Assumption/Commitment Rules for Networks of Asynchronously Communicating Agents[1]

Ketil Stølen, Frank Dederichs, Rainer Weber

Institut für Informatik
Technische Universität München
Postfach 20 24 20, 8000 München 2

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

Abstract

This report presents an assumption/commitment specification technique and a refinement calculus for networks of agents communicating asynchronously via unbounded FIFO channels in the tradition of [Kah74], [Kel78], [BDD+92]:

- We define two different types of (explicit) 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 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.
- 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.

The proposed specification formalism and refinement rules together with a number of related design principles presented in [Bro92d], [Bro92a] constitute a powerful design method which allows distributed systems to be developed in the same style as methods like [Jon90], [Mor90] allow for the design of sequential systems.

Contents

Chapter 1

Introduction

Ever since formal program development 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] and [BDD+92].

The motivation for insisting on an explicit assumption/commitment form varies from approach to approach. For example, 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.

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 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 object of this report is to present a set of decomposition rules for explicit assumption/commitment specifications with respect to networks of agents communicating asynchronously via unbounded FIFO channels in the style of [Kah74], [KM77], [Kel78]. Agents are modeled as sets of stream processing functions as explained in [Kel78], [Bro89], [BDD+92].

We distinguish between two types of such 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.

Our approach is compositional. This means that:

- Design decisions can be verified at the point where they are taken. This reduces the amount of backtracking needed during the design process.
- Specifications can be split into subspecifications which can be implemented separately.

We are interested in rules which can be used to reason about both safety and liveness properties. The report concentrates on the theoretical aspects. The authors plan to investigate the usefulness of the proposed rules in a number of case-studies.

The rest of the report is divided into four chapters and one appendix. The basic notation and the semantic model are introduced in Chapter 2. Chapter 3 presents decomposition rules for simple specifications, while the decomposition of general specifications is the topic of Chapter 4. Chapter 5 relates our approach to other proposals known from the literature. Proofs of soundness and semantic completeness can be found in the appendix.

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

Chapter 2

Basic Concepts and Notation

In this chapter we briefly explain the basic concepts of our approach and introduce some notation.

2.1 Streams

N denotes the set of natural numbers with 0 removed. B denotes 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 $(D_1^\omega \times D_2^\omega \times \ldots \times D_n^\omega)$. Observe that T^*, T^∞ and T^ω are sets of stream tuples and not sets of streams of data tuples.

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

- $s \circ t$ denotes the result of concatenating s and t, i.e. the j'th component $(s \circ t)_j$ is equal to the result of prefixing t_j with s_j 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. $\exists p \in T^\omega . s \circ p = t$;
- $A©r$ denotes the projection of r on A, data not occurring in A are deleted, e.g. $\{0,1\}$ © $\langle 0,1,2,0,4\rangle = \langle 0,1,0\rangle$;
- $\#r$ denotes the number of elements in r if $r \in D^*$, and ∞ otherwise;
- r_j denotes the j'th element of r if $j \leq \#r$;
- $\mathsf{dom}(r)$ denotes the set of indices corresponding to r, i.e. $\mathsf{dom}(r) = \{j \mid j \leq \#r\}$.

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. $Ch(T^\omega)$ denotes the set of chains over T^ω.

2.2 Predicates

A *predicate* is a boolean-valued function

$$P \in T^\omega \to \mathsf{B}.$$

When convenient the argument tuple of a predicate will be split into several argument tuples. For example a predicate of the form $P(s,t)$ will often be used to express the relation between stream tuples s and t.

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.

P is a *safety* predicate iff

$$\neg P(s) \Leftrightarrow \exists t \in T^*.\, t \sqsubseteq s \wedge \neg P(t).$$

This means, if some stream tuple s violates P, then there is a *finite* prefix of s that violates P. Thus the violation of safety predicates can always be detected by finite observations.

P is *admissible* iff for all chains $\hat{c}$ in T^ω

$$(\forall j \in \mathsf{N}.\, P(\hat{c}_j)) \Rightarrow P(\sqcup\hat{c}).$$

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 no safety predicate. $adm(P)$ holds iff P is admissible.

P is a *liveness* predicate iff

$$\forall s \in T^*.\, \exists t \in T^\omega.\, P(s \circ t).$$

This means, any finite stream tuple s can be extended by a stream tuple t such that P is fulfilled. Therefore, complete observations are necessary to detect the violation of liveness predicates.

2.3 Stream Processing Functions

A function

$$f \in I^\omega \to O^\omega$$

is called a *stream processing function* iff it is *prefix continuous*, i.e. iff for all chains $\hat{c}$ in I^ω:

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

Since continuity implies monotonicity, any stream processing function f is also *prefix monotonic*:

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

As pointed out by [Kah74], a stream processing function is an adequate means for describing (deterministic) agents that communicate asynchronously via unbounded FIFO-channels. The continuity constraint reflects the computational behavior of an agent: it consumes its input and produces its output in a step-wise manner. For partial output only partial input is necessary. This ensures that communicating agents can work in parallel. The set of all stream processing functions in $I^\omega \to O^\omega$ is denoted by

$$I^\omega \xrightarrow{c} O^\omega.$$

In the same way as for predicates, input and output tuples will be split into several tuples when convenient.

2.4 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. An agent may have no input channels but has always at least one output channel. The reason for the latter is of course that an agent without output channels is completely useless.

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 may 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.

Agents can be composed by three basic operators. These are introduced below.

2.4.1 Sequential Composition

Given two agents

$$F_1 : I^\omega \to X^\omega \quad \text{and} \quad F_2 : X^\omega \to O^\omega,$$

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

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

where

$$f_1 \circ f_2(i) \stackrel{\text{def}}{=} f_2(f_1(i)).$$

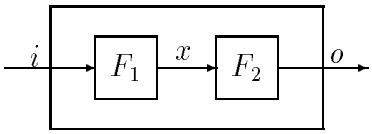

Figure 2.1: Sequential Composition.

Figure 2.1 shows the situation. Each arrow stands for a finite number of channels. Sequential composition is based on function composition. 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: F_1 reads some data from its input channels and produces some data on its output channels; while F_2 reads these data, F_1 can continue to work, e.g. read new inputs.

2.4.2 Parallel Composition

Given two agents

$$F_1 : I^\omega \to O^\omega \quad \text{and} \quad F_2 : R^\omega \to S^\omega,$$

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

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

where

$$f_1 \parallel f_2(i,r) \stackrel{\mathsf{def}}{=} (f_1(i), f_2(r)).$$

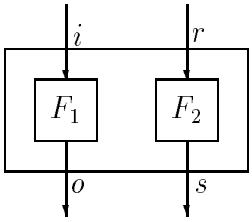

Figure 2.2: Parallel Composition.

Parallel composition is shown in Figure 2.2. F_1 and F_2 are simply put side by side and work independently without any mutual communication.

