NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Ph.D. Thesis

A Calculus of Communicating Systems with Atomicity and Recovery, for Protocol Specification and Design

By

Abdellatif OBAID

THESIS SUBMITTED TO THE SCHOOL OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF Ph.D. IN ELECTRICAL ENGINEERING

at the
University of Ottawa
August 1990

ISBN 0-315-68100-4

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Thesis Committee

Supervisor:

Prof. Luigi Logrippo, University of Ottawa

Readers:

Prof. Scott A. Smolka, State University of New York at Stonybrook

Prof. Moshe Krieger, University of Ottawa

Prof. Robert L. Probert, University of Ottawa

Prof. Murray C. Woodside, Carleton University

Acknowledgements

Dr Luigi Logrippo accepted me in his research group and initiated me to the delights of communication protocols and formal methods for their description and verification. He stimulated me in my progress as a researcher, while at the same time being available and constructive. Moreover, he had to endure the many errors in the drafting of this thesis. I want to express to him all my gratitude and my thanks.

Dr. Scott Smolka has accepted to be the external examiner of this thesis, and to share with me his knowledge as an expert in the area. I thank him warmly.

I would like to thank all members of my thesis committee: Dr. Moshe Krieger, Dr. Probert, and Dr. Murray Woodside for having provided much useful advice, and having accepted to read and evaluate this thesis, in spite of their many other engagements. Dr. Probert, as a coordinator of the research group at the University of Ottawa, accepted me in his laboratory and has greatly contributed to my education in this area.

Finally, I would like to thank all members of my family: my beloved wife Amal for being constantly supportive by enduring my moods during the years of this work; my lovely children Jehara and Sami for their affection; my Mother and my Sister Milouda who sacrificed so much to provide for my education during difficult times; and the rest of my family.

I dedicate this thesis to the memory of my father who lived happy for more than a hundred years. He used to repeat to me his favorite proverb:

" Science is a light,

Ignorance is a shame"

TABLE OF CONTENTS

Chapter 1:

Introduction

1.1 Motivation

The thesis is a contribution towards the use of formal methodologies for the specification, design, and validation of communication protocols for computer networks.

Although many examples in this thesis relate to protocols, applications of these techniques to other types of distributed systems, such as telephone networks, are also possible.

Protocols are the rules that govern the exchange of information between systems in a network. They are in general complicated. This complexity creates problems not only for specification, but also for testing and verification. The verification methods, for instance, are limited by the size of the specification. Thus, there is a need for modular specification and verification techniques that allow to specify, in a formal way, a large class of system behaviors. Also, system design (by design we mean decomposing systems into different functional components), is facilitated if modularity is provided. Several of the commonly used methods do not in general allow for modular specification and validation *while at the same time* staying formal enough to be used within a theory. Also,

most of these methods have shortcomings regarding the specification of data and message exchanges.

In this thesis, we use a particular class of techniques for the specification and the verification of protocols called **behavioral techniques**. In this type of technique we have the **Calculus of Communicating System (CCS)** introduced by Milner in [MIL 80] and a formal description technique for protocols called **A Language Of Temporal Ordering Specifications (LOTOS)** ([ISO 85], [ISO 87] and [BOL 87a]). We give a description of these methods in Chapter 3. We will emphasize CCS rather than LOTOS since CCS was the semantic model used for LOTOS at the time this work started. However, the contributions in this thesis can be extended to be used for the LOTOS language.

There are several contributions in our work both from the point of view of our model for communication, and from the point of view of implementing this model:

1) Theoretical contributions:

   a) With the intention of improving system design and specification in CCS and LOTOS, we introduce a new calculus, called **ATCCS** (for **Atomic Calculus of Communicating Systems**) that allows the description of behaviors of communicating systems with atomicity by means of useful features such as mutual exclusion, primitive functions (i.e. atomic functions), etc. At the same time, this calculus facilitates the design process by using appropriate semantics.

      The design of system specifications in LOTOS and CCS suffers from the fact that all systems are described in terms of elementary actions. It is important

to be able to construct systems from their components. This construction is not possible with these two techniques. The high degree of parallelism they have does not allow for such construction because we may end up with undesirable interleavings that do not allow the desired behavior to be specified. Furthermore when developing systems in a step-wise manner, it is desirable to be able to expand what appears to be a single action at a high level of abstraction into an equivalent, possibly complex process at a lower level of abstraction. These two features, and others, are made possible by the concept of atomicity that we introduced in our model.

b) As we developed the model, we found it very important to introduce, according to notion of atomicity, a mechanism for the description of fault tolerance. This was achieved by the introduction of recovery operators (capable of preserving the consistency of system states) in the semantics of the calculus.

c) The model includes a proof method based on an extended notion of observation equivalence as it was first introduced in the CCS calculus [MIL 80]. This method is shown to include the CCS equivalences with an additional equivalence related to the design of specifications using ATCCS.

2) Experimental contributions:

a) We have built a run-time system based on the theories of CCS and LOTOS. This work was a part of the University of Ottawa LOTOS interpreter [LOG 88] built for executing LOTOS specifications. This system, called **SINAPS** [OBA 87a], allows one to actually execute specifications, thus allowing one to interactively test their behaviors. Executability of specifications helps to

increase the confidence of the specifier on what he or she specified. We also extended this system to include our ATCCS calculus.

b) Furthermore, for the purpose of verification we built another system that, given a specification that is composed of concurrent processes, constructs another specification that has an equivalent behavior and that does not involve concurrency. This procedure is called **expansion**. Expansion is useful because in general, for verification purposes, it is more convenient to deal with processes that do not involve concurrency.

1.2 Protocol Specification and Verification

1.2.1 Desirable Properties of Specifications

A specification is a statement of a problem to be solved. Ideally, it should be at the same level of abstraction as the problem itself, that is it must be implementation-independent as much as possible. It has been argued that in order to increase the confidence of a specifier in what he is doing, it is preferable that a specification be **executable** [LOG 83].

In order to improve its readability and facilitate its possible modifications, a specification must be **modular**. This will allow for modular composition of specifications of subproblems, and will also aid design, understanding, and verification. Also, since a specification will be used as a vehicle of information between the specifier and the designer and/or the user, it must be **unambiguous** and **understandable**.

In order to allow for such properties, a specification must be as formal as possible: i.e. if possible, it must be based on a mathematical model. Therefore one should be able to describe the behavior of the system in terms of expressions which will be handled as mathematical objects. This will allow for mathematical verification of desired properties of the specified system.

Protocols in particular are usually expressed in terms of concurrent (or peer) entities which cooperate via an underlying medium in order to provide certain services. Therefore the description of **concurrency** and the modelling **communication** should be features of the specification.

A user of a system may not want in some cases to impose determinism in the behavior of the system because the latter may evolve in a **non-deterministic** way. Thus, the specification should be able to capture this property.

1.2.2 Service and Protocol Specifications

Protocols are rules that govern the exchange of information between systems, also called entities, in a distributed environment. The complexity of the systems makes it desirable that they be decomposed into smaller subsystems that perform some particular functions. In the context of computer networks, this decomposition consists of defining functional layers. Within the **International Organization for Standardization** (ISO for short), an architecture for **Open System Interconnection** (OSI for short) [ZIM 80] has been developed. This architecture identified seven different layers: from the layer that monitors the actual transfer of bit streams over a physical medium, to the layer that

defines the coordination between two different application processes in two different systems (Figure 1.1).

Each layer (say layer N) provides a number of functions (called **services**) to the layer above it (layer N+1). Layer N+1 uses the underlying layer together with its own functions to provide the services of its own layer through its **(N+1)-service access points.** This cascading of functions shows each layer as a black-box which provides a global service to its users in the layer above.

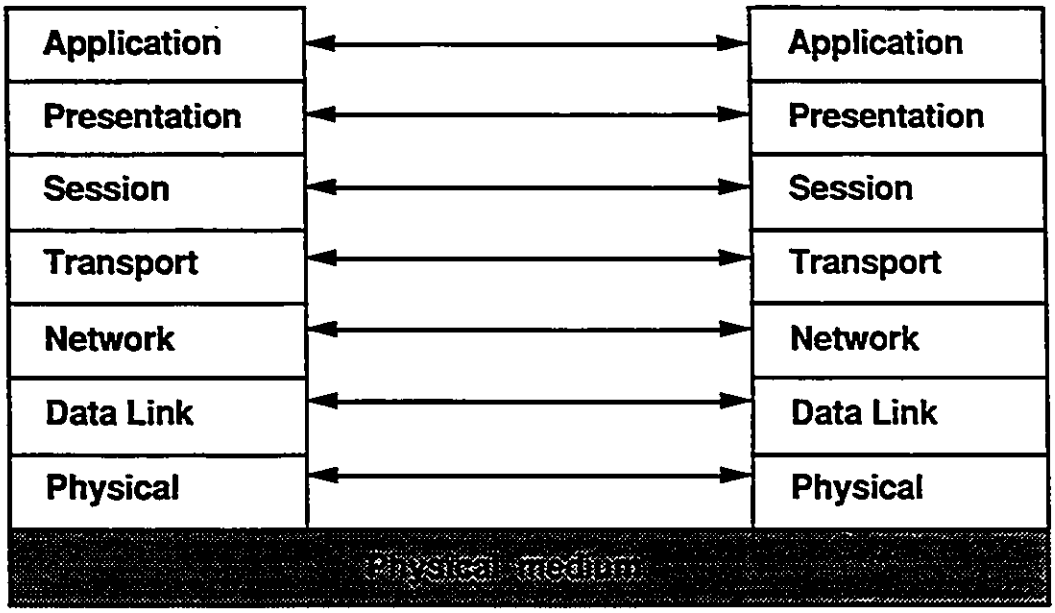


Figure 1.1: The OSI Reference Model

Each layer is composed of several entities which cooperate in order to provide the service. A description of the interactions between the entities to serve that purpose is called a **protocol specification.**

The description of the behavior of a layer in terms of its input/output interactions constitutes the **service specification**. These interactions are called **service primitives** For example a connection-oriented data transfer protocol should provide for "Connect", "Disconnect", "DataSend" and "DataReceive" service primitives. The execution of service primitives has temporal ordering constraints that any specification should reflect. From the description above we can see that a service specification is an abstraction of the protocol since the service specification does not describe how the service is achieved [VIS 85]. The structure of a service provider for Layer N, called **N-service Provider**, is shown in Figure 1.2.

