Specification and Verification of Communicating Systems with Value Passing

by

Dilian Borissov Gurov
Dipl. Eng., Higher Institute for Mechanical and Electrical Engineering, Sofia, Bulgaria, 1989

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

Dr. B.M. Kapron, Supervisor (Department of Computer Science)

Dr. H.A. Müller, Supervisor (Department of Computer Science)

Dr. M.H.M. Cheng, Departmental Member (Department of Computer Science)

Dr. N. Dimopoulos, Outside Member (Department of Electrical and Computer Engineering)

Dr. M. Greenstreet, External Examiner (Department of Computer Science, UBC)

University of Victoria

Supervisors: Dr. B.M. Kapron, Dr. H.A. Müller

ABSTRACT

The present Thesis addresses the problem of specification and verification of communicating systems with value passing. We assume that such systems are described in the well-known Calculus of Communicating Systems, or rather, in its value passing version. As a specification language we propose an extension of the Modal μ-Calculus, a poly-modal first-order logic with recursion. For this logic we develop a proof system for verifying judgements of the form $b \vdash E : \Phi$ where E is a sequential CCS term and b is a Boolean assumption about the value variables occurring free in E and Φ. Proofs conducted in this proof system follow the structure of the process term and the formula. This syntactic approach makes proofs easier to comprehend and machine assist. To avoid the introduction of global proof rules we adopt a technique of tagging fixpoint formulae with all relevant information needed for the discharge of reoccurring sequents. We provide such tagged formulae with a suitable semantics. The resulting proof system is shown to be sound in general and complete (relative to external reasoning about values) for a large class of sequential processes and logic formulae. We explore the idea of using tags to three different settings: value passing, extended sequents, and negative tagging.

Examiners:

Dr. B.M. Kapron, Supervisor (Department of Computer Science)

Dr. H.A. Müller, Supervisor (Department of Computer Science)

Dr. M.H.M. Cheng, Departmental Member (Department of Computer Science)

Dr. N. Dimopoulos, Outside Member (Department of Electrical and Computer Engineering)

Dr. M. Greenstreet, External Examiner (Department of Computer Science, UBC)

Contents

List of Figures

Acknowledgement

I would like to thank my supervisors Dr. Bruce Kapron and Dr. Hausi Müller for their guidance and support throughout the time of my graduate studies.

I would also like to thank Dr. John Ellis, Dr. Mantis Cheng and many graduate students from the Department for creating a stimulating research atmosphere.

Finally, I would like to thank my wife Elena and all my friends which made my stay at the University a remarkably pleasant experience.

Last but not least, I would like to acknowledge Dr. Iwan Tabakow, the person whom I owe my academic career.

Dedication

To Vladimir Vysotskij,
whose horses
never stopped.

Chapter 1

Introduction

Communicating Systems Since their invention, computers have evolved from simple calculators to extremely complex devices which have been entrusted with a wide range of information processing responsibilities. Alongside with information processing, *interaction* has become an increasingly crucial aspect of computer systems. For many types of systems this aspect is in fact the more important one. Communication protocols, telephone switching systems and mobile robot control systems are examples of systems, which fulfill their primary goals by interacting with other systems rather than by processing information. The terms *reactive* and *real-time* systems often refer to systems of the above type. In this thesis we use the term *communicating system* to refer to any system for which the information processing aspect can be considered less relevant, and whose interaction behaviour is of main interest.

Interaction can be understood differently depending on the properties of the communication medium. It can also be viewed on different levels of abstraction. Here we consider only a most basic type of interaction, namely *handshake*-type *atomic* interaction between two *agents*, on which Milner's Calculus of Communicating Systems (CCS) is based [Mil89]. This calculus has been designed to serve as a theoretical foundation for the study of concurrency rather than as a specification language for real-world applications; nonetheless, there are numerous practical examples where this economic language has proved adequate. We also assume that *actions* (as inter-

actions are called in CCS) can have parameters, so that values in some data domain can be transmitted via actions. This phenomenon is called *value passing*; the resulting version of CCS is usually referred to as Value Passing CCS.

One significant conceptual difference between communicating systems and information processing systems in general is presented by the rôle of *termination*. While termination is usually a desired property for information processing applications, indicating that some task has been successfully completed, for communicating systems it has to be considered rather a catastrophe: termination of a communicating system implies that no communication with the system is henceforth possible. Most existing formal techniques for functional analysis and synthesis of systems rely on the notion of termination, representing the behaviour of a system as a mapping from some set of allowable *initial configurations* to some set of desirable *final configurations*. This approach is not adequate for describing the ongoing behaviour of communicating systems. One alternative operational approach is to consider not the overall behaviour, but just the result of a single communication, as a mapping between configurations; the resulting formal notion is called a Labelled Transition System (LTS) and provides a semantic framework for many formal notations for describing ongoing systems behaviour, including CCS.

Another important characteristic of communicating systems is that they are inherently *distributed*: a communicating system is usually interacting with other systems of the same type, so the overall system consists of components which are communicating systems themselves. This brings about the question of how to represent adequately *global* configurations as (possibly structured) collections of *local* configurations, and how to compose component behaviours to form the behaviour of the composite system. The approach taken by Milner is to *interleave* these behaviours, but other approaches are also possible, notably the partial order semantics approach advocated by Petri in [Pet76].

Communicating Systems Design A rigorous systematic design methodology for communicating systems would include the following design phases:

- *Specification*: from an informal description of the requirements to the system's behaviour a formal specification, written in a suitable formal (e.g., logic) language is derived;
- *Modelling*: guided by the formal specification, a formal model of the desired system's behaviour is obtained;
- *Verification*: the formal model is checked against the formal specification to ensure all requirements are met. If these are not met, the Modelling phase is reentered;
- *Test Generation*: from the model, test suites are derived to test (validate) the system after it has been implemented;
- *Implementation*: the verified model is implemented in hardware/software (This phase can be performed concurrently with the previous one);
- *Validation*: the implemented system is tested using the test suites generated earlier.

Such a rigorous methodology is rarely used in practice due to the enormous competition and pressure for early delivery of software products. Producers of *safety-critical* systems, however, cannot afford risk and have to ensure the correct functioning of their products. This means that they have to follow a more or less rigorous design methodology like the above one. The techniques we develop here assume such a formal approach and address the specification and mainly the verification phases, but are necessarily related to the modelling phase as well, since these three phases cannot be considered separately.

There are two main approaches to formally specifying a communicating system. In the first of these, the specification is itself a model and is described in the same

modelling language in which the system is modelled later. This model describes the behaviour of the system as seen from the environment (in other words, its *interface* behaviour), and does not show the system's internal organization. Then, to verify the model resulting from the modelling phase means to show that the two models, i.e., specification and actual model, are equivalent according to some suitable notion of behavioural equivalence. One such notion is *bisimulation* equivalence (or observation congruence) [Mil89].

The second approach, which we follow here, is a logical one: a communicating system is specified with a set of logic formulae expressing the properties which the interface behaviour of the system is required to possess. Then, to verify the model means to check whether all formulae in the set hold in this particular model. The process of accomplishing this is usually called *model checking*.

These two approaches can complement each other if one wants to start with a very high-level specification, which is best given as a set of properties, and then to produce a high-level interface model, and finally to obtain through a sequence of refinements a model detailed enough to be implemented.

Aim of the Thesis This thesis addresses the following verification problem: Given a system, described in Value Passing CCS, and a system property, described as a formula in a suitable logic language, check whether the system possesses the property (i.e. the model *satisfies* the formula). The logic language we consider is the Modal μ-Calculus, introduced by Kozen in [Koz83], which we extend with appropriate (first-order) constructs to take into account the values being communicated. We present a proof system for proving satisfaction between a process and a formula which is sound and complete for a large class of sequential processes (i.e., processes not involving parallel composition[1]) and logic formulae. *Soundness* of the proof system means that one cannot derive satisfaction unless it really holds. *Completeness* on the other hand

[1]Parallel composition is handled separately with the help of an additional proof system; this shall be discussed later.

guarantees that one can always derive such a satisfaction when it holds. Together, these two properties of the proof system guarantee that one can prove (in this proof system) that a process satisfies a formula exactly when this is really the case. Due to the expressive power of the logic, our completeness result is necessarily a *relative* one: we assume that all reasoning concerning the values being communicated is done externally to the present proof system.

Proof systems of the above kind have been the focus of many research papers and dissertations. These papers differ from our approach in that they refer to the LTS of the system being verified instead of the process description itself (as for example in [Cle90, SW91, Bra92, BS92, And93, Dam93, HL95, Rat97, RH97]), or in that they consider propositional modal μ-calculus formulae only, i.e. do not address value passing (as in most of the above references, as well as in [ASW94, Dam95]).

Value Passing Our interest in value passing comes from the fact that many communicating systems do not just carry around values, but also use values to achieve a desired distributed behaviour. For example, in the Alternating Bit Protocol (ABP) special values are passed during the communication between sender, receiver, and medium, to assure the correct transmission of data. If the value domain necessary to achieve a desired distributed behaviour is finite (as it is the case with the ABP), one could abstract from the values thus simplifying the verification process. This, however, is not always possible. It would also require the formulae from the specification to be translated accordingly, which might result in huge and unreadable ones. Another drawback of abstracting from the values being communicated is that the resulting process description becomes even more abstract and unrealistic, creating a dangerous conceptual gap between model and implementation. After all, what matters is whether the final system operates correctly, and not just whether the model is correct. It is our belief that, if a proof system has been designed properly, the complexity of proving a system property should be affected by the complexities of

the property and the system's behaviour, and not so much by the complexity of the specification and modelling languages. So, if values are treated properly (for example symbolically) they should not make a proof more complicated unless the proof really *depends* on the values being communicated.

Compositionality The model checking problem is undecidable for infinite state systems in general and μ-calculus formulae. This means that verification for value passing processes is in general not fully automatable. It can be substantially machine assisted, but human guidance cannot be dispensed with completely. A *proof assistant* would reduce the initial verification goal, consisting in our case of a value passing CCS term and a μ-calculus formula, to sub-goals, and would repeat this process, guided by the user, until all sub-goals are evidently true (for example axioms or memorised theorems in the proof system). To be able to guide the derivation process, however, the user has to be able to interpret the intermediate sub-goals; in other words, the sub-goals should be represented in a way which is intuitive and meaningful to the user. A natural solution is to represent sub-goals in the same way as the initial goal, namely, in our case, as pairs consisting of a value passing CCS term and a μ-calculus formula (such pairs are usually called *sequents*). In this case the proof of a sequent could be guided not just by the structure of the formula but also by the structure of the process term. Such proof systems are usually termed *compositional.*[2]

There is also another reason for using compositional proof systems. An important aspect of a design methodology is *modularity.* A design of a compound system is called modular if the system's components have been designed independently by taking into account the requirements for putting them together. One immediate benefit from using a methodology which facilitates modular design is that in the case of repeated or similar components much effort can be saved. This is called *design reuse.* In our context, the important issue is *modular specification and verification*, which requires

[2]This term is also used in a weaker sense, meaning that it handles "compositionally" processes composed in parallel.

systems to be specified in such a way that the correctness of the components of a system imply the correctness of the composite system. Then, verifying a system reduces to verifying the components. Compositional proof systems facilitate modular verification in a natural way by reducing the proof of system properties to proving properties of components.

Another important problem which compositional proof systems aim to solve is the infamous *state explosion problem*: the global state space of a communicating system composed of several components running in parallel is of size roughly the product of the sizes of the state spaces of the constituent processes. This phenomenon makes verification intractable even for relatively small practical systems. Compositionality attacks this problem by reducing the verification of a global property of a system to verifying local properties of its components. Because of the significance of parallel composition to system design, and because of some technical reasons to be explained in later chapters, it is useful to separate the treatment of this operator on processes from the rest. Following Stirling [Sti87], we divide our proof system in two parts, the first of which treats all process combinators except parallel composition, and the second for inferring properties of a process from its parallel components.

Contributions The main contributions of the present thesis are the development of a suitable logic for specifying value passing processes, and the development of a compositional proof system which is sound and complete for a large class of processes. A unifying theme in our research has been the attempt to adapt the so called technique of *tagging* (to be explained in later chapters) to the different parts of our proof system. This technique allows global proof rules to be avoided, and simplifies considerably the machine-assistance of the proof system.

Credits The research presented here is partly joint work with Sergey Berezin from Carnegie Mellon University. More specifically, the second part of our proof system, namely the one dealing with parallel composition, was originally developed by Sergey

in his M.Sc. Thesis [Ber95]. For this part of the proof system, our contribution lies in collaborating with Sergey on modifying this proof system to employ tags, and adopting a suitable semantics for tagged formulae to facilitate an economic proof of soundness and completeness. The study of "negative" tagging presented here has been performed independently. Part of the results have been published in [GBK96, BG97].

Organisation The thesis is organised as follows. The next chapter presents the background necessary to understand and evaluate the contributions of the present work. Chapter 3 introduces a first-order extension of the Modal μ-Calculus as a suitable specification language for sequential CCS processes with value passing, presents a proof system which is compositional in the term structure of the processes and employs a technique known as *tagging* to eliminate the need for global inference rules, and illustrates the use of this proof system on some illuminating examples. The following chapter is dedicated to the correctness of the proof system. Chapter 5 investigates other settings in which tagging may be a suitable choice, notably *negative tagging*. The last chapter summarises the accomplishments of this thesis, draws conclusions about the merits and deficiencies of the chosen approach, and proposes directions for improvement and future research.

Chapter 2

Models, Logics, and Verification

In this chapter we give the background needed to appreciate the results presented later. We first present Labelled Transition Systems (LTS) as a semantic domain for representing the behaviour of communicating systems. We next present Milner's Calculus of Communicating Systems (CCS) [Mil89]. Then, we discuss the Modal μ-Calculus as a process logic. The last section explains the ideas behind model checking as a technique for verifying systems behaviour.

2.1 Labelled Transition Systems

The *semantics*, or meaning, of a process language or a process logic is best given in a well understood semantic domain for representing communicating systems behaviour. As discussed already in the Introduction, the approach of representing the behaviour of a system as a mapping from some set of allowable initial configurations to some set of desirable final configurations is not adequate for describing the ongoing behaviour of communicating systems. Instead, one can consider as a mapping between configurations the result of a single communication. But communicating systems are inherently distributed. This brings about the question how to treat local configurations. A conceptually simple approach is to abstract from these and to interleave local behaviours. Other approaches are also possible, notably the partial order semantics approach advocated by Petri [Pet76].

The interleaving approach described above leads to a simple model of behaviour in which the structure of states is irrelevant: a state is characterised only by the sequences of choices among communications which are offered from this state. The resulting mathematical structure, called *labelled transition system* (LTS), can be defined as a triple

$$(S, T, \{\stackrel{t}{\longrightarrow} \subseteq\ S \times S\ \mid\ t \in T\})$$

consisting of a set S of *states*, a set T of *transition labels*, and a set of *transition relations* for each transition label [Mil89]. We shall use the infix notation $s \stackrel{t}{\longrightarrow} s'$ for $(s, s') \in \stackrel{t}{\longrightarrow}$, and call s' a t-derivative of s.

If the size of S is small, an LTS can be visualised as a graph whose nodes are the states and whose edges are labelled. For example, consider a system which allows the user to depress one of two buttons and then, depending on the button depressed, makes a "beep" sound or a "boop" sound, and stops.[1] If we denote the four events as labels *depe*, *depo*, *beep*, and *boop*, respectively, an LTS describing the above behaviour could be graphically depicted as[2]:

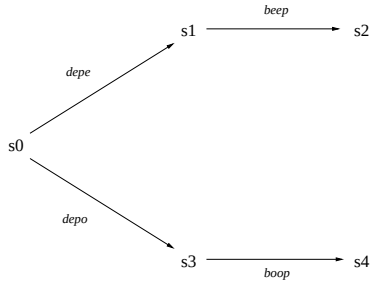


Figure 2.1: A small LTS

Such a semantics is called *branching-time semantics*, since it contains information about what choice of actions is available at any particular state. In the case of communicating systems, we shall interpret labels as "handshake"-type communications, also called *actions*. The set *Act* of actions is formed from a set $\mathcal{A}$ of *names*, the

[1] This is not really a communicating system, since these actions are not really "handshake"-type, but the notion of LTS is more general and does not interpret the nature of the labels.

[2] To be exact, the behaviour of the system is characterized by its initial state, namely by s_0.

corresponding set $\overline{\mathcal{A}}$ of *co-names*, and a pre-defined *silent* action τ. Since actions are handshakes between two agents, each action $l \in \mathcal{A} \cup \overline{\mathcal{A}}$ has its co-action $\overline{l}$; the result of a handshake is the silent action. If the above system is to be interpreted as a communicating system, then in its initial state it offers the user to communicate either through action *depe* or through action *depo*; the user (as a system) has consequently to offer the corresponding co-actions $\overline{depe}$ or $\overline{depo}$ to be able to engage in a communication with the system. A system engaging in the silent action τ means that there is internal communication taking place somewhere between components of the system. The user has no influence on this action, hence its identity is considered irrelevant; even more, its identity is considered as information that one would like to ignore in order to be able to produce manageable descriptions of large behaviours.[3]

A natural question is when to consider two states in an LTS as corresponding to the same behaviour (i.e., as being *behaviourally equivalent*). The usual automata-theoretic notion of equivalence (*trace equivalence*) is often too weak for practical purposes since it is not sensitive to deadlocks: two systems can be trace equivalent so that the first is deadlock-free while the second is not. The main reason for this insensitivity is that traces do not reflect the branching structure of behaviours. Since we found branching time semantics as being appropriate for communicating systems we should rather choose an equivalence notion sensitive to branching. The main guideline should be the question as to what constitutes a legal *experiment*, i.e., what do we consider to be our means of distinguishing between two behaviours? Then, two behaviours should be considered equivalent if and only if they cannot be distinguished by any allowable experiment. In the case of communicating systems it is natural to assume that experiments are sequences of interactions, and that experimenters are communicating systems themselves. If we assume that the experimenter is able to "see" at every state the choice of actions offered by the systems to be compared, then one naturally comes to the notion of *experimental equivalence* introduced by de

[3]This and the following considerations are introduced in [Mil89] at the level of CCS. They are, however, of semantic nature, and are therefore presented here.

Nicola and Hennessy [NH84]. If the experimenter is also allowed to create identical copies of the behaviours in any state, then to perform experiments on the copies, and finally to combine the results, one arrives at the notion of *equivalence with respect to duplicator experiments* [BM92]. Milner prefers an even finer equivalence, namely *bisimulation*, also called *observation congruence*, originally proposed by Park [Par81]. Its strong version, called *strong bisimulation*, is given in [Mil89] as follows:[4]

> P and Q are equivalent iff, for every action α, every α-derivative of P is equivalent to some α-derivative of Q, and conversely.

For practical purposes, strong bisimulation is too strong an equivalence notion because it does not abstract from the unobservable internal communication represented by silent actions taking place in a system. For this reason Milner also introduces the weaker equivalences *weak bisimulation* and observation congruence.

Many other equivalence notions have been proposed in the literature. Choosing a "good" one is particularly important if one has adopted the approach of describing both the specification and the model in the same process notation, and to do verification by showing the two descriptions equivalent. Then, one should choose an equivalence notion which is fine enough to catch important differences between specification and model, and is coarse enough to guarantee that verification would not fail because of unimportant ones. The experimenter is in this case the user of the system; it is hence the capabilities of the user which should also guide the choice of an appropriate equivalence.

2.2 Calculus of Communicating Systems

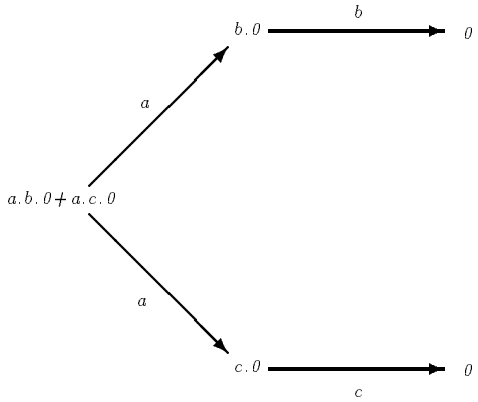
The process language on which we focus our attention is CCS and its value passing extension. CCS is an algebraic language which defines atomic behaviours and opera-

[4]The experiments corresponding to these notions of equivalence can be described nicely in terms of *games* [Sti96].

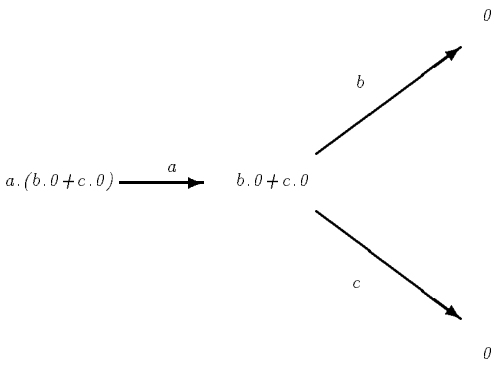
tors for constructing more complicated behaviours from simpler ones. We start with an informal overview of the language and its semantics.

Expressions in the language are termed *processes*. The only atomic process is the *nil* process, denoted $\mathbf{0}$, which is not capable of participating in any action. If a is an action, and P is a process, then we can construct out of them a new process $a.P$ which can initially engage in a only, and behave as P afterwards. So "$a.$" can be understood as a unary operator on processes, called *prefix*. So, process $tick.\mathbf{0}$ can just participate in one *tick* action. The behaviour of $tick.\mathbf{0}$ can be given by an LTS having as states the two processes $tick.\mathbf{0}$ and $\mathbf{0}$, and a single tuple $tick.\mathbf{0} \xrightarrow{tick} \mathbf{0}$.

If P and Q are processes, then $P + Q$ denotes a process which can behave either as P or as Q, depending on the first action chosen. For example, $a.P + b.Q$ offers to the environment a *choice* between participating in a or participating in b. If a is chosen, the process continues to behave as P, otherwise as Q. In the case of $a.b.\mathbf{0} + a.c.\mathbf{0}$ we have *non-deterministic* choice: after choosing a the process decides non-deterministically whether to continue as $b.\mathbf{0}$ or as $c.\mathbf{0}$. This behaviour is better explained by viewing its LTS:



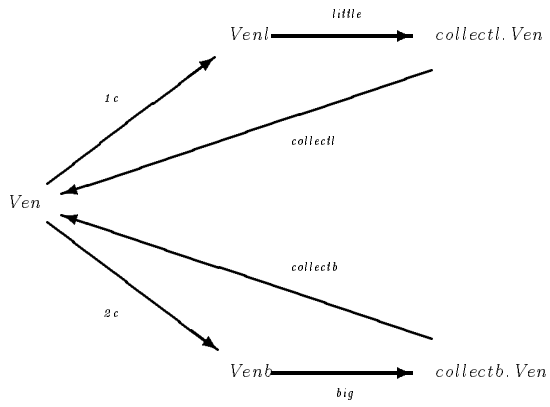
It should be distinguished from the behaviour of $a.(b.\mathbf{0} + c.\mathbf{0})$, where after a the choice between b and c is still to be resolved by the environment:



Using process constants and defining equations, one can give names to specific processes. For example, the notation $P \stackrel{\Delta}{=} a.b.\mathbf{0}$ defines process P as $a.b.\mathbf{0}$. Using recursive definitions one can define processes with ongoing behaviour: e.g., process $Clock \stackrel{\Delta}{=} tick.Clock$ has the ability to engage in an infinite sequence of consecutive *tick* actions. Another example is a simple vending machine, which can accept a one-cent or a two-cent coin, after which a little or a big button may be depressed depending on the coin inserted, and finally a little or a big item may be collected, upon which the vending machine enters its initial state:

$$\begin{aligned} Ven &\stackrel{\Delta}{=} 1c.Venl + 2c.Venb \\ Venl &\stackrel{\Delta}{=} little.collectl.Ven \\ Venb &\stackrel{\Delta}{=} big.collectb.Ven \end{aligned}$$

The LTS of process Ven is as follows:



Given two processes P and Q, process $P|Q$ denotes their *composition*, i.e., a new

process, which is like P and Q taken together, but allows also P and Q to interact. Let $Buffer_1 \stackrel{\Delta}{=} in.\overline{transmit}.Buffer_1$ and $Buffer_2 \stackrel{\Delta}{=} transmit.\overline{out}.Buffer_2$ be two one-element buffer processes. Then $Buffer_1|Buffer_2$ is a process, which can engage in an in action and become $\overline{transmit}.Buffer_1|Buffer_2$, or engage in $transmit$ and become $Buffer_1|\overline{out}.Buffer_2$. Process $\overline{transmit}.Buffer_1|Buffer_2$ can engage in both $transmit$ and $\overline{transmit}$, but can also engage in an internal communication, i.e. perform a τ-action, between $Buffer_1$ and $Buffer_2$ and become $Buffer_1|\overline{out}.Buffer_2$. If we wish to stop the environment from interfering in this internal communication, we can hide $transmit$ and $\overline{transmit}$ using the CCS *restriction* operator: process $(Buffer_1|Buffer_2)\backslash\{transmit\}$ can engage initially in in only, thus becoming $(\overline{transmit}.Buffer_1|Buffer_2)\backslash\{transmit\}$, then in τ only, becoming $(Buffer_1|\overline{out}.Buffer_2)\backslash\{transmit\}$, which can engage in both in and $\overline{out}$. A *renaming* operator is also provided to allow for reuse of already defined processes. For example, both $Buffer_1$ and $Buffer_2$ can be defined via $Buffer \stackrel{\Delta}{=} in.\overline{out}.Buffer$ as follows:

$$\begin{array}{lcl} Buffer_1 & \stackrel{\Delta}{=} & Buffer[\overline{transmit}/\overline{out}] \\ Buffer_2 & \stackrel{\Delta}{=} & Buffer[transmit/in] \end{array}$$

So, a two-element buffer can be defined using two one-element buffers as:

$$TwoBuffer \stackrel{\Delta}{=} (Buffer[\overline{transmit}/\overline{out}]|Buffer[transmit/in])\backslash\{transmit\}$$

Of course, at this high level of abstraction there is no difference between process $Buffer \stackrel{\Delta}{=} in.\overline{out}.Buffer$ and $Clock \stackrel{\Delta}{=} tick.\overline{tock}.Clock$ except for the different names of actions; what is missing in $Buffer$ to be really appreciated as a one-element buffer are the *values* being communicated. The version of CCS providing a notation for communicated values is called Value Passing CCS; in this language we could specify $Buffer$ as follows:

$$Buffer \stackrel{\Delta}{=} in(x).\overline{out}(x).Buffer$$

where x ranges over some pre-defined domain D of values. If D is the set of natural numbers, then *Buffer* can engage initially in any of the actions $in(1)$, $in(2)$, $in(3)$, etc., becoming respectively $\overline{out}(1).Buffer$, $\overline{out}(2).Buffer$, $\overline{out}(3).Buffer$, etc. Thus, input actions have binding power in the Value Passing CCS.

