrow$
        $ok\_in(\langle d' \rangle ^\frown y, \langle ? \rangle ^\frown i) = ok\_in(y, i),$
        $ok\_in(\langle ok \rangle ^\frown y, \langle ? \rangle ^\frown i) = \mathsf{false},$
        $ok\_in(\langle fail \rangle ^\frown y, \langle ? \rangle ^\frown i) = \mathsf{false},$
        $ok\_in(\langle d' \rangle ^\frown y, \langle d \rangle ^\frown i) = \mathsf{false},$
        $ok\_in(\langle ok \rangle ^\frown y, \langle d \rangle ^\frown i) = ok\_in(y, i),$
        $ok\_in(\langle fail \rangle ^\frown y, \langle d \rangle ^\frown i) = ok\_in(y, \langle d \rangle ^\frown i),$
$y \in (D \cup \{ok, fail\})^\infty \Rightarrow$
        $ok\_in(y, i) = \forall y' \in (D \cup \{ok, fail\})^*. \, y' \sqsubseteq y \Rightarrow ok\_in(y', i).$

Note that for every i, $\lambda y. ok\_in(y, i)$ is a safety predicate and hence admissible with respect to y.

The second auxiliary function

$$to\_ub : (D \cup \{ok, fail\})^\omega \times (D \cup \{?\})^\omega \xrightarrow{c} (D \cup \{?\})^\omega,$$

can be used to state that the server repeatedly sends the same data element until it receives an ok on its first input channel. For i, y and d as above, to_ub is defined by:

$to\_ub(y, \langle\rangle) = \langle\rangle$
$to\_ub(\langle\rangle, \langle ? \rangle ^\frown i) = \langle ? \rangle,$
$to\_ub(\langle\rangle, \langle d \rangle ^\frown i) = \langle d \rangle,$
$to\_ub(\langle d \rangle ^\frown y, \langle ? \rangle ^\frown i) = \langle ? \rangle ^\frown to\_ub(y, i),$
$to\_ub(\langle fail \rangle ^\frown y, \langle d \rangle ^\frown i) = \langle d \rangle ^\frown to\_ub(y, \langle d \rangle ^\frown i),$
$to\_ub(\langle ok \rangle ^\frown y, \langle d \rangle ^\frown i) = \langle d \rangle ^\frown to\_ub(y, i).$

These axioms define the behavior for compatible input, i.e. input accepted by ok_in. In some sense to_ub is the reverse of the auxiliary function *accept* which is employed in the specification of UB:

$\forall h \in \{ok, fail\})^\infty.$
    $ok\_in(y, i) \wedge \{ok, fail\} © y \sqsubseteq h \Rightarrow accept(to\_ub(y, i), h) \sqsubseteq D © i.$ $(*)$

It is also the case that

$$ok\_in(y, i) \wedge x = to\_ub(y, i) \Rightarrow \#\{?\} © x = \#\{?\} © i \vee \#y + 1 \leq \#x. \qquad (**)$$

Both lemmas follow by stream induction[4].

---

[4] By stream induction we mean induction on the length of a stream (or the sum of the lengths

Given that $i, x \in (D \cup \{?\})^\omega, y \in (D \cup \{ok, fail\})^\omega, o \in D^\omega$, then the server is characterized by

$$[A_{\mathrm{SRV}}, C_{\mathrm{SRV}}],$$

where

$$A_{\mathrm{SRV}}(y, i) \stackrel{\mathrm{def}}{=} ok\_in(y, i),$$
$$C_{\mathrm{SRV}}(y, i, x, o) \stackrel{\mathrm{def}}{=} o = D\copyright y \wedge x = to\_ub(y, i).$$

The idea behind the assumption and the second conjunct of the commitment should be clear from the discussion above. The first conjunct of the commitment requires the server to output any data element received on y along o.

To prove that this decomposition is correct, it must be shown that

$$[A_{\mathrm{RB}}, C_{\mathrm{RB}}] \rightsquigarrow [A_{\mathrm{UB}}, C_{\mathrm{UB}}]_{H_{\mathrm{UB}}} \otimes [A_{\mathrm{SRV}}, C_{\mathrm{SRV}}].$$

Rules 9 and 10 imply

$$[A_{\mathrm{RB}}, C_{\mathrm{RB}}] \rightsquigarrow [A_{\mathrm{RB}}, C_{\mathrm{RB}}]_{H_{\mathrm{UB}}},$$
$$[A_{\mathrm{SRV}}, C_{\mathrm{SRV}}]_{H_{\mathrm{UB}}} \rightsquigarrow [A_{\mathrm{SRV}}, C_{\mathrm{SRV}}].$$