2.4.3 Feedback

Let

$$F : I^\omega \times Y^\omega \times R^\omega \to O^\omega \times Y^\omega \times S^\omega$$

be an agent, where I denotes a $(p-1)$-ary tuple of data sets, O a $(q-1)$-ary tuple of data sets and Y, for the time being, a data set. Then the p-th input channel of F has the same type as the q-th output channel, and they can be connected as depicted in Figure 2.3. This is called *feedback*. The resulting construct $\mu_q^p F$ is of type $I^\omega \times R^\omega \to O^\omega \times Y^\omega \times S^\omega$, and its denotation is

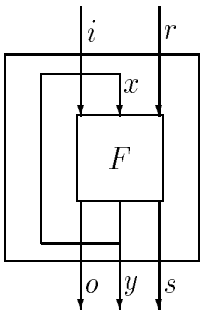

Figure 2.3: Feedback.

$$[\![\, \mu_q^p, F \,]\!] \stackrel{\mathsf{def}}{=} \{\mu_q^p f \mid f \in [\![\, F \,]\!]\},$$

where

$$\mu_q^p f(i,r) \stackrel{\mathsf{def}}{=} (o,y,s)$$

iff

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

In other words, (o,y,s) is the smallest stream tuple that satisfies the recursive equation $f(o,x',r) = (o',y',s')$. Obviously, for different inputs i,r different solutions will be obtained. (o,y,s) is called the *least fixed point* of f with respect to i,r. The continuity of stream processing functions ensures that there is a least fixed point.

It is easy to generalize the μ-operator to enable the feedback of more than one channel. For instance, if in the feedback definition above Y is not a single data set but an r-ary tuple of data sets we can write

$$\mu(r)_q^p F$$

to denote that r streams are fed back. Whenever it is clear from the context, which output channels are fed back to which input channels, we will just write μ without any decoration.

Kleene's theorem [Kle52] provides another characterization of least fixed points. It will be used in our soundness proofs.

Proposition 1 *Let $f \in I^\omega \times Y^\omega \times R^\omega \stackrel{c}{\to} O^\omega \times Y^\omega \times S^\omega$ be a stream processing function. Then for every $i \in I^\omega$, $r \in R^\omega$, f possesses a least fixed point (o,y,s) for which it holds*

$$(o,y,s) = \sqcup\{(\hat{o}_j, \hat{y}_j, \hat{s}_j) \,|\, j \in \mathsf{N}\},$$

where $(\hat{o}_1, \hat{y}_1, \hat{s}_1) = (\langle\rangle, \langle\rangle, \langle\rangle)$ and for all $j > 1$, $(\hat{o}_j, \hat{y}_j, \hat{s}_j) = f(i, \hat{y}_{j-1}, r)$. This chain is called the Kleene-chain.

The composition operators can be used to construct networks of agents — networks which themselves are agents.

Proposition 2 *The denotation of any network generated from some given basic agents using the operators $\circ$, $\|$ and μ is a set of stream processing functions. 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.

2.5 Basic Agents

In this report we distinguish between agents which are *syntactic* entities and their *semantic* representation as sets of stream processing functions. Networks of agents can be built using the operators for sequential and parallel composition plus feedback. These three operators 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.

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.

Programming notation for coding basic agents of the implementable sort can be defined in many ways. For example, [Ded92] proposes both a functional and a procedural language for this purpose. In this paper, we just *assume* that we have some sort of notation for describing basic agents. When we later prove semantic completeness of our logics, we assume that we can carry out certain deductions for basic agents. If we choose a particular representation, then this has to be shown.

Chapter 3

Decomposing Simple Specifications

As already pointed out, this report is concerned with explicit assumption/commitment specifications only. We distinguish between two different types of such specifications, namely simple and general specifications. This chapter deals with the former type, which can be used to specify any deterministic agent. It is first explained what a simple specification is. Then we introduce a set of refinement rules which allows simple specifications to be refined in a stepwise, top-down manner. The rules are semantically complete (in a weak sense) with respect to deterministic agents.

3.1 Simple Specifications

In our approach an agent communicates with its environment via unbounded FIFO channels. It receives input through a finite number of input channels and sends output through a finite number of output channels. An agent does not necessarily have any input channels, but has at least one output channel. Clearly, the only way an agent can be influenced by its environment is through its input channels, and the only way an agent can influence its environment is through its output channels, i.e. an agent is related to its environment as shown in Figure 3.1.

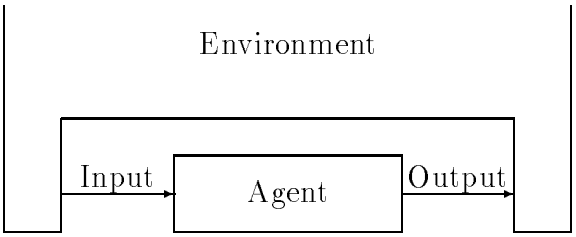

Figure 3.1: Assumption/Commitment Paradigm

Given this framework, 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:

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

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 BUF, where

$$A_{\mathrm{BUF}}(i) \stackrel{\mathsf{def}}{=} \forall i' \in (D \cup \{?\})^*.\ i' \sqsubseteq i \Rightarrow \#\{?\}ⓒi' \leq \#Dⓒi' \leq \#\{?\}ⓒi' + 1,$$

$$C_{\mathrm{BUF}}(i, o) \stackrel{\mathsf{def}}{=} o \sqsubseteq Dⓒi \wedge \#o = \#\{?\}ⓒi.$$

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).

It follows from the continuity constraint imposed on stream processing functions that the buffer will produce output, which satisfies the first conjunct of the commitment, as long as the input satisfies the assumption. Thus the above specification also constrains the buffer's behavior for inputs which falsify the assumption. □

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 $[\![\ [A, C]\ ]\!] \neq \emptyset$.

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 [Bro92b] 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

$$A(i) \stackrel{\text{def}}{=} \mathsf{true},$$
$$C(i,o) \stackrel{\text{def}}{=} \#i = \infty \Leftrightarrow \#o < \infty.$$

To see that this specification is non-feasible, assume the opposite. This means it is satisfied by at least one stream processing function f. f is continuous which implies that for every strictly increasing chain $\hat{i}$ we have:

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

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

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

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

The operators $\circ$, $\|$ and μ 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 three composition operators. For example, a mixed specification can be of the form

$$([A_1, C_1] \,\|\, [A_2, C_2]) \circ (F_1 \,\|\, \mu\,[A_3, C_3]).$$

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 (see Section 2.4.1, 2.4.2 and 2.4.3).

3.2 Refinement

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 be generalized to mixed specifications in a straightforward way: a mixed specification $Spec_2$ refines another mixed specification $Spec_1$ iff the denotation of $Spec_2$ is contained in or equal to the denotation of $Spec_1$.

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. In the next section, we will give a number of refinement rules geared towards a methodology of formal, stepwise, top-down refinement, i.e. an agent is designed from a specification in a series of refinement steps using mathematical tools. A stepwise refinement is depicted in Figure 3.2. The requirement specification $[A_0, C_0]$ is finally refined by a network of three agents, namely $\mu\,(F_3 \parallel (F_1 \circ F_2))$.

$$\begin{array}{ccccc} & & & & [A_4, C_4] \;\rightsquigarrow\; F_1 \\ & & [A_2, C_2] & \rightsquigarrow & \circ \\ & & & & [A_5, C_5] \;\rightsquigarrow\; F_2 \\ [A_0, C_0] & \rightsquigarrow & \mu\,[A_1, C_1] & \rightsquigarrow & \parallel \\ & & & & \\ & & [A_3, C_3] & \rightsquigarrow & F_3 \end{array}$$

Figure 3.2: Stepwise Refinement

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.