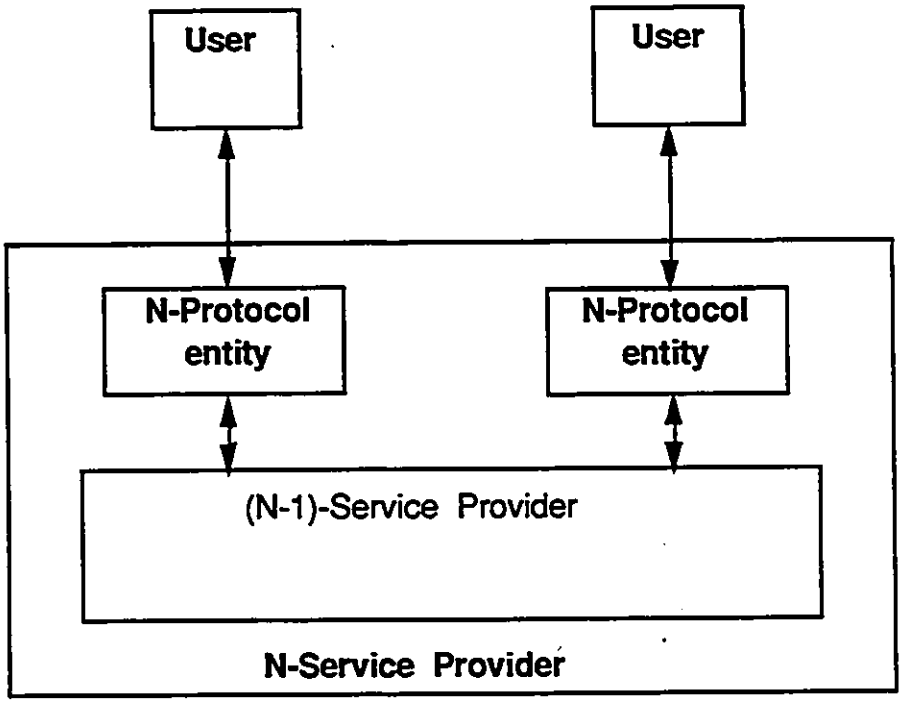


Figure 1.2: Structure of the N-Service Provider

A clear and precise definition of the functions of each layer implies the need for specification techniques for protocols and services. A variety of methods for protocol and service specifications have been proposed in recent years. Each of these methods is based on a particular model or simply uses some programming language. Some of them are reviewed in this thesis: Transition systems such as Finite State Machines (FSM's for short) ([BOC 78] and [DAN 77]) and Petri Nets [MER 79]; Programming language based models [STE 76]; Formal languages based models [HAR 78]; Abstract data type models [SUN 81]; Temporal logic based models [SM 80]; and Behavioral models such as the Calculus of Communicating Systems (CCS) [MIL 80], Communicating Sequential Processes (CSP) [HOA 85], the Algebra of Communicating Processes (ACP) ([BER 86a] and [BER 86b]), and the Language Of Temporal Ordering Specifications (LOTOS) ([ISO 85], [ISO 87] and [BOL 87a]).

1.2.3 Purpose of Verification

In order to increase user confidence that a system meets its specification, a verification process can be carried out. It is based on the specification and should consider (at least in theory) all possible situations the system will go through. Considering services and protocols, different problems must be addressed :

1) The protocol design must be verified against the service specification.

2) The design of the protocol must be verified by analysis of possible interactions of the layer entities, each one according to its protocol specification and to the underlying service specification.

3) The implementation of each protocol entity must be verified against its protocol specification.

Verification of the correctness of the protocol design requires the study and analysis of the communication of these entities given a correct underlying service provider. Verification of the implementation requires the study and analysis of the correctness of each entity separately.

What Is To Be Verified ?

We generally try to verify liveness and safety properties. Liveness properties have the form : "Something good will happen" [OWI 82]. Safety properties have the form : "Nothing bad will happen" [LAM 80]. In general the following properties are to be verified :

Absence of deadlocks : a deadlock is defined as being a state where all three of the following are true: all communicating entities are expecting certain messages to arrive; none of them is ready to send any message; and the communication media are empty. One should mention that, depending on the method that is used, the occurrence of deadlocks will be expressed in different ways. For example in CCS formalism, a deadlock is simply a situation where a system is unable to execute any action.

Progress properties : these properties concern the evolution of the system in time i.e. that the specified service functions will be completed in finite time. For instance, one such property is: "what is sent will eventually be received".

Absence of unspecified receptions: an unspecified reception occurs if at a certain state one entity is offered a next message that it cannot receive according to its specification.

Absence of nonexecutable receptions: a nonexecutable reception is a reception that an entity is unable to execute, either because it is specified at a state the entity cannot reach, or because the other entity never sends a corresponding message.

Absence of tempo-blocking: a tempo-blocking occurs when the protocol enters an infinite cycle accomplishing no useful work.

Correctness properties: these express the fact that if a protocol performs any actions, then these actions satisfy the service specification. For instance, in a system that offers sending and receiving functions one such property is: "What is received is a subset of what was sent".

Methods for verification vary with the specification method used. In Chapter 2, we describe these techniques and give descriptions of their validation techniques.

One should differentiate between two main validation techniques: verification and testing. Verification uses mathematical models (e.g. logic semantics, invariants, algebraic calculi,...) and state-space search or proof techniques. Testing, on the other hand, tries to derive, for a given system specification, a set of test cases that will hopefully reveal all errors present in the implementation [PRO 84].

The main goal in our thesis will be the study of specification techniques and verification methods rather than testing, although some discussion of related testing methods is given in Chapter 9.

1.3 Organization of the Thesis

Chapter 2 contains a review of the most important protocol specification and verification techniques. In Chapter 3, we give an idea of the approach taken in this work. In Chapter 4 we describe our model for atomicity. The application of this model for system design is described in Chapter 5. Chapter 6 contains the description of mechanisms for fault tolerance as applied to our model. Chapter 7 gives a study of the verification method used in our work for which a discussion on enhancements is given in Chapter 8. In Chapter 9, we describe the implementation details for the different concepts given in this thesis. Finally our conclusions are given in Chapter 10.

Chapter 2:

Previous Work

2.1 Review of Some Important Formal Methods

Most of the existing specification and verification methods focus on protocols and not on services. A variety of formalisms have been applied. Below, we give critical descriptions of some of them.

2.1.1 Transition-based Models

These are based on a model that views protocols as communicating entities which respond to events. The entities have states and events that cause state transitions. The responses depend on the current state of the entities.

2.1.1.1 Finite State Machine Models

The model

A system is described by using its set of input/output events, its (finite) set of states and its set of transitions. This method ([BSW 69], [BOC 78], [DAN 77]) allows to represent a

behavior by means of a directed graph whose nodes are system states and whose arcs are system transitions. A transition will be executed at a certain state when an event occurs. Each state describes an aspect of the behavior of the system (e.g. waiting for a message with a given sequence number).

In order to reduce the number of states, a notion of major state is defined which represents some control aspects (e.g. waiting state) of the system while the contextual information is described by means of contextual variables (e.g. sequence number). This version of the model is called EFSMfor Extended FSM [BOC 80]. ESTELLE [ISO 85b] is a formal description techniques used within the ISO organization based on the extended finite state machine model. This technique is quite popular among designers of standard protocols.

The verification method

In order to verify some properties, the method that is used is state exploration or reachability analysis. The entities connected together with the communication media are considered as a unique machine. Global state spaces of such machines are defined. A global state consists of the collection of the states of all communicating entities, including the media. Starting from the initial state, a reachability graph is then constructed. The traversal of the reachability graph is in fact the exploration of the complete interaction domain of the global system. Using this technique, one can verify properties of the protocol such as absence of deadlocks, absence of tempo-blocking, etc... [BOC 78].

Remarks

The major drawback of this verification technique is the state explosion. That is, the reachability graph may become very large if not infinite when data messages are used. To alleviate this problem one can impose certain restrictions, for example by assuming that the protocol entities are directly coupled [BOC 78], or by constructing classes of transitions and states [LAM 84], or by considering the different phases of the protocol one at a time [GOU 83]. In general the technique is used to validate protocols without considering contextual variables. Thus it can be used for small protocols such as the alternating bit protocol or for subsets of more complex ones [BOC 78], [DAN 77]. Its usefulness decreases as the importance of contextual variables increases.