Moreover, if

$$C'_{\mathrm{UB}}(x, y, h) \stackrel{\mathrm{def}}{=} \#x = \#y \wedge \forall j \in \mathsf{dom}(x).\, D\copyright(y|_j) \sqsubseteq accept(x|_j, h) \wedge$$
$$\{ok, fail\}\copyright(y|_j) \sqsubseteq h \wedge \#\{ok, fail\}\copyright(y|_j) = \#D\copyright(x|_j) \wedge$$
$$\#D\copyright(y|_j) = \#\{?\}\copyright(x|_j),$$

then Rule 12 implies

$$[A_{\mathrm{UB}}, C'_{\mathrm{UB}}]_{H_{\mathrm{UB}}} \rightsquigarrow [A_{\mathrm{UB}}, C_{\mathrm{UB}}]_{H_{\mathrm{UB}}}.$$

Thus, it follows from Rules 2 and 3 that it is enough to prove

$$[A_{\mathrm{RB}}, C_{\mathrm{RB}}]_{H_{\mathrm{UB}}} \rightsquigarrow [A_{\mathrm{UB}}, C'_{\mathrm{UB}}]_{H_{\mathrm{UB}}} \otimes [A_{\mathrm{SRV}}, C_{\mathrm{SRV}}]_{H_{\mathrm{UB}}}. \qquad (\dagger)$$

In the same way as in Example 4, it is necessary to use a consequence rule, in this case Rule 11, to strengthen the component assumptions with two invariants:

$$I_{\mathrm{UB}}(x, h, i) \stackrel{\mathrm{def}}{=} \exists y' \in (D \cup \{ok, fail\})^\omega.\, x = to\_ub(y', i) \wedge$$
$$\{ok, fail\}\copyright y' \sqsubseteq h \wedge ok\_in(y', i),$$

$$I_{\mathrm{SRV}}(y, i, h) \stackrel{\mathrm{def}}{=} \{ok, fail\}\copyright y \sqsubseteq h.$$

It follows from Rules 2,3 and 11 that ($\dagger$) holds if it can be shown that

---

of several streams) where one in addition to the usual premises has to show that the formula is admissible with respect to the stream (or tuple of streams) on which the induction is conducted.

$$[A_{\mathrm{RB}}, C_{\mathrm{RB}}]_{H_{\mathrm{UB}}} \leadsto [\exists i.\, A_{\mathrm{UB}} \wedge I_{\mathrm{UB}}, C'_{\mathrm{UB}}]_{H_{\mathrm{UB}}} \otimes [A_{\mathrm{SRV}} \wedge I_{\mathrm{SRV}}, C_{\mathrm{SRV}}]_{H_{\mathrm{UB}}}. \qquad (\ddagger)$$

According to Rule 16 ($\ddagger$) holds if it can be shown that

$$H_{\mathrm{UB}} \wedge A_{\mathrm{RB}} \Rightarrow adm(\lambda x.A_{\mathrm{SRV}} \wedge I_{\mathrm{SRV}}) \vee adm(\lambda y.A_{\mathrm{UB}} \wedge I_{\mathrm{UB}}),$$
$$H_{\mathrm{UB}} \wedge A_{\mathrm{RB}} \Rightarrow (A_{\mathrm{SRV}} \wedge I_{\mathrm{SRV}})[{}^{x}_{\langle\rangle}] \vee (A_{\mathrm{UB}} \wedge I_{\mathrm{UB}})[{}^{y}_{\langle\rangle}],$$
$$H_{\mathrm{UB}} \wedge A_{\mathrm{RB}} \wedge A_{\mathrm{SRV}} \wedge I_{\mathrm{SRV}} \wedge C_{\mathrm{SRV}} \wedge A_{\mathrm{UB}} \wedge I_{\mathrm{UB}} \wedge C'_{\mathrm{UB}} \Rightarrow C_{\mathrm{RB}},$$
$$H_{\mathrm{UB}} \wedge A_{\mathrm{RB}} \wedge A_{\mathrm{SRV}} \wedge I_{\mathrm{SRV}} \wedge C_{\mathrm{SRV}} \Rightarrow A_{\mathrm{UB}} \wedge I_{\mathrm{UB}},$$
$$H_{\mathrm{UB}} \wedge A_{\mathrm{RB}} \wedge A_{\mathrm{UB}} \wedge I_{\mathrm{UB}} \wedge C'_{\mathrm{UB}} \Rightarrow A_{\mathrm{SRV}} \wedge I_{\mathrm{SRV}}.$$