The original definition of CCS has instead of the binary choice operator + the operator $\sum_{i \in I}$ called *summation* (also having binding power), where I is an indexing set. In our value passing version of CCS we choose I to coincide with the domain D of values, and use values instead of indices.[5] For example, the process $\Sigma x\, \overline{out}(x).\mathbf{0}$ is ready to put out any value from the respective domain.

A more interesting example showing the higher modelling power of the value passing extension would be a simple teller machine which accepts and offers cash without giving credit:

$$\begin{aligned} Teller(balance) &\stackrel{\Delta}{=} Deposit(balance) + Withdrawal(balance) \\ Deposit(balance) &\stackrel{\Delta}{=} deposit(amount).Teller(balance + amount) \\ Withdrawal(balance) &\stackrel{\Delta}{=} \Sigma amount \\ &\quad \textbf{if } 0 < amount \leq balance \\ &\quad \textbf{then } \overline{withdraw}(amount).Teller(balance - amount) \end{aligned}$$

After this informal introduction to CCS,[6] we are ready to present the formal syntax and semantics of the language. We assume a set $\mathcal{A}$ of *names*, ranged over by a, each name having a non-negative arity. Let $\mathcal{L}$ denote the set $\mathcal{A} \cup \overline{\mathcal{A}}$ of *labels*, ranged over by l, and let $\overline{\overline{a}}$ denote a. We also assume a set D of *values*, value expressions e and Boolean expressions b built from variables $x, y, z, \ldots$ (possibly indexed), value constants d and arbitrary operator symbols defined in the domain. We use tt and ff to denote the usual Boolean constants "true" and "false."

[5] This does not decrease the expressiveness of the language, provided we drop the convention that input actions have binding power.

[6] We shall henceforth always understand under CCS the value passing version of the calculus.

Agent expressions E over $\mathcal{A}$ and D are generated by the grammar:[7]

$$E ::= \mathbf{0} \mid \pi.E \mid E + E \mid \Sigma\vec{x}\, E \mid E|E \mid E\backslash U \mid E\{\Xi\} \mid \textbf{if } b \textbf{ then } E \mid A(\vec{e})$$
$$\pi ::= a(\vec{x}) \mid \overline{a}(\vec{e}) \mid \tau$$

Here A are *agent constants*, each having a defining equation

$$A(\vec{x}) \triangleq E$$

where the right-hand side E may contain no free variables except the ones in $\vec{x}$. $U \subseteq \mathcal{L}$ are *restriction* sets, and $\Xi : \mathcal{L} \to \mathcal{L}$ are *relabelling* functions satisfying $\Xi(\overline{l}) = \overline{\Xi(l)}$ and $\Xi(\tau) = \tau$. Input actions and infinite summation have binding power. Closed agent expressions (i.e., expressions with no free variables) are termed *processes* and are ranged over by $P, Q, \ldots$

The semantics of a CCS process[8] is given in terms of an LTS whose states are processes, i.e., closed agent expressions, and whose transition labels are actions, i.e., either τ or of the form $l(\vec{d})$, where $l \in \mathcal{L}$ and $\vec{d} \in \vec{D}$. The set of actions is denoted Act and is ranged over by α. What remains to be defined is the set of transition relations of the LTS.[9] A denotational approach to giving semantics for a process would define the LTS of the process through the LTSs of the sub-expressions of the process; for example, it would define the LTS for $a.P$ through the LTS for P, possibly by extending the latter with the state $a.P$ and with the tuple $a.P \xrightarrow{a} P$. Milner preferred to give a *transitional semantics* to his calculus by giving *transition rules* for inferring the transitions of a composite process from the transitions of its component processes. Figure 2.2 presents such a set of transition rules. We use the usual notation for term substitution, and use $[\![\vec{e}\,]\!]$ and $[\![b]\!]$ to denote the values of $\vec{e}$ and b, respectively.

An LTS of the type described above is called *transition closed* if whenever the hypotheses of a rule are satisfied (i.e., there are processes in the set of states of the

[7] We use vectors of variables, expressions etc. when the arity is of no particular relevance.
[8] We shall not give a semantics for open agent expressions here.
[9] Since processes are states themselves in such a LTS, there is no need to identify an initial state.

$$\mathrm{R}(\tau)\ \frac{}{\tau.P \xrightarrow{\tau} P} \qquad \mathrm{R}(in)\ \frac{}{a(\vec{x}).E \xrightarrow{a(\vec{d})} E[\vec{d}/\vec{x}]} \qquad \mathrm{R}(out)\ \frac{}{\overline{a}(\vec{e}).P \xrightarrow{\overline{a}([\![\vec{e}]\!])} P}$$

$$\mathrm{R}(+l)\ \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 + P_2 \xrightarrow{\alpha} P_1'} \qquad \mathrm{R}(+r)\ \frac{P_2 \xrightarrow{\alpha} P_2'}{P_1 + P_2 \xrightarrow{\alpha} P_2'}$$

$$\mathrm{R}(\Sigma)\ \frac{E[\vec{d}/\vec{x}] \xrightarrow{\alpha} P}{\Sigma\vec{x}\ E \xrightarrow{\alpha} P} \qquad \mathrm{R}(|)\ \frac{P_1 \xrightarrow{l(\vec{d})} P_1' \qquad P_2 \xrightarrow{\overline{l}(\vec{d})} P_2'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2'}$$

$$\mathrm{R}(|l)\ \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1|P_2 \xrightarrow{\alpha} P_1'|P_2} \qquad \mathrm{R}(|r)\ \frac{P_2 \xrightarrow{\alpha} P_2'}{P_1|P_2 \xrightarrow{\alpha} P_1|P_2'}$$

$$\mathrm{R}(\backslash)\ \frac{P \xrightarrow{l(\vec{d})} P'}{P\backslash U \xrightarrow{l(\vec{d})} P'\backslash U}\ l,\overline{l} \notin U \qquad \mathrm{R}(\Xi)\ \frac{P \xrightarrow{l(\vec{d})} P'}{P\{\Xi\} \xrightarrow{l'(\vec{d})} P'\{\Xi\}}\ \Xi(l) = l'$$

$$\mathrm{R}(\textbf{if})\ \frac{P \xrightarrow{\alpha} P'}{\textbf{if}\ b\ \textbf{then}\ P \xrightarrow{\alpha} P'}\ [\![b]\!] = \mathtt{tt} \qquad \mathrm{R}(\triangleq)\ \frac{E[\vec{d}/\vec{x}] \xrightarrow{\alpha} P}{A(\vec{d}\,) \xrightarrow{\alpha} P}\ A(\vec{x}) \triangleq E$$

Figure 2.2: Transition rules for processes.

LTS which are in the corresponding transition relations), then the conclusion also holds (i.e., the two processes occurring in the conclusion are in the set of states of the LTS, and they are in the respective transition relation). Given a set of processes, we are interested in the least transition closed LTS containing these. This LTS has the property that two processes are in a given transition relation if and only if this can be derived using the rules. It is by such LTSs that we give meaning to processes. This style of providing a semantics for processes is not very intuitive, since it is not always obvious, as it is for example in the case with Petri nets, what transitions are *enabled* in a CCS process involving several parallel components. It is, however, technically very elegant and economic, which makes it preferable for theoretical investigations.

Let us see, for example, what behaviour these rules specify for process $Buffer \triangleq in(x).\overline{out}(x).Buffer$. One would expect to be able to establish $Buffer \xrightarrow{in(d)} P_d$ and $P_d \xrightarrow{\overline{out}(d)} Buffer$ for any $d \in D$ and appropriate processes P_d. This can be achieved as follows. Axiom rule R(in) implies that $in(x).\overline{out}(x).Buffer \xrightarrow{in(d)} \overline{out}(d).Buffer$, and axiom rule R($out$) implies that $\overline{out}(d).Buffer \xrightarrow{\overline{out}(d)} Buffer$. From the first of these follows by rule R($\triangleq$) that $Buffer \xrightarrow{in(d)} \overline{out}(d).Buffer$, and taking P_d to be $\overline{out}(d).Buffer$ establishes the expected transition relations.

2.3 Modal Logics and μ-Calculi

When specifying a communicating system, we are usually interested in the observable properties only, i.e., what sequences of choices of actions we observe when interacting with the system. Since we are talking about *capabilities* for interaction it is natural to use Modal Logics for describing such properties.

A simple modal logic for describing *local* capabilities for interaction is Hennessy-Milner Logic (HML) [HM80]. The formulae of HML are easy to define:

- the propositional constants tt and ff are formulae;
- if Φ and Ψ are formulae, then so are $\Phi \vee \Psi$ and $\Phi \wedge \Psi$; and
- if Φ is a formula and α is an action, then $\langle\alpha\rangle\,\Phi$ and $[\alpha]\,\Phi$ are formulae.

The meaning of these formulae can be given in terms of LTS by specifying when a process P of the LTS satisfies a formula Φ. This is denoted $P \models \Phi$, and can be defined as follows:

$P \models \mathsf{t}t$ always holds, i.e. tt means "true";
$P \models \mathsf{ff}$ never holds, i.e. ff means "false";
$P \models \Phi \vee \Psi$ iff $P \models \Phi$ or $P \models \Psi$;
$P \models \Phi \wedge \Psi$ iff $P \models \Phi$ and $P \models \Psi$;
$P \models \langle\alpha\rangle\,\Phi$ iff there is an α-derivative P' of P so that $P' \models \Phi$.
$P \models [\alpha]\,\Phi$ iff for every α-derivative P' of P, $P' \models \Phi$;

We shall refer to this style of giving semantics to formulae as *local*, or intensional, semantics.

For example, $P \models \langle a\rangle\,\mathsf{tt}$ means simply that P can engage in an a action, while $P \models [a]\,\mathsf{ff}$ means that it can not. For the vending machine process Ven described in the previous subsection, $Ven \models \langle 2c\rangle\,\langle big\rangle\,\langle collectb\rangle\,\mathsf{tt} \wedge [1c]\,[big]\,\mathsf{ff}$, i.e., Ven offers a

big item to be selected after 2c have been inserted, but not if only 1c has been inserted. The lack of negation in the logic is not incidental; it can easily be shown (by referring to deMorgan-type equivalences) that every HML formula involving negation (if we allow negation in the logic) has a negation-free equivalent.

HML is in fact a poly-modal logic: instead of having just the two modalities [ ] and $\langle\rangle$ as is the case with Classical Modal Logic where there is just a single "accessibility" (i.e., transition) relation, HML has a whole family of such modalities, namely a "box" modality and a "diamond" modality for each action corresponding to the respective transition relations.

The properties that are expressible in HML are *local*, or next-step, properties in the sense that they allow only the immediate capabilities for interaction (and internal action) of a process to be described. Properties of more general temporal character like "always Φ" or "eventually Φ" are not expressible in this logic. This concerns also silent actions. For example, we can express in HML a property of the form "process P can perform a silent action and offer a afterwards"; however, when specifying the interaction behaviour of a system we shouldn't discriminate between one or more silent actions in a row, but should rather be only able to speak of *internal activity* in general, like "process P can engage in some internal activity and offer a afterwards". Additional modalities have been suggested in the literature for specifying such properties [Sti96].

If we are to specify properties like "always Φ" or "eventually Φ" we enter the realm of Temporal Logics. Computation Tree Logic (CTL), proposed by Clarke and Emerson [CE81], is just one example for such a logic. It has explicit constructs for "always", "eventually" and "until," as well as path quantifiers. A more economic approach, leading at the same time to a more powerful logic, is to use HML as a basis and to add *recursion*; the price to be paid is the intuitiveness of the resulting logic. Consider the recursive equation $Z \equiv \langle a \rangle Z$, where Z is a propositional variable. A property would be a solution to this equation if every process satisfying it has

an a-derivative satisfying the same property. Fortunately every such equation has a solution; unfortunately however it is not necessarily unique. ff is one solution to the above equation, since $\mathsf{ff} \equiv \langle a \rangle \mathsf{ff}$, but this solution is of little interest. Another property satisfying the equation would be the capability of engaging in an infinite sequence of a actions. There can be other solutions as well; the mentioned two properties however (the second of which is not expressible in HML) are the *least* and the *greatest* ones w.r.t. logic implication, and are hence uniquely characterizable. In fact, any equation of the sort $Z \equiv \Phi$, where Φ is allowed to include occurrences of the propositional variable Z, has a least and a greatest solution denoted $\mu Z.\Phi$ and $\nu Z.\Phi$, respectively. This is an immediate consequence of Tarski's fixedpoint[10] theorem for complete lattices [Tar55].

The logic resulting from adding least and greatest fixpoint formulae to HML is called the Modal μ-Calculus, which was introduced by Kozen [Koz83], but was developed earlier by Park [Par69] in a more general relational setting. Stirling [Sti92] suggests a slight generalization of this logic by allowing sets K of actions to appear in the "box" and "diamond" modalities, and by using the notation "$-K$" to abbreviate "$Act - K$" and "$-$" to abbreviate "$Act - \{\}$" (i.e. Act itself). The resulting logic allows many other logics, like Dynamic Logic and CTL, to be conveniently encoded (see, e.g., [Dam94]).

Let us consider some properties which are often important in practical applications, and give their formalisation in (Stirling's extension of) the Modal μ-Calculus:

- A communicating system is called *deadlock free* if regardless of how we interact with it there is always a communication, possibly an internal one, in which the system can engage. Deadlock freedom can be expressed by the formula $\nu Z.\, \langle - \rangle \mathsf{tt} \wedge [-]\, Z$, saying that some action is enabled, and whatever action is taken the same property holds again. In other words, there is always some action which is enabled. Note that the least fixpoint wouldn't be of any use

[10] The term *fixedpoint* or *fixpoint* of a mapping f refers to any solution of the equation $f(x) = x$.

here since it is equivalent to "false".

- Action a is always enabled: $\nu Z.\, \langle a \rangle \mathtt{tt} \wedge [-]\, Z$. In general, $\nu Z.\Phi \wedge [-]\, Z$ formalises the *safety*, or *invariant*, property "always Φ".

- A *livelock* is the capability of engaging in internal chatter, i.e. in an infinite sequence of τ actions. Livelock freedom can be formalised as $\mu Z.\, [\tau]\, Z$.

- The property that action a can potentially become enabled is formalisable as $\mu Z.\, \langle a \rangle \mathtt{tt} \vee \langle - \rangle\, Z$ (i.e. either a is enabled right away, or otherwise we can do something so that the same property holds afterwards, and so on, but not forever).

- Action a is eventually to be chosen: $\mu Z.\, \langle - \rangle \mathtt{tt} \wedge [-a]\, Z$ (i.e. if we interact with the system but avoid choosing a, then sooner or later we will arrive at a point where only a is offered).

- There is a sequence of interactions so that a is enabled infinitely often along this sequence: $\nu Z.\mu Y.\, \langle a \rangle\, Z \vee \langle -a \rangle\, Y$. This property is not expressible in CTL.

As mentioned above, the formulae of this logic are often difficult to interpret. We gave little justification as to why the above formulae express the mentioned properties. Unlike CTL which formalises notions of time about which humans have a strong intuition, the Modal μ-Calculus is a typical example of a logic language which is the product of theoretical investigations in a search for expressive power, economy, and elegance, rather than intuitiveness. Therefore, using this formalism requires some training and thorough understanding of its formal semantics, which we are about to explain.

The formulae of the Modal μ-Calculus can be defined as follows:

- Propositional variables are formulae;

- if Φ and Ψ are formulae, then so are $\Phi \vee \Psi$ and $\Phi \wedge \Psi$;

- if Φ is a formula and α is an action, then $\langle\alpha\rangle\,\Phi$ and $[\alpha]\,\Phi$ are formulae; and
- if Φ is a formula and Z is a propositional variable, then $\mu Z.\Phi$ and $\nu Z.\Phi$ are also formulae.

The logic constants ff and tt are definable as $\mu Z.Z$ and $\nu Z.Z$, respectively.

It is not easy to give a local semantics for fixpoint formulae; this we do later by employing approximant formulae. It is far more convenient to present the semantics of Modal μ-Calculus formulae extensionally, i.e., by defining for every formula the set of processes satisfying it. We call this set the *denotation* of the formula. The denotation has to be defined relative to a LTS and a *valuation* $\mathcal{V}$ mapping subsets of the set of states of the LTS to propositional variables, since Φ or some sub-formulae of it can contain free occurrences of propositional variables. For a fixed LTS, we define the denotation $\|\Phi\|_{\mathcal{V}}$ of a Modal μ-Calculus formula Φ inductively as follows:

$$\begin{array}{rcl}
\|Z\|_{\mathcal{V}} & \stackrel{\Delta}{=} & \mathcal{V}(Z) \\
\|\Phi \vee \Psi\|_{\mathcal{V}} & \stackrel{\Delta}{=} & \|\Phi\|_{\mathcal{V}} \cup \|\Psi\|_{\mathcal{V}} \\
\|\Phi \wedge \Psi\|_{\mathcal{V}} & \stackrel{\Delta}{=} & \|\Phi\|_{\mathcal{V}} \cap \|\Psi\|_{\mathcal{V}} \\
\|\langle\alpha\rangle\,\Phi\|_{\mathcal{V}} & \stackrel{\Delta}{=} & \|\langle\alpha\rangle\|_{\mathcal{V}}\,(\|\Phi\|_{\mathcal{V}}) \\
\|[\alpha]\,\Phi\|_{\mathcal{V}} & \stackrel{\Delta}{=} & \|[\alpha]\|_{\mathcal{V}}\,(\|\Phi\|_{\mathcal{V}}) \\
\|\mu Z.\Phi\|_{\mathcal{V}} & \stackrel{\Delta}{=} & \mu X.\,\|\Phi\|_{\mathcal{V}[X/Z]} \\
\|\nu Z.\Phi\|_{\mathcal{V}} & \stackrel{\Delta}{=} & \nu X.\,\|\Phi\|_{\mathcal{V}[X/Z]}
\end{array}$$

where $\mu X.f(X)$ and $\nu X.f(X)$ denote the least and the greatest fixpoints of a mapping f, and where the following state transformers are used:

$$\begin{array}{rcl}
\|\langle\alpha\rangle\|_{\mathcal{V}} & \stackrel{\Delta}{=} & \lambda X.\{P \mid \exists P' \in X.\, P \stackrel{\alpha}{\longrightarrow} P'\} \\
\|[\alpha]\|_{\mathcal{V}} & \stackrel{\Delta}{=} & \lambda X.\{P \mid \forall P'.P \stackrel{\alpha}{\longrightarrow} P' \; \mathit{implies}\; P' \in X\}
\end{array}$$

The valuation $\mathcal{V}[X/Z]$ is as $\mathcal{V}$ but mapping the set X to Z. We can define satisfaction (now relative to a valuation) through denotation, namely:

$$P \models_{\mathcal{V}} \Phi \quad \stackrel{\Delta}{\equiv} \quad P \in \|\Phi\|_{\mathcal{V}}$$

As mentioned above, the existence of least and greatest fixpoints is guaranteed by Tarski's fixedpoint theorem for complete lattices [Tar55]. This theorem says that every monotone mapping over a complete lattice has a least and a greatest fixpoint. The lattice we have in our case is formed by the set S of states (processes) together with set inclusion $\subseteq$. It is simple to show that in the absence of negation in our logic the transformers $\lambda X.\, \|\Phi\|_{\mathcal{V}[X/Z]}$ are all monotone w.r.t. set inclusion (i.e., $X_1 \subseteq X_2$ implies $\|\Phi\|^X_{\mathcal{V}[X_1/Z]} \subseteq \|\Phi\|^X_{\mathcal{V}[X_2/Z]}$).

There are two main ways of characterising μf and νf for monotone mappings on complete lattices. One one hand, they can be presented as:

$$\begin{array}{rcl} \mu f & = & \bigcap \{X \mid X \supseteq f(X)\} \\ \nu f & = & \bigcup \{X \mid X \subseteq f(X)\} \end{array}$$

The other approach is to refer to *fixpoint approximants*. This characterisation often leads to a better understanding of the formulae of the Modal μ-Calculus. Let Ord denote the class of all *ordinals*, and let γ and λ range over ordinals and limit ordinals, respectively. Fixpoint approximants are defined inductively as follows:

$$\begin{array}{rcl@{\qquad}rcl} \mu^0 f & \stackrel{\Delta}{\equiv} & \{\} & \nu^0 f & \stackrel{\Delta}{\equiv} & S \\ \mu^{\gamma+1} f & \stackrel{\Delta}{\equiv} & f(\mu^\gamma f) & \nu^{\gamma+1} f & \stackrel{\Delta}{\equiv} & f(\nu^\gamma f) \\ \mu^\lambda f & \stackrel{\Delta}{\equiv} & \bigcup_{\gamma<\lambda} \mu^\gamma f & \nu^\lambda f & \stackrel{\Delta}{\equiv} & \bigcap_{\gamma<\lambda} \nu^\gamma f \end{array}$$

It can be shown that the following equations hold:

$$\begin{array}{rcl} \mu f & = & \bigcup_{\gamma \in Ord} \mu^\gamma f \\ \nu f & = & \bigcap_{\gamma \in Ord} \nu^\gamma f \end{array}$$

The term *approximant* is justified by the fact that (since f is monotone):

$$\mu^0 f \subseteq \mu^1 f \subseteq \mu^2 f \subseteq \ldots \subseteq \mu f$$

$$\nu^0 f \supseteq \nu^1 f \supseteq \nu^2 f \supseteq \ldots \supseteq \nu f$$

Furthermore, μf and νf are approximants themselves; this means that the above sequences stabilise after some ordinal, called the *closure* ordinal of μf and νf, respectively, and become equal to μf and νf. The closure ordinal is the first ordinal κ such that $\mu^\kappa f = \mu^{\kappa+1} f$ resp. $\nu^\kappa f = \nu^{\kappa+1} f.$, and its cardinality is bound by the cardinality of the carrier set of the complete lattice.

In the context of the Modal μ-Calculus we can define approximant formulae (using infinite conjunctions and disjunctions) as follows:

$$\begin{array}{rcl@{\qquad}rcl} \mu^0 Z.\Phi & \triangleq & \mathsf{ff} & \nu^0 Z.\Phi & \triangleq & \mathsf{ff} \\ \mu^{\gamma+1} Z.\Phi & \triangleq & \Phi[\mu^\gamma Z.\Phi/Z] & \nu^{\gamma+1} Z.\Phi & \triangleq & \Phi[\nu^\gamma Z.\Phi/Z] \\ \mu^\lambda Z.\Phi & \triangleq & \bigvee_{\gamma<\lambda} \mu^\gamma Z.\Phi & \nu^\lambda Z.\Phi & \triangleq & \bigwedge_{\gamma<\lambda} \nu^\gamma Z.\Phi \end{array}$$

and we obtain the following characterisation of satisfaction[11]:

$$\begin{array}{lll} P \models_\mathcal{V} \mu Z.\Phi & \text{iff} & P \models_\mathcal{V} \mu^\gamma Z.\Phi \quad \text{for some } \gamma. \\ P \models_\mathcal{V} \nu Z.\Phi & \text{iff} & P \models_\mathcal{V} \nu^\gamma Z.\Phi \quad \text{for all } \gamma. \end{array}$$

As a consequence, least fixpoint formulae are suitable for expressing *liveness* (i.e. eventuality) properties, while greatest fixpoint formulae are suitable for expressing *safety* (i.e. invariant) properties. The more complicated *reactivity* properties[12] usually necessary for specifying communicating systems require nesting (alternation) of fixpoints of different kind.

Let us now see how this definition helps in understanding Modal μ-Calculus formulae. Let us consider the formula $\mu Z.[a]Z$. Its first few fixpoint approximants are:

[11] These two clauses, together with the ones we gave above for HML formulae, can be considered as giving a local semantics for the Modal μ-Calculus.

[12] For a classification of program properties see for example [MP92].

$$\begin{aligned}
\mu^0 Z.\,[a]\,Z &\equiv \mathsf{ff} \\
\mu^1 Z.\,[a]\,Z &\equiv [a]\,\mathsf{ff} \\
\mu^2 Z.\,[a]\,Z &\equiv [a]\,[a]\,\mathsf{ff} \\
\mu^3 Z.\,[a]\,Z &\equiv [a]\,[a]\,[a]\,\mathsf{ff} \\
&\vdots
\end{aligned}$$

Process $a.a.\mathbf{0}$ satisfies $\mu^3 Z.\,[a]\,Z$ and hence also $\mu Z.\,[a]\,Z$. Obviously, every process that can engage only in finitely many consecutive a actions satisfies the formula. What is less obvious is that the opposite holds as well: if a process satisfies one of the approximants, then all its a-derivatives satisfy smaller approximants. Since the ordinals are well-founded (i.e. all strictly decreasing sequences of ordinal numbers are of finite length) this amounts to saying that a process satisfies $\mu Z.\,[a]\,Z$ if and only if it cannot engage in infinitely many consecutive a actions.

2.4 Model Checking

The idea of *model checking* was pioneered by Clarke and Emerson in [CE81], where it is described as a technique for automatic verification of finite state reactive systems specified in CTL. The term is now used in a wider sense, and refers here to determining satisfaction $P \models \Phi$ between a model P and a property Φ.

The characterisation of fixpoints using fixpoint approximants presented in the previous section suggests a straightforward procedure for computing the least and greatest fixpoints of a monotone mapping f on a complete lattice, which is guaranteed to terminate when the state space is finite: starting with the empty set (resp. the set of all states), compute its image under f, then the image of the image, and so on, until a set is found which is equal to the set obtained at the previous step, and stop. The set obtained in this way is the least (resp. the greatest) fixpoint of f. Of course, this procedure can be quite inefficient, and, if the state space is infinite, not even effective, but it serves as a systematic basis for automatic verification of finite state systems in the Modal μ-Calculus, since determining $P \models \Phi$ can be understood as computing the denotation $\|\Phi\|$ of Φ and checking whether P is in this set.