In [WES 78b] and [WES 78c], the finite state machine model is extended by considering two or more entities with implicit simplex channels (rather than explicit as in the previous techniques), one for each pair of entities and for each direction of communication. A global state is considered as the set of local states of the entities together with their input channels. The method has been applied to validate many protocols, among others, a call establishment procedure of X.21 and X.25 network interface specification [WES 78a], [WES 78c] and [ZAF 80].

As mentioned above, validation is carried out by building a reachability tree of the communicating entities. In constructing this tree, a global state that consists of the local states of each machine together with the contents of their input channels is built. This method is called the **perturbation method**. From the initial global state (i.e. initial local states and empty channels), a perturbation results in a new global state by executing a single transition leading to a change in the local state of the entity which executed the

transition. Analysis of the so generated tree may reveal certain protocol design errors such as presence of deadlocks, unspecified receptions and nonexecutable receptions. The difference between this method and the pure finite state machine model is the existence of the fifo channel that models the underlying service medium. A drawback of this method is that the reachability tree might grow infinitely unless we impose some restrictions such as limiting the size of the channels [RAF 83]. In any case for most protocols the reachability tree grows rather quickly. Another limitation of this method is that in order to reduce the complexity of validation, only systems with perfect channels were considered. In order words, cases of message loss or duplication are not considered.

In order to reduce the complexity of specifying protocols, the authors in [ZAF 80] developed a method to design protocols which have no unspecified or nonexecutable receptions by synthesizing their FSM's. Following a set of rules, they construct a system specification from incomplete specifications of its components.

In [BRA 83] a new approach is taken in order to analyze protocols modeled by FSM's. The authors build a **tree protocol** on which they define functions for each entity which give information about the set of states that could be reached by the other entities. This way, the authors could detect in an analytical way, mainly, the presence of unspecified receptions by giving necessary and sufficient conditions on the values of these functions.

Using this technique, validation can be only partially automated. It has been shown [BRA 83] that it is in general undecidable whether a given reception is executable or that there is no deadlock or that the reachability tree is finite (modulo some loops). However one can expect decidability if there are some restrictions on the number of messages in transit at any one time. For instance, if we know, for a given protocol, the bound on the

number of messages in transit at any one time, then these problems are all decidable. In [BRA 83] the authors showed that the problem is solvable for the class of protocols with two communicating machines and only one bounded channel.

2.1.1.2 Petri Nets

The model

A **Petri net** [PET 62], [PET 77] is a set of transitions and places with a finite number of tokens. Places represent states of the entities and messages in transit. Transitions represent state changes. A token in a state indicates the presence of an event in that state. A place can contain any number of tokens, therefore any number of events. A specification includes a Petri net and an initial distribution of tokens. A transition is said to be firable if all its input places contain at least one token. Once a firing occurs, a token is taken from each input place giving one more token to all output places. Petri nets representing several protocol entities (e.g. sender, receiver) can be combined into a single Petri net specification of the whole system. The composition is done by simply adding more places and transitions, which represent the communication link.

Petri nets are suitable in order to model systems with synchronization and concurrency. Also, the nondeterminism is an implicit aspect in Petri net models, in the sense that if there are several possible firable transitions, then the one that is fired depends on the implementation of the net.

Petri nets have been used to specify many protocols. Some examples are: the alternating bit protocol [BOC 77a], an OSI transport protocol [COU 84], a simple version of the ARPANET IMP-IMP link protocol [MER 76], a packet-switching call establishment

protocol [DAN 77], and the X.21 interface [RAZ 80]. One should notice that specification of large protocols, however, tends to give rise to large graphs that are often unreadable.

The verification method

Verification in case of Petri nets is basically a state exploration technique [DIA 82]. Starting from an initial marking of the net (i.e a placement of tokens in the places), one can construct a reachability graph called a **token machine.** This graph will be used to detect deadlocks, boundedness (i.e. no place gets flooded with tokens), mutual exclusion, and loops [MER 76].

Another verification technique, called **structural analysis**, uses linear algebra formalism on matrices (called incidence matrices) that represent dependencies between transitions of Petri nets. These matrices can be used to find place invariants or transition invariants. The latter are in turn used to determine if the net is safe, bounded or conservative (i.e. it keeps the same number of tokens).

Several extensions to the basic Petri net model have been proposed of which we will mention some of the most important:

Timed-Petri nets, described in [MER 76], consist of adding two time values *tmin* and *tmax* to each transition of the net. *tmin* denotes the minimal time that must elapse before a transition can fire. *tmax* is the maximum time that the transition is enabled and that the transition does not fire. This mechanism allows the modelling of timeouts [AZE 78]. Timed-Petri nets were used to specify the transport protocol of the CYCLADE network [DAN 80].

In [BOC 77a], Petri nets have been extended following [KEL 76]. Extensions involve state variables and predicates on these variables. Predicates must be true in order for a transition to fire. Any firing may change the values of these variables. The verification method for this extension consists of associating assertions with the system global states. These assertions are proved by induction on the set of global states.

Numerical Petri nets [SYM 80] are obtained by allowing tokens to have values of arbitrary data types and associating read/write memories with the net. Enabling conditions are also associated with transitions and transitions may reference tokens in the input places as well as values in the net memory. This allows to distinguish between tokens and firing conditions. This way the net has global variables as well as predicates as in the case of extended finite machines.

Numerical Petri nets were used to model a transport service in [BIL 82].

Remarks

Petri net models share several limitations with FSM models. When the set of states is large, the net becomes cumbersome. Also Petri nets cannot describe complex events used to model complex data transfer or timing considerations such as timeouts without suffering from complexity and/or explosion of the net size.

In spite of their ability to specify concurrency, Petri net models have disadvantages concerning modularity in specifications. It is in general not easy to build a net that represents a composition of several sub- nets. In fact there are several ways of achieving this composition [DIA 83]. However, they do not preserve the correctness properties of the net.

2.1.2 Programming Language Based Models

The model

In this approach, protocols are described in some (possibly suitably extended) high-level programming language [BOC 75], [STE 76], [BRA 78], [BUH 83]. Some of the most commonly used languages have been Pascal, Ada, or similar. The different components of the entities are represented by modules in the program and communication occurs via shared variables, monitors or by using rendezvous [BUH 83].

Describing protocols as programs allows one to describe control as well as parameters and data types. Moreover one can apply design methodologies known in the area of software engineering [PAR 77], [GUT 79] and the usual program verification techniques such as the Floyd and Hoare techniques [FLO 67], [HOA 69], [DIJ 65]

The verification method

In these techniques, global assertions are stated for the beginning and the end of the program as well as local assertions concerning the inside of the program. The global assertion at the beginning is a hypothesis. The local assertions must be proven using the hypothesis and other local assertions. A local assertion attached to a point inside a loop must be true on every iteration of the loop. When the program terminates the global assertion of the end must be true.

In [STEN 69], this method is used to verify correctness of message delivery and sequencing of messages sent by a transmitter.

Another verification technique is **symbolic execution** [BRA 78]. Symbolic execution explores all the possible execution paths and provides the results expressed in terms of symbolic input variables. The method as applied to protocols in [BRA 78] is based on the construction of a symbolic tree whose nodes contain symbolic values as well as an assertion to be verified at that node. Two primitives called **delay** and **wait** are used in order to control the concurrent execution of processes. The nondeterminism is dealt with by generating random events using a sort of random assignment statement.

Remarks

In these methods, by establishing assertions, one can verify safety properties. However, this requires expertise and intuition in order to select the appropriate assertions. Moreover the assertions may become cumbersome when the program is of reasonable size. Theorem provers can help in automating the verification of the assertions. However, the available theorem provers are still not developed enough to be widely used. Theorems that need to be proved for real life programs usually involve fairly complex knowledge.

Another drawback is that most programming languages do not include an adequate semantic model for concurrency. An attempt has been made in [OWI 76] for the verification of concurrent programs using an Algol-like language that allows communication via shared variables only. The proof mechanism is based on the axiomatic description introduced by Hoare [HOA 69] with some additional axioms for describing the assertions for critical regions and parallel execution of processes

described by means of the "cobegin-coend" construct. In [OWI 76], a proof of correctness of the five philosophers problem [DIJ 65] is shown.

While a large amount of work has been done on program verification, some characteristics of protocols make the verification proof using these methods very difficult. First, protocols are usually described by means of multiple modules that interact. In general these modules are physically separated and the communications cannot be described by means of shared variables. Moreover, most of the time the communication media are not reliable. This requires a proper treatment of media with nondeterministic behavior.

Also, in using these languages, one gets into detailed description of a system's operation. This makes it difficult to specify the abstract requirements of the protocol without getting into implementation details.

2.1.3 Temporal Logic Based Models

While all the techniques mentioned so far focused on safety properties, temporal logic [PNU 77], [MAN 81] is aimed at liveness properties as well as safety properties.

The model and the verification method

Conventional logic formulas as used in the invariant method described in section 2.1.2 cannot express liveness properties because they cannot refer to any other state than the current one.

One form of temporal logic includes three basic temporal operators: □ (henceforth),

✧ (eventually) and O (next). These operators are defined over the execution sequences of programs, and interpret these sequences as time. The **henceforth operator** is interpreted as: starting from the current time, a property is true forever. The **eventually operator** is interpreted as: there is at least one time in the future at which a formula is true. The **next operator** states that a property will be true in the next state. For example, in order to state that a formula is invariant one can write:

P => □ P

meaning that if P is true in the current state then it remains true forever.

A specification in temporal logic consists of a set of axioms that assert properties for all sequences of system execution [LAM 80]. In [SCH 81] and [MAN 81], beside the basic operators, more constructs are defined like the **Until operator** which states that a property is true at least until a certain time where another property is true. For instance, the axiom:

✧ at SEND Until empty(InQ)

states that the sender will keep sending as long as there is something to send (i.e. not(empty(InQ))).