3.3 Refinement Rules

Ideally, when developing an agent, one starts with a quite abstract specification which in a step-wise, top-down fashion is decomposed into a network of subspecifications amenable to be refined by communicating agents of adequate complexity. Refinement rules can be used to check the correctness of each decomposition step at the point where it is taken. As mentioned before, this report concentrates on the design of networks of agents, and rules (and program constructs) for implementing basic agents will not be given.

Generally our rules have the following form

$$\frac{\begin{array}{l} Premise_1 \\ \quad \vdots \\ Premise_n \end{array}}{Spec_1 \rightsquigarrow Spec_2}$$

stating, that provided the n premises hold, $Spec_1$ can be refined by $Spec_2$. The rules are sound in the following sense: given that the premises hold, then the conclusion holds. We distinguish between two kinds of rules, namely *consequence* and *decomposition* rules.

3.3.1 Consequence Rules

The first rule states that a specification's assumption can be weakened and its commitment can be strengthened.

Rule 1 :

$$\frac{A_1 \Rightarrow A_2 \quad A_1 \wedge C_2 \Rightarrow C_1}{[A_1, C_1] \rightsquigarrow [A_2, C_2]}$$

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 :

$$\frac{Spec_1 \rightsquigarrow Spec_2 \quad Spec_2 \rightsquigarrow Spec_3}{Spec_1 \rightsquigarrow Spec_3}$$

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 $Spec_1$.

3.3.2 Decomposition Rules

There is one rule for each of the composition operators $\circ, \|$ and μ. Each of them describes under which conditions the actual operator can be used to decompose a simple specification.

Given that the input/output variables are named in accordance with Figure 2.1 on Page 8, then the rule for sequential composition can be formulated as follows:

Rule 4 :

$$\frac{\begin{array}{l} A \Rightarrow A_1 \\ A \wedge C_1 \Rightarrow A_2 \\ A \wedge (\exists x.\, C_1 \wedge C_2) \Rightarrow C \end{array}}{[A, C] \rightsquigarrow [A_1, C_1] \circ [A_2, C_2]}$$

This rules 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 1 is a short-hand for the following rule:

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

Throughout this report, 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 $\exists x.\, C_1(i, x) \wedge C_2(x, o)$ 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 2.2 on Page 9, 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 5 :

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

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 2.3 on Page 9. This means that the component specification $[A_1, C_1]$ has $(i, x, r)/(o, y, s)$ as input/output variables, and that the overall specification $[A, C]$ has $(i, r)/(o, y, s)$ as input/output variables.

Rule 6 :

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

The rule is based on the stepwise computation of the feedback streams formally characterized by Proposition 1. Initially the feedback streams are empty. Then the agent starts to work by 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 10.

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 parallel composition with mutual feedback, depicted in Figure 3.3, can be modeled by combining the agents in parallel and then applying the feedback operator twice, i.e. by an agent of the form $\mu\,(\mu\,(f_1 \parallel f_2))$, from now on shortened to $\mu\,(f_1 \parallel f_2)$. It is possible to handle any such construct using the rules already introduced. However, from a methodological point of view, it is sensible to have a special rule

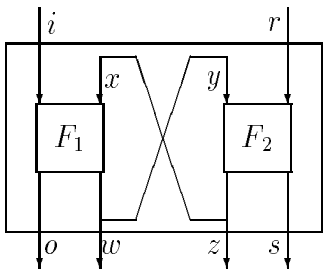

Figure 3.3: Parallel Composition with Mutual Feedback.

Rule 7 :

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

which applies to this coupling of two agents. The component specifications have respectively $(i, x)/(o, w)$ and $(y, r)/(z, s)$ as input/output variables. The overall specification has $(i, r)/(o, w, z, s)$ as input/output variables. In some sense, this rule can be seen as a "generalisation" of Rule 6. 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 $A_1 \wedge A_2$. The invariant is now $A_1 \vee A_2$. The invariant holds initially (second premise) and is maintained by each computation step (fourth and fifth premise), in which case it also holds after infinitely many computation steps (first premise). The conclusion is then a consequence of premise three, four and five.

Note, that without the existential quantifiers occurring in the component specifications, the rule becomes too weak. The problem is that 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 through its input channel, outputs the sum of all numbers received up to the actual 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

$$A_{\mathrm{SUM}}(r) \stackrel{\mathsf{def}}{=} \#r = \infty,$$
$$C_{\mathrm{SUM}}(r, o) \stackrel{\mathsf{def}}{=} \#o = \infty \wedge \forall j \in \mathsf{N}.\, o_j = \textstyle\sum_{k=1}^{j} r_k.$$

SUM can be refined by a network $\mu\,(\mathrm{PR0} \parallel \mathrm{ADD}) \circ \mathrm{STR}$ as depicted in Figure 3.4. 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. PR0, on the other hand, is required to specify an agent which outputs its input stream prefixed with 0. This means that if $A_{\mathrm{SUM}}(r)$ then

$$z = \langle \Sigma_{j=1}^{1} r_j \rangle \circ \langle \Sigma_{j=1}^{2} r_j \rangle \circ \ \ldots \ \circ \langle \Sigma_{j=1}^{n} r_j \rangle \circ \ \ldots \ .$$

where z is the right-hand side output stream of $\mu\,(\mathrm{PR0} \parallel \mathrm{ADD})$. Hence, it is enough to require STR to characterize an agent which outputs its second input stream. More formally:

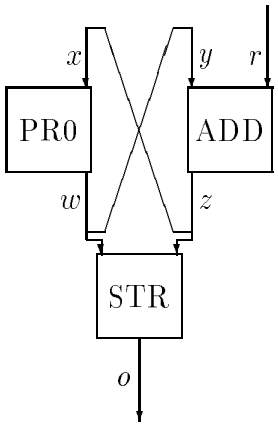

Figure 3.4: Network Refining SUM.

$[A_{\mathrm{PR0}}, C_{\mathrm{PR0}}]$,
$[A_{\mathrm{ADD}}, C_{\mathrm{ADD}}]$,
$[A_{\mathrm{STR}}, C_{\mathrm{STR}}]$,

where

$A_{\mathrm{PR0}}(x) \stackrel{\mathsf{def}}{=} \mathsf{true}$,
$C_{\mathrm{PR0}}(x, w) \stackrel{\mathsf{def}}{=} w = \langle 0 \rangle \circ x$,

$A_{\mathrm{ADD}}(y, r) \stackrel{\mathsf{def}}{=} \#r = \infty$,
$C_{\mathrm{ADD}}(y, r, z) \stackrel{\mathsf{def}}{=} \#z = \#y \wedge \forall j \in \mathsf{dom}(z).\, z_j = r_j + y_j$,

$A_{\mathrm{STR}}(w, z) \stackrel{\mathsf{def}}{=} \mathsf{true}$,
$C_{\mathrm{STR}}(w, z, o) \stackrel{\mathsf{def}}{=} o = z$.

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

$A'(r) \stackrel{\mathsf{def}}{=} A_{\mathrm{SUM}}(r)$,
$C'(r, w, z) \stackrel{\mathsf{def}}{=} C_{\mathrm{SUM}}(r, z)$.