It is easy to see that the first premise holds, since $\lambda y.A_{\mathrm{SRV}} \wedge I_{\mathrm{SRV}}$ is a safety predicate. That the second premise holds is obvious. That the antecedent of the third premise implies $o \sqsubseteq D\mathbb{C}i$ follows easily by the help of ($*$). That the same antecedent also implies $\#o = \#\{?\}\mathbb{C}i$ can be deduced by the help of ($**$). The correctness of premises four and five follows by stream induction. $\square$

7. Discussion

Techniques for writing explicit assumption/commitment specifications and composition principles for such specifications have already been proposed for a number of formalisms. What is new in this paper is that we have investigated the assumption/commitment paradigm in the context of nondeterministic Kahn-networks. Our results can be summed-up as follows:

- We have defined two types of assumption/commitment specifications, namely simple and general specifications.
- It has been shown that semantically, any deterministic agent can be uniquely characterized by a simple specification, and any nondeterministic agent can be uniquely characterized by a general specification.
- We have defined two sets of refinement rules, one for simple specifications and one for general specifications. The rules are Hoare-logic inspired. In particular the feedback rules employ an invariant in the style of a traditional while-rule.
- Both sets of rules have been proved to be sound and also semantically complete with respect to a chosen set of composition operators.
- We have defined conversion rules which allow the two logics to be combined. This means that general specifications and the rules for general specifications have to be introduced only at the point in a system development where they are really needed.

In addition, in a number of examples, we have illustrated how specifications can be written in this formalism and how decompositions can be proved correct using our rules.

We will now try to relate our results to assumption/commitment formalisms defined for other semantic models.

A number of approaches, like [MC81], [Jon83] and [Stø91], deal only with safety predicates and restricted types of liveness and are therefore less general than the logic described in this paper.

[Pnu85] presents an assumption/commitment formalism for a shared-state parallel language. A rule for a shared-state parallel operator is given. In fact

this seems to be the first paper which tries to handle general safety and liveness predicates in a compositional style. Roughly speaking, this parallel operator corresponds to our construct for mutual feedback as depicted in Figure 2. The rule differs from our Rule 8 (and 16) in that the induction is explicit, i.e. the user must himself find an appropriate well-ordering. A related rule is formulated in [Pan90]. There is a translation of these rules into our formalism, where the state is interpreted as the tuple of input/output streams, but the rules we then get are quite weak, i.e. incomplete, in the sense that we can only prove properties which hold for all fixpoints and not properties which hold for the least fixpoint only. [Sta85] also proposes a rule which seems to be a special case of Pnueli's rule.

More recently, [AL90] has proposed a general composition principle with respect to a shared-state model. This principle is similar to Rule 8 in that the induction is only implicit, but differs from Rule 8 in that the assumptions are required to be safety properties. It is shown in their paper that any "sensible" specification can be written in a normal form where the assumption is a safety property. A similar result holds for our specifications. However, at least with respect to our specification formalism, it is often an advantage to be able to state liveness constraints also in the assumptions. [AL93] proposes a slightly stronger rule which handles some liveness properties in the assumptions. However, this strengthening seems to be of little practical importance.

Our rules for the feedback operators can deal with at least some interesting liveness properties in the assumptions. This is clear because we only require the assumptions to be admissible with respect to the feedback channels. For example the assumption of the specification ADD on Page 12 is a liveness property. However, also our rules are not as strong as we would have liked. For example when using our specification formalism it may be helpful to state that the lengths of the input streams are related in a certain way. When using Rule 7 this can lead to difficulties (see its second premise where the empty stream is inserted for the feedback input). One way to handle this problem is to simulate the stepwise consumption of the overall input. The following rule is based on this idea:

Rule 7$'$:
$$adm(A_1)$$
$$A \Rightarrow A_1[{}^{x}_{\langle\rangle}\ {}^{i}_{t(i)_1}]$$
$$A \wedge A_1[{}^{x}_{y}] \wedge C_1[{}^{x}_{y}] \Rightarrow C$$
$$\frac{A \wedge A_1[{}^{i}_{t(i)_j}] \wedge C_1[{}^{i}_{t(i)_j}] \Rightarrow A_1[{}^{x}_{y}\ {}^{i}_{t(i)_{j+1}}]}{[A, C] \rightsquigarrow \mu\,[A_1, C_1]}$$

Here t is a function which takes a stream tuple as argument and returns a chain such that for all i, $\sqcup t(i) = i$. The idea is that t partitions i in accordance with how the input is consumed. Thus, the first element of $t(i)$ represents the consumption of input w.r.t. the first element of the Kleene chain, i.e. the empty stream; the second element of $t(i)$ represents the consumption of input w.r.t. the second element of the Kleene chain; etc. The rule can be made even stronger by characterizing the input consumption as a function of the tuple of input streams. Rules 8, 15 and 16 can be reformulated in a similar style. The rules are complete in the same sense as earlier.