Techniques as the above, where the whole denotation of a formula is computed, are termed *global*. More efficient (in average) are *local* techniques which consider only those states in the state space necessary for determining the required satisfaction. In the worst case all states have to be investigated anyway, but in many cases it is possible to obtain the needed result even when the whole state space is infinite and global model checking is not applicable. Local model checking was first suggested by Larsen [Lar88] for a sub-logic of the Modal μ-Calculus, but the term itself was proposed only later by Stirling and Walker [SW91], who deal with the full calculus.

The following approach can be used as a basis for local model checking in the Modal μ-Calculus. The goal $P \models \Phi$ to be checked is successively reduced to sub-goals until all resulting subgoals can be resolved trivially. This reduction can be performed for all non-fixpoint formulae (i.e. not of shape $\mu Z.\Phi$ or $\nu Z.\Phi$) following the local semantic clauses we gave for HML formulae. For example, checking $P \models \Phi \vee \Psi$ is reduced to checking $P \models \Phi$, and if this fails, then $P \models \Psi$ is checked. A goal of the form $P \models \mathsf{tt}$ can be resolved successfully.

The difficulty of local model checking lies in reducing fixpoint formulae. If the state space is finite, and if we know the closure ordinal of the respective formula (or at least some upper bound κ on the number of states, which would also provide an upper bound for the closure ordinal), then a naïve approach would be to use approximants by referring to the following local semantic clauses:

$$
\begin{array}{lll}
P \models \mu Z.\Phi & \text{iff} & P \models \mu^{\kappa} Z.\Phi \\
P \models \mu^{0} Z.\Phi & \text{false} & \\
P \models \mu^{\gamma+1} Z.\Phi & \text{iff} & P \models \Phi[\mu^{\gamma} Z.\Phi / Z]
\end{array}
$$

$$
\begin{array}{lll}
P \models \nu Z.\Phi & \text{iff} & P \models \nu^{\kappa} Z.\Phi \\
P \models \nu^{0} Z.\Phi & \text{true} & \\
P \models \nu^{\gamma+1} Z.\Phi & \text{iff} & P \models \Phi[\nu^{\gamma} Z.\Phi / Z]
\end{array}
$$

i.e. fixpoint formulae are *unfolded*, but at most κ times, and if the 0-th approximant is reached, the sub-goal is discharged successfully in the case of greatest fixpoints and

unsuccessfully otherwise.

Termination and correctness of this procedure follow immediately from the semantic clauses. The drawbacks of this naïve approach, however, are obvious: it works for finite state systems only (we would not be able to handle limit ordinals in the same way), and it is quite inefficient since one and the same state might be investigated for the same property more than once, which is certainly redundant.

A more sophisticated approach would be to use the approximants implicitly, by performing simple unfolding of the fixpoint formulae, justified by:

$$P \models \mu Z.\Phi \quad \text{iff} \quad P \models \Phi[\mu Z.\Phi/Z]$$
$$P \models \nu Z.\Phi \quad \text{iff} \quad P \models \Phi[\nu Z.\Phi/Z]$$

but keeping track (for each fixpoint formula separately) of the states at which we unfold the formulae. Finding a "loop" establishes a transitive dependence on the 0-th approximant, and the sub-goal can be discharged accordingly. Termination is still guaranteed for finite state spaces only, but now there is at least the possibility of successfully checking a local property even in an infinite state system. To keep track of the states visited, one can *tag* the fixpoint formulae, an idea first suggested by Winskel [Win91].

Let us illustrate the above ideas on a small example. Consider process $Clock$ defined by $Clock \stackrel{\Delta}{=} tick.Clock$, and let us prove that it has the property of being able to engage in infinitely many consecutive $tick$ actions, which is formalisable as $\nu Z.\langle tick\rangle Z$. Our initial goal is then to show that:

$$Clock \models \nu Z.\langle tick\rangle Z$$

We now unfold the formula, and tag the fixpoint just unfolded (with the singleton set $\{Clock\}$) to indicate that the formula has already been unfolded once at state $Clock$:

$$Clock \models \langle tick \rangle \nu Z\{Clock\}.\, \langle tick \rangle\, Z$$

According to the local semantic clause for the "diamond" modality, the above goal is equivalent to $P \models \nu Z\{Clock\}.\, \langle tick \rangle\, Z$ for some $tick$-derivative P of $Clock$. But there is only one such derivative, namely $Clock$ itself, so our new sub-goal can only be:

$$Clock \models \nu Z\{Clock\}.\, \langle tick \rangle\, Z$$

Here the tag tells us (without having to trace back our proof) that $\nu Z.\, \langle tick \rangle\, Z$ has already been unfolded at $Clock$. As discussed above, this establishes a direct dependence on the 0-th approximant of the formula, which for all greatest fixpoints is just "true", and the sub-goal can be trivially discharged as successful. Since there are no other sub-goals to be dealt with, this establishes also the truth of $Clock \models \nu Z.\, \langle tick \rangle\, Z$.

The above ideas are central, in one or another way, to most approaches to model checking finite state systems in the Modal μ-Calculus [Cle90, SW91, Win91, ASW94]. In our Thesis we deal predominantly with systems which are inherently infinite state due to the infinite domains of communicated values. These can be treated as above, but with no guarantee for termination. Since the model checking problem becomes undecidable in this case, and human intervention in the proof process unavoidable, presenting proofs in a way intuitive to the human becomes of crucial importance. On one hand, to limit the number of sub-goals simultaneously generated when treating some goal, one has to be able to treat sets of processes, for example by allowing goals of type $\mathcal{P} \models \Phi$, where $\mathcal{P}$ denotes a (possibly infinite) set of processes, and where the meaning of such goals is simply that all processes in $\mathcal{P}$ satisfy Φ. On the other hand, to limit the length of the proof, one has to introduce inference rules like *widening*, where a goal is reduced to a sub-goal having the same formula as the goal but a larger

set of processes, as well as more sophisticated rules for dealing with least fixpoints[13]. In this way it becomes possible to produce finite proofs even for infinite state systems, and these proofs can be nicely presented as *proof trees* [BS92, Bra92, And93, Sti96].

All the above considerations suggest that model checking in the Modal μ-Calculus for infinite state systems can be done conveniently in a proof-theoretic style employing *proof systems*. Proof systems consist of a set of *axioms* and a set of *inference rules*[14] for deriving *sequents* of the form $P \vdash \Phi$, the syntactical counterpart to $P \models \Phi$. Such a sequent is called *valid* if $P \models \Phi$ holds, and it is called *derivable* in a proof system if it can be constructed starting from the axioms and using the proof rules of this proof system, i.e. if there is a *proof* for this sequent. A proof system is called *sound* if derivability of $P \vdash \Phi$ implies its validity; if the reverse holds, the proof system is termed *complete*. Together, soundness and completeness guarantee that what we can derive really holds and vice versa. Notice that completeness only guarantees the existence of a proof, and does not say there is an effective procedure for constructing proofs for arbitrary sequents; if this is the case, the system is called *decidable*.

Proofs can be viewed nicely as finite proof trees (tableaux) with axiom leaves. Since we usually start with the goal to be proven, and then reduce this goal until axioms are obtained, it is convenient to present both the rules and the proofs in a goal directed fashion (i.e., as reversed trees).

In our example proof above we implicitly referred to the transition semantics of CCS when applying the rules. We had to do so, because the process was defined in CCS, but the semantics of the formula is given in terms of LTS. If we are to fully guarantee the correctness of our proof, we should refer to an external proof system for inferring semantic sets like the set of all α-derivatives of a CCS process. The presentation of a "complete" proof has to mix these two types of inferences. A more convenient and intuitive approach, in our opinion, is to require the proof system for

[13]The problem of dealing with least fixpoints in finitely many steps requires inductive reasoning to be used. This is the most complicated aspect of applying these ideas in practice; therefore, the greatest care should be taken to facilitate easy and intuitive application of induction.

[14]Axioms can also be represented as proof rules, but with an empty set of premises.

sequents of type $P \vdash \Phi$, where P is a CCS term, to refer directly to the structure of P to determine the resulting sub-goals. Such proof systems are called *compositional* and inevitably have a larger set of inference rules since all combinations of possible forms of P and Φ have to be covered by such rules.

There are at least two other important reasons for using compositional proof systems. The first is that they greatly facilitate reuse of proofs, which is part of *design reuse*. For example, suppose we have constructed a proof for $P + Q \vdash \Phi$ by reducing it to $P \vdash \Phi_1$ and $Q \vdash \Phi_2$. If for some reason the specification of Q had to be changed without changing the one for $P + Q$, then we would only have to redo the proof for $Q \vdash \Phi_2$, and not the whole proof.

The second and more important problem at the solving of which compositional proof systems aim is the infamous *state explosion problem*: given a communicating system composed of several processes running in parallel, its global state space would be of size roughly the product of the sizes of the constituent processes. This phenomenon makes verification intractable even for relatively small practical systems. One way to avoid this exponential blowup of states is not to construct the global state space at all, but rather to represent it in some economic form suitable for manipulation. A breakthrough in the magnitude of the number of states that can be model checked automatically in reasonable time was achieved through the introduction of *symbolic model checking* [McM92], where the global state graph is represented as a Binary Decision Diagram [Bry86]. This technique, however, has proven successful in hardware verification mainly, due to the higher architectural regularity of hardware systems compared with software systems.

Since the only operator of CCS causing state explosion is parallel composition, reducing a goal of type $P|Q \vdash \Phi$ to sub-goals $P \vdash \Phi_1$ and $Q \vdash \Phi_2$ would essentially solve the problem, provided of course that the size of these formulae is comparable. Unfortunately, and not incidentally, it is exactly the parallel composition operator of CCS which is quite difficult to be treated in such a purely compositional way. The

core of the problem is that finding algorithmically appropriate formulae Φ_1 and Φ_2 in the above sequents is, in the worst case, as expensive a problem as checking the global state space. In practice, however, they can often be guessed if one understands the system well. The following approach, suggested by Stirling [Sti87], seems particularly attractive: a separate proof system is introduced for proving sequents of the form $\Phi_1, \Phi_2 \vdash \Phi$ meaning that *any* two processes satisfying Φ_1 and Φ_2, when put in parallel, satisfy Φ as a composite process. We can then reduce $P|Q \vdash \Phi$ to proving $P \vdash \Phi_1$ and $Q \vdash \Phi_2$ and $\Phi_1, \Phi_2 \vdash \Phi$ for suitable formulae Φ_1 and Φ_2. The state explosion problem can thus be avoided, or to be more precise, be converted into the problem of finding the local properties of the components yielding the corresponding global property of the whole system.

We are now ready to present the difficulties arising in the presence of value passing, and our approach to verifying communicating systems with value passing.

Chapter 3

Verification of Value Passing CCS Processes

In this chapter we address the problem of extending the techniques presented in the previous chapter to the case when processes are capable of communicating and storing values, and making decisions as to how to continue based on these values. As a process description language we assume the value passing extension of the CCS presented earlier.

Value passing can significantly influence the behaviour of a system. For example, consider the teller machine process defined in the previous chapter:

$$\begin{aligned} Teller(balance) &\triangleq Deposit(balance) + Withdrawal(balance) \\ Deposit(balance) &\triangleq deposit(amount).Teller(balance + amount) \\ Withdrawal(balance) &\triangleq \Sigma amount \\ &\quad \textbf{if } 0 < amount \leq balance \\ &\quad \textbf{then } \overline{withdraw}(amount).Teller(balance - amount) \end{aligned}$$

If we assume the domain of values to be the set of integers, then it is a property of this system that it cannot participate in an infinite sequence of withdrawals. If we abstract from the values, however, we obtain the following value-free version of the same process:

$$Teller \triangleq deposit.Teller + \overline{withdraw}.Teller$$

which no longer has the above property.

In this example the property did not explicitly mention any values. However, this is usually not the case when specifying value passing processes. Consider for example the following process modelling a memory cell [Dam93]:

$$Mem(x) \stackrel{\Delta}{\equiv} \overline{out}(x).Mem(x) + in(y).Mem(y)$$

It has the property that the value being read out is always the last value which has been written. This is a property which cannot be described in the propositional Modal μ-Calculus.

These considerations motivated our search for a suitable extension of the Modal μ-Calculus for specification, and a compositional proof system for verification of communicating systems described in the Value Passing CCS.

3.1 A μ-Calculus for Value Passing Processes

In searching for a suitable extension of the Modal μ-Calculus to handle values it seems to us a natural approach first to augment HML with values, and then to add fixpoint formulae.

The definition of the memory process given above describes in fact a whole family of processes. For example, it specifies process $Mem(3)$ as $\overline{out}(3).Mem(3) + in(y).Mem(y)$. Process $Mem(3)$ has the property that it can put out 3, but it cannot put out 4. In HML this property is formalisable as:

$$\Phi \stackrel{\Delta}{\equiv} \langle \overline{out}(3) \rangle \, \mathsf{tt} \wedge [\overline{out}(4)] \, \mathsf{ff}$$

In fact, $Mem(3)$ cannot put out any value different from 3. This property, however, cannot be expressed in HML unless we add quantifiers to the language. Using value

variables and quantifiers, it is expressible as:

$$\Phi_3 \stackrel{\Delta}{\equiv} \forall y.\, [\overline{out}(y)]\, y = 3$$

i.e., if y is an arbitrary value in the value domain, and if y can be put out, then y equals 3.

Consider now the following simple process:

$$P(x) \stackrel{\Delta}{=} \overline{out}(x).P(x+1)$$

$P(1)$ is capable of putting out successively all natural numbers. Expressing this property requires fixpoint formulae to be used. We introduce fixpoints again by considering modal equations. Let Z range now not just over propositions, but also over predicates over the value domain. Then the above property would be the $Z(1)$ element of the greatest solution to the following system of modal equations:

$$\forall x.(Z(x) \equiv \langle \overline{out}(x) \rangle\, Z(x+1))$$

We would like, however, to express this as a single equation having just Z on the left-hand side. To achieve this, we use the well-known *lambda notation*, i.e., we add lambda abstraction and application to our logic. The equation can then be written as:

$$Z \equiv \lambda x.\, \langle \overline{out}(x) \rangle\, Z(x+1)$$

The greatest solution to this equation is denoted $\nu Z.\lambda x.\, \langle \overline{out}(x) \rangle\, Z(x+1)$, and the property of being able to put out successively all natural numbers becomes formalisable as:

$$\Phi \stackrel{\Delta}{\equiv} (\nu Z.\lambda x.\, \langle \overline{out}(x) \rangle\, Z(x+1))(1)$$

In other words, the greatest shift in thinking implied by these considerations is that we have fixpoint predicates instead of fixpoint formulae. It is now predicates that have fixpoint approximants, and these approximants are predicates of the same arity. For example, the 0-th approximant of the above predicate $\Pi \stackrel{\Delta}{\equiv} \nu Z.\lambda x.\, \langle \overline{out}(x) \rangle\, Z(x+1)$ is not tt but the one-argument predicate $\lambda x.\mathsf{tt}$. The first few approximants are (after simplification using β-reduction):

$$\Pi^0 \equiv \lambda x.\mathsf{tt}$$
$$\Pi^1 \equiv \lambda x.\, \langle \overline{out}(x) \rangle\, \mathsf{tt}$$
$$\Pi^2 \equiv \lambda x.\, \langle \overline{out}(x) \rangle\, \langle \overline{out}(x+1) \rangle\, \mathsf{tt}$$
$$\Pi^3 \equiv \lambda x.\, \langle \overline{out}(x) \rangle\, \langle \overline{out}(x+1) \rangle\, \langle \overline{out}(x+2) \rangle\, \mathsf{tt}$$

When applied to 1, these approximants yield:

$$\Pi^0(1) \equiv \mathsf{tt}$$
$$\Pi^1(1) \equiv \langle \overline{out}(1) \rangle\, \mathsf{tt}$$
$$\Pi^2(1) \equiv \langle \overline{out}(1) \rangle\, \langle \overline{out}(2) \rangle\, \mathsf{tt}$$
$$\Pi^3(1) \equiv \langle \overline{out}(1) \rangle\, \langle \overline{out}(2) \rangle\, \langle \overline{out}(3) \rangle\, \mathsf{tt}$$

which justifies our claim that $\Phi \stackrel{\Delta}{\equiv} \Pi(1)$ formalises the property of being able to put out successively all natural numbers.

As another example, the property of the teller-machine process of not being able to offer infinitely many successive withdrawals can be formalised as:

$$\mu Z.\forall amount.\, \left[\overline{withdraw}(amount)\right] Z$$

After this intuitive introduction we are ready to present the syntax and semantics of the resulting logic. We again assume a set $\mathcal{A}$ of names, ranged over by a, each name having a non-negative arity. $\mathcal{L}$ denotes the set $\mathcal{A} \cup \overline{\mathcal{A}}$ of labels, ranged over by l, and $\overline{\overline{a}} \triangleq a$. We also assume a set D of values, value expressions e and Boolean expressions b built from value variables $x, y, z, \ldots$ (possibly indexed), value constants d and arbitrary operator symbols defined in the domain. Vectors $\vec{e}$ of expressions or $\vec{x}$ of values are used where the arity is not important.

3.1.1 Syntax

In the syntax of formulae we distinguish the following syntactic categories: Boolean expressions b, value expressions e, action expressions π, predicates Π, and formulae Φ. The syntax for Boolean and value expressions is left open.

The formulae of the logic are defined inductively as follows:

- Boolean expressions b are formulae;
- if Φ and Ψ are formulae, then so are $\Phi \vee \Psi$ and $\Phi \wedge \Psi$;
- if Φ is a formula and π is an action expression of the form $a(\vec{e})$, $\overline{a}(\vec{e})$, or τ, then $\langle \pi \rangle \Phi$ and $[\pi] \Phi$ are formulae;
- if Φ is a formula, and x is a value variable, then $\exists x.\Phi$ and $\forall x.\Phi$ are formulae;
- zero-ary predicates are formulae;
- if Π is a predicate, and $\vec{e}$ is a value expression of the same arity, then $\Pi\vec{e}$ is a formula.

where predicates and their arity are defined by:

- predicate variables Z are predicates of no specific arity;
- formulae are zero-ary predicates;

- if Φ is a formula, and $\vec{x}$ are value variables, then $\lambda\vec{x}.\Phi$ is a predicate of arity the length of $\vec{x}$;

- if Π is a predicate, and Z is a predicate variable, then $\mu Z.\Pi$ and $\nu Z.\Pi$ are predicates of the same arity as Π.

To illustrate these rules, we show that $(\nu Z.\lambda x.\, \langle\overline{out}(x)\rangle\, Z(x+1))(1)$ is a formula. Since Z is a predicate of no fixed arity, $Z(x+1)$ is a formula. Then $\langle\overline{out}(x)\rangle\, Z(x+1)$ is also a formula, and $\lambda x.\, \langle\overline{out}(x)\rangle\, Z(x+1)$ is consequently a one-argument predicate, and so is $\nu Z.\lambda x.\, \langle\overline{out}(x)\rangle\, Z(x+1)$. Therefore $(\nu Z.\lambda x.\, \langle\overline{out}(x)\rangle\, Z(x+1))(1)$ is a formula.

We identify zero-argument predicates with formulae. Note that if we deal with 0-ary syntactic categories only, the above syntax coincides with the one of the propositional Modal μ-Calculus.

The notions of free and bound variables are defined as usual. We also assume the usual notion $\Phi[\vec{e}/\vec{x}]$ of capture-avoiding substitution.

3.1.2 Semantics

The semantics of formulae is given, as in the propositional case, by their denotation relative to transition closed LTS with labels being either τ or of the form $l(\vec{d})$, where $l \in \mathcal{L}$ and $\vec{d} \in \vec{D}$, and relative to a valuation $\mathcal{V}$, which has to provide values not only for predicate variables, but also for value variables. So, the models $\mathcal{M}$ which provide interpretations for our formulae are pairs $(\mathcal{T}, \mathcal{V})$, where $\mathcal{T}$ is a transition closed LTS, and $\mathcal{V}$ is a valuation.

Given a model $\mathcal{M}$, the semantics of a formula Φ is defined inductively by the denotation $\|\Phi\|_{\mathcal{M}}$ of Φ w.r.t. $\mathcal{M}$, i.e. by the set of states in S satisfying Φ. We write $\|\Phi\|_{\mathcal{V}}^{\mathcal{T}}$ for $\|\Phi\|_{\mathcal{M}}$ and often omit the superscript when understood from the context.

The denotation of the different syntactic categories is of the following type:

$\|b\|_{\mathcal{V}} \in \{\mathsf{tt}, \mathsf{ff}\}$
$\|\vec{e}\|_{\mathcal{V}} \in \vec{D}$,
$\|\pi\|_{\mathcal{V}} \in Act$,
$\|\Phi\|_{\mathcal{V}} \subseteq S$, and
$\|\Pi\|_{\mathcal{V}} : \vec{D} \to \wp(S)$.

We assume that closed Boolean expressions b have a fixed truth value $[\![b]\!] \in \{\mathsf{tt}, \mathsf{ff}\}$, and closed value expressions e have a fixed value $[\![e]\!] \in D$. The notation $b[\mathcal{V}]$ and $e[\mathcal{V}]$ denotes the expressions obtained from b and e by substituting all free variable occurrences according to $\mathcal{V}$.

Let $\mathcal{E}_{\vec{D}}$, $\mathcal{E}'_{\vec{D}}$ etc. range over the set $[\vec{D} \to \wp(S)]$ of mappings from $\vec{D}$ to subsets of S, and let $\mathcal{E}_{\vec{d}}$ stand for $\mathcal{E}_{\vec{D}}(\vec{d})$. We define the partial ordering $\sqsubseteq$ on $[\vec{D} \to \wp(S)]$ by

$$\mathcal{E}_{\vec{D}} \sqsubseteq \mathcal{E}'_{\vec{D}} \stackrel{\Delta}{\equiv} \forall \vec{d} \in \vec{D}.\, \mathcal{E}_{\vec{d}} \subseteq \mathcal{E}'_{\vec{d}}$$

Let $E \subseteq [\vec{D} \to \wp(S)]$. We define:

$\sqcup E \stackrel{\Delta}{\equiv} \lambda \vec{d}.\, \bigcup_{\mathcal{E}_{\vec{D}} \in E} \mathcal{E}_{\vec{d}}$
$\sqcap E \stackrel{\Delta}{\equiv} \lambda \vec{d}.\, \bigcap_{\mathcal{E}_{\vec{D}} \in E} \mathcal{E}_{\vec{d}}$

The *denotation* of formulae Φ of the logic is defined inductively as shown in Figure 3.1.

The domain of predicates of a certain arity, together with $\sqsubseteq$, forms a complete lattice. Consequently, by referring to Tarski's fixpoint theorem [Tar55], the semantics of fixpoint formulae can indeed be given by the corresponding fixpoints on the predicate transformers $\lambda \mathcal{E}_{\vec{D}}.\, \|\Pi\|_{\mathcal{V}[\mathcal{E}_{\vec{D}}/Z]}$.

The denotation of closed formulae does not depend on valuations; we can hence write $\|\Phi\|$ instead of $\|\Phi\|_{\mathcal{V}}$ when Φ is closed.

$$\|\vec{e}\|_{\mathcal{V}} \triangleq [\![\vec{e}[\mathcal{V}]]\!] \quad \|Z\|_{\mathcal{V}} \triangleq \mathcal{V}(Z)$$

$$\|a(\vec{e})\|_{\mathcal{V}} \triangleq a(\|\vec{e}\|_{\mathcal{V}}) \quad \|\overline{a}(\vec{e})\|_{\mathcal{V}} \triangleq \overline{a}(\|\vec{e}\|_{\mathcal{V}}) \quad \|\tau\|_{\mathcal{V}} \triangleq \tau$$

$$\|b\|_{\mathcal{V}} \triangleq \begin{cases} S & \text{if } [\![b[\mathcal{V}]]\!] = \mathsf{tt} \\ \{\} & \text{otherwise} \end{cases}$$

$$\|\Phi_1 \wedge \Phi_2\|_{\mathcal{V}} \triangleq \|\Phi_1\|_{\mathcal{V}} \cap \|\Phi_2\|_{\mathcal{V}} \quad \|\Phi_1 \vee \Phi_2\|_{\mathcal{V}} \triangleq \|\Phi_1\|_{\mathcal{V}} \cup \|\Phi_2\|_{\mathcal{V}}$$

$$\|[\pi]\,\Phi\|_{\mathcal{V}} \triangleq \|[\|\pi\|_{\mathcal{V}}]\|\ \|\Phi\|_{\mathcal{V}} \quad \|\langle\pi\rangle\,\Phi\|_{\mathcal{V}} \triangleq \|\langle\|\pi\|_{\mathcal{V}}\rangle\|\ \|\Phi\|_{\mathcal{V}}$$

$$\|\forall x.\Phi\|_{\mathcal{V}} \triangleq \bigcap_{d\in D} \|\Phi\|_{\mathcal{V}[d/x]} \quad \|\exists x.\Phi\|_{\mathcal{V}} \triangleq \bigcup_{d\in D} \|\Phi\|_{\mathcal{V}[d/x]}$$

$$\|\Pi\,\vec{e}\|_{\mathcal{V}} \triangleq \|\Pi\|_{\mathcal{V}}\ \|\vec{e}\|_{\mathcal{V}} \quad \|\lambda\vec{x}.\Phi\|_{\mathcal{V}} \triangleq \lambda\vec{d}.\,\|\Phi\|_{\mathcal{V}[\vec{d}/\vec{x}]}$$

$$\|\nu Z.\Pi\|_{\mathcal{V}} \triangleq \nu\mathcal{E}_{\vec{D}}.\,\|\Pi\|_{\mathcal{V}[\mathcal{E}_{\vec{D}}/Z]} \quad \|\mu Z.\Pi\|_{\mathcal{V}} \triangleq \mu\mathcal{E}_{\vec{D}}.\,\|\Pi\|_{\mathcal{V}[\mathcal{E}_{\vec{D}}/Z]}$$

Figure 3.1: Denotation of formulae.