Since

$A_{\mathrm{SUM}}(r) \Rightarrow A'(r)$,
$C'(r, w, z) \Rightarrow A_{\mathrm{STR}}(w, z)$,
$\exists w, z \in \mathsf{N}^{\omega}.\, C'(r, w, z) \wedge C_{\mathrm{STR}}(w, z, o) \Rightarrow C_{\mathrm{SUM}}(r, o)$,

it follows from Rule 4 that

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

The right-hand side component $[A', C']$ can be refined further by observing that

$$adm(A_{\mathrm{PR0}}(x)) \vee adm(\lambda y.A_{\mathrm{ADD}}(y, r))$$
$$A'(r) \Rightarrow A_{\mathrm{PR0}}(\langle\rangle) \vee A_{\mathrm{ADD}}(\langle\rangle, r),$$
$$C_{\mathrm{PR0}}(x, w) \wedge A_{\mathrm{ADD}}(y, r) \wedge C_{\mathrm{ADD}}(y, r, z) \Rightarrow C'(r, w, z),$$
$$A'(r) \wedge A_{\mathrm{ADD}}(y, r) \wedge C_{\mathrm{ADD}}(y, r, z) \Rightarrow A_{\mathrm{PR0}}(z),$$
$$A'(r) \wedge C_{\mathrm{PR0}}(x, w) \Rightarrow A_{\mathrm{ADD}}(w, r),$$

in which case it follows from Rule 7 that

$$[A', C'] \rightsquigarrow \mu\,([A_{\mathrm{PR0}}, C_{\mathrm{PR0}}] \parallel [A_{\mathrm{ADD}}, C_{\mathrm{ADD}}]).$$

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

$$[A_{\mathrm{SUM}}, C_{\mathrm{SUM}}] \rightsquigarrow \mu\,([A_{\mathrm{PR0}}, C_{\mathrm{PR0}}] \parallel [A_{\mathrm{ADD}}, C_{\mathrm{ADD}}]) \circ [A_{\mathrm{STR}}, C_{\mathrm{STR}}].$$

Thus, the proposed decomposition is valid. Further refinements of the three component specifications ADD, PR0 and STR may now be carried out in isolation. □

3.4 Completeness

Informal soundness proofs have been given above. More detailed proofs for Rule 6 and 7 can be found in Section A.1.1 of the appendix. In this section we will deal with completeness issues.

3.4.1 Semantic Completeness

In the examples above a predicate calculus related *assertion language* has been employed for writing specifications. However, in this report 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 *semantically complete* in the following sense: if F is a deterministic agent built from basic deterministic agents using the operators for sequential composition, parallel composition and feedback, and

$$[A, C] \rightsquigarrow F,$$

then F can be deduced from $[A, C]$ using Rule 1-6, given that

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

See section A.1.2 of the appendix for a detailed proof. Proposition 9 shows that Rule 7 is complete in a similar sense.

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.

3.4.2 Adaptation Completeness

Another completeness result we would have liked our logic to satisfy is what is usually referred to as *adaptation completeness* [Zwi89]. For our logic to be adaptation complete it must be possible to show that whenever

$$[A_1, C_1] \rightsquigarrow [A_2, C_2],$$

then $[A_2, C_2]$ can be deduced from $[A_1, C_1]$, using Rule 1 - 6. Unfortunately this is not possible:

Example 4 :
Let

$$A_1(i) \stackrel{\mathsf{def}}{=} \mathsf{true},$$
$$C_1(i,o) \stackrel{\mathsf{def}}{=} \#i = \#o \wedge (\#i = \infty \Rightarrow o \in \{1\}^\infty) \wedge (\#i \neq \infty \Rightarrow o \in \{1\}^*),$$

$$A_2(i) \stackrel{\mathsf{def}}{=} \mathsf{true},$$
$$C_2(i,o) \stackrel{\mathsf{def}}{=} \#i = \#o \wedge (\#i = \infty \Rightarrow o \in \{1\}^\infty),$$

then it follows from the continuity of stream processing functions that

$$[A_1, C_1] \rightsquigarrow [A_2, C_2].$$

However, $[A_2, C_2]$ cannot be deduced from $[A_1, C_1]$ using Rule 1 - 6. □

If we assume, as for example in [BDD+92], that our assertion language has variables over domains of stream processing functions, then we can get adaptation completeness by adding the following rule:

Rule 8 :

$$\frac{\forall f.\, (\forall i.\, A_2 \Rightarrow C_2[^o_{f(i)}]) \Rightarrow (\forall i.\, A_1 \Rightarrow C_1[^o_{f(i)}])}{[A_1, C_1] \rightsquigarrow [A_2, C_2]}$$

which basically restates the semantics of a simple specification in the assertion language. Here f ranges over the set of type-correct stream processing functions, and both specifications are assumed to have i/o as input/output variables. Rule 1 is a special case of Rule 8. Unfortunately, Rule 8 is often difficult to use in practice, which is why it has not been introduced earlier. Moreover, we believe that in most practical situations Rule 1 is sufficiently strong.

In some sense Rule 8 is just transferring a problem from our specification formalism into the assertion language without really giving any strategy for how to find a proof. Rule 1 - 7 on the other hand, depend upon premises which are relatively easy to discharge.

Observe, that if the assertion language has operators corresponding to $\circ$, $\|$ and μ, then the premises of Rule 4-7 could have been formulated in a similar style. But again, these rules would not be very helpful from a practical point of view.

Chapter 4

Decomposing General Specifications

In the previous chapter we introduced a formalism for the specification of and reasoning about networks of agents. It can be used to derive networks of agents from assumption/commitment specifications by stepwise refinement. It has been proved that the given development rules are semantically complete with respect to deterministic agents, i.e. for any deterministic agent F, if there is a simple specification $[A, C]$ such that

$$[A, C] \rightsquigarrow F,$$

then under the assumptions stated above, F can be deduced from $[A, C]$ using Rule 1 - 6. For nondeterministic agents this does not hold. In this chapter we introduce a more general formalism which provides semantic completeness also for nondeterministic agents.

4.1 Symmetric Specifications

In Section 3.2 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 27. □

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 $A, C \in I^\omega \times O^\omega \to \mathsf{B}$, and

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

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[1].

Another more serious problem is that also symmetric specifications are insufficiently expressive. Consider the following example (taken from [Bro92c]):

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

$$\begin{array}{l} f_1(\langle\rangle) \stackrel{\mathsf{def}}{=} f_2(\langle\rangle) \stackrel{\mathsf{def}}{=} \langle 1\rangle, \\ f_1(\langle 1\rangle) \stackrel{\mathsf{def}}{=} f_4(\langle 1\rangle) \stackrel{\mathsf{def}}{=} \langle 1,1\rangle, \\ f_3(\langle\rangle) \stackrel{\mathsf{def}}{=} f_4(\langle\rangle) \stackrel{\mathsf{def}}{=} \langle\rangle, \\ f_2(\langle 1\rangle) \stackrel{\mathsf{def}}{=} f_3(\langle 1\rangle) \stackrel{\mathsf{def}}{=} \langle 1\rangle, \\ y = \langle 1,1\rangle \circ x \Rightarrow f_1(y) \stackrel{\mathsf{def}}{=} f_2(y) \stackrel{\mathsf{def}}{=} f_3(y) \stackrel{\mathsf{def}}{=} f_4(y) \stackrel{\mathsf{def}}{=} \langle 1,1\rangle. \end{array}$$

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 $\mu\, F_1$ may either output $\langle 1,1\rangle$ or $\langle\rangle$, while $\mu\, F_2$ may either output $\langle 1\rangle$ or $\langle\rangle$. □

The expressiveness problem described in Example 6 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 21:

[1] It can be argued that the simple specification $[\mathsf{false}, P]$ suffers from exactly the same problem. However, there is a slight difference. $[\mathsf{false}, P]$ is satisfied by any agent. The same does not hold for $(*)$. As argued in [Bro92b], 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.