The P-A logic of [PJ91] gives rules for both asynchronous and synchronous communication with respect to a CSP-like language. Also in this approach the

assumptions are safety predicates. Moreover, general liveness predicates can only be derived indirectly from the commitment via a number of additional rules.

We are using sets of monotonic and continuous functions to model agents. There are certain time dependent components like non-strict fair merge which cannot be modeled in this type of semantics [Kel78], i.e. they are not agents as agents are defined here. In [BS94] this problem is dealt with by using a more sophisticated semantics based on timed streams [Par83]. The rules proposed in [BS94] can be seen as a generalization of the rules introduced above.

Some case-studies have been carried out. In particular a non-trivial production cell has been successfully specified and decomposed using the proposed formalism [Phi93], [FP95].

8. Acknowledgement

References

[AL88]   M. Abadi and L. Lamport. The existence of refinement mappings. Technical Report 29, Digital, Palo Alto, 1988.

[AL90]   M. Abadi and L. Lamport. Composing specifications. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proc. REX Workshop on Stepwise Refinement of Distributed Systems, Lecture Notes in Computer Science 430*, pages 1–41, 1990.

[AL93]   M. Abadi and L. Lamport. Conjoining specifications. Technical Report 118, Digital, Palo Alto, 1993.

[AS85]   B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[BA81]   J. D. Brock and W. B. Ackermann. Scenarios: A model of non-determinate computation. In J. Diaz and I. Ramos, editors, *Proc. Formalization of Programming Concepts, Lecture Notes in Computer Science 107*, pages 252–259, 1981.

[BDD$^+$93]   M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The design of distributed systems — an introduction to Focus (revised version). Technical Report SFB 342/2/92 A, Technische Universität München, 1993.

[BKP84]   H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proc. Sixteenth ACM Symposium on Theory of Computing*, pages 51–63, 1984.

[Bro89]   M. Broy. Towards a design methodology for distributed systems. In M. Broy, editor, *Proc. Constructive Methods in Computing Science, Summerschool, Marktoberdorf*, pages 311–364. Springer, 1989.

[Bro92]   M. Broy. Functional specification of time sensitive communicating systems. In M. Broy, editor, *Proc. Programming and Mathematical Method, Summerschool, Marktoberdorf*, pages 325–367. Springer, 1992.

[Bro94]   M. Broy. A functional rephrasing of the assumption/commitment specification style. Technical Report SFB 342/10/94 A, Technische Universität München, 1994.

[BS94]   M. Broy and K. Stølen. Specification and refinement of finite dataflow networks — a relational approach. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Proc. FTRTFT'94, Lecture Notes in Computer Science 863*, pages 247–267, 1994.

[CM88]   K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.

[Ded92]   F. Dederichs. *Transformation verteilter Systeme: Von applikativen zu prozedu-ralen Darstellungen*. PhD thesis, Technische Universität München, 1992. Also available as SFB-report 342/17/92 A, Technische Universität München.

[dR85]   W. P. de Roever. The quest for compositionality, formal models in programming. In F. J. Neuhold and G. Chroust, editors, *Proc. IFIP 85*, pages 181–205, 1985.

[FP95]   M. Fuchs and J. Philipps. Formal development of a production cell in Focus – a case study. In C. Lewerenz and T. Lindner, editors, *Formal Development of Reactive Systems: Case Study Production Cell, Lecture Notes in Computer Science 891*, pages 187–200. 1995.

[Hoa69]   C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[Jon83]   C. B. Jones. Specification and design of (parallel) programs. In R.E.A. Mason, editor, *Proc. Information Processing 83*, pages 321–331. North-Holland, 1983.

[Jon90]   C. B. Jones. *Systematic Software Development Using VDM, Second Edition*. Prentice-Hall, 1990.

[Kah74]   G. Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Proc. Information Processing 74*, pages 471–475. North-Holland, 1974.

[Kel78]   R. M. Keller. Denotational models for parallel programs with indeterminate operators. In E. J. Neuhold, editor, *Proc. Formal Description of Programming Concepts*, pages 337–366. North-Holland, 1978.

[Kle52]   S. C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.

[KM77]   G. Kahn and D. B. MacQueen. Corutines and networks of parallel processes. In B. Gilchrist, editor, *Proc. Information Processing 77*, pages 993–998. North-Holland, 1977.

[Man74]   Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[MC81]   J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7:417–426, 1981.

[Mor88]   C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10:403–419, 1988.