In [HAL 83] a notion of **history variables** is introduced which represent input and output messages exchanged between modules in the specification. The safety invariant would be that the output history of the sender should be included in the input history of the receiver. A liveness statement would be that any message sent will be eventually received.

Temporal logic has been used to specify the alternating bit protocol [SCH 81], [SCH 82] and to specify and verify the Stenning data transfer protocol [HAL 80] and [HAL 83].

Remarks

The method is formal enough to describe safety and liveness properties of protocols. However, its use requires a lot of expertise. A specification of even a small system involves a relatively large number of temporal axioms. When there are many operators involved, a specification becomes hard to understand. To our knowledge, automated provers are not developed enough to be used on large specifications. There is however, an automatic prover which uses a new version of temporal logic called **linear branching temporal logic**. The system is based on a finite state model [CLA 86]. As well, this system uses propositional logic (i.e. a logic without variables) which is not useful to express predicates that contain variables. Also, it does not allow to express non-deterministic behaviors.

2.1.4 Behavioral Models

We introduce the best known behavioral methods (also known as **Process Algebras**) in this section. They were selected because they are conceptually related and mutually comparable.

2.1.4.1 Communicating Sequential Processes (CSP)

CSP was introduced by Hoare in [HOA 72], the theoretical model of which is described in [HOA 85].

The model

CSP is a specification language which describes the behavior of communicating machines by synchronization and interleaving. Each process behavior is described in terms of the set of its possible actions, called its **alphabet**. The choice between the different actions can be either deterministic (i.e. fully determined by the environment) or non-deterministic. Communication can occur when one process performs an input action and the other an output action. The actual communication is a multi-way rendezvous where more than two processes can participate in the rendezvous.

In [HOA 85] all the operators (e.g. guarding, choice, concurrency....) are described in terms of functional programs in LISP.

The verification method

Verification in CSP is done by associating to each process two sets: the set of all execution sequences called the **trace set**, and the set of refused actions called the **refusal set**. A **failure set** which combines both the trace set and the refusal set is built. Typically, two processes are equivalent if they have the same failure sets. Properties of systems are stated on the domain of those traces. A property is proved on the basis of laws relating operators to traces.

OCCAM [OCC 86], a programming language based on a version of CSP has been used to specify and verify, using the traces of the system, some simple protocols [ROS 85]. TCSP [BRO 84], the theoretical CSP, has been used to specify and verify a simple Data Link protocol [BRE 88].

2.1.4.2 The Calculus of Communicating Systems (CCS)

In this section we give a brief description of CCS [MIL 80] and [MIL 89]. More details are discussed in Chapter 3.

The model

The motivation for CCS was to come up with a unified approach for the description of communicating systems. A system is considered as a **black box** which communicates with the outside world via gates. In order for two machines to communicate they must share some gates and offer matching actions at these gates.

One of the strengths of CCS resides in the fact that its semantics is well defined in terms of operational laws [PLO 81], expressed in the form of inference rules. These define the exact effects of the execution of actions on processes and have been used for generating execution sequences in [OBA 85] and for simulating process behaviors [OBA 87b].

CCS has been used to specify (in simplified form) the OSI network service [SHI 83] and a CSMA/CD protocol in [PARR 87].

The verification method

Verification aspects are carried out by using the notion of **observation equivalence** defined in [MIL 80]. Two processes are said to be equivalent if they are not distinguishable by experiments. This equivalence is useful in reducing the complexity of behavior

expressions by substituting a process for an equivalent one. A practical technique,for proving observation equivalences , called **bisimulation**, is described in [PAR 81]. It was adopted by Milner for CCS in [MIL 89].

A demonstration of the practical usefulness of CCS is provided by the fact that an early version of LOTOS, which was based on an extension of CCS, has been used for very substantial specifications, especially in the area of OSI protocols. This version of LOTOS (LOTOS-85) is no longer used, although the current version is still strongly based on CCS.

In [MIL 83] more general calculi are described: SCCS (for Synchronous CCS) and ACCS (for Asynchronous CCS). In SCCS for example one can specify simultaneous actions as a product of actions from several processes. In ACCS and SCCS a **delay** operator is defined (a process can either idle for a certain number of time units or perform an action). The semantics of CCS is extended in order to deal with timing and delay specifications. These calculi have been applied mainly in hardware descriptions.

The latest version of CSP [HOA 85] has become in some respects similar to CCS. There are however some fundamental differences, one of them concerning the ways in which communication occurs. For example, in CCS two systems sharing gates can communicate and the result of this communication becomes an internal event. This allows to explicitly express the communication using a particular action. In CSP, instead, the communication is a multi-way rendezvous. That is, a process can communicate at the same time with several processes. This feature, although useful in systems design, has some effects on the semantics of CSP. In CCS, if one wants to restrict the communication to two communicating systems, this is possible by simply hiding the gates involved in that communication. This hiding does not apply to the internal event resulting from that communication, and internal events can be used to model nondeterminism. In CSP, this

nondeterminism cannot be expressed. To overcome this problem, CSP has an operator that models the nondeterministic choice between behaviors, so spontaneous decisions can be made. For a given process, the hiding operator transforms the expression of the behavior of a system into another that contains nondeterministic choices. In fact, there are three choice operators: the non-deterministic choice, the deterministic choice, and the general one. This solution leads to complicated semantics. However, some of these operations can be expressed in terms of CCS and this was applied to the LOTOS semantics [ISO 87].

2.1.4.3 The Algebra of Communicating Processes (ACP)

This method, introduced by Bergstra and Klop describes systems by means of an algebraic semantics [BER 86a] .

The model

ACP [BER 86a] is an equational specification model for communicating processes. It was introduced as a general framework for the description of cooperating processes that can communicate on an asynchronous basis. ACP has several operators. For example, s*equencing* is expressed by using a multiplication operator denoted by · whereby a process, upon its termination, enables another process to execute (this is somehow similar to the ATCCS *enable operator* described in Section 4.7.1). Also one can describe the notion of abstraction as a way to represent communication hiding (i.e. communication that prevents external processes from participating). This operator was actually given in an improved version of ACP called ACP_τ [BER 86b] . Concurrency is also described by a composition operator ||. The communication between two processes is captured by the communication function γ defined over atomic actions. The semantics

describe the effect of the operators by means of equations that show the set of operators and their effects on processes. For instance, in order to describe the choice between two possible behaviors one has to provide the set of axioms that describe this choice and some of its important properties (e.g. commutativity, associativity, ...) in several contexts (e.g. sequencing, parallel composition, ...).

The verification method

A nice feature of ACP_τ is that it provides a proof method using behavior graphs and an adapted notion of bisimulation (see Chapter 7). These graphs are acyclic graphs, which makes this model suitable to represent finite processes only. By using the axioms, one can do verification by showing equivalences between algebraic expressions.

2.2 Summary and Conclusions

The methods that we have reviewed have been used in different ways for the specification of (distributed) systems. The transition based methods were the first ones for which a formal model was developed. Their main limitation resides in the number of global states that are involved when one wants to analyze the behavior of realistic communicating systems. In spite of this, this model has been quite useful, especially in consideration of the fact that it has a well developed verification methodology. The logic-based models are very powerful and formal enough to express all sorts of properties of systems. However they are very complex to use and have not been automated yet. The behavioral models (CCS, LOTOS, CSP, ...) seem to be more attractive to some people in the protocol community for two main reasons. The first is that their semantics can be used to produce a run time system for system prototyping and also to carry out some formal

verifications. The second reason is that they can be used to represent transition systems to some extent. The main limitation of this approach is that there is no common discipline for designing systems. Also, some behavioral specifications might not be executable as the result of using certain types of constructs (e.g. infinitely branching processes).

The different techniques will be used depending on the preferences of the user, the type of applications, and the desired results. The implementors will likely use transition models because they are easily translatable to conventional languages and also because the testing process has been well studied for these models [SAR 82]. The logic based models will be used mainly for their mathematical formalisms. They have already been used in several applications for the generation of test cases and for the validation of protocols ([PRO 84], [URA 86]).

Methods such as behavioral techniques will be used for their expressive power and for their formal semantics. These trends will be enforced if design tools and automatic proof systems can be implemented in an effective way for these methods. There are already some encouraging results in this area [BOL 87b] and [KAN 90]. In addition, behavioral methods such as CCS, LOTOS and CSP will be used because they allow highly structured specifications, and algebraic proof techniques. Also, the existence of design techniques for a given model is an important feature.

Chapter 3:

An Introduction to CCS

3.1 Features of CCS

CCS is a calculus in which one can define behaviors of communicating systems modeled by **processes.** Each process is described in terms of its interactions with its **environment** (e.g. other processes) via its interaction points. There are available several operators for combining processes for the description of complex system behaviors.

An extended version of CCS, called CCS*, was at the basis of an earlier version of LOTOS. It is this version of CCS that is used in this thesis. More recent developments of LOTOS make it (somehow) closer to Hoare's CSP [HOA 85]. LOTOS was primarily designed for specifying ISO protocols and services. It allows for modularity, system decomposition and system hierarchy. Also it supports abstract data type definitions. This part is defined in terms of abstract data types supported by the ACT ONE data type definition language [EHR 85].