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 $Spec_1$, 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

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

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

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

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

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

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.

4.2 General Specifications

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 3.4.1, any deterministic agent can be uniquely characterized by a simple specification, we define a *general specification* as a set of simple specifications:

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

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

$A \in I^\omega \times T \to \mathsf{B}$,
$C \in I^\omega \times T \times O^\omega \to \mathsf{B}$,
$H \in T \to \mathsf{B}$.

T is the type of the indices and H, the *hypothesis predicate*, is a predicate of this type. Its *denotation*

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

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

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

$$H_{\mathrm{UB}}(h) \stackrel{\mathsf{def}}{=} \mathsf{true}$$

as hypothesis predicate. Thus, every infinite stream over $\{ok, fail\}$ is a legal hypothesis. Since any hypothesis h is infinite, the n'th data element occurring in an input stream i straightforwardly corresponds to the n'th element of h, which is either equal to ok or $fail$. Now, if for a particular pair of input i and hypothesis h a data element d in i 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 needed. Let

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

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

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

$$accept(\langle\rangle, h) = \langle\rangle,$$
$$accept(\langle ? \rangle \circ i, h) = accept(i, h),$$
$$accept(\langle d \rangle \circ i, \langle fail \rangle \circ h) = accept(i, h),$$
$$accept(\langle d \rangle \circ i, \langle ok \rangle \circ h) = \langle d \rangle \circ accept(i, h).$$

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 $h_{\#(D \copyright i)+1}$ denotes the element of the hypothesis stream which corresponds to d in the sense explained above. For any finite input stream i and any hypothesis h, $state(i, h)$ returns the buffer's state after it has processed i 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 the following general specification:

$$[A_{\mathrm{UB}}, C_{\mathrm{UB}}]_{H_{\mathrm{UB}}}$$

where

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

$$C_{\mathrm{UB}}(i, h, o) \stackrel{\mathsf{def}}{=} D \copyright o \sqsubseteq accept(i, h) \wedge \{ok, fail\} \copyright o \sqsubseteq h \wedge \#\{ok, fail\} \copyright o = \#D \copyright i \\ \wedge \#D \copyright o = \#\{?\} \copyright i.$$

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.

Note that the assumption is a safety property while the commitment also contains a liveness part. Although *ok* and *fail* occur in the buffer's output as well as in the hypothesis, there is a fundamental difference between these two kinds of use. In the first case *ok* and *fail* represent messages the buffer sends to the environment. In the second case *ok* and *fail* model internal decisions. Since the messages are supposed to inform the environment

about the internal decisions, the same symbols have been used.

The specification UB is satisfied by a buffer which rejects all data elements it receives. To avoid that it is enough to strengthen the hypotheseses predicate as follows:

$$H_{\mathrm{UB}'}(h) \stackrel{\mathsf{def}}{=} \#\{ok\}ⓒh = \infty.$$

□

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

As the following example shows, the relationship between the assumption, the commitment and the hypothesis predicate of a general specification can be quite subtle.

Example 8 :
Let $T = \{even, odd\}$ be the hypothesis type, and let

$$A : \mathsf{N}^\omega \times T \to \mathsf{B},$$
$$C : \mathsf{N}^\omega \times T \times \mathsf{N}^\omega \to \mathsf{B},$$

be two predicates such that

$$A(i, even) \stackrel{\mathsf{def}}{=} \forall j \in \mathbf{dom}(i).\, i_j \bmod 2 = 0,$$
$$A(i, odd) \stackrel{\mathsf{def}}{=} \forall j \in \mathbf{dom}(i).\, i_j \bmod 2 = 1,$$
$$C(i, h, o) \stackrel{\mathsf{def}}{=} i = o.$$

Then, the following statement is true:

$$[A, C]_{(h=even \vee h=odd)} \rightsquigarrow [A, C]_{(h=even)}.$$

In the first specification the assumption admits as input a stream of even numbers as well as a stream of odd numbers, whereas in the second specification only an even stream is allowed. In fact, by changing the hypothesis predicate we have implicitly strengthened the assumption, and thus restricted the environment. Is this a legal refinement? It is, not only formally, but also in an intuitive sense. In the first specification, whatever the environment sends as input, it can never be sure that the agent's output fulfills the commitment: if it sends a stream of even numbers, the agent may choose to react properly only to streams of odd numbers, and if it sends a stream of odd numbers, the agent may choose the other alternative. There is no way for the environment to influence the agent's choice. So this specification is not very helpful. The second specification, on the other hand, is more demanding with respect to the input. If the environment sends a stream of even numbers, then it knows that the output will be in accordance with the commitment.

□

The following observation is helpful for relating the refinement rules of the previous chapter to those for general specifications given in the next section:

Proposition 3 *Given two general specifications Spec, Spec′, with respectively T, T' as hypothesis types and H, H' as hypothesis predicates, then Spec $\rightsquigarrow$ 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. $H'(h) \Rightarrow Spec_{l(h)} \rightsquigarrow Spec'_h$.

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

The importance of this proposition is that since a simple descendant is a simple specification, the rules of the previous chapter can be used to verify the second condition. Thus the logic for simple specifications, which has been proved sound, can be employed to prove soundness of the logic for general specifications.

To see that the proposition 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 $[\![\ ]\!]$, $f \in [\![Spec]\!]$.

Proposition 3 can of course easily be generalized to the case where $Spec'$ is the result of composing several general specifications using the three basic composition operators. The proof is again straightforward.

4.3 Refinement Rules

This section presents a set of refinement rules for general specifications. Most of these rules are straightforward translations of the rules for simple specifications.

4.3.1 Relationship to Previous Logic

In the preceding section the close relationship between simple and general specifications was described. This is reflected by the following two rules:

Rule 9 :

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

Rule 10 :

$$\frac{\begin{array}{l}\exists h.\, H \\ H \wedge A_1 \Rightarrow A_2 \\ H \wedge A_1 \wedge C_2 \Rightarrow C_1\end{array}}{[A_1, C_1]_H \rightsquigarrow [A_2, C_2]}$$

Rule 9 can be used to refine a simple specification by a general specification, while Rule 10 allows a general specification to be refined by a simple specification.

For the special case that H holds for exactly one $h_0 \in T$, the two rules can be used to deduce the equivalence of $[A, C]_H$ and $[A_{h_0}, C_{h_0}]$, where $[A_{h_0}, C_{h_0}]$ is a simple descendant as defined on Page 26.

4.3.2 Consequence Rules

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. The general rule captures all these aspects:

Rule 11 :

$$\frac{\begin{array}{l}H_2 \Rightarrow H_1[^q_{l(h)}] \\ H_2 \wedge A_1[^q_{l(h)}] \Rightarrow A_2 \\ H_2 \wedge A_1[^q_{l(h)}] \wedge C_2 \Rightarrow C_1[^q_{l(h)}]\end{array}}{[A_1, C_1]_{H_1} \rightsquigarrow [A_2, C_2]_{H_2}}$$

Here $l : T_2 \to T_1$ 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 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 Proposition 3 on page 30, and premise two and three together with Rule 1 imply the second condition of Proposition 3, it follows that the rule is sound.

Rule 2 and 3 remain valid.