3.2 A Compositional Proof System

In this section we present a compositional proof system for sequential value passing CCS processes. To simplify the analysis of soundness and completeness, we shall not treat here the CCS operators of renaming and restriction. Our main interest is in value passing, but these two operators do not affect the values being communicated.[1], and their treatment is standard [ASW94, Bru93]

A main concern in designing the proof system has been to allow for proofs to be conducted in a way which follows our intuition nicely. For this reason, we started our investigation with designing intuitive pseudo-proofs for many communicating systems and their properties, we then selected from these a set of candidate proof rules, and tried to justify them semantically. After several iterations we arrived at a set of proof rules which we find intuitive, and for which we can prove that they are sound and complete relative to external reasoning about the value domain. We also aimed at keeping the rules as syntactic as possible, i.e., replacing semantic relationships between objects in sequents with syntactically checkable ones wherever this was possible. In our opinion, this simplifies the verification process since it minimises the

[1]Only the channel names are renamed or restricted.

cases in which a proof assistant would request help from an external tool handling the reasoning about the values.

3.2.1 Sequents

Following [HL95], the *judgements*, or *sequents*, of the proof system are of the form $b \vdash E : \Phi$, where b is a Boolean expression, called *assumption*, E is a value passing CCS term, and Φ is a formula in our extension of the Modal μ-Calculus. The intended semantic counterpart $b \models E : \Phi$ of such judgments is: whenever b holds, then E satisfies Φ. Consider for example the teller machine process described in the previous chapter. The judgement

$$amount \leq balance \models Teller(balance) : \left\langle \overline{withdraw}(amount) \right\rangle \mathtt{tt}$$

means that if some amount is not greater than the current balance of the teller machine, then this amount can be withdrawn. We find judgements of this type very intuitive. The Boolean conditions in sequents are needed to handle the Boolean conditions that might occur in CCS expressions. Note that in the above example we have an open Boolean condition, process term, and formula, but we somehow assumed that variables occurring in different parts of a sequent denote the same value. To capture this intuition formally, we have to refer again to valuations $\mathcal{V}$. We have also to give meaning to open process terms relative to a valuation. We hence define the *denotation* $\|E\|_{\mathcal{V}}$ of an agent expression E relative to a valuation $\mathcal{V}$ as the process which is obtained from E by substituting all free variables in E according to $\mathcal{V}$, i.e.,

$$\|E\|_{\mathcal{V}} \stackrel{\Delta}{=} E[\mathcal{V}]$$

and define the meaning of judgements by:

$$b \models E : \Phi \stackrel{\Delta}{\equiv} for\ any\ \mathcal{V},\ \|b\|_{\mathcal{V}} = \mathtt{tt}\ implies\ \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}}$$

3.2.2 Rules

The proof rules of our proof system can be grouped according to the parts of a judgement on whose structure they depend. There are also some rules which do not look into the structure of any of these parts; they are referred to as *general rules*. All rules are given in Figure 3.2 and are presented, for convenience of application, in a "goal-oriented" fashion, i.e., with the conclusion above the premises. *Axioms* are presented as rules with an empty premise, denoted by a dot. The names of the rules appear on the left-hand side of each rule, while side conditions appear on the right-hand side.

We shall briefly discuss the rules and explain how they are to be applied. Several small but instructive example proofs are given in the next section.

The first group of rules are the rules that look into the structure of the process term only. We call these rules *process rules*, or E-rules. The first rule allows the guarding condition of a process to be ignored if it is implied by the assumption of the sequent. The notation $b \Rightarrow b'$ means that b implies b' under all valuations. The second process rule deals with process constants by substituting them according to their definitions. Note that the substitution of expressions for variables might have to perform renaming of some variables to avoid variable capture.

The second group of inference rules are the so called *logic rules*, or Φ-rules, which look only into the structure of the formula. The rules for dealing with conjunctions and disjunctions are standard; the only peculiarity compared with applying these rules in the propositional case is that in the presence of values each application of the rules for disjunction will generally have to be preceded by an application of the cut-rule given below. Universal quantifiers can be eliminated due to the adopted semantics of sequents where free variables are implicitly universally quantified, but the quantifier variable has to be renamed if it occurs free somewhere in the sequent. Existential

$$\mathrm{E}(\mathbf{if})\ \frac{b \vdash \mathbf{if}\ b'\ \mathbf{then}\ E\ :\ \Phi}{b \vdash E\ :\ \Phi}\ b \Rightarrow b' \qquad \mathrm{E}(\triangleq)\ \frac{b \vdash A(\vec{e})\ :\ \Phi}{b \vdash E[\vec{e}/\vec{x}]\ :\ \Phi}\ A(\vec{x}) \triangleq E$$

$$\Phi(\wedge)\ \frac{b \vdash E\ :\ \Phi_1 \wedge \Phi_2}{b \vdash E\ :\ \Phi_1 \qquad b \vdash E\ :\ \Phi_2}$$

$$\Phi(\vee l)\ \frac{b \vdash E\ :\ \Phi_1 \vee \Phi_2}{b \vdash E\ :\ \Phi_1} \qquad \Phi(\vee r)\ \frac{b \vdash E\ :\ \Phi_1 \vee \Phi_2}{b \vdash E\ :\ \Phi_2}$$

$$\Phi(\forall)\ \frac{b \vdash E\ :\ \forall x.\Phi}{b \vdash E\ :\ \Phi[y/x]}\ y - fresh \qquad \Phi(\exists)\ \frac{b \vdash E\ :\ \exists x.\Phi}{b \vdash E\ :\ \Phi[e/x]}$$

$$\Phi(b)\ \frac{b \vdash E\ :\ b'}{\cdot}\ b \Rightarrow b' \qquad \Phi(\beta)\ \frac{b \vdash E\ :\ (\lambda\vec{x}.\Phi)\vec{e}}{b \vdash E\ :\ \Phi[\vec{e}/\vec{x}]}$$

$$\Phi(\nu 0)\ \frac{b \vdash E\ :\ (\nu Z\{L\}.\Pi)\vec{e}}{\cdot}\ \mathcal{C}_0 \qquad \Phi(\nu 1)\ \frac{b \vdash E\ :\ (\nu Z\{L\}.\Pi)\vec{e}}{b \vdash E\ :\ (\Pi[\nu Z\{l, L\}.\Pi/Z])\vec{e}}\ \mathcal{C}_\nu$$

$$\Phi(\mu 0)\ \frac{b \vdash E\ :\ (\mu Z\{L\}.\Pi)\vec{e}}{\cdot}\ \mathcal{C}_0 \qquad \Phi(\mu 1)\ \frac{b \vdash E\ :\ (\mu Z\{L\}.\Pi)\vec{e}}{b' \vdash E'\ :\ (\Pi[\mu Z\{l, L\}.\Pi/Z])\vec{e}\,'}\ \mathcal{C}_\mu$$

$$\mathrm{E}\Phi(\mathbf{0}, [\,])\ \frac{b \vdash \mathbf{0}\ :\ [\pi]\,\Phi}{\cdot} \qquad \mathrm{E}\Phi(\pi, [\,])\ \frac{b \vdash \pi.E\ :\ [\pi']\,\Phi}{\cdot}\ \pi \not\sim \pi'$$

$$\mathrm{E}\Phi(\tau, \langle\rangle)\ \frac{b \vdash \tau.E\ :\ \langle\tau\rangle\,\Phi}{b \vdash E\ :\ \Phi} \qquad \mathrm{E}\Phi(\tau, [\,])\ \frac{b \vdash \tau.E\ :\ [\tau]\,\Phi}{b \vdash E\ :\ \Phi}$$

$$\mathrm{E}\Phi(a, \langle\rangle)\ \frac{b \vdash a(\vec{x}).E\ :\ \langle a(\vec{e})\rangle\,\Phi}{b \vdash E[\vec{e}/\vec{x}]\ :\ \Phi} \qquad \mathrm{E}\Phi(a, [\,])\ \frac{b \vdash a(\vec{x}).E\ :\ [a(\vec{e})]\,\Phi}{b \vdash E[\vec{e}/\vec{x}]\ :\ \Phi}$$

$$\mathrm{E}\Phi(\overline{a}, \langle\rangle)\ \frac{b \vdash \overline{a}(\vec{e}).E\ :\ \langle\overline{a}(\vec{e})\rangle\,\Phi}{b \vdash E\ :\ \Phi} \qquad \mathrm{E}\Phi(\overline{a}, [\,])\ \frac{b \vdash \overline{a}(\vec{e}).E\ :\ [\overline{a}(\vec{e}\,')]\,\Phi}{b \wedge (\vec{e} = \vec{e}\,') \vdash E\ :\ \Phi}$$

$$\mathrm{E}\Phi(\Sigma, \langle\rangle)\ \frac{b \vdash \Sigma\vec{x}\,E\ :\ \langle\pi\rangle\,\Phi}{b \vdash E[\vec{e}/\vec{x}]\ :\ \langle\pi\rangle\,\Phi} \qquad \mathrm{E}\Phi(\Sigma, [\,])\ \frac{b \vdash \Sigma\vec{x}\,E\ :\ [\pi]\,\Phi}{b \vdash E[\vec{y}/\vec{x}]\ :\ [\pi]\,\Phi}\ \vec{y} - fresh$$

$$\mathrm{E}\Phi(+l, \langle\rangle)\ \frac{b \vdash E_1 + E_2\ :\ \langle\pi\rangle\,\Phi}{b \vdash E_1\ :\ \langle\pi\rangle\,\Phi} \qquad \mathrm{E}\Phi(+r, \langle\rangle)\ \frac{b \vdash E_1 + E_2\ :\ \langle\pi\rangle\,\Phi}{b \vdash E_2\ :\ \langle\pi\rangle\,\Phi}$$

$$\mathrm{E}\Phi(+, [\,])\ \frac{b \vdash E_1 + E_2\ :\ [\pi]\,\Phi}{b \vdash E_1\ :\ [\pi]\,\Phi \qquad b \vdash E_2\ :\ [\pi]\,\Phi}$$

$$\mathrm{E}\Phi(\mathbf{if}, [\,])\ \frac{b \vdash \mathbf{if}\ b'\ \mathbf{then}\ E\ :\ [\pi]\,\Phi}{b \wedge b' \vdash E\ :\ [\pi]\,\Phi}$$

$$\mathrm{G}(\mathsf{ff})\ \frac{b \vdash E\ :\ \Phi}{\cdot}\ b \equiv \mathsf{ff} \qquad \mathrm{G}(\mathsf{Sub})\ \frac{b[\vec{e}/\vec{x}] \vdash E[\vec{e}/\vec{x}]\ :\ \Phi[\vec{e}/\vec{x}]}{b \wedge (\vec{x} = \vec{e}) \vdash E\ :\ \Phi}\ \vec{x} - fresh$$

$$\mathrm{G}(\mathsf{Cut})\ \frac{b \vdash E\ :\ \Phi}{b' \vdash E\ :\ \Phi \qquad b'' \vdash E\ :\ \Phi}\ b \Rightarrow b' \vee b''$$

$$\mathrm{G}(\equiv)\ \frac{b \vdash E\ :\ \Phi}{b \vdash E'\ :\ \Phi'}\ E \equiv_b E',\ \Phi \equiv_b \Phi'$$

Figure 3.2: Proof Rules.

quantifiers can also be eliminated, but the quantifier variable has to be substituted with a suitable expression. The validity of a sequent having a Boolean expression as formula obviously depends only on the assumption. Hence, the rule for dealing with such sequents does not generate sub-goals, i.e., it is an axiom rule. Lambda expressions are naturally dealt with using β-reduction. So, the only logic rules requiring special attention are the fixpoint rules; we therefore defer their consideration.

The only case where we cannot reduce a sequent without looking into the structure of the process is when the formula is of the form $[\pi]\,\Phi$ or $\langle\pi\rangle\,\Phi$, since the semantics of these modalities concerns the immediate next-step behaviour of processes. For treating this case there is a group of rules, usually referred to as *dynamic rules*, or EΦ-rules. The "nil" process $\mathbf{0}$ has no derivatives, and therefore possesses vacuously all "box" properties. This explains the simplicity of the first rule which is an axiom. A similar case is when the process starts with a prefix action π which is not *compatible* with the action π' of a leading "box" modality, denoted $\pi \not\sim \pi'$, i.e., when either exactly one of them is τ, or otherwise if $\pi = l(\vec{e})$ and $\pi' = l'(\vec{e}\,')$, then $l \neq l'$. The τ-rules are obvious. So are the other rules for treating prefices; the difference between treating input and output actions is only due to the adopted convention that input actions have binding power while output actions have not. It should only be noted, that applications of rule E$\Phi(\overline{a}, \langle\rangle)$ have usually to be preceded by an application of rule G($\equiv$), given below, to unify the two value expressions. The binary choice operator on processes is handled similarly to disjunction in the logic rules, if the formula is of type $\langle\pi\rangle\,\Phi$, and similarly to conjunction otherwise, which is a simple consequence of their semantics. A similar correspondence exists between the rules for infinite summation on one hand, and the logical quantifier rules on the other. The last dynamic rule is for dealing with processes having guarding conditions. Note that there is no dynamic rule for such processes for formulae of type $\langle\pi\rangle\,\Phi$ - this case is covered by rule E(**if**) discussed above.

The fourth and last group of rules are the so-called *general* rules for dealing with

value expressions and assumptions. The first rule allows vacuously true sequents, i.e., sequents based on a false assumption, to be discharged. The second rule allows expressions to be "moved" to the assumption, and plays an important rôle in our proof of completeness. The third rule is a "cut" rule. The last general rule is for unifying expressions occurring in different parts of a sequent. Its side condition requires E to be identical to E' up to equivalence of terms under b, and the same for Φ and Φ'. For example, $E(x^2) \equiv_{x=1} E(x)$ holds since x^2 and x are equal when $x = 1$.

The greatest difficulty in designing such a proof system is presented by the treatment of fixpoint formulae. For this reason we present our approach in detail, starting with intuitive considerations. To allow for a simpler treatment of fix-point formulae, we impose the restriction on the syntax of formulae that all fix-point sub-predicates of the root formula be closed. Note that all rules preserve this property. This restriction does not affect the expressive power of the logic, since every formula containing fix-point sub-predicates with free object variables can easily be converted into an equivalent formula in which all fix-point sub-predicates are closed. For example, $\sigma Z.\langle a(x)\rangle\, Z$ is equivalent to $(\sigma Z.\lambda x.\langle a(x)\rangle\, Z(x))(x)$. We begin with the easier case of treating greatest fixpoints.

3.2.3 Greatest Fixpoints

Consider a parametrised process capable of putting out infinitely many successive numbers:

$$P(x) \stackrel{\Delta}{=} \overline{out}(x).P(x+1)$$
$$\Pi \stackrel{\Delta}{\equiv} \nu Z.\lambda x.\langle \overline{out}(x)\rangle\, Z(x+1)$$

and let us attempt to show that

$$\vdash\ P(x)\ :\ \Pi(x)$$

Before focusing on the process term, we manipulate the formula using logic rules until

a formula of the form $[\pi]\,\Phi$ or $\langle\pi\rangle\,\Phi$ is obtained. We start by unfolding the fixpoint predicate:

$$\vdash\ P(x)\ :\ (\lambda x.\,\langle\overline{out}(x)\rangle\,\Pi(x+1))(x)$$

and apply β-reduction to obtain:

$$\vdash\ P(x)\ :\ \langle\overline{out}(x)\rangle\,\Pi(x+1)$$

Now we unfold the process constant according to its definition:

$$\vdash\ \overline{out}(x).P(x+1)\ :\ \langle\overline{out}(x)\rangle\,\Pi(x+1)$$

After applying the respective dynamic rule we obtain:

$$\vdash\ P(x+1)\ :\ \Pi(x+1)$$

which is actually a partial case of the initial goal: this can be made explicit by applying universal substitution (a derived rule presented below):

$$\vdash\ P(x)\ :\ \Pi(x)$$

We have reached the same sequent from which we started, but implicitly we descended the approximant hierarchy, thus allowing the sequent to be discharged successfully.

To avoid the necessity for using global proof rules, i.e., investigating the whole proof tree, and to simplify the theoretical investigation of our proof system, we employ Winskel's approach of *tagging* greatest fixpoint formulae [Win91], and generalise it to the case with value passing. The difficult question is what information to include in these tags to allow loops in the proof to be detected, and what would be a suitable semantics for tagged formulae.

Since in our case we unfold predicates rather than formulae, it is natural to assume that it is the predicates that should be tagged. Hence, when unfolding a predicate Π in a sequent $b\ \vdash\ E\ :\ \Pi\vec{e}$, the tag should "remember" all components of the sequent except Π itself. Consequently, tags should be sets L of triples of the form

$(b, E, \vec{e})$. This is as far as the syntax of tagged formulae is concerned. Finding a suitable semantics is far more complicated. Winskel suggested tags to be understood as hypotheses. Following his idea, we interpret tags as predicates: triples $(b, E, \vec{e})$ are interpreted as predicates, and tags, being sets of such triples, are interpreted as the disjunction of these predicates. Intuitively, the meaning of a triple $(b, E, \vec{e})$ should be the least predicate Π' for which $b \models E : \Pi'\vec{e}$. A formal justification of this intuition is given in the next chapter; here we only give the definitions. With a triple $l = (b, E, \vec{e})$ we associate the indexed set $\mathcal{E}^l_{\vec{D}}$ of processes defined by:

$$\mathcal{E}^l_{\vec{D}} \stackrel{\Delta}{=} \lambda\vec{d}.\{P \mid \exists\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \wedge \|E\|_{\mathcal{V}} = P \wedge \|\vec{e}\|_{\mathcal{V}} = \vec{d})\}$$

The semantics of tagged predicates, both for greatest and least fixpoints, is defined as follows:

$$\|\sigma Z\{L\}.\Pi\|_{\mathcal{V}} \stackrel{\Delta}{=} \sigma\mathcal{E}_{\vec{D}}.\sqcup L \sqcup \|\Pi\|_{\mathcal{V}[\mathcal{E}_{\vec{D}}/Z]}$$

where $\sqcup L \stackrel{\Delta}{=} \bigsqcup_{l\in L} \mathcal{E}^l_{\vec{D}}$. If L is empty, $\sqcup L = \lambda\vec{d}.\{\}$, and hence $\sigma Z.\Pi \equiv \sigma Z\{\}.\Pi$.

We are now ready to explain the rules for handling greatest fixpoint formulae. Axiom rule $\Phi(\nu 0)$ allows a sequent to be successfully discharged if $(b, E, \vec{e}) \in L$, which is abbreviated as $\mathcal{C}_0$. Rule $\Phi(\nu 1)$ is for unfolding combined with tagging; its side condition $\mathcal{C}_\nu$, however, is stronger than simply requiring $(b, E, \vec{e}) \notin L$. As justified by our analysis of completeness in the next chapter, we require that $\hat{l} \notin \hat{L}$, where $\hat{l}$ and $\hat{L}$ are the value-free versions of $(b, E, \vec{e})$ and L. This does not decrease the inferential power of the system, but prevents fixpoint formulae from being unfolded unnecessarily.

3.2.4 Least Fixpoints

The treatment of least fixpoint formulae is more complicated. Consider the following process which is capable of putting out only finite-length sequences of numbers:

$$P(x) \stackrel{\Delta}{\equiv} \textbf{if } x > 0 \textbf{ then } \overline{out}(x).P(x-1)$$
$$\Phi \stackrel{\Delta}{\equiv} \mu Z.\forall x.\, [\overline{out}(x)]\, Z$$

and let us attempt to infer the following goal

$$\vdash\ P(x)\ :\ \Phi$$

Unfolding Φ yields

$$\vdash\ P(x)\ :\ \forall x.\, [\overline{out}(x)]\, \Phi$$

We can eliminate the universal quantifier by introducing the fresh variable y, thus obtaining

$$\vdash\ P(x)\ :\ [\overline{out}(y)]\, \Phi$$

Unfolding $P(x)$ according to its definition yields

$$\vdash\ \textbf{if } x > 0 \textbf{ then } \overline{out}(x).P(x-1)\ :\ [\overline{out}(y)]\, \Phi$$

The guarding condition $x > 0$ can be moved to the sequent assumptions

$$x > 0\ \vdash\ \overline{out}(x).P(x-1)\ :\ [\overline{out}(y)]\, \Phi$$

and after applying the corresponding dynamic rule we arrive at

$$(x > 0) \wedge (x = y)\ \vdash\ P(x-1)\ :\ \Phi$$

which can be further simplified to

$$x > 0\ \vdash\ P(x-1)\ :\ \Phi$$

This is the crucial point in the example, where one would expect to be able to resolve this goal, since further unfolding is obviously redundant. The following inductive argument seems to suggest that we can discharge the goal successfully. Substitute any value d of the value domain for x in the goal sequent. One could repeat the above derivation sequence for d. This means, that $d > 0 \models P(d-1) : \Phi$ implies $\models P(d) : \Phi$. Note that if $d \leq 0$ the former sequent can be discharged using axiom rule G(ff). So, $\models P(d) : \Phi$ holds for all non-positive values d, and it holds for all other values d if it holds for $d-1$. By induction, therefore, $\models P(d) : \Phi$ holds for all values d, and hence $\models P(x) : \Phi$. This inductive argument, however, is not really present in the above derivation sequence: the semantics of sequents we have chosen does not support such "pointwise" reasoning, i.e., inference for every particular value of the domain. The desired effect can be achieved if one substitutes the symbol c for x in the above derivation, c being interpreted as some arbitrary *constant* of the value domain.

We employ an idea by H.Andersen [And93] to use tagging for inductive reasoning. The rationale behind it is that, with the semantics of tagged predicates chosen, tags can be interpreted as being assumptions (hypotheses) and can hence be used for encoding of the induction hypothesis of an inductive argument. However, instead of doing induction on sets of processes explicitly, as this was done in [And93], we do this implicitly by using induction on the domain of values. In this way we avoid the introduction of infinitary proof rules in our proof system. One has, however, to introduce into the language a new syntactic category, namely *constants*. These are to be treated as constants from the domain of values. In this way we obtain an implicit second level of universal quantification which is necessary to conduct an inductive pointwise argument over the domain of values.

We are ready to explain the side condition $\mathcal{C}_\mu$ for applying rule $\Phi(\mu 1)$. First, as in $\Phi(\nu 1)$, we require $\hat{l} \notin \hat{L}$. And second, there have to exist variables $\vec{x}$ (which may be termed *induction variables*) occurring free in b, E, or $\vec{e}$, fresh arbitrary constants

$\vec{c}$, and fresh variables $\vec{x}'$, so that $b' = b[\vec{c}/\vec{x}]$, $E' = E[\vec{c}/\vec{x}]$, and $\vec{e}' = \vec{e}[\vec{c}/\vec{x}]$, and the new triple l inserted in the tag is $(b''[\vec{c}/\vec{x}], E[\vec{x}'/\vec{x}], \vec{e}[\vec{x}'/\vec{x}])$, for some b'' implying $b[\vec{x}'/\vec{x}]$ under all valuations and inducing a well-founded relation $\prec$ on $\vec{D}$ defined by $\vec{d}' \prec \vec{d} \triangleq b''[\vec{d}/\vec{x}, \vec{d}'/\vec{x}']$. Well-foundedness means that $\prec$ has no infinite decreasing chains.

The formulation of the rule is rather complicated, but the idea behind it is not. The variables $\vec{x}'$, called the *primed version* of $\vec{x}$, can be understood as capturing the "new values" of $\vec{x}$ after unfolding a least fixpoint, and the arbitrary constants $\vec{c}$ are intended to "store" the old values. In the tag we encode the induction hypothesis that the sequent is valid for all values less than $\vec{c}$ w.r.t. $\prec$. Reducing the validity of a sequent for any fixed values of its variables to the validity of the same sequent but for smaller values w.r.t. some well-founded relation establishes, by Noetherian induction, the general validity of this sequent. The main difficulty in applying the rule is finding a suitable triple l for which one expects to be able to reach the corresponding sequent allowing the axiom rule $\Phi(\mu 0)$ to be applied. The use of the rule is exemplified in the next section.

Note that rule $\Phi(\mu 1)$ can always be applied trivially by choosing $\vec{x}$ to be a null-ary vector, and b'' to be false; this is equivalent to simple unfolding without changing the tag.

3.2.5 Extensions and Derived Rules

There are two main approaches to defining new operators in the process language: by explicitly defining their transitional semantics by giving transitional rules, or by defining them in terms of the already existing operators [Mil89]. While the first approach is more flexible and more powerful in general, in the second case one can derive proof rules for the new operators from the already existing rules. The same can be said about defining new constructs in the logic.

A useful and common extension of the value passing CCS is the construction

if b **then** E_1 **else** E_2. It can be defined in terms of **if** b **then** E as follows [Mil89]:

$$\begin{array}{lcl} \textbf{if } b \textbf{ then } E_1 \textbf{ else } E_2 & \triangleq & \textbf{if } b \textbf{ then } E_1 \\ & + & \textbf{if } \neg b \textbf{ then } E_2 \end{array}$$

Using rules E$\Phi(+l, \langle\rangle)$, E$\Phi(+l, \langle\rangle)$, and E(**if**), we derive the rules E$\Phi(\textbf{ifte}l, \langle\rangle)$ and E$\Phi(\textbf{ifte}r, \langle\rangle)$, given in Figure 3.3. In the same way, using rules E$\Phi(+, [])$ and E$\Phi(\textbf{if}, [])$, rule E$\Phi(\textbf{ifte}, [\,])$ is obtained.

$$\mathrm{E}\Phi(\textbf{ifte}l, \langle\rangle)\ \frac{b \vdash \textbf{if } b' \textbf{ then } E_1 \textbf{ else } E_2 \ :\ \langle\pi\rangle\,\Phi}{b \vdash E_1 \ :\ \langle\pi\rangle\,\Phi}\ b \Rightarrow b'$$

$$\mathrm{E}\Phi(\textbf{ifte}r, \langle\rangle)\ \frac{b \vdash \textbf{if } b' \textbf{ then } E_1 \textbf{ else } E_2 \ :\ \langle\pi\rangle\,\Phi}{b \vdash E_2 \ :\ \langle\pi\rangle\,\Phi}\ b \Rightarrow \neg b'$$

$$\mathrm{E}\Phi(\textbf{ifte}, [\,])\ \frac{b \vdash \textbf{if } b' \textbf{ then } E_1 \textbf{ else } E_2 \ :\ [\pi]\,\Phi}{b \wedge b' \vdash E_1 \ :\ [\pi]\,\Phi \qquad b \wedge \neg b' \vdash E_2 \ :\ [\pi]\,\Phi}$$

$$\mathrm{G}(\Rightarrow)\ \frac{b \vdash E \ :\ \Phi}{b' \vdash E \ :\ \Phi}\ b \Rightarrow b' \qquad \mathrm{G}(\mathsf{US})\ \frac{b[\vec{e}/\vec{x}] \vdash E[\vec{e}/\vec{x}] \ :\ \Phi[\vec{e}/\vec{x}]}{b \vdash E \ :\ \Phi}$$

Figure 3.3: Derived Rules.