CCS allows one to describe a wide variety of systems from operating systems [DOE 83] to protocols [SHI 83], [PARR 87]. In fact the model is powerful enough to be used as a target of a mapping from several of the description techniques known up to now. For example, the

finite state machine given in Figure 3.1 models a simplified stop-and-wait protocol: a message is taken from a sending user (get) and then sent. If an acknowldgement message (ack) is received then the next message is sent (send), otherwise, a lost event leads to retransmission. The same protocol is specified in Figure 3.2 using a Petri net. It can be specified in CCS as :

```
sender  := get  ; sending
sending := send ;( ack ; sender + lost ; sending)
```

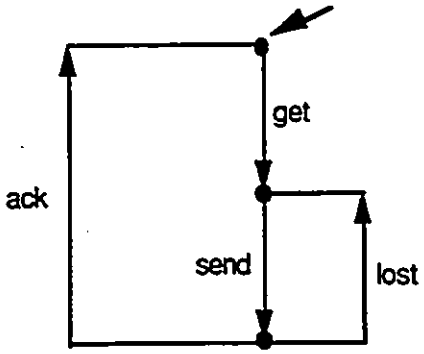


Figure 3.1: An FSM for the stop-and-wait protocol

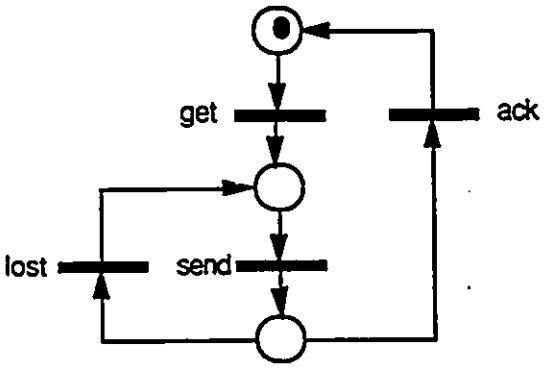


Figure 3.2: A Petri net model for the stop-and-wait protocol

CCS allows modular specifications to be built. For instance, in an FSM based model, a service based on the sender of Figure 3.1, and a corresponding receiver, would require that the two entities offering sending and receiving functions be connected together by a channel modeling the underlying service provider. This channel is not specified. It is implicit and usually assumed to be a FIFO channel.

In CCS, such a service would be a composition of a sender, a medium, and a receiver. All three elements can be fully and independently specified as expressions in the CCS language:

```
service := ( sender | medium | receiver)
```

Furthermore, the semantics of the language provides for an execution model for such a composition (see Chapter 9) .

Moreover, specification of a service with two identical peer entities can be done easily by creating two instances of a single protocol entity specified as a process. This is done by means of a special renaming operator.

CCS can be said to be a black-box oriented specification technique since it allows to encapsulate processes inside others. This is a useful feature for software specification and design. The communication between the encapsulated entities is not observable from the outside. This allows to analyze a system in terms of its observable input/output interactions. This capability was enforced in CCS by a proof mechanism based on the notion of observation equivalence (see chapter 7).

Also, the nondeterminism is modeled easily and so the behavior of systems having "spontaneous" transitions can be formalized.

This model has inspired many researchers who work in the area of formal specification models for distributed systems [BLO 84], [BOU 85], [CAS 85], and [HEN 85]. For example, in [HEN 85] a notion of acceptance is defined as a means to describe concurrency and nondeterminism. In [LAR 88], CCS is combined with probabilistic choices making the model suitable for concurrent systems where probabilistic behaviors must be specified.

In our opinion, CCS contains desirable features of specification languages to a greater extent than other languages.

3.2 Overview of CCS and CCS*

In this and the next sections we give an overview of CCS. The description includes semantic aspects of the language. We illustrate them with examples.

The simplest process is *stop* which does nothing. It models deadlock.

A more complex process has a set of **gates** through which it can communicate with its environment. A process is declared by means of a **process abstraction** which consists of a **process identifier** followed by its list of parameters, an operator denoted by :=, and its body called a **behavior expression**. A process may have parameters passed to it by value from a calling process. Also a process may call itself recursively.

The basic element of a process behavior is the **action offer** (or simply action) which consists of a gate name and a (possibly empty) list of events of two types : *!V* meaning that the process is offering a value *V*, and ?X:t meaning that the process is expecting a value of type t. However in one action we may have several data exchanges. For instance the action `a!true?X:int` is an action which consists of offering the value `true` at gate `a` and simultaneously expecting a value of type `int`. If another process wants to communicate by participating in this action, it must accept a value of type `boolean` and offer a value of type `int`. We say that the two processes offer **matching actions**. The result of the matching will be to bind variables for the process expecting values for them.

These types of actions are called **observable actions** since they are observed (or controlled) by the environment. The other type of actions are the **internal actions** represented by i the occurrence of which is "spontaneous" and not observed by any environment. They are normally used to model internal events such as shutdowns, lost events, timeouts, internal choices or internal communications between processes.

A process may execute an action and evolve to its "next state". This sequencing is described by means of the **sequencing operator ;**. For instance, process `buffer1` defined as:

```
buffer1 := in?X:int ;  out!X ;  buffer1
```

specifies the behavior of a one place error-free simplex buffer. It inputs a value of type `int` (integer) at gate `in`, outputs it at gate `out`, and loops.

A process may contain a choice between different actions to be executed. This is expressed by using the **choice operator** denoted by **+**. For instance, process `multiplexer` defined as:

```
multiplexer :=        in1?X:int ; out!X ; multiplexer
                    + in2?Y:int ; out!Y ; multiplexer
```

models a multiplexer that inputs messages either from gate `in1` or from gate `in2`, and outputs them at gate `out`. The environment may determine which alternative is taken by providing an offer on either gate `in1`or gate `in2`. However, if it offers both actions, then the choice is made nondeterministically.

In order for two processes to communicate, they must be linked by the **composition operator** ‖ and they must share some common gates. The composition means that they can synchronize and communicate by exchanging data. This communication results in an internal event. Also the composition describes the actual interleaving of the two processes. Although the two processes communicate with each other, there is no reason (unless otherwise specified) to implicitly prevent them from communicating with other processes. This type of communication is **two-way rendez-vous** where only two processes can synchronize. Other specification techniques such as CSP and LOTOS allow more than two processes to participate in what is called a **multi-way rendez-vous.**

Process `sender` defined below gets a message via gate `get`, outputs it via gate `out`, and loops:

```
sender :=  get?V:int ; out!V ; sender
```

In order to express the parallel composition of `sender` and `buffer1`, we must relabel their communication gates by a **relabelling operator.** Relabelling consists of applying a substitution to the list of the selected gates. Relabelling in a process creates a new

instance of an abstraction (or a declaration) called **process instantiation**. The relabeling of gate `in` by gate `com` in `buffer1` is `buffer1[com/in]` and the relabelling of gate `out` by gate `com` in process `sender` is `sender[com/out]`. This way the two processes can communicate via gate `com` according to the composition:

```
(sender[com/out] | buffer1[com/in] )
```

Processes `sender` and `buffer1` offer now matching actions at gate `com`. Therefore they can communicate. The result of this communication is the internal event. Since the composition includes also the possibility of communication with other processes as the result of their interleaving, the possibility of communication of `buffer1` and `sender` at gate `com` does not prevent other processes from participating. In order to restrict processes to communicate *only between them*, we can use the **hiding operator** denoted by \ followed by the set of gates to be hidden. These gates are no longer accessible from the outside. Only internal actions resulting from communications at these gates are possible. Figure 3.3 shows a service specification called `sender-receiver`. In this specification, we introduce process `receiver` that receives messages from `buffer1` and delivers them to a potential user via gate `deliver`. Processes `sender` and `buffer1` are those defined above. One notices also that the only externally visible gates are `get` and `deliver`.

```
simple_service :=
        (    sender[com1/out]
         | buffer1[com1/in,com2/out]
         | receiver[com2/in]
        ) \ {com1,com2}
 where
  receiver := in?Z:int ; deliver!Z ; receiver
```

Figure 3.3: A service specification

The system will evolve as follows:

1) In the first action, it will request a value V on gate `get` from the environment.
2) If the environment participates, this value will be sent by process `sender` via gate `out` (renamed by `com1`) to `buffer1` via its gate `in` (also renamed by `com1`). Then this value will be given by `buffer1` via gate `out` (renamed by `com2`) to process `receiver` via its gate `out` (also renamed by `com2`). Meanwhile since gate `get` is not hidden the environment can participate by adding a new value for V. But since `buffer1` is a one-slot buffer, this value cannot be transmitted until the first one is delivered.
3) The value that is passed to process `receiver` can be received by the environment via gate `deliver` which is not hidden. And the system behavior continues.

CCS* has an additional operator called the **disabling operator** denoted by [> whereby a process can take the place of another process during its execution. In general, it is used to model processes such as interrupts, disconnect phases and shutdowns. For instance in:

```
p [> danger
where
  p        :=   go ; p
  danger   :=   down ; out!"Byecruelworld" ; stop
```

process `danger` can take over at any time while process `p` is running.

Also in CCS* there exists a **guarding operator** denoted by -> used in expressions of the form: *[G]* -> *B* whereby behavior expression *B* is executed if the boolean guard *G* is true.

Nondeterminism

There are some cases where different choices can be made nondeterministically, either because they are internal to the system and therefore non observable (internal nondeterminism), or because they represent the same action prefixing different choices (external nondeterminism). The case of internal nondeterminism may occur as a result of communication of a process with several others. For instance the following specification shows a composition of three processes. The first and the third one send value 1 to the second one.

```
(out!1 ; stop) | (out?X:int ; stop) | (out!1 ; stop)
```

At the beginning the two senders will be ready to send values to the buffer. The system will have to decide between which one of the two offers is accepted.