4.3.3 Decomposition Rules

As in the case of simple specifications there are three basic decomposition rules plus one rule for parallel composition with mutual feedback. As for Rule 11 their soundness follows straightforwardly from (the general version of) Proposition 3 and the corresponding rules of the previous chapter.

Rule 12 :

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

Rule 13 :

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

Rule 14 :

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

Rule 15 :

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

4.4 Completeness

The completeness results of Chapter 3 can now be generalized.

4.4.1 Semantic Completeness

The logic for general specifications is semantically complete with respect to nondeterministic agents: if F is an agent built from basic agents using the operators for sequential composition, parallel composition and feedback, and

$$[A, C] \rightsquigarrow F,$$

then F can be deduced from $[A, C]$ using Rule 2, 3, 11, 12, 13 and 14, given that

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

See section A.2.2 for a proof. Rule 15 is complete in a similar sense.

Given a nondeterministic agent F, it is straightforward to write a general specification $Spec$ which is semantically equivalent to F. It is enough to choose $I^\omega \xrightarrow{c} O^\omega$ as the hypothesis type. Then

$$[\![Spec]\!] = [\![F]\!]$$

if

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

Hence, for any function $f \in [\![F]\!]$, $[A_{Spec}, C_{Spec}]_{H_{Spec}}$ contains a simple descendant which characterizes only f. Roughly speaking, our specification technique uses a set of relations in the same sense as [BDD+92] employs a set of functions to get around the compositionality problems of relations reported in [BA81].

4.4.2 Adaptation Completeness

As for simple specifications it is easy to formulate a sufficiently strong adaptation rule:

Rule 16 :

$$\frac{\forall f.\,(\exists h.\,H_2 \wedge (\forall i.\,A_2 \Rightarrow C_2[{}^{o}_{f(i)}])) \Rightarrow (\exists h.\,H_1 \wedge (\forall i.\,A_1 \Rightarrow C_1[{}^{o}_{f(i)}]))}{[A_1, C_1]_{H_1} \rightsquigarrow [A_2, C_2]_{H_2}}$$

This rule deals with general specifications. Rules which relate simple and general specifications can be formulated in a similar way.

Chapter 5

Conclusions

Logics for explicit assumption/commitment specifications have been studied for a long time. To begin with, the emphasize was on sequential program design in the style of Hoare-logic. Then the interest turned towards development of parallel programs. Of the latter, many approaches, like [MC81], [Jon83] and [Stø91], deal only with safety predicates and restricted types of liveness predicates and are therefore less general than the logic described in this paper.

[Pnu85], which presents a rule for a shared-state parallel operator, was the first to handle general safety and liveness predicates in a compositional style. Roughly speaking, this operator corresponds to our construct for mutual feedback as depicted in Figure 3.3. The rule differs from our Rule 7 (and 14) in that the induction is explicit, i.e. the user must himself find an appropriate well-orderering. A related rule is formulated in [Pan90]. There is a (naive) 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 in the sense that we can only prove properties which hold for all fixpoints, i.e. not properties which hold for the least fixpoint only.

More recently, [AL90] has proposed a general composition principle with respect to a shared-state model. This principle is similar to Rule 7 in that the induction is only implicit, but differs from Rule 7 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. In fact it can be argued that also our rules are to restrictive in this respect. For example it is often helpful to state that the lengths of the input streams are related in a certain way. When using Rule 6 this can lead to difficulties (see the 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 by restating Rule 6 as below:

Rule 6′ :

$$\frac{\begin{array}{l} adm(A_1) \\ A \Rightarrow A_1[{}^{x\ \ i}_{\langle\rangle\ t(i)_1}] \\ A \wedge A_1[{}^{x}_{y}] \wedge C_1[{}^{x}_{y}] \Rightarrow C \\ A \wedge A_1[{}^{i}_{t(i)_j}] \wedge C_1[{}^{i}_{t(i)_j}] \Rightarrow A_1[{}^{x\ \ i}_{y\ \ t(i)_{j+1}}] \end{array}}{[A, C] \rightsquigarrow \mu\,[A_1, C_1]}$$

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. Rule 7, 13 and 14 can be reformulated in a similar style. The rules are semantically complete in the same sense as earlier. As for all such deduction systems, there are many variations of these rules. We have at the moment not sufficient experience to say which set of rules is the better one from a practical point of view. Some case-studies are currently being carried out.

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 continuous stream processing functions to model agents. There are certain time dependent programs 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. As explained in [Bro92c], such programs can be handled by using timed stream processing functions at the semantic level. The specification formalism and refinement calculus introduced above carry over straightforwardly.

5.1 Acknowledgement

We would like to thank M. Broy who has influenced this work in a number of ways. P. Collette and C.B. Jones have read earlier drafts and provided valuable feedback. Valuable comments have also been received from O.J. Dahl, M. Krětínský, O. Owe, F. Plášil, W.P. de Roever and Q. Xu.

Bibliography

[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.

[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+92] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The design of distributed systems — an introduction to Focus. Technical Report SFB 342/2/92 A, Technische Universität München, 1992.

[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*, pages 311–364. Springer, 1989.

[Bro92a] M. Broy. Compositional refinement of interactive systems. Working Material, International Summer School on Program Design Calculi, August 1992.

[Bro92b] M. Broy. A functional rephrasing of the assumption/commitment specification style. Manuscript, November 1992.

[Bro92c] M. Broy. Functional specification of time sensitive communicating systems. In M. Broy, editor, *Proc. Programming and Mathematical Method*, pages 325–367. Springer, 1992.

[Bro92d] M. Broy. (Inter-) action refinement: the easy way. Working Material, International Summer School on Program Design Calculi, August 1992.

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

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

[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.

[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.

[Mor90] C. Morgan. *Programming from Specifications.* Prentice-Hall, 1990.

[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.

[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.

[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.

Appendix A

Proofs

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

A.1 Logic for Simple Specifications

This section contains proofs related to the logic for simple specifications.

A.1.1 Soundness

Theorem 1 *The logic for simple specifications is sound.*

Proof: The soundness proofs for Rule 1, 2, 3, 4, 5 and 8 should by now be trivial. The soundness of Rule 6 and 7 follow from Proposition 4 and 5.

end of proof

In Proposition 4 we consider only the case that there are no channels corresponding to r and s of Figure 2.3. The proposition can easily be extended to handle the full generality of the feedback operator.