[OG76]   S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[Pan90]   P. K. Pandya. Some comments on the assumption-commitment framework for compositional verification of distributed programs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proc. REX Workshop on Stepwise Refinement of Distributed Systems, Lecture Notes in Computer Science 430*, pages 622–640, 1990.

[Par83]   D. Park. The "fairness" problem and nondeterministic computing networks. In J. W. de Bakker and J van Leeuwen, editors, *Proc. 4th Foundations of Computer Science, Mathematical Centre Tracts 159*, pages 133–161. Mathematisch Centrum Amsterdam, 1983.

[Phi93]   J. Philipps. Spezifikation einer Fertigungszelle — eine Fallstudie in Focus. Master's thesis, Technische Universiät München, 1993.

[PJ91]   P. K. Pandya and M. Joseph. P-A logic — a compositional proof system for distributed programs. *Distributed Computing*, 5:37–54, 1991.

[Pnu85]   A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Proc. Logics and Models of Concurrent Systems*, pages 123–144. Springer, 1985.

[SDW93]   K. Stølen, F. Dederichs, and R. Weber. Assumption/commitment rules for networks of asynchronously communicating agents. Technical Report SFB 342/2/93 A, Technische Universität München, 1993.

[Sta85]   E. W. Stark. A proof technique for rely/guarantee properties. In S. N. Maheshwari, editor, *Proc. 5th Conference on the Foundation of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science 206*, pages 369–391, 1985.

[Stø91]   K. Stølen. A method for the development of totally correct shared-state parallel programs. In J. C. M. Baeten and J. F. Groote, editors, *Proc. CONCUR'91, Lecture Notes in Computer Science 527*, pages 510–525, 1991.

[Zwi89]   J. Zwiers. *Compositionality, Concurrency and Partial Correctness: Proof Theories for Networks of Processes and Their Relationship*, volume 321 of *Lecture Notes in Computer Science*. 1989.

A. Proofs

The object of this appendix is to give proofs for claims made elsewhere in the paper.

A.1. Proof of Theorem 1

The soundness proofs for Rules 1 - 6 are trivial. The soundness of Rules 7 and 8 follow from Lemmas 1 and 2.

Lemma 1. If

$$A(i) \Rightarrow adm(\lambda x \in Y^\omega. A_1(i,x)), \tag{1}$$
$$A(i) \Rightarrow A_1(i,\langle\rangle), \tag{2}$$
$$A(i) \wedge A_1(i,y) \wedge C_1(i,y,o,y) \Rightarrow C(i,o,y), \tag{3}$$
$$A(i) \wedge A_1(i,x) \wedge C_1(i,x,o,y) \Rightarrow A_1(i,y), \tag{4}$$

then

$$[A,C] \rightsquigarrow \mu\,[A_1,C_1]. \tag{5}$$

Proof. Assume that 1 - 4 hold, and that f, i, o and y are such that

$$f \in [\![\, [A_1,C_1] \,]\!], \tag{6}$$
$$A(i) \wedge \mu\,f(i) = (o,y). \tag{7}$$

The monotonicity of f implies that there are chains $\hat{o}$, $\hat{y}$ such that

$$(\hat{o}_1, \hat{y}_1) \stackrel{\mathsf{def}}{=} (\langle\rangle, \langle\rangle), \tag{8}$$
$$(\hat{o}_j, \hat{y}_j) \stackrel{\mathsf{def}}{=} f(i, \hat{y}_{j-1}) \qquad \text{if } j > 1. \tag{9}$$

Kleene's theorem [Kle52] implies

$$\sqcup(\hat{o}, \hat{y}) = (o, y). \tag{10}$$

Assume for an arbitrary $j \geq 1$

$$A_1(i, \hat{y}_j). \tag{11}$$

6, 8, 9 and 11 imply

$$C_1(i, \hat{y}_j, \hat{o}_{j+1}, \hat{y}_{j+1}). \tag{12}$$

4, 7, 11 and 12 imply

$$A_1(i, \hat{y}_{j+1}).$$

Thus, for all $j \geq 1$

$$A_1(i, \hat{y}_j) \Rightarrow A_1(i, \hat{y}_{j+1}). \tag{13}$$

2, 7, 13 and induction on j imply for all $j \geq 1$

$$A_1(i, \hat{y}_j). \tag{14}$$

1, 7, 10 and 14 imply

$$A_1(i, y). \tag{15}$$

6, 7 and 15 imply

$$C_1(i, y, o, y). \tag{16}$$

3, 7, 15 and 16 imply

$$C(i, o, y).$$

Thus, it has been shown that

$$A(i) \wedge \mu f(i) = (o, y) \Rightarrow C(i, o, y). \tag{17}$$

17 and the way f, i, o and y were chosen imply 5. $\square$

Lemma 2. If

$$A(i, r) \Rightarrow adm(\lambda x \in X^\omega.\, A_1(i, x, r)) \vee adm(\lambda y \in Y^\omega.\, A_2(y, r, i)), \tag{18}$$