Internal nondeterminism is shown in the next example where action `a?X:int` is possible in the two alternatives. The process will output either `2*X` or `X`. The choice will be made nondeterministically:

```
a?X:int ; b!2*X ; stop  +  a?X:int ; b!X  ; stop
```

3.3 An Introduction to the Semantics of CCS*

The semantics of CCS is described in operational terms [PLO 81]. The effect of an action on a process is expressed as a set of rules (also called **inference rules**). Each inference rule

determines the relation between processes, the operators linking them and an action that they can execute. In these rules observable actions belong to a set called *Act*. For instance, for the sequencing operator, the following rule:

(Rule 1) $B = a;B' \quad -a->B' \quad a \in Act \cup \{i\}$

says that B moves to B' if action a is executed. We also say that B' is a **derivative** of B or that B **infers to** B' by action a. In other words B' is the result of execution of this action, which will be executed if there is an environment that will communicate with process B at gate a.

The rule given above is rather a simplification of the reality. In fact, action a may contain variables to be bound. For instance action a could be: `gate?X:int`. In this case the execution of this action requires that the environment offers a value of type `int`. Therefore all free occurrences of variable `X` must be replaced by the value provided by the environment in B'. On the other hand, action a may be an output interaction such as `gate!2` for example. Then process B is offering that value to its environment. In this case there is no variable binding. However, if a is the internal event, then the process may evolve by "itself" without the participation of any environment.

The following rules:

(Rule 2) $A+B \text{ -a-> } A' \text{ if } A \text{ -a-> } A' \quad a \in Act \cup \{i\}$

(Rule 3) $A+B \text{ -a-> } B' \text{ if } B \text{ -a-> } B' \quad a \in Act \cup \{i\}$

say that the derivative of a choice between two processes for an action is the same as the derivative of the process which accepts that action. Notice the nondeterminism in the

case where both processes admit the same action.

The semantics of the parallel composition operator is expressed by means of the following rules:

(Rule 4) $A \mid B \xrightarrow{a} A' \mid B$ if $A \xrightarrow{a} A'$ $\quad a \in Act \cup \{i\}$

(Rule 5) $A \mid B \xrightarrow{a} A \mid B'$ if $B \xrightarrow{a} B'$ $\quad a \in Act \cup \{i\}$

(Rule 6) $A \mid B \xrightarrow{i} A' \mid B'$ if $A \xrightarrow{a1} A'$ and $B \xrightarrow{a2} B'$

and a1 "matches" a2 and a1, a2 $\in$ Act

Rules 4 and 5 specify the interleaving while Rule 6 specifies the communication. The exact definition of "matches" will be given in Chapter 4. These three rules allow to model systems that can run in parallel and possibly communicate at their common gates.

The disable operator is described by the following rules:

(Rule 7) $A \,[>\, B \xrightarrow{a} A' \,[>\, B$ if $A \xrightarrow{a} A'$ and a is not an exit action

(Rule 8) $A \,[>\, B \xrightarrow{a} B'$ if $B \xrightarrow{a} B'$

(Rule 9) $A \,[>\, B \xrightarrow{a} A'$ if $A \xrightarrow{a} A'$ and a is an exit action

An exit action is a particular action occuring at a particular gate (named δ) where a process can signal its successful termination (see also Section 4.4 for the description of the corresponding operator in ATCCS). Tese rules say that as long as A does not finish B can take over.

The inference rules for all the CCS* operators are given in [ISO 84].

The semantics also may allow, for a given expression that is built using the parallel composition operator, to express another behavior that is equivalent and that does not involve any concurrency. This amounts to **expanding** the initial expression using what is called the **expansion theorem** [MIL 80]. This theorem formalizes the idea that, given a behavior expression B, we can always find, using the Rules 1-9 and others, the set of behavior expressions that result from the executions of its actions. For example, assuming that actions `a1` and `a2` match, we give, in Figure 3.4, a definition of `p` and its derivations. In this figure, actions at the same level of indentation express choices, and the sequencing is shown by indenting to the right.

```
p := a1 ; b ; stop | a2 ; stop
     -a1->  b ; stop | a2 ; stop          By using Rule 1 and Rule 4
         -b-> stop | a2  ; stop           By using Rule 1 and Rule 4
            -a2->  stop | stop            By using Rule 1 and Rule 4
         -a2-> b ; stop | stop            By using Rule 1 and Rule 5
            -b-> stop | stop              By using Rule 1 and Rule 4
     -a2->  a1 ; b ; stop | stop          By using Rule 1 and Rule 5
         -a1-> b ; stop | stop            By using Rule 1 and Rule 4
            -b->  stop | stop             By using Rule 1 and Rule 4
     -i->  b ; stop | stop                By Using Rule 1 twice and Rule 4
         -b->  stop | stop                By Using Rule 1 and Rule 4
```

Figure 3.4: `Expansion of process p`

Given the derivations above, we can see that process `p` has the same behavior (in terms of the actions that it can execute) as process `p1` defined as:

```
p1 := a1;(b; a2;stop + a2;b;stop) + a2;a1;b;stop  + i;b;stop
```

Formally, the expansion theorem can be stated as follows. Given two behavior expressions A and B written as sums of sequence behavior expressions as follows :

$$A = \sum_{i \in IA} a_i ; A_i \qquad a_i \in Act \cup \{i\}$$

and $$B = \sum_{j \in IB} b_j ; B_j \qquad b_j \in Act \cup \{i\}$$

Then the composition A | B can be written as:

$$A \mid B = \sum_{i \in IA} \{a_i ; (A_i \mid B)\} + \sum_{j \in IB} \{b_j ; (A \mid B_j)\} + \sum_{i \in IA \; j \in IB} \{i : (A_i \mid B_j)\ a_i \text{ "matches" } b_j \text{ and } a_i \in Act \text{ and } b_j \in Act \}$$

The expansion theorem is very useful because it allows rewriting any CCS* behavior expression involving composition operators to an expression consisting in a sum of alternatives (called *summands*). Each summand shows what is its first action, and the behavior expression that results from executing that action. If the behavior expressions involved in the composition are finite (i.e. do not involve recursion), repeated applications of the expansion theorem will result in a behavior expression that contains no parallel composition operators. However if they are infinite, there are cases where we may not be able to eliminate the composition operators.

One of the most interesting characteristics of CCS is that it has not only a precise operational semantics, given by inference rules such as the ones given above, but also a

rich set of algebraic properties, such as the expansion theorems, that will be discussed later.

One should mention that CCS, CCS*, and LOTOS are based on the so called **interleaving semantics** where behaviors of compound processes result from the shuffling of behaviors of their components plus their communication. This type of semantics allows one to express expansion theorems such as the one outlined above. Other models are based on "true concurrency" where systems can execute in "parallel".

Chapter 4:

An Atomic Calculus of Communicating Systems

4.1 Introduction and Motivation

In previous chapters we gave reviews of the syntax and semantics of CCS. We also discussed LOTOS briefly.

In these description techniques, systems are described in terms of actions that they can perform in common with the environment, actions which are assumed to be atomic. However in some instances we may want to extend the concept of **atomic action** to compound actions. For example in the semantics of CCS, the parallel operator specifies interleaving at the elementary action level. In other words, elementary actions are only viewed as atomic or noninterruptible. Although larger atomic actions (used to implement critical sections) can be specified by using constructs such as semaphores, we contend that this presents some disadvantages, among others from the point of view of modular design. For example, the parallel composition of two processes may

introduce some unwanted interleaving, thus limiting the usefulness of this way of composing modules.

Furthermore, one of the main concepts of step-wise development of systems is to be able to progressively expand what appears to be a single action A at a high level of abstraction into a functionally equivalent, possibly complex compound action B at a lower level of abstraction. This presents a problem when concurrency is present: because, if A is considered to be atomic, so must be B in order to be equivalent to it. Semaphores can of course be used for this purpose, but it is cumbersome to introduce them in the stepwise development process. This problem also affects LOTOS.

Our solution to this problem has been to introduce a primitive concept of atomicity in CCS. We have extended CCS by some operators apt to define noninterruptible composition of actions. In the resulting new calculus, called **ATCCS** for **ATomic CCS** [OBA 87b], one can specify behaviors of systems with some sub-behaviors being atomic or non-interruptible.

One of the advantages of our calculus is to make it possible to map specifications from ESTELLE to LOTOS by adding similar constructs to LOTOS as we did to ATCCS.

In ESTELLE there is the concept of block of actions that are supposed to be executed in one thread without disruption. This is a transition in ESTELLE's terminology. A transition in ESTELLE has the form shown in Figure 4.1 [ISO 85] where:

- <state$_1$> specifies the current state,

- <state$_2$> specifies the next state,

-<input_action> is an event, the occurrence of which enables the transition to be activated.

-<condition> is a condition that must hold in order for an activated transition to be executed.

-the body of the transition is the sequence of actions, <action $_1$>,....,<action$_n$> included in the begin-end block. The actions may include input/output interactions such as sending an acknowledgement, starting a timer, or any other statement of the language. The block should be considered as a critical region that must be executed without interruption in any context.

from <state$_1$>
when <input_action>
provided <condition>
to <state$_2$>

begin
<action$_1$>
...
<action$_n$>
end

Figure 4.1: A simplified syntax of a transition in ESTELLE

Such a construct is a natural one and is easy to handle for building implementation oriented specifications. In a transition we may have complicated structures or calls to procedures. Unfortunately it is not possible to directly map such a construct into CCS or LOTOS sequences of actions. The semantics of these description techniques may imply that the sequences of action can be interrupted by (or interleaved with) any other

sequence of actions if the process that is specified is placed in a context of parallel composition.