We shall often use the widening rule G($\Rightarrow$), which can be derived from rule G(Cut) by taking b'' to be false, and rule G(ff). The last rule, called *universal substitution*, is also a widening rule since its conclusion is a specialisation of the premise due to the adopted semantics of sequents where free variables are implicitly universally quantified. It can be derived using G(Sub) and G($\Rightarrow$).

3.3 Example Proofs

In this section we present some example proofs conducted in the proof system given above. Let us outline the general proof scheme followed in these proofs. This scheme is derived in a natural way from the analysis of completeness performed in the next chapter, yielding what one might call *canonical proofs*.

1. We apply logic rules until the formula becomes of the form $[\pi]\,\Phi$ or $\langle\pi\rangle\,\Phi$, or the resulting sub-goal can be discharged with an axiom. The rules for disjunction have usually to be preceded by a suitable application of the cut rule, while the fixpoint rules might have to be preceded by applications of more than one general rule.

2. Next, process and dynamic rules are applied until the process becomes of the form $\pi.E$, or the resulting sub-goal can be discharged with an axiom. The rules for binary choice have usually to be preceded by a suitable application of the cut rule.

3. At this point, the process is of the form $\pi.E$, and the formula is of the form $[\pi]\,\Phi$ or $\langle\pi\rangle\,\Phi$. The corresponding dynamic rule is applied, and the first step is re-entered. Rule $\mathrm{E}\Phi(\overline{a},\langle\rangle)$ has usually to be preceded by a suitable application of $\mathrm{G}(\equiv)$ to unify the respective expressions.

Our first example is the memory-cell process described at the beginning of this chapter [Dam93]:

$$Mem(x) \stackrel{\Delta}{\equiv} \overline{out}(x).Mem(x) + in(y).Mem(y)$$

It has the property that it cannot put out a value different from the value x currently stored in it:

$$\Phi_x \stackrel{\Delta}{\equiv} \forall y.\,[\overline{out}(y)]\,y = x$$

We can prove that $Mem(x)$ satisfies Φ_x as follows, proofs being presented as goal-directed tableaux:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash\ Mem(x)\ :\ \forall y.\,[\overline{out}(y)]\,y = x}\ \Phi(\forall)}{\vdash\ Mem(x)\ :\ [\overline{out}(y)]\,y = x}}{\vdash\ \overline{out}(x).Mem(x) + in(y).Mem(y)\ :\ [\overline{out}(y)]\,y = x}\ \mathrm{E}(\stackrel{\Delta}{=})}{\vdash\ \overline{out}(x).Mem(x)\ :\ [\overline{out}(y)]\,y = x}\ \mathrm{E}\Phi(+,[\,])}{x = y\ \vdash\ Mem(x)\ :\ y = x}\ \mathrm{E}\Phi(\overline{a},[\,])}{\Phi(\mathrm{b})}\qquad \dfrac{\vdash\ in(y).Mem(y)\ :\ [\overline{out}(y)]\,y = x}{\cdot}\ \mathrm{E}\Phi(\pi,[\,])$$

The proof follows "blindly" the above outlined steps. The only place where external reasoning about values is required is the application of rule $\Phi(\mathrm{b})$. The application is justified on the account that $x = y$ implies $y = x$.

As an example for handling greatest fixpoints, consider process P defined as follows:

$$\begin{aligned} P &\stackrel{\Delta}{=} Q(0) \\ Q(x) &\stackrel{\Delta}{=} R(x, x+1) \\ R(x,y) &\stackrel{\Delta}{=} \overline{out}(x+2, y+3).R(x+1, y+2) \end{aligned}$$

This process is capable of engaging in the infinite sequence of interactions

$$s_0 \stackrel{\overline{out}(2,4)}{\longrightarrow} s_1 \stackrel{\overline{out}(3,6)}{\longrightarrow} s_2 \stackrel{\overline{out}(4,8)}{\longrightarrow} \cdots$$

i.e., of putting out all consecutive pairs $(x, 2x)$ starting with $x = 2$. This property can be described as $\Pi(2)$ where:

$$\Pi \stackrel{\Delta}{=} \nu Z.\lambda x.\,\langle \overline{out}(x, 2x)\rangle\, Z(x+1)$$

The proof could be conducted as follows:

$$
\begin{array}{cl}
\vdash P \,:\, \Pi(2) & \\ \hline
\vdash Q(0) \,:\, \Pi(2) & \mathrm{E}(\triangleq) \\ \hline
\vdash R(0, 0+1) \,:\, \Pi(2) & \mathrm{E}(\triangleq) \\ \hline
\vdash R(0, 2\times 0+1) \,:\, \Pi(0+2) & \mathrm{G}(\equiv) \\ \hline
\vdash R(x, 2x+1) \,:\, \Pi(x+2) & \mathrm{G}(\mathsf{US}) \\ \hline
\vdash R(x, 2x+1) \,:\, (\lambda x.\, \langle \overline{out}(x, 2x)\rangle\, \Pi^l(x+1))(x+2) & \Phi(\nu 1) \\ \hline
\vdash R(x, 2x+1) \,:\, \langle \overline{out}(x+2, 2(x+2))\rangle\, \Pi^l((x+2)+1) & \Phi(\beta) \\ \hline
\vdash \overline{out}(x+2, (2x+1)+3).R(x+1, (2x+1)+2) \,:\, \langle \overline{out}(x+2, 2(x+2))\rangle\, \Pi^l((x+2)+1) & \mathrm{E}(\triangleq) \\ \hline
\vdash \overline{out}(x+2, 2x+4).R(x+1, 2x+3) \,:\, \langle \overline{out}(x+2, 2x+4)\rangle\, \Pi^l(x+3) & \mathrm{G}(\equiv) \\ \hline
\vdash R(x+1, 2x+3) \,:\, \Pi^l(x+3) & \mathrm{E}\Phi(\overline{a}, \langle\rangle) \\ \hline
\vdash R(x+1, 2(x+1)+1) \,:\, \Pi^l((x+1)+2) & \mathrm{G}(\equiv) \\ \hline
\vdash R(x, 2x+1) \,:\, \Pi^l(x+2) & \mathrm{G}(\mathsf{US}) \\ \hline
. & \Phi(\nu 0)
\end{array}
$$

where $\Pi^l \triangleq \nu Z\{l\}.\lambda x.\, \langle \overline{out}(x, 2x)\rangle\, Z(x+1)$ with $l \triangleq (\epsilon, R(x, 2x+1), x+2)$. It is worth noticing the way in which rule G(US) is used in combination with rule G($\equiv$): in the first case as a widening, allowing the proof to be completed with just one unfolding of the ν-formula, and in the second, to allow for rule $\Phi(\nu 0)$ to be applied.

Here is an example for using the least fixpoint rules. Consider process P defined by:

$$
\begin{array}{rcl}
P & \triangleq & in(x).Q(x) \\
Q(x) & \triangleq & \mathbf{if}\ x > 0\ \mathbf{then}\ \overline{out}(x).Q(x-1)
\end{array}
$$

and its property (for any integer x) of not being able to engage in an infinite sequence of the form

$$
s_0 \xrightarrow{in(x)} s_1 \xrightarrow{\overline{out}(x)} s_2 \xrightarrow{\overline{out}(x-1)} s_3 \xrightarrow{\overline{out}(x-2)} s_4 \xrightarrow{\overline{out}(x-3)} \cdots
$$

formalisable as:

$$
\begin{array}{l}
\Phi \triangleq \forall x.\, [in(x)]\, \Pi(x) \\
\Pi \triangleq \mu Z.\lambda x.\, [\overline{out}(x)]\, Z(x-1)
\end{array}
$$

The proof goes as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash\ P\ :\ \forall x.\,[in(x)]\,\Pi(x)}\,\Phi(\forall)}{\vdash\ P\ :\ [in(y)]\,\Pi(y)}\,\mathrm{E}(\triangleq)}{\vdash\ in(x).Q(x)\ :\ [in(y)]\,\Pi(y)}\,\mathrm{E}\Phi(a,[\,])}{\vdash\ Q(y)\ :\ \Pi(y)}\,\Phi(\mu 1)}{\vdash\ Q(c)\ :\ (\lambda x.\,[\overline{out}(x)]\,\Pi^c(x-1))(c)}\,\Phi(\beta)}{\vdash\ Q(c)\ :\ [\overline{out}(c)]\,\Pi^c(c-1)}\,\mathrm{E}(\triangleq)}{\vdash\ \mathbf{if}\ c>0\ \mathbf{then}\ \overline{out}(c).Q(c-1)\ :\ [\overline{out}(c)]\,\Pi^c(c-1)}\,\mathrm{E}\Phi(\mathbf{if},[\,])}{c>0\ \vdash\ \overline{out}(c).Q(c-1)\ :\ [\overline{out}(c)]\,\Pi^c(c-1)}\,\mathrm{E}\Phi(\overline{a},[\,])}{(c>0)\wedge(c=c)\ \vdash\ Q(c-1)\ :\ \Pi^c(c-1)}\,\mathrm{G}(\Rightarrow)}{c>0\ \vdash\ Q(c-1)\ :\ \Pi^c(c-1)}\,\mathrm{G}(\mathsf{Sub})}{(c>0)\wedge(y'=c-1)\ \vdash\ Q(y')\ :\ \Pi^c(y')}\,\Phi(\mu 0)$$

.

where $\Pi^c \triangleq \mu Z\{l_c\}.\lambda x.\,[\overline{out}(x)]\,Z(x-1)$ with $l_c \triangleq ((c>0)\wedge(y'=c-1), Q(y'), y')$. Rule $\Phi(\mu 1)$ has been applied with y, c, and $(y>0)\wedge(y'\leq y-1)$ for $\vec{x}$, $\vec{c}$, and b'' in the description of condition $\mathcal{C}_\mu$, respectively. The binary relation $\prec$ on D defined by $y' \prec y \triangleq b''$ is well-founded.

Our last example is a simple communication protocol composed of a sender and a receiver. As a system, the protocol behaves as a one-element buffer, i.e., it can repeatedly input a natural number and afterwards output the same number. The sender accepts a number n, engages afterwards in n consecutive *tick* communications with the receiver, informs the receiver through a *last* action about the last *tick*, waits for an acknowledgement *ack* from the receiver that n has been read out, and finally enters the initial state:

$$\begin{array}{rcl} Sender & \triangleq & in(x).Sender'(x) \\ Sender'(x) & \triangleq & \mathbf{if}\ \ x>0\ \ \mathbf{then}\ \ \overline{tick}.Sender'(x-1) \\ & & \mathbf{else}\ \ \overline{last}.ack.Sender \end{array}$$

The receiver repeatedly accepts *tick* actions, incrementing a counter, until a *last* action is accepted, then the value of the counter is put out, and an acknowledgement is given to the sender:

$$\begin{array}{rcl} Receiver & \stackrel{\Delta}{=} & Receiver'(0) \\ Receiver'(x) & \stackrel{\Delta}{=} & tick.Receiver'(x+1) \\ & + & last.\overline{out}(x).\overline{ack}.Receiver \end{array}$$

The protocol hides the internal communication between sender and receiver:

$$Protocol \stackrel{\Delta}{=} (Sender|Receiver)\backslash\{tick, last, ack\}$$

Many character-oriented communication protocols exchange data packets character-by-character at the physical layer, and this behaviour is only slightly more complex then the behaviour of $Protocol$.

We first show that $Protocol$ can repeatedly input a natural number n and engage afterwards in exactly n $\overline{tick}$ actions, followed by ack (here Π_Ψ should be understood as a formula scheme):

$$\Phi \stackrel{\Delta}{\equiv} \nu Z.\forall x.\, \langle in(x)\rangle\, \Pi_Z(x)$$
$$\Pi_\Psi \stackrel{\Delta}{\equiv} \mu Y.\lambda y.((y > 0 \wedge \left\langle \overline{tick} \right\rangle Y(y-1)) \vee (y = 0 \wedge \left\langle \overline{last} \right\rangle \langle ack\rangle\, \Psi))$$

Here is a proof tableau:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash Sender : \nu Z.\forall x.\, \langle in(x)\rangle\, \Pi_Z(x)}{\vdash Sender : \forall x.\, \langle in(x)\rangle\, \Pi_{\Phi'}(x)}\,\Phi(\nu 1)}{\vdash Sender : \langle in(x)\rangle\, \Pi_{\Phi'}(x)}\,\Phi(\forall)}{\vdash in(x).Sender'(x) : \langle in(x)\rangle\, \Pi_{\Phi'}(x)}\,\mathrm{E}(\stackrel{\Delta}{=})}{\vdash Sender'(x) : \Pi_{\Phi'}(x)}\,\mathrm{E}\Phi(a, \langle\rangle)}{\vdash Sender'(c) : (\lambda y.((y > 0 \wedge \left\langle \overline{tick} \right\rangle \Pi^c_{\Phi'}(y-1)) \vee (y = 0 \wedge \left\langle \overline{last} \right\rangle \langle ack\rangle\, \Phi')))(c)}\,\Phi(\mu 1)}{\vdash Sender'(c) : (c > 0 \wedge \left\langle \overline{tick} \right\rangle \Pi^c_{\Phi'}(c-1)) \vee (c = 0 \wedge \left\langle \overline{last} \right\rangle \langle ack\rangle\, \Phi')}\,\Phi(\beta)}{(1) \quad (2)}\,\mathrm{G}(\mathsf{Cut})$$

where sub-tableau (1) is:

$$\dfrac{\dfrac{\dfrac{c>0 \vdash Sender'(c) \;:\; (c>0 \wedge \langle\overline{tick}\rangle \Pi^c_{\Phi'}(c-1)) \vee (c=0 \wedge \langle\overline{last}\rangle\langle ack\rangle \Phi')}{c>0 \vdash Sender'(c) \;:\; c>0 \wedge \langle\overline{tick}\rangle \Pi^c_{\Phi'}(c-1)}\,\Phi(\vee l)}{c>0 \vdash Sender'(c) \;:\; c>0}\,\Phi(\wedge) \qquad \dfrac{c>0 \vdash Sender'(c) \;:\; \langle\overline{tick}\rangle \Pi^c_{\Phi'}(c-1)}{c>0 \vdash \begin{array}{l}\textbf{if } c>0 \quad \textbf{then}\, \overline{tick}.Sender'(c-1)\\ \qquad\qquad \textbf{else } \overline{last}.ack.Sender\end{array} \;:\; \langle\overline{tick}\rangle \Pi^c_{\Phi'}(c-1)}}{\dfrac{\dfrac{\dfrac{c>0 \vdash \overline{tick}.Sender'(c-1) \;:\; \langle\overline{tick}\rangle \Pi^c_{\Phi'}(c-1)}{c>0 \vdash Sender'(c-1) \;:\; \Pi^c_{\Phi'}(c-1)}\,\mathrm{E}\Phi(\overline{a},\langle\rangle)}{(c>0)\wedge(x'=c-1) \vdash Sender'(x') \;:\; \Pi^c_{\Phi'}(x')}\,\mathrm{G}(\mathsf{Sub})}{\cdot}\,\Phi(\mu 0)}$$

and sub-tableau (2) is:

$$\dfrac{\dfrac{\dfrac{c=0 \vdash Sender'(c) \;:\; (c>0 \wedge \langle\overline{tick}\rangle \Pi^c_{\Phi'}(c-1)) \vee (c=0 \wedge \langle\overline{last}\rangle\langle ack\rangle \Phi')}{c=0 \vdash Sender'(c) \;:\; c=0 \wedge \langle\overline{last}\rangle\langle ack\rangle \Phi'}\,\Phi(\vee r)}{c=0 \vdash Sender'(c) \;:\; c=0}\,\Phi(\wedge) \qquad \dfrac{c=0 \vdash Sender'(c) \;:\; \langle\overline{last}\rangle\langle ack\rangle \Phi'}{c=0 \vdash \begin{array}{l}\textbf{if } c>0 \quad \textbf{then}\, \overline{tick}.Sender'(c-1)\\ \qquad\qquad \textbf{else } \overline{last}.ack.Sender\end{array} \;:\; \langle\overline{last}\rangle\langle ack\rangle \Phi'}}{\dfrac{\dfrac{\dfrac{c=0 \vdash \overline{last}.ack.Sender \;:\; \langle\overline{last}\rangle\langle ack\rangle \Phi'}{c=0 \vdash ack.Sender \;:\; \langle ack\rangle \Phi'}\,\mathrm{E}\Phi(\overline{a},\langle\rangle)}{c=0 \vdash Sender \;:\; \Phi'}\,\mathrm{E}\Phi(a,\langle\rangle)}{\vdash Sender \;:\; \Phi'}\,\mathrm{G}(\Rightarrow)}{\cdot}\,\Phi(\nu 0)}$$

where Φ' and Π^c_Ψ are as Φ and Π_Ψ, but the first being additionally tagged with $(\epsilon, Sender, \epsilon)$, and the second with $((c>0)\wedge(x'=c-1), Sender'(x'), x')$. Note how rule G(Cut) has been applied in combination with rules $\Phi(\vee l)$ and $\Phi(\vee r)$ to allow the proof to be completed.

Process *Receiver* has the property, that it can repeatedly accept *tick* actions, thereby incrementing its counter (which is initialized to zero), until a *last* action is accepted, after which the value of the counter is output, and an acknowledging $\overline{ack}$ action is performed:

$$\Phi \stackrel{\Delta}{\equiv} \nu Z.\Pi_Z(0)$$
$$\Pi_\Psi \stackrel{\Delta}{\equiv} \nu Y.\lambda x.(\langle tick\rangle Y(x+1) \wedge \langle last\rangle\langle\overline{out}(x)\rangle\left\langle\overline{ack}\right\rangle \Psi)$$

We can show that *Receiver* satisfies Φ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash \mathit{Receiver} \;:\; \nu Z.\Pi_Z(0)}{\vdash \mathit{Receiver} \;:\; \Pi_{\Phi'}(0)}\,\Phi(\nu 1)}{\vdash \mathit{Receiver}'(0) \;:\; \Pi_{\Phi'}(0)}\,\mathrm{E}(\stackrel{\Delta}{=})}{\vdash \mathit{Receiver}'(x) \;:\; \Pi_{\Phi'}(x)}\,\mathrm{G}(\mathsf{US})}{\vdash \mathit{Receiver}'(x) \;:\; (\lambda x.(\langle tick\rangle \Pi'_{\Phi'}(x+1) \wedge \langle last\rangle \langle \overline{out}(x)\rangle \left\langle \overline{ack}\right\rangle \Phi'))(x)}\,\Phi(\nu 1)}{\vdash \mathit{Receiver}'(x) \;:\; \langle tick\rangle \Pi'_{\Phi'}(x+1) \wedge \langle last\rangle \langle \overline{out}(x)\rangle \left\langle \overline{ack}\right\rangle \Phi'}\,\Phi(\beta)}{(1)\quad(2)}\,\Phi(\wedge)$$

where sub-tableau (1) is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash \mathit{Receiver}'(x) \;:\; \langle tick\rangle \Pi'_{\Phi'}(x+1)}{\vdash \begin{array}{ll} & tick.Sender'(x+1) \\ + & last.\overline{out}(x).\overline{ack}.Receiver \end{array} \;:\; \langle tick\rangle \Pi'_{\Phi'}(x+1)}\,\mathrm{E}(\stackrel{\Delta}{=})}{\vdash tick.Receiver'(x+1) \;:\; \langle tick\rangle \Pi'_{\Phi'}(x+1)}\,\mathrm{E}\Phi(+l,\langle\rangle)}{\vdash \mathit{Receiver}'(x+1) \;:\; \Pi'_{\Phi'}(x+1)}\,\mathrm{E}\Phi(a,\langle\rangle)}{\vdash \mathit{Receiver}'(x) \;:\; \Pi'_{\Phi'}(x)}\,\mathrm{G}(\mathsf{US})}{\cdot}\,\Phi(\nu 0)$$

and sub-tableau (2) is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash \mathit{Receiver}'(x) \;:\; \langle last\rangle \langle \overline{out}(x)\rangle \left\langle \overline{ack}\right\rangle \Phi'}{\vdash \begin{array}{ll} & tick.Sender'(x+1) \\ + & last.\overline{out}(x).\overline{ack}.Receiver \end{array} \;:\; \langle last\rangle \langle \overline{out}(x)\rangle \left\langle \overline{ack}\right\rangle \Phi'}\,\mathrm{E}(\stackrel{\Delta}{=})}{\vdash last.\overline{out}(x).\overline{ack}.Receiver \;:\; \langle last\rangle \langle \overline{out}(x)\rangle \left\langle \overline{ack}\right\rangle \Phi'}\,\mathrm{E}\Phi(+r,\langle\rangle)}{\vdash \overline{out}(x).\overline{ack}.Receiver \;:\; \langle \overline{out}(x)\rangle \left\langle \overline{ack}\right\rangle \Phi'}\,\mathrm{E}\Phi(a,\langle\rangle)}{\vdash \overline{ack}.Receiver \;:\; \left\langle \overline{ack}\right\rangle \Phi'}\,\mathrm{E}\Phi(\overline{a},\langle\rangle)}{\vdash \mathit{Receiver} \;:\; \Phi'}\,\mathrm{E}\Phi(\overline{a},\langle\rangle)}{\cdot}\,\Phi(\nu 0)$$

where Φ' and Π'_Ψ are like Φ and Π_Ψ, but the first being additionally tagged by $(\epsilon, \mathit{Receiver}, \epsilon)$, and the second by $(\epsilon, \mathit{Receiver}'(x), x)$.

Showing that *Protocol* behaves as expected, i.e., proving

$$\vdash \mathit{Protocol} \;:\; \nu Z.\forall x.\, \langle in(x)\rangle \langle \overline{out}(x)\rangle Z$$

can be reduced, using compositional reasoning, to the local properties of *Sender* and *Receiver* we just verified. We address this problem in Chapter 5.

The examples considered in this section give an insight into how the proof system is to be used, and to what extent proofs can be guided and assisted by a machine.

Chapter 4

Correctness of the Proof System

In this chapter we investigate the correctness of the proof system presented in the previous chapter, namely whether it is sound and complete. As explained earlier, *soundness* means that every sequent derivable in the proof system is valid. A proof system which is not sound is hardly of any use for verification. Since all proof rules in our system are local, proving soundness can be reduced to proving the individual soundness of each rule, i.e., showing that axioms are valid and that all other rules preserve validity. *Completeness* means the reverse, namely that all valid sequents are derivable. Together, the two properties imply that the valid sequents are exactly the ones derivable in the proof system. Proving completeness is usually far more complicated a task than proving soundness. The approach taken here is standard: we show completeness by showing that every valid sequent can be reduced to valid sequents which are in some sense simpler.

4.1 Soundness

In our proof of soundness we refer to the following results.

Property 4.1 *For any B, E, $\vec{e}$ and indexed set $\mathcal{E}_{\vec{D}}$ the following holds:*

$$\mathcal{E}_{\vec{D}}^{(B,E,\vec{e})} \sqsubseteq \mathcal{E}_{\vec{D}} \;\equiv\; \forall \mathcal{V}.(\|B\|_{\mathcal{V}} = \mathsf{tt} \;\rightarrow\; \|E\|_{\mathcal{V}} \in \mathcal{E}_{\|\vec{e}\|_{\mathcal{V}}})$$

Lemma 4.2 (Reduction lemma) *For any monotone* $f : [\vec{D} \to \wp(S)] \to [\vec{D} \to \wp(S)]$ *and any* $U_{\vec{D}} : \vec{D} \to \wp(S)$ *:*

$$U_{\vec{D}} \sqsubseteq \nu f \;\equiv\; U_{\vec{D}} \sqsubseteq f(\nu\mathcal{E}_{\vec{D}}.(U_{\vec{D}} \sqcup f(\mathcal{E}_{\vec{D}})))$$

Proof. Our proof is based on the following two relationships:

$$\begin{array}{lll} (1) & U_{\vec{D}} \sqcup f(\nu\mathcal{E}_{\vec{D}}.(U_{\vec{D}} \sqcup f(\mathcal{E}_{\vec{D}}))) \;=\; \nu\mathcal{E}_{\vec{D}}.(U_{\vec{D}} \sqcup f(\mathcal{E}_{\vec{D}})) & \{\text{fix-point property}\} \\ (2) & \nu f \;\sqsubseteq\; \nu\mathcal{E}_{\vec{D}}.(U_{\vec{D}} \sqcup f(\mathcal{E}_{\vec{D}})) & \{\text{easy to show}\} \end{array}$$

($\to$) Follows immediately from (2) and monotonicity of f.