$$A(i, r) \Rightarrow A_1(i, \langle\rangle, r) \vee A_2(\langle\rangle, r, i), \tag{19}$$

$$A(i, r) \wedge A_1(i, x, r) \wedge A_2(y, r, i) \wedge C_1(i, x, o, y) \wedge C_2(y, r, x, s) \Rightarrow$$
$$\qquad C(i, r, o, y, x, s), \tag{20}$$

$$A(i, r) \wedge A_1(i, x, r) \wedge C_1(i, x, o, y) \Rightarrow A_2(y, r, i), \tag{21}$$

$$A(i, r) \wedge A_2(y, r, i) \wedge C_2(y, r, x, s) \Rightarrow A_1(i, x, r), \tag{22}$$

then

$$[A, C] \rightsquigarrow [\exists r \in R^\omega.\, A_1, C_1] \otimes [\exists i \in I^\omega.\, A_2, C_2]. \tag{23}$$

Proof. Assume that 18 - 22 hold, and that f_1, f_2, i, r, o, y, x and s are such that

$$f_1 \in [\![\, [\exists r \in R^\omega.\, A_1, C_1]\,]\!], \tag{24}$$

$$f_2 \in [\![\, [\exists i \in I^\omega.\, A_2, C_2]\,]\!], \tag{25}$$

$$A(i, r) \wedge f_1 \otimes f_2(i, r) = (o, y, x, s). \tag{26}$$

The monotonicity of f_1 and f_2 implies that there are chains $\hat{o}, \hat{y}, \hat{x}$ and $\hat{s}$ such that

$$(\hat{o}_1, \hat{y}_1, \hat{x}_1, \hat{s}_1) \stackrel{\text{def}}{=} (\langle\rangle, \langle\rangle, \langle\rangle, \langle\rangle), \tag{27}$$

$$(\hat{o}_j, \hat{y}_j, \hat{x}_j, \hat{s}_j) \stackrel{\text{def}}{=} f_1(i, \hat{x}_{j-1}) \parallel f_2(\hat{y}_{j-1}, r) \qquad \text{if } j > 1. \tag{28}$$

Kleene's theorem implies

$$\sqcup(\hat{o}, \hat{y}, \hat{x}, \hat{s}) = (o, y, x, s). \tag{29}$$

Assume for an arbitrary $j \geq 1$

$$A_1(i, \hat{x}_j, r). \tag{30}$$

24, 27, 28 and 30 imply

$$C_1(i, \hat{x}_j, \hat{o}_{j+1}, \hat{y}_{j+1}). \tag{31}$$

21, 26, 30 and 31 imply

$$A_2(\hat{y}_{j+1}, r, i).$$

Thus, for all $j \geq 1$

$$A_1(i, \hat{x}_j, r) \Rightarrow A_2(\hat{y}_{j+1}, r, i). \tag{32}$$

By a similar argument it follows that for all $j \geq 1$

$$A_2(\hat{y}_j, r, i) \Rightarrow A_1(i, \hat{x}_{j+1}, r). \tag{33}$$

19, 26, 32, 33 and induction on j imply that for all $j \geq 1$

$$\exists k.\, k > j \wedge A_1(i, \hat{x}_k, r) \wedge \exists k.\, k > j \wedge A_2(\hat{y}_k, r, i). \tag{34}$$

Assume

$$adm(\lambda x \in X^\omega.\, A_1(i, x, r)). \tag{35}$$

29, 34 and 35 imply

$$A_1(i, x, r). \tag{36}$$

24, 26 and 36 imply

$$C_1(i, x, o, y). \tag{37}$$

21, 26, 36 and 37 imply

$$A_2(y, r, i). \tag{38}$$

25, 26 and 38 imply

$$C_2(y, r, x, s). \tag{39}$$

20, 26, 36, 37, 38 and 39 imply

$$C(i, r, o, y, x, s). \tag{40}$$

Assume

$$adm(\lambda y \in Y^\omega.\, A_2(y, r, i)). \tag{41}$$

By an argument similar to the one above, 40 may again be deduced.
Since 18 and 26 imply that either 35 or 41 hold, it has been shown that

$$A(i, r) \wedge f_1 \otimes f_2(i, r) = (o, y, x, s) \Rightarrow C(i, r, o, y, x, s). \tag{42}$$

42 and the way f_1, f_2, i, r, o, y, x and s were chosen imply 23. $\square$

A.2. Proof of Theorem 2

Follows straightforwardly from Lemmas 3 - 6.