Also available in ESTELLE are primitive functions used to model service primitives. Such functions are also viewed as "atomic".

By introducing the concept of atomic actions in CCS, we make it possible to describe transitions as critical regions and therefore a direct translation from ESTELLE into ATCCS becomes possible.

In Chapter 5 we show the use of such a mapping from a specification originally written in ESTELLE into another written in ATCCS, and the mapping to LOTOS can be done in the same way with the corresponding modifications to the language. Basically a transition can be mapped to the atomic sequence of actions that constitutes this transition. Although we do not develop this point further in our thesis we believe that the concepts we have developed can be extended to a general procedure for this mapping .

Atomic actions can also be useful in the specification of other types of distributed systems such as data base management systems. In these systems we have the concept of **transaction** which may involve a number of actions or even a number of processes to be executed as a unit in order to avoid losing consistency of data. We discuss this application of ATCCS in Chapter 6.

4.2 Syntax of the Calculus

In order to define the syntax of the calculus we use the following notations:

Notations:

- B, B', B_1, B_2, ..., represent behavior expressions. That is expressions that describe the behavior of a system in terms of its actions.

- e_1, e_2, ... , represent value expressions (considered as terms written in an abstract data specification language such as ACT ONE [EHR 85]).

- An action of a behavior is represented by an **action denotation** which consists of a gate name optionally followed by a sequence of *input-output* events: $g\alpha_1...\alpha_n$. As in LOTOS an *input* is denoted by the symbol ? and an *output* is denoted by !. The input action for a variable X specifies the type of this variable. That is we have:

  $\alpha_i = ?X_i : t_i$ where t_i is a type identifier or

  $\alpha_i = !v_i$ where v_i is a value expression.

  The variables X_i are to be bound by values of the specified type t_i during a communication. For example, `a!2?X:int` denotes the action of offering the value 2 and expecting an integer for variable X. The action takes place at gate `a`.

- We have two particular types of actions.

  Internal actions are represented by the letter **i**.

  Exit actions are used by processes to signal **successful termination**. This action is used by a process that terminates and "exports" value expressions. The semantics of the **exit** will not be formally defined until later in the thesis. Note that we use the action label **exit** instead of the symbol δ of CCS* and LOTOS.

- Every type has its identifier. For example `int` can be considered as an identifier for integer values. For every type identifier t we define Dom(t), **the domain** of t, as being the set of possible values of that type according to a type definition that we do not define in this thesis.

- For an action a we define :

label(a) = g if $a = g\ \alpha_1 ... \alpha_n$

label(i) = i

label(a) = exit if $a = exit(e_1,.....e_n)$

- The set of labels is represented by L. The set of possible actions is designated by **Act**. Actions are represented by a, b, ... The internal action **i** and the **exit** action do not belong to Act. Note that Act is not necessarily finite.

- S represents a substitution of the form $[a_1/a'_1,.....a_n/a'_n]$ where a_i and a'_i are gate names. The intuitive meaning of S is that a_i replaces a'_i , for $1 \leq i \leq n$.

- We also have substitutions of values for variables of the form $[e_1/X_1,.....e_m/X_m]$ where e_i are value expressions and X_i are variable names.

- Process names are represented by lower case identifiers such as p,q.... ○

Formal Grammar

Spec ::= $p(X_1,....X_n) := B$

| **atomicproc** $p(X_1,....X_n) := B$

B ::= stop
| $exit(e_1,....e_n)$
| exit
| a;B $a \in Act$
| a*B $a \in Act$
| i;B
| i*B
| B+B
| [G]->B where G is a boolean value expression
| B||B
| B|LB
| B|RB
| B>>B
| B>> **accept** $X_1 : t_1, ... X_n : t_n$ **in** B
| B|>B
| BL>B
| B\L where L is a set of gates
| B[S]
| $p(e_1,....e_n)$ where $e_1,.....\ e_n$ are value expressions

We assume the following increasing sequence of precedences :

\. relabelling
>>
|>. L>
||, |R, |L, <>
+
->
:, *

In case of equality of precedences we use right-to-left evaluation. For example, the expression: `a * b ; c ; p [ a1 /a, b1 /b]+ c ; p \ { c }` is interpreted as :

`(a * (b ; (p [ a1 /a, b1 /b])))+ (c ; (p \ { c } ))`

4.3 An Introduction to the Semantics of ATCCS

We describe the semantics of a calculus that allows us to specify atomicity. In order to do that, we extend CCS* with a **strong sequencing operator** denoted by *. For instance `a*b;stop` means that after action a is executed, action b *must immediately* follow in this order in any context and then the process stops. The semantics of this new operator must be defined in terms of the two CCS* operators that involve concurrency, i.e. parallel composition and disabling. To do this, we have defined auxiliary operators expressing the fact that, in the presence of strong sequencing, evaluation of an expression must continue along the sub-expression where strong sequencing is present.

In this section we give an introduction to the semantics of our calculus. The full semantics will be given in Section 4.4.

The strong sequencing operator has the following law:

$$a * B \text{ -a-> } B$$

meaning that a*B **derives** to B if action a is executed.

In order to have a complete axiomatization of our calculus, we add auxiliary operators among which we have:

i) The **left composition operator**, denoted by |L, is like the CCS* composition operator except that its takes its next action from the left process.

ii) The **right composition operator**, noted |R, takes its next action from the right process.

iii) The **communication operator**, noted <>, models communication within atomic actions.

In terms of derivations, the semantics of these three operators can be introduced as follows:

1: (a;A) |LB -a-> A || B

2: (a*A) |LB -a-> A |LB

3: A |R (a;B) -a-> A || B

4: A |R (a*B) -a-> A |RB

5: a*A <> a*B -i-> A <> B

This semantics is not complete of course as we have to define the effects of these three operators on all behavior expressions that can be obtained by the grammar given above.

The use of the left (and right) composition, and communication operators seems a natural way to define the semantics of our calculus. Other authors use a similar concept [BER 86a].

Note that these operators may not be found in a given ATCCS expression. They are used to define the semantics of strong sequencing combined with composition by the following derivations:

6: (a*A || B) -a-> A |LB

7: (A || b*B) -b-> A |R B

8: (a; A || a;B) -i-> A || B

9: (a* A || a*B) -i-> A <> B

Rule 6 states that, after execution of action a, the next action should be taken from A as defined in the semantics of the left composition operator in Rule 1. Rule 7 says that the next action should be taken from the right process as defined by the right composition operator. Rules 8 and 9 express the communication between processes. This will be discussed later on.

In order to model systems that should not be disabled in the middle of a particular sequence, we introduce a new disable operator that is called **left disabling operator**, denoted by **L>**. This operator says that the next action is to be taken from the left process. The disable operator allows a process to take over another one during its execution. There is a notion of non-interruptibility in this operator. The semantics of the disable with atomicity is expressed as follows: If an atomic sequence is disabled by a process then the disabling should be delayed until the end of the sequence. The operational semantics of the disable operation is expressed in terms of the following rule:

10: A L> B -a-> A' [> B if A-a-> A'

Strong sequencing with disable operation can be defined as follows:

11: (a * A [> B) -a-> A L> B

Just as the right and left composition operators, the **left disabling operator** denoted by **L>** is only used to define the semantics of the disabling in the presence of atomicity. Intuitively this means that a process cannot be disabled in the middle of an atomic sequence of actions.

4.3.1 Examples

Before going on to the formal definition of the semantics of these operators, we provide some examples of their use. In the examples we will use the following identities that will be studied later on:

B ||| stop = B

stop ||| B = B

stop [> B = B

The composition operators will be described later on in Section 4.4. In the following examples, indentation expresses the sequencing and alternatives on the same indentation level express a choice between derivations. Notice that this type of representation is similar to the way TTCN language represents the temporal ordereing of events for the description of test suites [PRO 88].

Example 4.1

```
a*b;stop || c*d*f;stop
    -a-> (b;stop |L c*d*f;stop)
        -b-> (stop || c*d*f;stop)
            -c-> d*f;stop
                -d-> f;stop
```

```
            -f-> stop
      -c-> (a*b;stop |R d*f;stop)
          -d-> (a*b;stop |R f;stop)
            -f-> (a*b;stop || stop)
              -a-> b;stop
                -b-> stop
```

The starting expression in Example 4.1 specifies a composition between two atomic sequences `a*b` and `c*d*f`. As we can see from the derivations, action `a` is always immediately followed by action `b` and action `c` is always immediately followed by `d`, which is always immediately followed by `f`. The two atomic sequences are executed in any order with respect to each other. Notice that if, for example, the environment offers `c` after having offered `a`, the result is a deadlock.

Example 4.2

```
a*b;stop ||c;d;stop
        -a-> (b;stop |L c;d;stop)
           -b-> (stop ||c;d;stop)
            -c-> d;stop
             -d-> stop
         -c-> (a*b;stop ||d;stop)
             -d-> (a*b;stop ||stop)
               -a-> b;stop
                -b-> stop
             -a->(b;stop |L d;stop)
                -b-> (stop ||d;stop)
                  -d-> stop
```

The starting expression in Example 4.2 is a composition between an atomic sequence `a*b` and a nonatomic one `c;d`. As we can see sequence `a*b` cannot be interrupted while sequence `c;d` be interrupted by interleaving with `a*b`.