($\leftarrow$) Let $U_{\vec{D}} \sqsubseteq f(\nu\mathcal{E}_{\vec{D}}.(U_{\vec{D}} \sqcup f(\mathcal{E}_{\vec{D}})))$. Then $f(\nu\mathcal{E}_{\vec{D}}.(U_{\vec{D}} \sqcup f(\mathcal{E}_{\vec{D}}))) = \nu\mathcal{E}_{\vec{D}}.(U_{\vec{D}} \sqcup f(\mathcal{E}_{\vec{D}}))$ because of (1), and hence $\nu\mathcal{E}_{\vec{D}}.(U_{\vec{D}} \sqcup f(\mathcal{E}_{\vec{D}}))$ is a fix-point of f and is thereby less or equal to νf since the latter is the greatest fix-point of f. It follows by (2), that $\nu f \;=\; \nu\mathcal{E}_{\vec{D}}.(U_{\vec{D}} \sqcup f(\mathcal{E}_{\vec{D}}))$ and consequently:

$$U_{\vec{D}} \sqsubseteq f(\nu\mathcal{E}_{\vec{D}}.(U_{\vec{D}} \sqcup f(\mathcal{E}_{\vec{D}}))) = \nu\mathcal{E}_{\vec{D}}.(U_{\vec{D}} \sqcup f(\mathcal{E}_{\vec{D}})) = \nu f$$

which completes the proof. □

Corollary 4.3 *The following equivalence holds for any predicate* Π *and any valuation* $\mathcal{V}$*:*

$$\mathcal{E}^l_{\vec{D}} \sqsubseteq \|\nu Z\{L\}.\Pi\|_{\mathcal{V}} \;\equiv\; \mathcal{E}^l_{\vec{D}} \sqsubseteq \sqcup L \sqcup \|\Pi[\nu Z\{l, L\}.\Pi/Z]\|_{\mathcal{V}}$$

Proof. The equivalence is established as follows:

$$\begin{array}{lll} & \mathcal{E}^l_{\vec{D}} \sqsubseteq \|\nu Z\{L\}.\Pi\|_{\mathcal{V}} & \\ \equiv & \mathcal{E}^l_{\vec{D}} \sqsubseteq \nu\mathcal{E}_{\vec{D}}.(\sqcup L \sqcup \|\Pi\|_{\mathcal{V}[\mathcal{E}_{\vec{D}}/Z]}) & \{\text{Def. tagged predicates}\} \\ \equiv & \mathcal{E}^l_{\vec{D}} \sqsubseteq \sqcup L \sqcup \|\Pi\|_{\mathcal{V}[\nu\mathcal{E}_{\vec{D}}.((\sqcup L \sqcup \mathcal{E}^l_{\vec{D}}) \sqcup \|\Pi\|_{\mathcal{V}[\mathcal{E}_{\vec{D}}/Z]})/Z]} & \{\text{Reduction Lemma}\} \\ \equiv & \mathcal{E}^l_{\vec{D}} \sqsubseteq \sqcup L \sqcup \|\Pi\|_{\mathcal{V}[\|\nu Z\{l,L\}.\Pi\|_{\mathcal{V}}/Z]} & \{\text{Def. tagged predicates}\} \\ \equiv & \mathcal{E}^l_{\vec{D}} \sqsubseteq \sqcup L \sqcup \|\Pi[\nu Z\{l, L\}.\Pi/Z]\|_{\mathcal{V}} & \{\text{Prop. of substitution}\} \end{array}$$

□

Our system is sound, as the following theorem states.

Theorem 4.4 (Soundness) *If sequent $B \vdash E : \Phi$ is derivable in the proof system, then $B \models E : \Phi$ holds.*

Proof. It is sufficient to show by case analysis that all rules are individually sound. The result then follows by induction on the height of the derivation tree. We give only the more interesting cases here.

Case E(**if**). Let the side condition $b \Rightarrow b'$ hold. Then:

$$
\begin{array}{lll}
 & b \models E : \Phi & \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}}) & \{\text{Def. } \models\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|b'\|_{\mathcal{V}} = \mathsf{tt} \wedge \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}}) & \{\ b \Rightarrow b'\ \} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|\textbf{if } b' \textbf{ then } E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}}) & \{\text{Def. transition rules}\} \\
\equiv & b \models \textbf{if } b' \textbf{ then } E : \Phi & \{\text{Def. } \models\}
\end{array}
$$

Case E($\stackrel{\Delta}{=}$). Let the side condition $A(\vec{x}) \stackrel{\Delta}{=} E$ hold. Then:

$$
\begin{array}{lll}
 & b \models E[\vec{e}/\vec{x}] : \Phi & \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E[\vec{e}/\vec{x}]\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}}) & \{\text{Def. } \models\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow E[[\![\vec{e}[\mathcal{V}]]\!]/\vec{x}] \in \|\Phi\|_{\mathcal{V}}) & \{\text{Def. denotation}\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow A([\![\vec{e}[\mathcal{V}]]\!]) \in \|\Phi\|_{\mathcal{V}}) & \{\text{Def. transition rules}\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|A(\vec{e})\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}}) & \{\text{Def. denotation}\} \\
\equiv & b \models A(\vec{e}) : \Phi & \{\text{Def. } \models\}
\end{array}
$$

Case $\Phi(\forall)$. Let y be fresh in $b \vdash E : \forall x.\Phi$. Then:

$$
\begin{array}{lll}
 & b \models E : \Phi[y/x] & \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\Phi[y/x]\|_{\mathcal{V}}) & \{\text{Def. } \models\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}[\mathcal{V}(y)/x]}) & \{\text{Substitution}\} \\
\equiv & \forall \mathcal{V}.\forall d \in D.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}[d/x]}) & \{y \text{ - fresh}\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \forall d \in D.\, \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}[d/x]}) & \{\text{Calculus}\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \bigcap\limits_{d \in D} \|\Phi\|_{\mathcal{V}[d/x]}) & \{\text{Calculus}\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\forall x.\Phi\|_{\mathcal{V}}) & \{\text{Def. denotation}\} \\
\equiv & b \models E : \forall x.\Phi & \{\text{Def. } \models\}
\end{array}
$$

Case $\Phi(\exists)$.

$$
\begin{array}{lll}
& b \models E : \Phi[e/x] & \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\Phi[e/x]\|_{\mathcal{V}}) & \{\text{Def. } \models\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}[\|e\|_{\mathcal{V}}/x]}) & \{\text{Substitution}\} \\
\rightarrow & \forall \mathcal{V}.\exists d \in D.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}[d/x]}) & \{\|e\|_{\mathcal{V}} \in D\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \exists d \in D.\, \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}[d/x]}) & \{\text{Calculus}\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \bigcup_{d \in D} \|\Phi\|_{\mathcal{V}[d/x]}) & \{\text{Calculus}\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\exists x.\Phi\|_{\mathcal{V}}) & \{\text{Def. denotation}\} \\
\equiv & b \models E : \exists x.\Phi & \{\text{Def. } \models\}
\end{array}
$$

Case $\mathrm{E}\Phi(\tau, \langle\rangle)$.

$$
\begin{array}{lll}
& b \models E : \Phi & \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}}) & \{\text{Def. } \models\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|\tau.E\|_{\mathcal{V}} \in \|\langle\tau\rangle\|_{\mathcal{V}}\, \|\Phi\|_{\mathcal{V}}) & \{\text{Def. transition rules}\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|\tau.E\|_{\mathcal{V}} \in \|\langle\tau\rangle\, \Phi\|_{\mathcal{V}}) & \{\text{Def. denotation}\} \\
\equiv & b \models \tau.E : \langle\tau\rangle\, \Phi & \{\text{Def. } \models\}
\end{array}
$$

Case $\mathrm{E}\Phi(a, \langle\rangle)$.

$$
\begin{array}{lll}
& b \models E[\vec{e}/\vec{x}] : \Phi & \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E[\vec{e}/\vec{x}]\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}}) & \{\text{Def. } \models\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|a(\vec{x}).E\|_{\mathcal{V}} \in \|\langle a(\vec{e})\rangle\|_{\mathcal{V}}\, \|\Phi\|_{\mathcal{V}}) & \{\text{Def. transition rules}\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|a(\vec{x}).E\|_{\mathcal{V}} \in \|\langle a(\vec{e})\rangle\, \Phi\|_{\mathcal{V}}) & \{\text{Def. denotation}\} \\
\equiv & b \models a(\vec{x}).E : \langle a(\vec{e})\rangle\, \Phi & \{\text{Def. } \models\}
\end{array}
$$

Case $\mathrm{E}\Phi(\overline{a}, [\,])$.

$$
\begin{array}{lll}
& b \wedge (\vec{e} = \vec{e}\,') \models E : \Phi & \\
\equiv & \forall \mathcal{V}.(\|b \wedge (\vec{e} = \vec{e}\,')\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}}) & \{\text{Def. } \models\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \wedge \|\vec{e} = \vec{e}\,'\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}}) & \{\text{Def. denotation}\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow (\|\vec{e} = \vec{e}\,'\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}})) & \{\text{Calculus}\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow (\|\vec{e}\|_{\mathcal{V}} = \|\vec{e}\,'\|_{\mathcal{V}} \rightarrow \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}})) & \{\text{Def. denotation}\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|\overline{a}(\vec{e}).E\|_{\mathcal{V}} \in \|[\overline{a}(\vec{e}\,')]\|_{\mathcal{V}}\, \|\Phi\|_{\mathcal{V}}) & \{\text{Def. transition rules}\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|\overline{a}(\vec{e}).E\|_{\mathcal{V}} \in \|[\overline{a}(\vec{e}\,')]\, \Phi\|_{\mathcal{V}}) & \{\text{Def. denotation}\} \\
\equiv & b \models \overline{a}(\vec{e}).E : [\overline{a}(\vec{e}\,')]\, \Phi & \{\text{Def. } \models\}
\end{array}
$$

Case $\mathrm{E}\Phi(\Sigma, \langle\rangle)$.

$$
\begin{array}{lll}
& b \models E[\vec{e}/\vec{x}] : \langle\pi\rangle\,\Phi & \\
\equiv & \forall\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E[\vec{e}/\vec{x}]\|_{\mathcal{V}} \in \|\langle\pi\rangle\,\Phi\|_{\mathcal{V}}) & \{\text{Def. } \models\} \\
\equiv & \forall\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E[\|\vec{e}\|_{\mathcal{V}}/\vec{x}]\|_{\mathcal{V}} \in \|\langle\pi\rangle\|_{\mathcal{V}}\,\|\Phi\|_{\mathcal{V}})) & \{\text{Def. denotation}\} \\
\rightarrow & \forall\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|\Sigma\vec{x}\,E\|_{\mathcal{V}} \in \|\langle\pi\rangle\|_{\mathcal{V}}\,\|\Phi\|_{\mathcal{V}})) & \{\text{Def. transition rules}\} \\
\equiv & \forall\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|\Sigma\vec{x}\,E\|_{\mathcal{V}} \in \|\langle\pi\rangle\,\Phi\|_{\mathcal{V}} & \{\text{Def. denotation}\} \\
\equiv & b \models \Sigma\vec{x}\,E : \langle\pi\rangle\,\Phi & \{\text{Def. } \models\}
\end{array}
$$

Case E$\Phi(\Sigma, [\,])$.

$$
\begin{array}{lll}
& b \models E[\vec{y}/\vec{x}] : [\pi]\,\Phi & \\
\equiv & \forall\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E[\vec{y}/\vec{x}]\|_{\mathcal{V}} \in \|[\pi]\,\Phi\|_{\mathcal{V}}) & \{\text{Def. } \models\} \\
\equiv & \forall\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E[\|\vec{y}\|_{\mathcal{V}}/\vec{x}]\|_{\mathcal{V}} \in \|[\pi]\|_{\mathcal{V}}\,\|\Phi\|_{\mathcal{V}})) & \{\text{Def. denotation}\} \\
\equiv & \forall\mathcal{V}.\forall\vec{d} \in \vec{D}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \left\|E[\vec{d}/\vec{x}]\right\|_{\mathcal{V}} \in \|[\pi]\|_{\mathcal{V}}\,\|\Phi\|_{\mathcal{V}})) & \{\vec{y} \text{ - fresh}\} \\
\equiv & \forall\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \forall\vec{d} \in \vec{D}.\left\|E[\vec{d}/\vec{x}]\right\|_{\mathcal{V}} \in \|[\pi]\|_{\mathcal{V}}\,\|\Phi\|_{\mathcal{V}})) & \{\text{Calculus}\} \\
\equiv & \forall\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|\Sigma\vec{x}\,E\|_{\mathcal{V}} \in \|[\pi]\|_{\mathcal{V}}\,\|\Phi\|_{\mathcal{V}})) & \{\text{Def. transition rules}\} \\
\equiv & \forall\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|\Sigma\vec{x}\,E\|_{\mathcal{V}} \in \|[\pi]\,\Phi\|_{\mathcal{V}} & \{\text{Def. denotation}\} \\
\equiv & b \models \Sigma\vec{x}\,E : [\pi]\,\Phi & \{\text{Def. } \models\}
\end{array}
$$

Case G(Sub).

$$
\begin{array}{lll}
& b \wedge (\vec{x} = \vec{e}) \models E : \Phi & \{\text{Def. } \models\} \\
\equiv & \forall\mathcal{V}.(\|b \wedge (\vec{x} = \vec{e})\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}}) & \{\text{Def. denotation}\} \\
\equiv & \forall\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \wedge \|\vec{x} = \vec{e}\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}}) & \{\text{Def. denotation}\} \\
\equiv & \forall\mathcal{V}.(\|\vec{x} = \vec{e}\|_{\mathcal{V}} = \mathsf{tt} \rightarrow (\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}})) & \{\text{Calculus}\} \\
\equiv & \forall\mathcal{V}.(\|b\|_{\mathcal{V}[\|\vec{e}\|_{\mathcal{V}}/\vec{x}]} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}[\|\vec{e}\|_{\mathcal{V}}/\vec{x}]} \in \|\Phi\|_{\mathcal{V}[\|\vec{e}\|_{\mathcal{V}}/\vec{x}]}) & \{\vec{x} \text{ - fresh}\} \\
\equiv & \forall\mathcal{V}.(\|b[\vec{e}/\vec{x}]\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E[\vec{e}/\vec{x}]\|_{\mathcal{V}} \in \|\Phi[\vec{e}/\vec{x}]\|_{\mathcal{V}}) & \{\text{Substitution}\} \\
\equiv & b[\vec{e}/\vec{x}] \models E[\vec{e}/\vec{x}] : \Phi[\vec{e}/\vec{x}] & \{\text{Def. } \models\}
\end{array}
$$

The soundness of $\Phi(\nu 0)$ and $\Phi(\nu 1)$ can be established as follows (assuming $\nu Z\{L\}.\Pi$ is closed):

$$
\begin{array}{lll}
& b \models E : (\nu Z\{L\}.\Pi)\vec{e} & \\
\equiv & \forall\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|(\nu Z\{L\}.\Pi)\vec{e}\|_{\mathcal{V}}) & \{\text{Def. } \models\} \\
\equiv & \forall\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\nu Z\{L\}.\Pi\|\,\|\vec{e}\|_{\mathcal{V}}) & \{\text{Def. denotation}\} \\
\equiv & \mathcal{E}^{(b,E,\vec{e})}_{\vec{D}} \sqsubseteq \|\nu Z\{L\}.\Pi\| & \{\text{Property 4.1}\} \\
\equiv & \mathcal{E}^{(b,E,\vec{e})}_{\vec{D}} \sqsubseteq \sqcup L \sqcup \|\Pi[\nu Z\{l, L\}.\Pi/Z]\| & \{\text{Corollary 4.3}\}
\end{array}
$$

From this it follows that:

$$
\begin{array}{lll}
 & (b, E, \vec{e}) \in L & \\
\rightarrow & \mathcal{E}_{\vec{D}}^{(b,E,\vec{e})} \sqsubseteq \sqcup L & \{\text{Def. } \sqcup L\} \\
\rightarrow & \mathcal{E}_{\vec{D}}^{(b,E,\vec{e})} \sqsubseteq \sqcup L \sqcup \|\Pi[\nu Z\{l, L\}.\Pi/Z]\| & \{\text{Monotonicity}\} \\
\equiv & b \models E : (\nu Z\{L\}.\Pi)\vec{e} & \{\text{Above equivalence}\}
\end{array}
$$

i.e., that $\Phi(\nu 0)$ is sound. If $\mathcal{C}_\nu$ holds, i.e., $\hat{E} \notin \hat{L}$, then $\mathcal{E}_{\vec{D}}^{(b,E,\vec{e})} \sqcap \sqcup L = \lambda \vec{d}.\{\}$ follows directly from the definitions, and hence:

$$
\begin{array}{lll}
 & b \models E : (\Pi[\nu Z\{l, L\}.\Pi/Z])\vec{e} & \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|(\Pi[\nu Z\{l, L\}.\Pi/Z])\vec{e}\|_{\mathcal{V}}) & \{\text{Def. } \models\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\Pi[\nu Z\{l, L\}.\Pi/Z]\| \, \|\vec{e}\|_{\mathcal{V}}) & \{\text{Def. denotation}\} \\
\equiv & \mathcal{E}_{\vec{D}}^{(b,E,\vec{e})} \sqsubseteq \|\Pi[\nu Z\{l, L\}.\Pi/Z]\| & \{\text{Property 4.1}\} \\
\equiv & \mathcal{E}_{\vec{D}}^{(b,E,\vec{e})} \sqsubseteq \sqcup L \sqcup \|\Pi[\nu Z\{l, L\}.\Pi/Z]\| & \{\hat{E} \notin \hat{L}\} \\
\equiv & b \models E : (\nu Z\{L\}.\Pi)\vec{e} & \{\text{Above equivalence}\}
\end{array}
$$

i.e., $\Phi(\nu 1)$ is sound.

Case $\Phi(\mu 0)$ is similar to $\Phi(\nu 0)$.

Case $\Phi(\mu 1)$ is the most involved one. The idea behind it, however, is rather simple, and is based on the well-known principle of Noetherian induction. Assume condition $\mathcal{C}_\mu$ holds for some $\vec{x}$, $\vec{c}$, $\vec{x}'$, and b'', and assume the notation chosen in the explanation of the condition. One could consider the sequent $b \models E : (\mu Z\{L\}.\Pi)\vec{e}$ as a predicate φ on $\vec{x}$, or more exactly, as $\forall \vec{x}.\varphi(\vec{x})$. Given a well-founded relation $\prec$ on $\vec{D}$, one could use Noetherian induction to establish $\forall \vec{x}.\varphi(\vec{x})$ if one could show that for arbitrary constants $\vec{c}$, $\forall \vec{x}' \prec \vec{c}.\varphi(\vec{x}')$ implies $\varphi(\vec{c})$. This is exactly what $b' \models E' : (\Pi[\mu Z\{l, L\}.\Pi/Z])\vec{e}\,'$ amounts to, since the assumption $\forall \vec{x}' \prec \vec{c}.\varphi(\vec{x}')$ is encoded as a triple l in the tag, and $\varphi(\vec{c})$ corresponds to $b' \models E' : (\Pi[\mu Z\{L\}.\Pi/Z])\vec{e}''$.

We now give a formal version of the proof. Let l' denote $(b, E, \vec{e})$.

$$
\begin{array}{lll}
& b' \models E' : (\Pi[\mu Z\{l, L\}.\Pi/Z])\vec{e}\,' & \\
\equiv & \forall \mathcal{V}.(\|b'\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E'\|_{\mathcal{V}} \in \|(\Pi[\mu Z\{l, L\}.\Pi/Z])\vec{e}\,'\|_{\mathcal{V}}) & \{\text{Def. } \models\} \\
\equiv & \forall \mathcal{V}.(\|b'\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E'\|_{\mathcal{V}} \in \|\Pi[\mu Z\{l, L\}.\Pi/Z]\| \, \|\vec{e}\,'\|_{\mathcal{V}}) & \{\text{Def. denot.}\} \\
\equiv & \mathcal{E}_{\vec{D}}^{l'[\vec{c}/\vec{x}]} \sqsubseteq \|\Pi[\mu Z\{l, L\}.\Pi/Z]\| & \{\text{Prop. 4.1}\} \\
\equiv & \forall \vec{d} \in \vec{D}. \left(\mathcal{E}_{\vec{D}}^{l'[\vec{d}/\vec{x}]} \sqsubseteq \left\|\Pi[\mu Z\{l[\vec{d}/\vec{c}], L\}.\Pi/Z]\right\|\right) & \{\text{Calculus}\} \\
\rightarrow & \forall \vec{d} \in \vec{D}. \left(\mathcal{E}_{\vec{D}}^{l'[\vec{d}/\vec{x}]} \sqsubseteq \left\|\mu Z\{l[\vec{d}/\vec{c}], L\}.\Pi\right\|\right) & \{\text{Monoton.}\} \\
\rightarrow & \forall \vec{d} \in \vec{D}. \left(\mathcal{E}_{\vec{D}}^{l[\vec{d}/\vec{c}]} \sqsubseteq \|\mu Z\{L\}.\Pi\| \rightarrow \mathcal{E}_{\vec{D}}^{l'[\vec{d}/\vec{x}]} \sqsubseteq \|\mu Z\{L\}.\Pi\|\right) & \{5.1\ [\text{And93}]\} \\
\rightarrow & \forall \vec{d} \in \vec{D}.(\forall \vec{d}\,' \prec \vec{d}. \left(\mathcal{E}_{\vec{D}}^{l'[\vec{d}\,'/\vec{x}]} \sqsubseteq \|\mu Z\{L\}.\Pi\|\right) \rightarrow & \\
& \mathcal{E}_{\vec{D}}^{l'[\vec{d}/\vec{x}]} \sqsubseteq \|\mu Z\{L\}.\Pi\|) & \{\text{Cond. } \mathcal{C}_\mu\} \\
\rightarrow & \forall \vec{d} \in \vec{D}. \left(\mathcal{E}_{\vec{D}}^{l'[\vec{d}/\vec{x}]} \sqsubseteq \|\mu Z\{L\}.\Pi\|\right) & \{\text{Noeth. ind.}\} \\
\equiv & \mathcal{E}_{\vec{D}}^{l'} \sqsubseteq \|\mu Z\{L\}.\Pi\| & \{\text{Calculus}\} \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|(\mu Z\{L\}.\Pi)\vec{e}\|_{\mathcal{V}}) & \{\text{Prop. 4.1}\} \\
\equiv & b \models E : (\mu Z\{L\}.\Pi)\vec{e} & \{\text{Def. } \models\}
\end{array}
$$

This concludes the proof of soundness.

□

4.2 Completeness

In this section we investigate completeness of the proof system we propose. More precisely, we establish the conditions under which completeness holds. Indeed, only a relativised completeness result is obtainable, since in the general case of an infinite domain of values and arbitrary languages for Boolean and value expressions there cannot be a sound, and at the same time complete, proof system. We therefore assume that all reasoning about the value domain is performed externally to our proof system. But this is not the only restriction we shall impose. Instead of listing these restrictions here at once, we shall explicate them one-by-one during our discussion, and summarize them at the end of the section. Here it suffices to say that, since we are not giving proof rules for parallel composition, restriction and relabelling, we consider only CCS terms without these combinators.

Proving completeness of a proof system can in many cases be established using the

following argument. First, one shows that for each valid sequent there is a rule, or a set of rules, and a way of applying these "backwards", so that the resulting sequents are guaranteed to be valid as well. If one can prove that following the prescribed way of applying the rules is guaranteed to terminate, then one has essentially established completeness, since there is no way of terminating (under the above conditions) other than by eliminating all sub-goals by applying axiom rules, i.e., by constructing a *proof*. Note that it is only required that there *be* a way of applying the rules, and it is not required that we know *how* to find it effectively.[1]

For some simple logics there are proof systems all proof rules of which have premises which are in some sense simpler than the corresponding conclusions. For example this is the case when the premises of each rule are strict sub-formulae of the rule's conclusion, or are formulae of a strictly smaller size. In this case all proof trees are finite, and termination is guaranteed. In our proof system, however, this simple argument is not applicable, since we have rules for unfolding process constants and fixpoint formulae. This unfolding increases the size of the process, respectively the formula. Nevertheless, a more complicated completeness argument along the same lines can still be made even in this case.

In the following subsection we investigate the conditions under which for every valid sequent there is an applicable proof rule or set of rules yielding valid sub-goals. From this investigation we extract a schema for applying the proof rules, i.e., for constructing what one might call *canonical proofs*. In the second subsection we investigate the conditions (i.e., restrictions) under which the construction is guaranteed to terminate. The last subsection summarizes all these restrictions, yielding a relative completeness result.

[1] If the latter is the case (as it is in fact in many proof systems, but not in ours), then one has also established *semi-decidability*, namely the existence of a procedure for proving valid sequents. For the system to be *decidable*, the decision procedure has also to terminate (with a negative response) when the initial sequent is not valid.

4.2.1 Canonical Proofs

To show that for every valid sequent there is an applicable proof rule or set of rules yielding valid sub-goals, we look one level into the structure of the process and the formula in the sequent, and make sure that every possible case has been covered. Naturally, the inference rules which are backward sound substantially simplify the process of finding how canonical proofs should look. Most of our rules are indeed backward sound, as can be seen from our proof of soundness where many of the individual soundness proofs were established through sequences of equivalences. Here, we present the remaining cases only.

Case $(E,\, \Phi_1 \vee \Phi_2)$. Let $b \vdash E : \Phi_1 \vee \Phi_2$ be valid, i.e., $b \models E : \Phi_1 \vee \Phi_2$. Let b' and b'' be the weakest Boolean expressions expressible in the given language such that $b' \models E : \Phi_1$ and $b'' \models E : \Phi_2$. It is natural to require, that the language for Boolean expressions be *complete*, i.e., that every function of type $\vec{D} \to \{\mathsf{ff}, \mathsf{tt}\}$ is expressible in this language. In this case $b \Rightarrow b' \vee b''$ holds, and hence $b \vdash E : \Phi_1 \vee \Phi_2$ can be reduced by applying G(Cut), $\Phi(\vee l)$ and $\Phi(\vee r)$ to the valid sequents $b' \vdash E : \Phi_1$ and $b'' \vdash E : \Phi_2$.

Case $(E_1 + E_2,\, \langle\pi\rangle\, \Phi)$ is similar.

Case $(E,\, \exists x.\Phi)$. Let $b \vdash E : \exists x.\Phi$ be valid. It is also natural to require that the language for value expressions be *complete*, i.e., that every function of type $\vec{D} \to D$ be expressible in this language. Then, there is an expression e yielding for every valuation an "appropriate" value for the free occurrences of x in Φ, so that the sequent $b \vdash E : \Phi[e/x]$ resulting from applying rule $\Phi(\exists)$ is also valid.

Case $(\Sigma\vec{x}\; E,\, \langle\pi\rangle\, \Phi)$ is similar.