Proposition 4 *If*

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

then

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

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

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

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

$$(\hat{o}_1, \hat{y}_1) \stackrel{\text{def}}{=} (\langle\rangle, \langle\rangle), \tag{A.8}$$

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

Proposition 1 on Page 10 implies

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

Assume for an arbitrary $j \geq 1$

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

A.6, A.8, A.9 and A.11 imply

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

A.4, A.7, A.11 and A.12 imply

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

Thus, for all $j \geq 1$

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

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

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

A.1, A.7, A.10 and A.14 imply

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

A.6, A.7 and A.15 imply

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

A.3, A.7, A.15 and A.16 imply

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

Thus, it has been shown that

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

A.17 and the way f, i, o and y were chosen imply A.5.

end of proof

Proposition 5 *If*

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

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

$$A(i, r) \wedge A_1(i, z, r) \wedge A_2(w, r, i) \wedge C_1(i, z, o, w) \wedge C_2(w, r, z, s) \Rightarrow C(i, r, o, w, z, s), \tag{A.20}$$

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

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

then

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

Proof: Assume that A.18 - A.22 hold, and that f_1, f_2, i, r, o, w, z and s are such that

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

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

$$A(i,r) \wedge \mu\, f_1 \parallel f_2(i,r) = (o,w,z,s). \tag{A.26}$$

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

$$(\hat{o}_1, \hat{w}_1, \hat{z}_1, \hat{s}_1) \stackrel{\mathsf{def}}{=} (\langle\rangle, \langle\rangle, \langle\rangle, \langle\rangle), \tag{A.27}$$

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

Proposition 1 on Page 10 implies

$$\sqcup(\hat{o}, \hat{w}, \hat{z}, \hat{s}) = (o,w,z,s). \tag{A.29}$$

Assume for an arbitrary $j \geq 1$

$$A_1(i, \hat{z}_j, r) \vee A_2(\hat{w}_j, r, i). \tag{A.30}$$

A.24, A.25, A.27, A.28 and A.30 imply

$$(A_1(i, \hat{z}_j, r) \wedge C_1(i, \hat{z}_j, \hat{o}_{j+1}, \hat{w}_{j+1})) \vee (A_2(\hat{w}_j, r, i) \wedge C_2(\hat{w}_j, r, \hat{z}_{j+1}, \hat{s}_{j+1}). \tag{A.31}$$

A.21, A.22, A.26 and A.31 imply

$$A_1(i, \hat{z}_{j+1}, r) \vee A_2(\hat{w}_{j+1}, r, i).$$

Thus, for all $j \geq 1$

$$A_1(i, \hat{z}_j, r) \vee A_2(\hat{w}_j, r, i) \Rightarrow A_1(i, \hat{z}_{j+1}, r) \vee A_2(\hat{w}_{j+1}, r, i). \tag{A.32}$$

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

$$A_1(i, \hat{z}_j, r) \vee A_2(\hat{w}_j, r, i). \tag{A.33}$$

A.18, A.26, A.29 and A.33 imply

$$A_1(i, z, r) \vee A_2(w, r, i).$$

Without loss of generality, assume

$$A_1(i, z, r). \tag{A.34}$$

A.24, A.26 and A.34 imply

$$C_1(i, z, o, w). \tag{A.35}$$

A.21, A.26, A.34 and A.35 imply

$$A_2(w, r, i). \tag{A.36}$$

A.25, A.26 and A.36 imply

$$C_2(w, r, z, s). \tag{A.37}$$

A.20, A.26, A.34, A.35, A.36 and A.37 imply

$$C(i, r, o, w, z, s).$$

Thus, it has been shown that

$$A(i,r) \wedge \mu\, f_1 \parallel f_2(i,r) = (o,w,z,s) \Rightarrow C(i,r,o,w,z,s). \tag{A.38}$$

A.38 and the way f_1, f_2, i, r, o, w, z and s were chosen imply A.23.

end of proof

A.1.2 Semantic Completeness

Theorem 2 *If F is a deterministic agent built from basic deterministic agents using the operators for sequential composition, parallel composition and feedback, and*

$$[A, C] \rightsquigarrow F,$$

then F can be deduced from $[A, C]$ using Rule 1 - 6, given that

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

Proof: Follows straightforwardly from Proposition 6 - 8.
end of proof

Proposition 6 *If*

$$f_1 \circ f_2 \in [\![[A, C]]\!], \tag{A.39}$$

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

$$f_1 \in [\![[A_1, C_1]]\!], \tag{A.40}$$
$$f_2 \in [\![[A_2, C_2]]\!], \tag{A.41}$$
$$A(i) \Rightarrow A_1(i), \tag{A.42}$$
$$A(i) \wedge C_1(i, x) \Rightarrow A_2(x), \tag{A.43}$$
$$A(i) \wedge \exists x \in X^\omega . C_1(i, x) \wedge C_2(x, o) \Rightarrow C(i, o). \tag{A.44}$$

Proof: Assume A.39. Let

$$A_1(i) \stackrel{\text{def}}{=} \mathsf{true},$$
$$A_2(x) \stackrel{\text{def}}{=} \mathsf{true},$$
$$C_1(i, x) \stackrel{\text{def}}{=} f_1(i) = x,$$
$$C_2(x, o) \stackrel{\text{def}}{=} f_2(x) = o.$$

Since

$$A(i) \Rightarrow C_1(i, f_1(i))$$

is equivalent to

$$A(i) \Rightarrow f_1(i) = f_1(i)$$

it follows that A.40 holds. A.41 follows by a similar argument. A.42 and A.43 hold trivially. To prove A.44, firstly observe that the antecedent of A.44 is equivalent to

$$A(i) \wedge \exists x \in X^\omega . f_1(i) = x \wedge f_2(x) = o. \tag{A.45}$$

A.45 implies

$$A(i) \wedge f_1 \circ f_2(i) = o.$$

A.39 implies

$$A(i) \wedge f_1 \circ f_2(i) = o \Rightarrow C(i, o).$$

Thus, A.44 holds.
end of proof

Proposition 7 *If*

$$f_1 \parallel f_2 \in [\![[A, C]]\!], \tag{A.46}$$

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

$$f_1 \in [\![[A_1, C_1]]\!], \tag{A.47}$$
$$f_2 \in [\![[A_2, C_2]]\!], \tag{A.48}$$
$$A(i, r) \Rightarrow A_1(i) \wedge A_2(r), \tag{A.49}$$
$$A(i, r) \wedge C_1(i, o) \wedge C_2(r, s) \Rightarrow C(i, r, o, s). \tag{A.50}$$

Proof: Assume A.46. Let

$$A_1(i) \stackrel{\text{def}}{=} \mathsf{true},$$
$$A_2(r) \stackrel{\text{def}}{=} \mathsf{true},$$
$$C_1(i, o) \stackrel{\text{def}}{=} f_1(i) = o,$$
$$C_2(r, s) \stackrel{\text{def}}{=} f_2(r) = s.$$

It follows trivially that A.47-A.50 hold.
end of proof

Proposition 8 *If*

$$\mu f \in [\![[A, C]]\!], \tag{A.51}$$

then there are A_1 *and* C_1 *such that*

$$f \in [A_1, C_1], \tag{A.52}$$
$$A(i) \Rightarrow adm(\lambda x \in Y^\omega . A_1(i, x)), \tag{A.53}$$
$$A(i) \Rightarrow A_1(i, \langle\rangle), \tag{A.54}$$
$$A(i) \wedge A_1(i, y) \wedge C_1(i, y, o, y) \Rightarrow C(i, o, y), \tag{A.55}$$
$$A(i) \wedge A_1(i, x) \wedge C_1(i, x, o, y) \Rightarrow A_1(i, y). \tag{A.56}$$

Proof: Assume A.51. Let

$$A_1(i,x) \stackrel{\text{def}}{=} (\exists j \in \mathsf{N}.\, K_j(i,x)) \vee (\exists \hat{y} \in Ch(Y^\omega).\, x = \sqcup \hat{y} \wedge \forall j \in \mathsf{N}.\, K_j(i,\hat{y}_j)),$$
$$K_1(i,x) \stackrel{\text{def}}{=} x = \langle\rangle,$$
$$K_j(i,x) \stackrel{\text{def}}{=} \exists x' \in Y^\omega.\, \exists o \in O^\omega.\, K_{j-1}(i,x') \wedge f(i,x') = (o,x) \qquad if\ j > 1,$$
$$C_1(i,x,o,y) \stackrel{\text{def}}{=} f(i,x) = (o,y).$$