Lemma 3. If

$$f_1 \circ f_2 \in [\![\, [A, C] \,]\!], \tag{43}$$

then there are A_1, A_2, C_1 and C_2 such that

$$f_1 \in [\![\, [A_1, C_1] \,]\!], \tag{44}$$
$$f_2 \in [\![\, [A_2, C_2] \,]\!], \tag{45}$$
$$A(i) \Rightarrow A_1(i), \tag{46}$$
$$A(i) \wedge C_1(i, x) \Rightarrow A_2(x), \tag{47}$$
$$A(i) \wedge C_1(i, x) \wedge C_2(x, o) \Rightarrow C(i, o). \tag{48}$$

Proof. Assume 43. Let

$$A_1(i) \stackrel{\text{def}}{=} \text{true},$$

$$A_2(x) \stackrel{\text{def}}{=} \text{true},$$

$$C_1(i, x) \stackrel{\text{def}}{=} f_1(i) = x,$$

$$C_2(x, o) \stackrel{\text{def}}{=} f_2(x) = o.$$

It follows trivially that 44-48 hold. $\square$

Lemma 4. If

$$f_1 \parallel f_2 \in [\![\, [A, C] \,]\!], \tag{49}$$

then there are A_1, A_2, C_1 and C_2 such that

$$f_1 \in [\![\, [A_1, C_1] \,]\!], \tag{50}$$
$$f_2 \in [\![\, [A_2, C_2] \,]\!], \tag{51}$$
$$A(i, r) \Rightarrow A_1(i) \wedge A_2(r), \tag{52}$$
$$A(i, r) \wedge C_1(i, o) \wedge C_2(r, s) \Rightarrow C(i, r, o, s). \tag{53}$$

Proof. Assume 49. Let

$$A_1(i) \stackrel{\text{def}}{=} \text{true},$$

$$A_2(r) \stackrel{\text{def}}{=} \text{true},$$

$$C_1(i, o) \stackrel{\text{def}}{=} f_1(i) = o,$$

$$C_2(r, s) \stackrel{\text{def}}{=} f_2(r) = s.$$

It follows trivially that 50-53 hold. $\square$

Lemma 5. If

$$\mu f \in [\![\, [A, C] \,]\!], \tag{54}$$

then there are A_1 and C_1 such that

$$f \in [\![\, [A_1, C_1] \,]\!], \tag{55}$$
$$A(i) \Rightarrow adm(\lambda x \in Y^\omega. \, A_1(i, x)), \tag{56}$$
$$A(i) \Rightarrow A_1(i, \langle \rangle), \tag{57}$$
$$A(i) \wedge A_1(i, y) \wedge C_1(i, y, o, y) \Rightarrow C(i, o, y), \tag{58}$$
$$A(i) \wedge A_1(i, x) \wedge C_1(i, x, o, y) \Rightarrow A_1(i, y). \tag{59}$$

Proof. Assume 54. Let

$$A_1(i, x) \stackrel{\text{def}}{=} (\exists j \in \mathsf{N}. \, K_j(i, x)) \vee (\exists \hat{y} \in Ch(Y^\omega).$$
$$\quad x = \sqcup \hat{y} \wedge \forall j \in \mathsf{N}. \, K_j(i, \hat{y}_j)),$$

$$K_1(i, x) \stackrel{\text{def}}{=} x = \langle \rangle,$$

$$K_j(i, x) \stackrel{\text{def}}{=} \exists x' \in Y^\omega. \, \exists o \in O^\omega. \, K_{j-1}(i, x') \wedge f(i, x') = (o, x) \qquad if \ j > 1,$$

$$C_1(i, x, o, y) \stackrel{\text{def}}{=} f(i, x) = (o, y).$$

Basically, $K_j(i, x)$ characterizes the j'th element x of the Kleene-chain for the function f and the given input i. This means that (i, x) satisfies A_1 iff x is an element of the Kleene-chain or its least upper bound for the input i. 55 holds trivially. 56 follows from the second disjunct of A_1's definition, while 57 is a direct consequence of the definition of K_1. To prove 58, observe that the antecedent of 58 is equivalent to

$$A(i) \wedge A_1(i, y) \wedge f(i, y) = (o, y). \tag{60}$$

Since A_1 characterizes the Kleene-chain or its least upper bound for a given input i, 60 implies