Case $(\mathbf{if}\; b\; \mathbf{then}\; E,\, \langle\pi\rangle\, \Phi)$. We have:

$$
\begin{array}{ll}
 & b \models \mathbf{if}\; b'\; \mathbf{then}\; E : \langle\pi\rangle\, \Phi \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|\mathbf{if}\; b'\; \mathbf{then}\; E\|_{\mathcal{V}} \in \|\langle\pi\rangle\|_{\mathcal{V}}\, \|\Phi\|_{\mathcal{V}}) \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|b'\|_{\mathcal{V}} = \mathsf{tt} \wedge \|E\|_{\mathcal{V}} \in \|\langle\pi\rangle\|_{\mathcal{V}}\, \|\Phi\|_{\mathcal{V}}) \\
\equiv & \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|b'\|_{\mathcal{V}} = \mathsf{tt}) \wedge \forall \mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\langle\pi\rangle\|_{\mathcal{V}}\, \|\Phi\|_{\mathcal{V}}) \\
\equiv & b \Rightarrow b' \wedge b \models E : \langle\pi\rangle\, \Phi
\end{array}
$$

which implies that a valid sequent $b \vdash \mathbf{if}\ b'\ \mathbf{then}\ E : \langle\pi\rangle\,\Phi$ is reducible, by applying E(**if**), to the valid sequent $b \vdash E : \langle\pi\rangle\,\Phi$.

Case $(\overline{a}(\vec{e}).E,\ \langle\overline{a}(\vec{e}\,')\rangle\,\Phi)$. The reason why we have to consider this case is that rule E$\Phi(\overline{a}, \langle\rangle)$ covers only the case when $\vec{e} = \vec{e}\,'$. In the general case we have:

$$
\begin{array}{ll}
 & b \models \overline{a}(\vec{e}).E : \langle\overline{a}(\vec{e}\,')\rangle\,\Phi \\
\equiv & \forall\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|\overline{a}(\vec{e}).E\|_{\mathcal{V}} \in \|\langle\overline{a}(\vec{e}\,')\rangle\,\Phi\|_{\mathcal{V}}) \\
\equiv & \forall\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|\vec{e}\|_{\mathcal{V}} = \|\vec{e}\,'\|_{\mathcal{V}} \wedge \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}}) \\
\equiv & \forall\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|\vec{e}\|_{\mathcal{V}} = \|\vec{e}\,'\|_{\mathcal{V}}) \wedge \forall\mathcal{V}.(\|b\|_{\mathcal{V}} = \mathsf{tt} \rightarrow \|E\|_{\mathcal{V}} \in \|\Phi\|_{\mathcal{V}}) \\
\equiv & \vec{e} \equiv_b \vec{e}\,' \wedge b \models E : \Phi
\end{array}
$$

which implies that a valid sequent $b \vdash \overline{a}(\vec{e}).E : \langle\overline{a}(\vec{e}\,')\rangle\,\Phi$ can be reduced, by applying G($\equiv$) followed by E$\Phi(\overline{a}, \langle\rangle)$, to the valid sequent $b \vdash E : \Phi$.

Case $(E, (\nu Z\{L\}.\Pi)\vec{e})$. This case has to be partitioned into two sub-cases[2], namely $\hat{E} \notin \hat{L}$ and $\hat{E} \in \hat{L}$. The first case is handled by rule $\Phi(\nu 1)$, the backward soundness of which is obvious from the proof of its soundness. The second case, however, is not completely covered by rule $\Phi(\nu 0)$, since its side condition $\mathcal{C}_0$ has the stronger requirement that $(b, E, \vec{e}) \in L$. A certain discipline of applying rules $\Phi(\nu 1)$ and $\Phi(\nu 0)$ is required to guarantee that the case $\hat{E} \in \hat{L}$ is reducible, by applying general rules only, to the case $(b, E, \vec{e}) \in L$. Before proposing one such discipline, an important note is due. So far, when saying that a sequent $b \vdash E : \Phi$ is *valid*, we meant that $b \models E : \Phi$ holds. What is actually important, however, is that $b \models E : \Phi^{tf}$ holds, where Φ^{tf} is the *tag-free* version of Φ, since tags are not part of the specification language and are only introduced to facilitate the proofs. We call this latter type of validity *tf-validity*. From the semantics of tags follows that tf-validity implies validity. Since in our proofs we always start with tag-free formulae, and the way we apply the proof rules outlined so far preserves not just validity, but tf-validity as well, we have only to make sure that the rule G(Cut) and the derived rule G($\Rightarrow$)

[2]Recall that the "hat" operator $\hat{}$ abstracts from the values and the expressions forming and using them, so the triples in tags are reduced to value-free processes.

are used in a way that does not widen the assumption of a sequent "too much" to lose tf-validity. After this note we can describe our discipline of using the greatest fixpoint rules. Assume $\hat{E} \notin \hat{L}$. Let $\vec{e}_1$ consist of all expressions occurring in E, and let

$$b \vdash E(\vec{e}_1) : (\nu Z\{L\}.\Pi)\vec{e}_2$$

be tf-valid. We apply rule G(Sub) to put the process and the formula in a form easier to be handled later:

$$b \wedge (\vec{x}_1 = \vec{e}_1) \wedge (\vec{x}_2 = \vec{e}_2) \vdash E(\vec{x}_1) : (\nu Z\{L\}.\Pi)\vec{x}_2$$

We now use G($\Rightarrow$) to perform the maximal possible widening still yielding a tf-valid sequent; let the resulting assumption be denoted b_1:

$$b_1 \vdash E(\vec{x}_1) : (\nu Z\{L\}.\Pi)\vec{x}_2$$

and unfold the fixpoint predicate using rule $\Phi(\nu 1)$ with $l \stackrel{\Delta}{=} (b_1, E(\vec{x}_1), \vec{x}_2)$:

$$b_1 \vdash E(\vec{x}_1) : (\Pi[\nu Z\{l, L\}.\Pi/Z])\vec{x}_2$$

Suppose that later in the construction of the proof we reach a tf-valid sub-goal of the form:

$$b_2 \vdash E(\vec{e}_3) : (\nu Z\{L'\}.\Pi')\vec{e}_4$$

where Π' is as Π but possibly differing in the tag sets. Then it is guaranteed that $l \in L'$ and consequently also $\hat{E} \in \hat{L}'$ holds. We apply rule G(Sub) to obtain, possibly after some renaming of variables, the tf-valid sequent:

$$b_2 \wedge (\vec{x}_1 = \vec{e}_3) \wedge (\vec{x}_2 = \vec{e}_4) \vdash E(\vec{x}_1) : (\nu Z\{L'\}.\Pi')\vec{x}_2$$

We had a similar goal (up to Boolean expressions) above, and due to the maximal widening performed there it is the case that $b_2 \wedge (\vec{x}_1 = \vec{e}_3) \wedge (\vec{x}_2 = \vec{e}_4) \Rightarrow b_1$. We can hence apply rule G($\Rightarrow$) to obtain:

$$b_1 \vdash E(\vec{x}_1) : (\nu Z\{L'\}.\Pi')\vec{x}_2$$

The triple $(b_1, E(\vec{x}_1), \vec{x}_2)$ is already in L', and therefore the axiom rule $\Phi(\nu 0)$ is applicable here, thus eliminating the sub-goal. The so presented discipline of using the greatest fixpoint rules justifies their choice. As the example proofs in the previous chapter suggest, in the above scheme, rules G($\equiv$) and G(US) can be used instead of G(Sub) and G($\Rightarrow$), respectively, giving the proofs a more syntactic flavour[3].

Case $(E, (\mu Z\{L\}.\Pi)\vec{e})$. This case is treated similarly to the previous one, but with some additional complications. Assume again $\hat{E} \notin \hat{L}$, and let

$$b \vdash E(\vec{e}_1) : (\mu Z\{L\}.\Pi)\vec{e}_2$$

be tf-valid. We apply rule G(Sub) to obtain:

$$b \wedge (\vec{x}_1 = \vec{e}_1) \wedge (\vec{x}_2 = \vec{e}_2) \vdash E(\vec{x}_1) : (\mu Z\{L\}.\Pi)\vec{x}_2$$

and perform the maximal possible widening using G($\Rightarrow$) still resulting in a tf-valid sequent:

$$b_1 \vdash E(\vec{x}_1) : (\mu Z\{L\}.\Pi)\vec{x}_2$$

[3]It is an interesting problem under what conditions this is always the case.

We unfold the fixpoint predicate using $\Phi(\mu 1)$ as follows. We choose the variables in $\vec{x}_1$ and $\vec{x}_2$ as induction variables, and let $\vec{x}'_1$ and $\vec{x}'_2$ be their primed versions. To define a suitable b'' (according to the notation adopted in the description of the rules) we first define the mapping $cl : \vec{D}_1 \times \vec{D}_2 \to Ord$ so that $cl(\vec{d}_1, \vec{d}_2)$ is the least ordinal λ such that:

$$b_1[\vec{d}_1/\vec{x}_1, \vec{d}_2/\vec{x}_2] \models E(\vec{d}_1) : (\mu^{\lambda} Z\{L\}.\Pi^{tf})\vec{d}_2$$

i.e., cl gives the *closure ordinal* for the last sequent under a given valuation. This mapping is well defined because of tf-validity of the sequent. Now, we construct b'' as a Boolean expression containing the variables in $\vec{x}_1$, $\vec{x}_2$, $\vec{x}'_1$, and $\vec{x}'_2$, so that:

$$b'' \equiv b_1[\vec{x}'_1/\vec{x}_1, \vec{x}'_2/\vec{x}_2] \wedge cl(\vec{x}'_1, \vec{x}'_2) \prec cl(\vec{x}_1, \vec{x}_2)$$

where $\prec$ is the usual (well-founded!) ordering on ordinals. The conditions that b'' has to satisfy according to the description of rule $\Phi(\mu 1)$ follow immediately from the construction. So, applying rule $\Phi(\mu 1)$ results in the tf-valid sequent:

$$b_1[\vec{c}_1/\vec{x}_1, \vec{c}_2/\vec{x}_2] \vdash E(\vec{c}_1) : (\Pi[\mu Z\{l, L\}.\Pi/Z])\vec{c}_2$$

where the triple l inserted in the tag is $(b''[\vec{c}_1/\vec{x}_1, \vec{c}_2/\vec{x}_2], E(\vec{x}'_1), \vec{x}'_2)$. Suppose now that later in the construction of the proof we reach a tf-valid sub-goal of the form:

$$b_2 \vdash E(\vec{e}_3) : (\mu Z\{L'\}.\Pi')\vec{e}_4$$

where Π' is as Π but possibly differing in the tag sets. Then it is guaranteed that $l \in L'$ and consequently also $\hat{E} \in \hat{L}'$ holds. We apply rule G(Sub) to obtain, possibly

after some renaming of variables, the tf-valid:

$$b_2 \wedge (\vec{x}'_1 = \vec{e}_3) \wedge (\vec{x}'_2 = \vec{e}_4) \vdash E(\vec{x}'_1) : (\mu Z\{L'\}.\Pi')\vec{x}'_2$$

Due to the maximal widening performed before, it is the case that:

$$b_2 \wedge (\vec{x}'_1 = \vec{e}_3) \wedge (\vec{x}'_2 = \vec{e}_4) \Rightarrow b_1[\vec{x}'_1/\vec{x}_1, \vec{x}'_2/\vec{x}_2]$$

holds here. Assume that after applying rule $\Phi(\mu 1)$ no strict widening was applied, i.e., if some widening rule was applied, this was done in the most conservative (equivalence preserving) way. Then, by going from the sequent at which $\Phi(\mu 1)$ was applied to the last sequent, we descend the fixpoint approximation hierarchy, and therefore, $cl(\vec{x}'_1, \vec{x}'_2) \prec cl(\vec{c}_1, \vec{c}_2)$, implying:

$$b_2 \wedge (\vec{x}'_1 = \vec{e}_3) \wedge (\vec{x}'_2 = \vec{e}_4) \Rightarrow b''[\vec{c}_1/\vec{x}_1, \vec{c}_2/\vec{x}_2]$$

holds here, and we can hence apply rule G($\Rightarrow$) to the last sequent to obtain:

$$b''[\vec{c}_1/\vec{x}_1, \vec{c}_2/\vec{x}_2] \vdash E(\vec{x}'_1) : (\mu Z\{L'\}.\Pi')\vec{x}'_2$$

But the triple $(b''[\vec{c}_1/\vec{x}_1, \vec{c}_2/\vec{x}_2], E(\vec{x}'_1), \vec{x}'_2)$ is already in L', and therefore the axiom rule $\Phi(\mu 0)$ is applicable here, thus eliminating the sub-goal.

From the above considerations we can extract the following proof discipline, yielding proofs that we can term *canonical*.[4] Starting from the goal sequent:

1. We apply logic rules until the formula becomes of the form $[\pi]\,\Phi$ or $\langle\pi\rangle\,\Phi$, or the resulting sub-goal can be discharged with an axiom. The rules for disjunction

[4] It should be noted that canonical proofs are not necessarily the most economic ones.

have possibly to be preceded by a suitable application of the cut rule, while the fixpoint rules have possibly to be preceded by applications of general rules according to the description given above.

2. Next, process and dynamic rules are applied until the process becomes of the form $\pi.E$, or the resulting sub-goal can be discharged with an axiom. The rules for binary choice have possibly to be preceded by a suitable application of the cut rule.

3. At this point, the process is of the form $\pi.E$, and the formula is of the form $[\pi']\,\Phi$ or $\langle\pi'\rangle\,\Phi$. The corresponding dynamic rule is applied, and the first step is re-entered. Rule $\mathrm{E}\Phi(\overline{a}, \langle\rangle)$ has possibly to be preceded by a suitable application of $\mathrm{G}(\equiv)$ to unify the respective expressions.

Unfortunately we made in case $(E\ ,\ (\mu Z\{L\}.\Pi)\vec{e})$ an assumption about widening, which interferes with the fixpoint reasoning we suggest in the above proof discipline. Hence we can only justify completeness for processes and formulae which do not require widening to be applied before unfolding of fixpoint formulae between an application of $\Phi(\mu 1)$ and the consequent application of $\Phi(\mu 0)$. These cases are not easy to characterise. We give here just one such case. Assume the process is finite-state (this would usually be the case when the domain of values is finite). Then it is sufficient to use $\Phi(\mu 1)$ for simple unfolding, i.e., without tagging [ASW94]. Induction can still be applied, but it could be viewed merely as a means of making some proofs shorter.

The completeness proof given in [And93] however suggests that with a more sophisticated, but far less intuitive, argument one could still prove completeness for the general case. We leave this as a topic for future investigation.

4.2.2 Termination

The proof discipline outlined in the preceding section has the property that it yields, provided the goal sequent is valid, and provided that it terminates, a proof for the goal sequent, i.e., a proof tree with the goal sequent as its root. The important question, in view of achieving completeness, becomes: under what conditions is this discipline guaranteed to terminate?

An important observation is that all rules other than rule E(**if**), the fixpoint rules and the general rules, reduce a sequent to new sequents with a strictly smaller size of the process and/or the formula, while none of the two is increased. This is true for any reasonable choice of the notion of size, assuming only that it ignores all Boolean or value expressions and the tags (e.g. the number of logic combinator occurrences in a formula gives such a measure). Termination of the proof discipline depends then only on whether process constants and fixpoint formulae can be unfolded infinitely many times.

To make sure that, when attempting to construct a canonical proof, we do not stay forever in the second step of the proof discipline, two natural restrictions can be made. First, we allow only finite sets of agent constants to be used when specifying a system. And second, we demand that, in the right-hand side of defining equations, agent constants appear only within the scope of some prefix operator. Expressions of this sort are termed *guarded* [Mil89]. Under these restrictions it is obvious that any sequence of applications of process and dynamic rules eventually leads to a sequent with a process term which is a prefix, i.e., of the form $\pi.E$.

The above two restrictions have the additional effect that the space of value-free abstractions $\hat{E}$ of agent expressions or sub-expressions E occurring in the specification of a system is guaranteed to be finite. Such processes are usually referred to as *finite-control*, or *bound*. An important consequence of this is that the tags are also guaranteed to be of finite length, since we always start with empty tags and add a triple to a tag only if its value-free abstraction is not already there, and there are only

finitely many of these. Consequently, fixpoint formulae can only be unfolded finitely many times!

As a result, under these restrictions the above described proof discipline is guaranteed to terminate whenever the goal sequent is valid, thus yielding a canonical proof.

4.2.3 Completeness Conditions

The two preceding sub-sections give sufficient conditions under which there is a canonical proof for every valid sequent. In other words, our proof system is guaranteed to be *complete* if:

First, the languages for constructing value expressions e and Boolean expressions b are *complete*, and the value domain is finite.

Second, the agent expressions in a system specification satisfy the following restrictions:

- agents are *sequential*, i.e., without concurrent composition;
- the set of agent constants used is finite;
- the defining agent expressions in agent constant definitions are *guarded*.

And third, the formula in the goal sequent are:

- tag-free;
- closed w.r.t. predicate variables;
- all its fixpoint sub-formulae are closed.

This concludes our analysis of correctness of the proof system proposed in the preceding chapter.

Chapter 5

Extensions

Motivated by the advantages the technique of *tagging* seems to offer in keeping fixpoint reasoning "local" (i.e., avoiding global rules), we investigate in this chapter applications of this technique to other settings. The first section proposes a way of tagging fixpoint formulae for a different kind of sequents and offers a semantics for such formulae. Inference rules for handling fixpoint formulae are proposed and shown sound. The second section considers the problem of "negative" tagging of least fixpoint formulae, which is a technique for preventing proof trees from growing unboundedly, i.e., for ensuring termination of proof search. This idea has been explored in [ASW94] for sequents of type $P \vdash \Phi$, where P is a single process. We investigate under what conditions this approach can be extended to sets of processes, which is important when value passing or parallel composition is considered.

5.1 Compositionality

The proof system presented and analysed in the previous two chapters considers sequential processes only. The reason for this is that parallel composition cannot be treated "compositionally" in the same framework. Rather, one needs sequents of a different type. One possibility for this is to use sequents of a more general type as suggested by Dam [Dam95]. Another approach, advocated by Stirling [Sti87], is to use another kind of sequents only for parallel composition. We explain and follow the

latter approach below.

Stirling's suggestion is to use a separate proof system for inferring sequents of the shape

$$\Phi, \Psi \vdash \Theta$$

meaning that *any* two processes satisfying Φ and Ψ respectively, when composed in parallel, yield a process satisfying Θ. Then, proving $P|Q \vdash \Theta$ can be reduced to proving $P \vdash \Phi$ and $Q \vdash \Psi$ for some suitable formulae Φ and Ψ for which we can infer $\Phi, \Psi \vdash \Theta$. In other words, we introduce the rule

$$\mathrm{E}(|) \ \frac{P|Q \vdash \Theta}{P \vdash \Phi \quad \Phi, \Psi \vdash \Theta \quad Q \vdash \Psi}$$

to connect the proof systems for the two kinds of sequents. This treatment of parallel composition is not a "truly" compositional one, in the sense that the new formulae Φ and Ψ are not necessarily subformulae of Θ, and have usually to be guessed. On the other hand, due to the complex nature of parallel composition, there seems to be no such treatment, and the proposed way of dealing with parallel composition is justifiable.

In [Sti87], proof rules are given for sequents of the new type, but only for formulae in Hennessy-Milner Logic. It is a challenging problem to find rules for dealing with fixpoint formulae. A proof system of this kind was proposed by Berezin in [Ber95], but the treatment of fixpoint formulae proposed there was found not satisfactory. In a later paper co-authored by the present author [BG97] we proposed the use of tags much the same way as shown in the preceding chapter. We describe here only our own contribution, namely the treatment of the fixpoint formulae.

Intuitively, formulae can be identified with their denotation, i.e., with sets of processes, and for this setting the tagging approach has been used successfully. One would expect the rules shown in Figure 5.1 to be justifiable, where L is a list of formula pairs. The rules $(\mu l0)$ and $(\mu l1)$ have symmetric counterparts, not shown here for brevity. We also omit the rules for unfolding least fixpoints on the right-hand side

and for unfolding greatest fixpoints on the left-hand side of the turnstyle symbol; these are simple unfoldings without tagging. It is an important question whether this is sufficient to achieve completeness. Kozen's proof system [Koz83] seems to indicate that it is, but a rigorous proof is to be given elsewhere. Note also that to simplify the account we consider just the propositional version of the Modal μ-Calculus, but the ideas presented here generalise easily to the logic presented in Chapter 3.

$$(\nu r1)\ \frac{\Phi, \Psi \vdash \nu Z\{L\}.\Theta}{\Phi, \Psi \vdash \Theta[\nu Z\{(\Phi,\Psi), L\}.\Theta/Z]}\ (\Phi,\Psi) \notin L$$

$$(\mu l1)\ \frac{\Phi, \mu Z\{L\}.\Psi \vdash \Theta}{\Phi, \Psi[\mu Z\{(\Phi,\Theta), L\}.\Psi/Z] \vdash \Theta}\ (\Phi,\Theta) \notin L$$

$$(\nu r0)\ \frac{\Phi, \Psi \vdash \nu Z\{L\}.\Theta}{\cdot}\ (\Phi,\Psi) \in L \quad (\mu l0)\ \frac{\Phi, \mu Z\{L\}.\Psi \vdash \Theta}{\cdot}\ (\Phi,\Theta) \in L$$

Figure 5.1: Fixpoint Rules.

The task we are faced with is finding a suitable semantics for tagged formulae which would justify the rules just presented.

Definition 5.1 *Let $\mathcal{P}, \mathcal{P}_1, \mathcal{P}_2, \ldots$ denote sets of CCS terms. We define the following two operations on such sets:*

$$\mathcal{P}_1|\mathcal{P}_2 \stackrel{\Delta}{=} \{P|Q \mid (P,Q) \in (\mathcal{P}_1 \times \mathcal{P}_2) \cup (\mathcal{P}_2 \times \mathcal{P}_1)\}$$
$$\mathcal{P}_1/\mathcal{P}_2 \stackrel{\Delta}{=} \{P \mid \forall Q \in \mathcal{P}_2.\ P|Q \in \mathcal{P}_1 \wedge Q|P \in \mathcal{P}_1\}$$

In other words, these are a symmetric product and division operator on sets of processes. These two operators are dual, as the following proposition shows.

Property 5.2 *For any sets $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ of processes the following equivalence holds:*

$$\mathcal{P}_1 \subseteq \mathcal{P}_3/\mathcal{P}_2 \equiv \mathcal{P}_1|\mathcal{P}_2 \subseteq \mathcal{P}_3 \equiv \mathcal{P}_2 \subseteq \mathcal{P}_3/\mathcal{P}_1$$

Proof. We show the first equivalence; the second one is just a symmetric version.

$$\begin{array}{lll} & \mathcal{P}_1|\mathcal{P}_2 \subseteq \mathcal{P}_3 & \\ \equiv & \{P|Q \mid (P,Q) \in (\mathcal{P}_1 \times \mathcal{P}_2) \cup (\mathcal{P}_2 \times \mathcal{P}_1)\} \subseteq \mathcal{P}_3 & \{\text{Definition 5.1}\} \\ \equiv & \forall P \in \mathcal{P}_1.\, \forall Q \in \mathcal{P}_2.\, P|Q \in \mathcal{P}_3 \wedge Q|P \in \mathcal{P}_3 & \{\text{Set theory}\} \\ \equiv & \mathcal{P}_1 \subseteq \{P \mid \forall Q \in \mathcal{P}_2.\, P|Q \in \mathcal{P}_3 \wedge Q|P \in \mathcal{P}_3\} & \{\text{Set theory}\} \\ \equiv & \mathcal{P}_1 \subseteq \mathcal{P}_3/\mathcal{P}_2 & \{\text{Definition 5.1}\} \end{array}$$

□

Using this operators one can give a concise definition of validity of sequents of the shape $\Phi, \Psi \vdash \Theta$.

Definition 5.3 *A sequent of the shape* $\Phi, \Psi \vdash \Theta$ *is termed* valid, *denoted* $\Phi, \Psi \models \Theta$, *iff for all valuations* $\mathcal{V}$*:*

$$\|\Phi\|_{\mathcal{V}} \mid \|\Psi\|_{\mathcal{V}} \subseteq \|\Theta\|_{\mathcal{V}}$$

We repeat here the Reduction Lemma, giving also its dual version for least fixpoints.

Lemma 5.4 (Reduction Lemma) *Let* $f : S \to S$ *be a monotone mapping on the complete lattice* $(S, \sqsubseteq)$*. Then for any* $U \in S$*:*

(i) $U \sqsubseteq \nu X.f(X) \equiv U \sqsubseteq f(\nu X.(U \sqcup f(X)))$

(ii) $U \sqsupseteq \mu X.f(X) \equiv U \sqsupseteq f(\mu X.(U \sqcap f(X)))$

We are now ready to give a meaning to tagged fixpoint formulae.

Definition 5.5 *Let* L *be a (possibly empty) list of pairs of closed formulae. The semantics of tagged predicates is defined as follows:*

(i) $\|\nu Z\{L\}.\Phi\|_{\mathcal{V}} \stackrel{\Delta}{=} \nu X.\ \cup L \cup \|\Phi\|_{\mathcal{V}[X/Z]}$

(ii) $\|\mu Z\{L\}.\Phi\|_{\mathcal{V}} \stackrel{\Delta}{=} \mu X.\ \cap L \cap \|\Phi\|_{\mathcal{V}[X/Z]}$

where $\cup L \stackrel{\Delta}{=} \bigcup_{(\Psi,\Theta)\in L} \|\Psi\| \mid \|\Theta\|$ *and* $\cap L \stackrel{\Delta}{=} \bigcap_{(\Psi,\Theta)\in L} \|\Theta\| \,/\, \|\Psi\|$.

With this semantics we can show that the rules suggested in the beginning of the section for handling fixpoint formulae in sequents of the shape $\Phi, \Psi \vdash \Theta$ are sound. The soundness proof uses the following corollary.