Basically, $K_j(i,x)$ characterizes the j'th element x of the Kleene-chain for the function f and the given input i (see Page 10 for the definition of the Kleene-chain). 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. A.52 holds trivially. A.53 follows from the second disjunct of A_1's definition, while A.54 is a direct consequence of the definition of K_1. To prove A.55, observe that the antecedent of A.55 is equivalent to

$$A(i) \wedge A_1(i,y) \wedge f(i,y) = (o,y). \tag{A.57}$$

Since A_1 characterizes the Kleene-chain or its least upper bound for a given input i, A.57 implies

$$A(i) \wedge A_1(i,y) \wedge \mu\, f(i) = (o,y). \tag{A.58}$$

A.51 implies

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

Thus A.55 holds. To prove A.56, let i, x, o and y be such that

$$A(i) \wedge A_1(i,x) \wedge C_1(i,x,o,y). \tag{A.59}$$

A.59 implies

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

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 A.60 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

$$K_j(i,x). \tag{A.61}$$

A.60 and A.61 imply

$$K_{j+1}(i,y). \tag{A.62}$$

A.62 implies

$$A_1(i,y). \tag{A.63}$$

This proves A.56.

end of proof

A.1.3 Additional Proofs

Proposition 9 shows that Rule 7 is complete in the same sense as Rule 4-6.

Proposition 9 *If*

$$\mu\,(f_1 \parallel f_2) \in [\![\ [A, C]\ ]\!] \tag{A.64}$$

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

$$f_1 \in [\![\ [\exists r \in R^\omega.\, A_1, C_1]\ ]\!], \tag{A.65}$$
$$f_2 \in [\![\ [\exists i \in I^\omega.\, A_2, C_2]\ ]\!], \tag{A.66}$$
$$A(i,r) \Rightarrow adm(\lambda x \in Z^\omega.\, A_1(i,x,r)) \vee adm(\lambda y \in W^\omega.\, A_2(y,r,i)), \tag{A.67}$$
$$A(i,r) \Rightarrow A_1(i, \langle\rangle, r) \vee A_2(\langle\rangle, r, i), \tag{A.68}$$
$$A(i,r) \wedge A_1(i,z,r) \wedge A_2(w,r,i) \wedge C_1(i,z,o,w) \wedge C_2(w,r,z,s) \Rightarrow C(i,r,o,w,z,s), \tag{A.69}$$
$$A(i,r) \wedge A_1(i,x,r) \wedge C_1(i,x,o,w) \Rightarrow A_2(w,r,i), \tag{A.70}$$
$$A(i,r) \wedge A_2(y,r,i) \wedge C_2(y,r,z,s) \Rightarrow A_1(i,z,r). \tag{A.71}$$

Proof: Let

$$A_1(i,x,r) \stackrel{\text{def}}{=} (\exists j \in \mathsf{N}.\, \exists y \in W^\omega.\, K_j(i,r,x,y)) \vee (\exists \hat{x} \in Ch(Z^\omega).\, \exists \hat{y} \in Ch(W^\omega).\, x = \sqcup \hat{x} \wedge \forall j \in \mathsf{N}.\, K_j(i,r,\hat{x}_j,\hat{y}_j)),$$
$$A_2(y,r,i) \stackrel{\text{def}}{=} (\exists j \in \mathsf{N}.\, \exists x \in Z^\omega.\, K_j(i,r,x,y)) \vee (\exists \hat{x} \in Ch(Z^\omega).\, \exists \hat{y} \in Ch(W^\omega).\, y = \sqcup \hat{y} \wedge \forall j \in \mathsf{N}.\, K_j(i,r,\hat{x}_j,\hat{y}_j)),$$
$$K_1(i,r,x,y) \stackrel{\text{def}}{=} x = \langle\rangle \wedge y = \langle\rangle,$$
$$K_j(i,r,x,y) \stackrel{\text{def}}{=} \exists x' \in Z^\omega.\, \exists y' \in W^\omega.\, \exists o \in O^\omega.\, \exists s \in S^\omega.\, K_{j-1}(i,r,x',y') \wedge f_1(i,x') = (o,y) \wedge f_2(y',r) = (x,s),$$
$$C_1(i,x,o,w) \stackrel{\text{def}}{=} f_1(i,x) = (o,w),$$
$$C_2(y,r,z,s) \stackrel{\text{def}}{=} f_2(y,r) = (z,s).$$

A.65-A.71 can now be deduced from A.64 by an argument similar to that of Proposition 8.

end of proof

A.2 Logic for General Specifications

This section contains proofs related to the logic for general specifications.

A.2.1 Soundness

Theorem 3 *The logic for general specifications is sound.*

Proof: The soundness of Rule 9 and 10 follow from Proposition 10 and 11. The soundness of Rule 11-15 follow easily from Proposition 3 and the soundness of the corresponding rules for simple specifications. Rule 16 is trivially sound.

end of proof

Proposition 10 *If*

$$H(h) \wedge A(i) \Rightarrow A'(i,h), \tag{A.72}$$
$$H(h) \wedge A(i) \wedge C'(i,h,o) \Rightarrow C(i,o) \tag{A.73}$$

then

$$[A,C] \rightsquigarrow [A',C']_H. \tag{A.74}$$

Proof: Assume that A.72-A.73 hold, and that f is such that

$$f \in [\![\ [A',C']_H\ ]\!]. \tag{A.75}$$

A.75 implies there is a hypotheses h such that

$$H(h) \wedge f \in [\![\ [A'_h,C'_h]\ ]\!]. \tag{A.76}$$

A.72, A.73, A.76 and Rule 1 imply

$$f \in [\![\ [A,C]\ ]\!]. \tag{A.77}$$

A.77 and the way f and h were chosen imply A.74.

end of proof

Proposition 11 *If*

$$\exists h \in T.\, H(h), \tag{A.78}$$
$$H(h) \wedge A(i,h) \Rightarrow A'(i), \tag{A.79}$$
$$H(h) \wedge A(i,h) \wedge C'(i,o) \Rightarrow C(i,h,o) \tag{A.80}$$

then

$$[A,C]_H \rightsquigarrow [A',C']. \tag{A.81}$$

Proof: Assume that A.78-A.80 hold, and that f is such that

$$f \in [\![\ [A',C']\ ]\!]. \tag{A.82}$$

A.78 and A.82 imply there is a hypotheses h such that

$$H(h) \wedge f \in [\![\ [A',C']\ ]\!]. \tag{A.83}$$

A.79, A.80, A.83 and Rule 1 imply

$$H(h) \wedge f \in [\![\ [A,C]\ ]\!]. \tag{A.84}$$

A.84 and the way f and h were chosen imply A.81.

end of proof

A.2.2 Semantic Completeness

Theorem 4 *If F is an agent built from basic agents using the operators for sequential composition, parallel composition and feedback, and*

$$[A, C]_H \rightsquigarrow F,$$

then F can be deduced from $[A, C]_H$ using Rule 2, 3, 11, 12, 13 and 14, given that

- *such a deduction can always be carried out for basic agents,*
- *any valid formula in the assertion logic is provable,*
- *any predicate we need can be expressed in the assertion language.*

Proof: 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 Propositions 6-8 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.

end of proof