$$A(i) \wedge A_1(i, y) \wedge \mu f(i) = (o, y). \tag{61}$$

54 implies

$$A(i) \wedge \mu f(i) = (o, y) \Rightarrow C(i, o, y).$$

Thus 58 holds. To prove 59, let i, x, o and y be such that

$$A(i) \wedge A_1(i, x) \wedge C_1(i, x, o, y). \tag{62}$$

62 implies

$$A(i) \wedge A_1(i, x) \wedge f(i, x) = (o, y). \tag{63}$$

It follows from the definition of A_1 that there are two cases to consider. If x is the least upper bound of the Kleene-chain for the input i, it follows that $x = y$, in which case 63 implies

$$A_1(i, y).$$

On the other hand, if x is an element of the Kleene-chain for the input i, then there is a $j \geq 1$ such that

$$K_j(i, x). \tag{64}$$

63 and 64 imply

$$K_{j+1}(i, y). \tag{65}$$

65 implies

$$A_1(i, y). \tag{66}$$

This proves 59. $\quad\square$

Lemma 6. If

$$f_1 \otimes f_2 \in [\![[A, C]]\!] \tag{67}$$

then there are A_1, A_2, C_1 and C_2 such that

$$f_1 \in [\![[\exists r \in R^\omega. A_1, C_1]]\!], \tag{68}$$

$$f_2 \in [\![[\exists i \in I^\omega. A_2, C_2]]\!], \tag{69}$$

$$A(i, r) \Rightarrow adm(\lambda x \in X^\omega. A_1(i, x, r)) \vee adm(\lambda y \in Y^\omega. A_2(y, r, i)), \tag{70}$$

$$A(i, r) \Rightarrow A_1(i, \langle\rangle, r) \vee A_2(\langle\rangle, r, i), \tag{71}$$

$$A(i,r) \wedge A_1(i,x,r) \wedge A_2(y,r,i) \wedge C_1(i,x,o,y) \wedge C_2(y,r,x,s) \Rightarrow$$
$$C(i,r,o,y,x,s), \tag{72}$$
$$A(i,r) \wedge A_1(i,x,r) \wedge C_1(i,x,o,y) \Rightarrow A_2(y,r,i), \tag{73}$$
$$A(i,r) \wedge A_2(y,r,i) \wedge C_2(y,r,x,s) \Rightarrow A_1(i,x,r). \tag{74}$$

Proof. Let

$$A_1(i,x,r) \stackrel{\text{def}}{=} (\exists j \in \mathsf{N}.\, \exists y \in Y^\omega.\, K_j(i,r,x,y)) \vee$$
$$(\exists \hat{x} \in Ch(X^\omega).\, \exists \hat{y} \in Ch(Y^\omega).\, x = \sqcup\hat{x} \wedge \forall j \in \mathsf{N}.\, K_j(i,r,\hat{x}_j,\hat{y}_j)),$$

$$A_2(y,r,i) \stackrel{\text{def}}{=} (\exists j \in \mathsf{N}.\, \exists x \in X^\omega.\, K_j(i,r,x,y)) \vee$$
$$(\exists \hat{x} \in Ch(X^\omega).\, \exists \hat{y} \in Ch(Y^\omega).\, y = \sqcup\hat{y} \wedge \forall j \in \mathsf{N}.\, K_j(i,r,\hat{x}_j,\hat{y}_j)),$$

$$K_1(i,r,x,y) \stackrel{\text{def}}{=} x = \langle\rangle \wedge y = \langle\rangle,$$

$$K_j(i,r,x,y) \stackrel{\text{def}}{=} \exists x' \in X^\omega.\, \exists y' \in Y^\omega.\, \exists o \in O^\omega.\, \exists s \in S^\omega.$$
$$K_{j-1}(i,r,x',y') \wedge f_1(i,x') = (o,y) \wedge f_2(y',r) = (x,s),$$

$$C_1(i,x,o,y) \stackrel{\text{def}}{=} f_1(i,x) = (o,y),$$

$$C_2(y,r,x,s) \stackrel{\text{def}}{=} f_2(y,r) = (x,s).$$

68-74 can now be deduced from 67 by an argument similar to that of Lemma 5. □

A.3. Proof of Theorem 4

The soundness of Rules 9 and 10 follows trivially. The soundness of Rules 11-16 follows easily from Lemma 3 and the soundness of the corresponding rules for simple specifications.

A.4. Proof of Theorem 5

Since Rule 11 allows us to extend the set of hypotheses, we may assume that there is an injective mapping m from $[\![\, F \,]\!]$ to the set of hypotheseses characterized by H such that for all $f \in [\![\, F \,]\!]$

$$H(m(f)) \wedge f \in [\![\, [A_{m(f)}, C_{m(f)}] \,]\!].$$

Under this assumption Lemmas 3-6 can be used to construct sets of simple specifications, i.e. general specifications, in the same way as they were used to construct simple specifications in the proof of Theorem 2.