Corollary 5.6 *For any formulae Φ, Ψ and Θ holds:*

(i) $\|\Phi\| \mid \|\Psi\| \subseteq \|\nu Z\{L\}.\Theta\| \equiv \|\Phi\| \mid \|\Psi\| \subseteq \cup L \cup \|\Theta[\nu Z\{(\Phi,\Psi), L\}.\Theta/Z]\|$

(ii) $\|\Theta\| / \|\Phi\| \supseteq \|\mu Z\{L\}.\Psi\| \equiv \|\Theta\| / \|\Phi\| \supseteq \cap L \cap \|\Psi[\mu Z\{(\Phi,\Theta), L\}.\Psi/Z]\|$

Proof. Again we show the first equivalence only. Let $\mathcal{V}$ be an arbitrary valuation.

$$
\begin{array}{lll}
 & \|\Phi\| \mid \|\Psi\| \subseteq \|\nu Z\{L\}.\Theta\|_{\mathcal{V}} & \\
\equiv & \|\Phi\| \mid \|\Psi\| \subseteq \nu X.\ \cup L \cup \|\Theta\|_{\mathcal{V}[X/Z]} & \{\text{Definition 5.5 (i)}\} \\
\equiv & \|\Phi\| \mid \|\Psi\| \subseteq \cup L \cup \|\Theta\|_{\mathcal{V}[\nu X.\, \cup L \cup (\|\Phi\| \mid \|\Psi\|) \cup \|\Theta\|_{\mathcal{V}[X/Z]}/Z]} & \{\text{Lemma 5.4 (i)}\} \\
\equiv & \|\Phi\| \mid \|\Psi\| \subseteq \cup L \cup \|\Theta\|_{\mathcal{V}[\|\nu Z\{(\Phi,\Psi),L\}.\Theta\|_{\mathcal{V}}/Z]} & \{\text{Definition 5.5 (i)}\} \\
\equiv & \|\Phi\| \mid \|\Psi\| \subseteq \cup L \cup \|\Theta[\nu Z\{(\Phi,\Psi), L\}.\Theta/Z]\|_{\mathcal{V}} & \{\text{Substit. Property}\}
\end{array}
$$

□

Theorem 5.7 *The rules ($\nu r1$), ($\mu l1$), ($\nu r0$) and ($\mu l0$) presented above are sound.*

Proof. (i) Rule ($\nu r1$).

$$
\begin{array}{lll}
 & \Phi, \Psi \models \Theta[\nu Z\{(\Phi,\Psi), L\}.\Theta/Z] & \\
\equiv & \|\Phi\| \mid \|\Psi\| \subseteq \|\Theta[\nu Z\{(\Phi,\Psi), L\}.\Theta/Z]\| & \{\text{Definition 5.3 (i)}\} \\
\Rightarrow & \|\Phi\| \mid \|\Psi\| \subseteq \cup L \cup \|\Theta[\nu Z\{(\Phi,\Psi), L\}.\Theta/Z]\| & \{\text{Set theory}\} \\
\equiv & \|\Phi\| \mid \|\Psi\| \subseteq \|\nu Z\{L\}.\Theta\| & \{\text{Corollary 5.6 (i)}\} \\
\equiv & \Phi, \Psi \models \nu Z\{L\}.\Theta & \{\text{Definition 5.3 (i)}\}
\end{array}
$$

(ii) Rule ($\mu l1$). Similar to (i).

(iii) Rule ($\nu r0$).

$$
\begin{array}{lll}
 & (\Phi,\Psi) \in L & \\
\Rightarrow & \|\Phi\| \mid \|\Psi\| \subseteq \cup L & \{\text{Definition 5.5}\} \\
\Rightarrow & \|\Phi\| \mid \|\Psi\| \subseteq \cup L \cup \|\Theta[\nu Z\{(\Phi,\Psi), L\}.\Theta/Z]\| & \{\text{Set theory}\} \\
\equiv & \|\Phi\| \mid \|\Psi\| \subseteq \|\nu Z\{L\}.\Theta\| & \{\text{Corollary 5.6 (i)}\} \\
\equiv & \Phi, \Psi \models \nu Z\{L\}.\Theta & \{\text{Definition 5.3 (i)}\}
\end{array}
$$

(iv) Rule ($\mu l0$). Similar to (iii).

□

It should be noted here that the side conditions to the rules, as presented here, are too strong as they require (Φ, Ψ) to be syntactically identical to some pair in L, including the tags of Φ and Ψ! These conditions can be relaxed as shown in [BG97]. For the above discussion, however, we chose the simpler setting.

5.2 Negative Tagging

When model checking finite state systems it is not necessary to use reasoning based on induction for least fixpoint formulae. To obtain completeness it is sufficient to perform simple unfolding. Inductive reasoning can reduce the size of a proof significantly, but makes proof search more difficult. It makes sense, however, to record the states at which a least fixpoint formula has already been unfolded. It is generally desirable to terminate a branch in the current proof tree whenever there is sufficient evidence that the respective sequent is not valid and that elaborating it further cannot possibly lead to success. For example, the proof system presented in [ASW94] has a rule of the form:

$$(\mu)\ \frac{s \vdash \mu Z\{L\}.\Phi}{s \vdash \Phi[\mu Z\{s, L\}.\Phi/Z]}\ s \notin L$$

which prevents unfolding a least fixpoint formula twice at the same state. Such a rule can be justified semantically by defining tags L to denote sets of states, and by defining the denotation of tagged least fixpoint formulae as follows:

$$\|\mu Z\{L\}.\Phi\|_{\mathcal{V}} \stackrel{\Delta}{\equiv} \mu X.(\|\Phi\|_{\mathcal{V}[X/Z]} - L)$$

Rule (μ) is sound and backward sound due to the following equivalence:

$$s \in \mu X.f(X) \equiv s \in f(\mu X.(f(X) - \{s\}))$$

which holds for any monotone mapping $f : \wp(S) \to \wp(S)$. We refer to tagging used in this way as *negative tagging*, since tags are in some sense negative assumptions:

we assume that the states in the tag do *not* belong to the denotation of the tagged least fixpoint formula.

Unfortunately, this equality holds only for single states, and not for sets of states in general. It does not justify a rule like (μ) in proof systems for value passing processes or for sequents with formulae only (like the ones considered in the previous section), since parametrised processes and formulae are understood as sets of states. This is why it is an interesting problem to investigate for what semantics of tags and tagged formulae and for what relationship *Rel* between a set of states U and a tag L could a rule of shape

$$(\mu*)\ \frac{U \vdash \mu Z\{L\}.\Phi}{U \vdash \Phi[\mu Z\{U,L\}.\Phi/Z]}\ U\ Rel\ L$$

be justified (even in the more general case of infinite state spaces), and to what other settings could it be adapted.

Let us start by analysing why it is that the above equality fails for sets of states. If we adopt the notation $\mu X\{U\}.f(X)$ for $\mu X.(f(X) - U)$, the previous equivalence could be rewritten as:

$$s \in \mu X.f(X) \ \equiv\ s \in f(\mu X\{s\}.f(X))$$

Consider the following LTS:

$$s_3 \longrightarrow s_2 \longrightarrow s_1 \longrightarrow s_0$$

and the formula $\mu Z.[-]Z$, the denotation of which is the least fixpoint μf of the state transformer $f \stackrel{\Delta}{=} \lambda X.\|[-]\|\, X$. We have $\mu X\{s_2\}.f(X) = \{s_0, s_1\}$ and hence $f(\mu X\{s_2\}.f(X)) = f(\{s_0, s_1\}) = \{s_0, s_1, s_2\}$ includes s_2. In terms of fixpoint approximants, $\mu X\{s\}.f(X)$ contains $\mu^\alpha f$ for the greatest ordinal α such that $\mu^\alpha f$ does not include s, since this is the first point in the iterative construction of the fixpoint that s comes into play[1] (in this example α equals two). Since f is monotone, $s \in \mu f$

[1]Or dually, $\alpha + 1$ is the least ordinal such that $\mu^{\alpha+1} f$ includes s.

implies:

$$s \in \mu^{\alpha+1} f = f(\mu^{\alpha} f) \subseteq f(\mu X\{s\}.f(X))$$

and therefore $s \in f(\mu X\{s\}.f(X))$. This is exactly the point where we cannot extend this reasoning to any set of states U: if α is the greatest ordinal[2] for which $\mu^{\alpha} f$ does not intersect U, then $U \subseteq \mu^{\alpha+1} f$ is guaranteed only when U is a singleton set. For example, for $U = \{s_1, s_2\}$ we have $\mu X\{U\}.f(X) = \{s_0\}$ and hence $f(\mu X\{U\}.f(X)) = \{s_0, s_1\}$ which includes s_1 but does not include s_2. On the other hand, the following observation can be made: a relationship of the shape

$$U \subseteq \mu^{\alpha+1} f = f(\mu^{\alpha} f) \subseteq f(\mu X\{U\}.f(X))$$

would still hold if we redefine:

- α to be the greatest ordinal (if there is such) so that $\mu^{\alpha} f$ does not contain (rather than "does not intersect") U. Then $U \subseteq \mu^{\alpha+1}$.
- tags to be sets of states U denoting not themselves, but rather those elements of U only which are not in $\mu^{\alpha} f$. Then $\mu^{\alpha} f \subseteq \mu X\{U\}.f(X)$ and therefore $f(\mu^{\alpha} f) \subseteq f(\mu X\{U\}.f(X))$.

We now proceed to formalise the above intuitive ideas. Let S be a set (of states), and let $f : \wp(S) \to \wp(S)$ be monotone.

Definition 5.8 *Let $U \subseteq S$ be a set of states. The* closure ordinal $co_f U$ *and* closure elements $ce_f U$ *of U w.r.t. f are defined as follows:*

$co_f U \triangleq$ *the least ordinal α such that $U \cap \mu f \subseteq \mu^{\alpha} f$*

$ce_f U \triangleq U - \bigcup_{\beta < co_f U} \mu^{\beta} f$

Note 5.9 *In the latter defining equation the term $\bigcup_{\beta < co_f U} \mu^{\beta} f$ equals $\mu^{\alpha} f$ whenever $co_f U$ is the successor ordinal of α.*

[2] It should also be noted here, that such a greatest ordinal is guaranteed to exist only when U is finite, a complication that does not arise for finite state spaces.

Property 5.10 *Let $U \subseteq S$ be a set of states. Then:*

(i) $(U \cap \mu f) \subseteq \mu^{co_f U} f$.

(ii) $ce_f U \cap \mu f$ *is non-empty if and only if* $co_f U$ *is a successor ordinal.*

(iii) If U is finite, then $ce_f U$ is not *a limit ordinal.*

(iv) If $s \in S$, then $ce_f\{s\} = \{s\}$.

Proof. These properties are established as follows.

(i) Follows directly from the definition of $co_f U$.

(ii) We have:

$$
\begin{array}{lll}
 & ce_f U \cap \mu f \neq \emptyset & \\
\equiv & ce_f U \cap \mu f \not\subseteq \bigcup_{\beta < co_f U} \mu^\beta f & \{\text{Def. } ce_f U,\ \mu^\alpha f \subseteq \mu f\} \\
\equiv & U \cap \mu f \neq \emptyset \ \wedge \ \bigcup_{\beta < co_f U} \mu^\beta f \neq \mu^{co_f U} f & \{\text{From (i)}\} \\
\equiv & co_f U \neq 0 \ \wedge \ co_f U \text{ is not a limit ordinal} & \{\text{Def. fixp. approximant}\} \\
\equiv & \exists \alpha \in Ord.\ co_f U = \alpha + 1 & \{\text{Def. ordinal}\}
\end{array}
$$

(iii) From the definition of fixpoint approximants follows immediately that the closure ordinal for singleton sets is not a limit ordinal. If U is finite, the closure ordinals of the singletons formed by the elements of U have a greatest element α which is not a limit ordinal. This ordinal is also the closure ordinal of U.

(iv) This is a direct consequence of (iii).

□

Definition 5.11 *Let $U \subseteq S$. We define* tagged mappings *as follows:*

$$f_{\{U\}} \stackrel{\Delta}{=} \lambda X.(f(X) - ce_f U)$$

and use the notation $f_{\{U,V_1,\ldots,V_n\}}$ for $(f_{\{V_1,\ldots,V_n\}})_{\{U\}}$.

Note 5.12 *In the chosen notation $\mu f_{\{U\}}$ equals $\mu X\{ce_f U\}.f(X)$. Because of Property 5.10 (iv) the new semantics of tags coincides with the old one in the case of singleton sets.*

Property 5.13 *Let $U \subseteq S$ be a set of states. Then:*

(i) $\mu f_{\{U\}} \subseteq \mu f$

(ii) if $co_f U = \alpha + 1$ *for some ordinal* α*, then* $\mu^\alpha f = \mu^\alpha f_{\{U\}}$.

Proof. These properties are established as follows.

(i) Follows directly from the equation:

$$\mu f = \bigcap \{X \mid f(X) \subseteq X\}$$

(ii) Let $co_f U = \alpha + 1$. Then $ce_f U \cap \mu^\alpha f = \emptyset$ by Definition 5.8 and Note 5.9. Consequently $ce_f U \cap \mu^\beta f = \emptyset$ holds for all ordinals $\beta \leq \alpha$. Then the result holds by a simple inductive argument.

□

The following property will be used to justify the side condition of the new proof rule $(\mu*)$.

Property 5.14 *For any finite non-empty set U holds the inequality:*

$$U \not\subseteq \mu f_{\{V_1,\ldots,U,\ldots,V_n\}}$$

Proof. By induction on n. The base case (i.e., empty tag) holds vacuously. The induction hypothesis assumes the property for an arbitrary k. Assume U is a finite non-empty set. If $U = V_i$ for some i such that $2 \leq i \leq k+1$ then the property holds, since $\mu f_{\{V_2,\ldots,V_{k+1}\}} \subseteq \mu f_{\{V_1,V_2,\ldots,V_{k+1}\}}$ by Property 5.13 (i) and $U \not\subseteq \mu f_{\{V_2,\ldots,U,\ldots,V_{k+1}\}}$ by the induction hypothesis. The case that remains to be considered is $U = V_1$. Let g denote $\mu f_{\{V_2,\ldots,V_{k+1}\}}$. We have to show that $U \not\subseteq \mu g_{\{U\}}$. According to Property 5.10 (iii), since U is finite, $co_f U$ is not a limit ordinal. Since U is not empty, either there are elements in U which are not in μf, or $ce_f U$ is not empty, and in either case $U \not\subseteq \mu g_{\{U\}}$. □

The following lemma plays the same rôle as Kozen's Reduction Lemma.

Lemma 5.15 (Reduction Lemma) *For any set $U \subseteq S$ the following equality holds:*

$$U \subseteq \mu f \;\equiv\; U \subseteq f(\mu f_{\{U\}})$$

Proof. The two directions are established as follows:

($\Leftarrow$) This direction holds simply because $f(\mu f_{\{U\}}) \subseteq f(\mu f) = \mu f$.

($\Rightarrow$) If $ce_f U \cap \mu f$ is empty, then the implication holds trivially since in this case $\mu f = \mu f_{\{U\}} = f(\mu f) = f(\mu f_{\{U\}})$. If $ce_f U \cap \mu f$ is not empty, then by Property 5.10 (ii) $co_f U$ is the successor of some ordinal α. Then:

$$\begin{array}{rcll} U \subseteq \mu f & \equiv & U \subseteq \mu^{co_f U} f & \{\text{Property 5.10 (i)}\} \\ & \equiv & U \subseteq \mu^{\alpha+1} f & \{co_f U = \alpha + 1\} \\ & \equiv & U \subseteq f(\mu^{\alpha} f) & \{\text{Def. fixpoint approximants}\} \\ & \equiv & U \subseteq f(\mu^{\alpha} f_{\{U\}}) & \{\text{Property 5.13 (ii)}\} \\ & \Rightarrow & U \subseteq f(\mu f_{\{U\}}) & \{\mu^{\alpha} f_{\{U\}} \subseteq \mu f_{\{U\}}\} \end{array}$$

□

We are now ready to give a suitable semantics to formulae tagged with lists of sets of states.

Definition 5.16 *The* denotation *of negatively tagged formulae is defined as follows:*

$\|\mu Z\{V_1, \ldots, V_n\}.\Phi\|_{\mathcal{V}} \stackrel{\Delta}{=} \mu f_{\{V_1,\ldots,V_n\}}$, *where* $f = \lambda X.\, \|\Phi\|_{\mathcal{V}[X/Z]}$

Due to Note 5.12 this semantics is equivalent to the one already given for the case when the tag sets are singletons, and is hence a proper generalisation of the latter. It gives rise to the following (goal directed) inference rule:

$$(\mu*)\ \frac{U \;\vdash\; \mu Z\{V_1, \ldots, V_n\}.\Phi}{U \;\vdash\; \Phi[\mu Z\{U, V_1, \ldots, V_n\}.\Phi/Z]}\ U - finite \;\Rightarrow\; \forall i.\, V_i \not\subseteq U$$

the soundness of which is established in the following theorem.

Theorem 5.17 *Rule ($\mu*$) is sound and backward sound.*

Proof. If we ignore the side condition, soundness and backward soundness of the rule are a straightforward consequence of Definition 5.16 and the Reduction Lemma. Assume the side condition does not hold, i.e., U is finite and that some set V_i in the tag is a subset of U. But then V_i is also finite, and according to Property 5.14 the sequent $V_i \vdash \mu Z\{V_1, \ldots, V_n\}.\Phi$ cannot be valid, and hence neither can $U \vdash \mu Z\{V_1, \ldots, V_n\}.\Phi$ be valid. □

Rule ($\mu*$) is easily seen to be a proper generalisation of rule (μ). The most interesting question that immediately offers itself is whether finiteness of U is really important for terminating a branch in a proof tree as unsuccessful. This turns out to be the case, as the following example shows. Consider the infinite state LTS with states S:

$$\cdots \longrightarrow s_3 \longrightarrow s_2 \longrightarrow s_1 \longrightarrow s_0$$

and the formula $\mu Z.[-]Z$. The sequent $S \vdash \mu Z.[-]Z$ is valid, but if we start reducing it we very soon arrive at the sequent $S \vdash \mu Z\{S\}.[-]Z$. It would still make sense to stop at this point from purely practical reasons, but it would certainly not be sound to conclude that the sequent is not valid.

What remains a topic for future research is investigating in what settings the new rule is useful. An obvious candidate would be a proof system along the lines of the one presented by Andersen [And93], but for the case of finite state systems.

Chapter 6

Conclusion

6.1 Summary

This thesis addressed the problem of specification and verification of communicating systems with value passing. We assumed that such systems are described in the well-known Calculus of Communicating Systems, or rather, in its value passing extension. As a specification language we proposed an extension of the Modal μ-Calculus, a poly-modal first-order logic with least and greatest fixpoints. For this logic we developed a proof system for verifying judgements of the form $b \vdash E : \Phi$ where E is a sequential CCS term and b is a Boolean assumption about the value variables occurring free in E and Φ. Proofs conducted in this proof system follow the structure of the process term and the formula. This syntactic approach makes proofs easier to comprehend and machine assist. To avoid the introduction of global proof rules we adopted the technique of tagging fixpoint formulae with all relevant information needed for the discharge of reoccurring sequents. We provided such tagged formulae with a suitable semantics. The resulting proof system was shown to be sound in general and complete, relative to external reasoning about values, for a large class of sequential processes. The problem of verifying processes involving parallel composition was only addressed partially here. We proposed a way of tagging, and a semantics for tagged formulae, in the context of sequents of the shape $\Phi, \Psi \vdash \Theta$. To facilitate termination in proof search we also investigated negative tagging in a more general setting than this has

previosly been done.

6.2 Evaluation

The work presented here extends existing techniques for specification and verification of communicating systems. These can be viewed as explorations of the idea of using tags to three different settings: value passing, extended sequents, and negative tagging.

The *benefits* from using tags are manifold: it eliminates the need for global proof rules for dealing with fixpoint formulae, thus simplifying both the use and the theoretical investigation of the proof system; it helps proof search by giving syntactic proof termination criteria; it allows inductive reasoning to be performed by encoding the induction hypothesis into the tag. Although formulae seem to become less readable in the presence of tags, in an actual implementation of a proof assistant tags do not have to be shown explicitly, but can rather be kept internally.

There are also some *shortcomings* to using tags in the way it is done here. One of these is that loops in proofs are detected only when the formula in a sequent is a fixpoint formula, i.e., of the shape $\sigma Z.\Phi$, and hence not necessarily at the earliest possible point. Another deficiency is the relative complexity of handling least fixpoints. A much more attractive approach is the one that was recently proposed in [DF98] and developed further in [DFG98]. There, inductive datatypes (such as integers, lists etc.) are treated coinductively (i.e., in the way greatest fixpoints are treated here) instead of inductively, which drastically simplifies verification. One might wonder where all the complexity of dealing with least fixpoints goes in this case. The crucial point here is that the properties of inductive datatypes (usually expressible as least fixpoint formulae) needed for proving a property of a process are usually well-known properties which can be *assumed* and hence be moved to the assumptions of the sequent. Dealing with least fixpoints on the left-hand side of the turnstyle symbol is similar to the way greatest fixpoints are handled on the other side, namely coinductively.

6.3 Directions for Improvement

One important question which we left open is whether the restriction of having a finite domain of values is really necessary for obtaining completeness. A more sophisticated analysis along the lines of [And93] might give a completeness proof without this restriction. Future efforts are also needed to address the shortcomings described above. For example, one has to solve the problem of verifying systems with *dynamic processes structure*. This requires the proof systems for the two different types of sequents considered here to be glued together in a way that respects the tags. Achieving this in a way which does not loose inferential power and is still semantically justifiable (recall the different semantics for tags employed so far in the two proof systems) presents an enormous challenge.

Another direction for improvement comes from the observation that some inference rules require early choices to be made that can only be resolved by looking deeper into the structure of the formula or the process term. In other words, these are choices that one would rather like (for proof search purposes) to postpone. Such rules are the rules for disjunction, for existential quantifier, the rules for binary choice and "diamond", and most notably the rule for unfolding least fixpoints (i.e., induction). Introducing some "lazyness" into the proof system by means of Gentzen-style sequents, existential variables, etc. would greatly facilitate machine-assisted proof search.

Of course, these techniques remain to be supported by appropriate tools. As a next step, they also remain to be evaluated industrially, this being the only true way of determining their value.

Concluding this thesis, we would like to express our hope that Formal Methods will be developed in the near future to meet the great challenges posed by the recent developments in software and hardware technology. We would like the work presented here to be seen as a (however small) contribution to such an effort.

Bibliography

[And93] Henrik Reif Andersen. *Verification of Temporal Properties of Concurrent Systems*. PhD thesis, Computer Science Department, Aarhus University, Denmark, June 1993.

[ASW94] Henrik Reif Andersen, Colin Stirling, and Glynn Winskel. A compositional proof system for the modal mu-calculus. In *Proceedings of LICS'94*, 1994.

[Ber95] Sergey Berezin. Hierarchical verification of parallel processes using the modal mu-calculus. Master's thesis, Novosibirsk State University, Novosibirsk, Russia, June 1995. In Russian.

[BG97] Sergey Berezin and Dilian Gurov. A compositional proof system for the modal mu-calculus and CCS. Technical Report CMU-CS-97-105, School of Computer Science, Carnegie Mellon University, USA, January 1997.

[BM92] Bard Bloom and Albert Meyer. Experimenting with process equivalence. *Theoretical Computer Science*, 101:223–237, 1992.

[Bra92] Julian Bradfield. *Verifying Temporal Properties of Systems*. Birkhauser, 1992.

[Bru93] Glenn Bruns. A practical technique for process abstraction. *In Proceedings of CONCUR'93, Lecture Notes in Computer Science*, 715:37–49, 1993.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[BS92] Julian Bradfield and Colin Stirling. Local model checking for infinite state spaces. *Theoretical Computer Science*, 96:157–174, 1992.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and verification of synchronization skeletons using branching time temporal logic. *In Logics of Programs, Lecture Notes in Computer Science*, 131:52–71, 1981.

[Cle90] Rance Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27:725–747, 1990.

[Dam93] Mads Dam. Model checking mobile processes. *In Proceedings of CONCUR'93, Lecture Notes in Computer Science*, 715:22–36, 1993.

[Dam94] Mads Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126:77–96, 1994.

[Dam95] Mads Dam. Compositional proof systems for model checking infinite state processes. *In Proceedings of CONCUR'95, Lecture Notes in Computer Science*, 962:12–26, 1995.

[DF98] Mads Dam and Lars-åke Fredlund. On the verification of open distributed systems. In *Proceedings of ACMSAC'98*, 1998.

[DFG98] Mads Dam, Lars-åke Fredlund, and Dilian Gurov. Toward parametric verification of open distributed systems. In H. Langmaack, A. Pnueli, and W.-P. De Roever, editors, *Compositionality: The Significant Difference*. Springer Verlag, 1998. To appear.

[GBK96] Dilian Gurov, Sergey Berezin, and Bruce Kapron. A modal mu-calculus and a proof system for value passing processes. *Electronic Notes in Theoretical Computer Science*, 5, 1996.

[HL95] Matthew Hennessy and Xinxin Liu. A modal logic for message passing processes. *Acta Informatica*, 32:375–393, 1995.

[HM80] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. *Lecture Notes in Computer Science*, 85:295–309, 1980.

[Koz83] Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Lar88] Kim G. Larsen. Proof systems for Hennessy-Milner logic with recursion. *In Proceedings of CAAP'88, Lecture Notes in Computer Science*, 299:215–230, 1988.

[McM92] Ken McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, School of Computing Science, Carnegie-Mellon University, USA, 1992.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[NH84] Rocco De Nicola and Matthew Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Par69] David Park. Fixpoint induction and proofs of program properties. *Machine Intelligence*, 5:59–78, 1969.

[Par81] David Park. Concurrency and automata on infinite sequences. *In Proceedings of the 5th GI-Conference on Theoretical Computer Science, Lecture Notes in Computer Science*, 104:167–183, 1981.

[Pet76] Carl Adam Petri. Nicht-sequentielle prozesse. *Arbeitsberichte des IMMD*, 9(8):57–82, 1976. Lecture given at the IMMD Jubilee Colloquium on Parallelism in Computer Science, Unversitaet Erlangen-Nuernberg.

[Rat97] Julian Rathke. *Symbolic Techniques for Value-passing Calculi*. PhD thesis, School of Cognitive and Computing Sciences, University of Sussex, United Kingdom, March 1997.

[RH97] Julian Rathke and Matthew Hennessy. Local model checking for value-passing processes. In *Proceedings of TACS'98*, 1997.

[Sti87] Colin Stirling. Modal logics for communicating systems. *Theoretical Computer Science*, 49:311–347, 1987.

[Sti92] Colin Stirling. Modal and temporal logics. In *Handbook of Logic in Computer Science, Volume 2*. Clarendon Press, Oxford, 1992.

[Sti96] Colin Stirling. Modal and temporal logics for processes. *In Logics for Concurrency, Lecture Notes in Computer Science*, 1043:149–237, 1996.

[SW91] Colin Stirling and David Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, 1991.

[Tar55] Alfred Tarski. A lattice-theoretical fixedpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Win91] Glynn Winskel. A note on model checking the modal nu-calculus. *Theoretical Computer Science*, 83:157–167, 1991.