Example 4.3

```
a*b;stop || a*c;stop
      -a-> (b;stop |L a*c;stop)
         -b-> (stop || a*c;stop)
           -a-> c;stop
             -c-> stop
      -a-> (a*b;stop |R c;stop)
          -c-> (a*b;stop || stop)
            -a-> b;stop
              -b-> stop
       -i-> ( b ; stop <> c ; stop)   (Deadlock!)
```

This example shows the case of a composition with atomic actions that start communicating. When communication becomes impossible in the middle of an atomic sequence, deadlock results.

Example 4.4

```
a*b;stop [> c*d;stop
     -a->(b;stop L> c*d;stop)
       -b-> (stop [> c*d;stop)
         -c-> d;stop
           -d-> stop
```

```
-c-> d;stop

  -d-> stop
```

In this example, sequence `a*b` is not disabled by sequence `c*d`.

Discussion

The formal semantics of the * operator is complicated by the presence of the choice operator, since a process can be defined as a choice between two different alternatives, one of which is atomic and the other not. For instance the expression :

```
(a * b ; stop + c ; g ; stop) || a * f ; stop      (E1)
```

shows a case where we cannot decide which parallel composition to apply after each action (i.e. left or right or parallel composition). If left composition is applied, then `c;g` is created as atomic, while if normal composition is applied, then `a*b` will be interrupted. In order to deal with this kind of expressions, we will introduce an operational semantics with attributes which will carry the *atomicity nature* of each expression.

4.4 An Inference System for ATCCS

In this section we give the semantics of some of the most important operators of the calculus. The other operators will be specified later on. The semantics are expressed in an operational way. This allows one to derive the set of actions that a process can

perform by the operational rules associated with the operators that are involved in the behavior expression of the process. These rules are called **inference rules** [PLO S1].

Notation

In order to be able to build an inference system for ATCCS, we use *parameterized inference rules*. The parameter can be considered as a synthesized attribute which will indicate the nature of the derivative of an expression. This attribute has two possible values: "at" to indicate an atomic derivation and "int" to indicate an interruptible derivation. This means that for a given (non deadlocked) process whose behavior is described by the expression B, we can find a behavior B' and an action a such that:

B -a-(at)-> B' or B -a-(int)-> B'

For simplicity we will use the name *any* to denote a variable that ranges over {*at*,*int*}. Each occurrence of this variable represents the same value within the scope of the rule where it appears. The values *int* and *at* represent the **atomicity of derivations.**

We first give a definition that we will use in the description of the semantics.

Definition 4.1

We say that B **admits a derivative** if there exists a behavior expression B' and an action a ∈ Act ∪ { exit, i } such that:

B -a-(any)-> B'

This is is called a **derivation for B.**

If there is no such a derivation for B we say that B is **deadlocked.** On the other hand if B' is deadlocked and *any* is equal to at. we say that B' has reached a **premature deadlock** ○

The notion of **premature deadlock** will be used to characterize atomic sequences of actions that reach a deadlock before completion.

Inference rules:

In the rules given below, all actions, represented by a belong to Act $\cup$ {i} unless otherwise specified.

Stop process:

the *stop* process has no inference rule.

Process *stop* has no action. It is deadlocked. Therefore it has no derivative.

Sequencing:

s-1: $i ; B \text{-}i\text{-}(int)\text{->} B$

s-2: $i * B \text{-}i\text{-}(at)\text{->} B$

s-3: $g\,\alpha_1 \ldots \alpha_n ; B \text{-}g\,v_1 \ldots v_n\text{-}(int)\text{->} B[e_1/X_1, \ldots, e_m/X_m]$

s-4: $g\ \alpha_1 \ldots \alpha_n * B \text{-}g v_1 \ldots v_n\text{-}(at)\text{->} B[e_1/X_1, \ldots, e_m/Xm]$

where $v_i = e_i$ if $\alpha_1 = !e_i$ for $1 \le i \le n$

and $v_i \in Dom(t_i)$ if $\alpha_i = ?X_i : t_i$ for $1 \le i \le n$

and $e_i = v_j$ if $\alpha_i = ?X_j : t_j$ for $1 \le i \le n$ and $1 \le j \le m$

Rule s-1 states that the derivation by sequencing with the internal event is non-atomic. Rule s-2 says that the derivation by strong sequencing is atomic. Rule s-3 (resp. Rule s-4) defines the derivation by sequencing (resp. strong sequencing) in the case of general actions. They also specify the binding of variables of certain types to values belonging to that type. The bindingcan occur with any value belonging to that type. We take $X_1, X_2, \ldots, X_m$ to be the free variables occuring, in this order, in an action. As in LOTOS, we assume that each variable can occur at most once in each action.

Choice:

c-1: A+B -a-(any)-> A' if A -a-(any)-> A'

c-2: A+B -a-(any)-> B' if B -a-(any)-> B'

Rules c-1 and c-2 state that the derivative of a choice and its atomicity is the same as the derivative and the atomicity of one of its alternatives.

Restriction:

r-1: A \ L -a-(any)-> A' \ L if A-a-(any)->A' and label(a) $\notin$ L

This operator restricts the set of actions that a process can execute to those actions that occur at gates that are not "hidden". Rule r-1 says that an action has an effect only when it is not hidden.

Relabelling:

In the definition of this operator we use a substitution S which maps a set of gates to another set of gates. This is not necessarily a one-to-one mapping. This operator creates a copy of a given process behavior. We write S(a)=b if b is substitution of a.

rn-1 : A[S] -a-(any)-> A'[S] if A-b-(any)->A' and S(b)=a

where S is a gate substitution.

The effect of relabelling an action in a process is the same as the effect of the relabelled action on the relabelling of the process.

Guarding:

g-1: [G] -> B -a-(any)-> B' if G=true and B-a-(any)->B'

Composition:

Composition:

pc-1: A|[]|B -a-(at)-> A' |[L]| B if A-a-(at)-> A'

pc-2: A|[]|B -a-(at)-> A |[R]| B' if B-a-(at)-B'

pc-3: A | | B -a-(int)-> A' | | B if A -a-(int)->A'

pc-4: A | | B -a-(int)-> A | | B' if B-a-(int)->B'

pc-5: A | | B -i-(int)-> A' | | B' if A-a-(int)->A' and B-a-(int)->B' $a \in Act$

pc-6: A | | B -i-(at)-> A'<>B' if A-a-(at)->A' and B-a-(at)->B' $a \in Act$

Note that, according to this semantics, atomic sequences can synchronize only with atomic sequences.

Left composition:

lc-1: A |L B -a-(int)-> A' | | B if A-a-(int)->A'

lc-2: A |L B -a-(at)-> A' |L B if A -a-(at)-> A'

Rules lc-1 and lc-2 state that left composition forces the next action to be taken from the left expression. The atomicity of derivation is the same as the atomicity of derivation of the left process.

Right composition:

rc-1: A |R B -a-(int)-> A | | B' if B-a-(int)-> B'

rc-2: A |R B -a-(at)-> A |R B' if B-a-(at)-> B'

The right composition forces the next action to be taken from the right process. The atomicity of derivation is preserved.

Communication:

com-1: A <> B -i-(int)-> A' || B' if A-a-(int)-> A' and B-a-(int)-> B' a ∈ Act

com-2: A <> B -i-(at)-> A' <> B' if A-a-(at)-> A' and B-a-(at)-> B' a ∈ Act

com-3: A <> B -i-(at)-> A' <> B if A-i-(at)-> A'

com-4: A <> B -i-(at)-> A <> B' if B-i-(at)-> B'

The communication operator guarantees matching between atomic actions. As long as they match in an atomic way, the communication continues. Once one action terminates or the sequences stop matching we obtain a deadlock. The semantics of this operator also eliminates unwanted internal actions. For example an atomic sequence like `a*b` will match with sequence `a*i*b` since the internal event does not participate in interactions with the environment. This mechanism is illustrated in the following example:

Let `p1 = a*i*b;c;stop` and `p2 = a*b;d;stop`. The composition `p1||p2` will offer, among others, the following sequence of derivations:

```
p1||p2 -i(at)-> (i*b;c;stop <> b;d;stop)        ( communication at a)
          -i(at)-> (b;c;stop <> b;d;stop)       ( action i in p1)
                -i(int)-> (c;stop || d;stop)    ( communication at b)
```

Disabling:

Disabling:

d-1: A |> B -a-(at)-> A' L> B if A-a-(at)-> A' and label(a) ≠ exit

d-2: A [> B -a-(int)-> A' [> B if A-a-(int)-> A' and label(a) ≠ exit

d-3: A [> B -a-(any)-> A' if A-a-(any)-> A' and label(a)=exit

d-4: A [> B -a-(any)-> B' if B-a-(any)-> B'

When the disabled process starts an atomic sequence, the next action is to be taken from it (Rule d-1) if it has not terminated. If a process is terminated then the disabling has no effect (Rule d-3).

Left disabling:

ld-1: A L> B -a-(at)-> A' L> B if A-a-(at)-> A'

ld-2: A L> B -a-(int)-> A' [> B if A-a-(int)-> A'

ld-3: A L> B -a-(any)-> A' if A-a-(any)-> A' and label(a) = exit

The left disabling forces the action to be taken from the left process. Thus, a process cannot be disabled during an atomic sequence.

Process instantiation:

pi-1: $b(e_1,....e_n)$-a-(any)->B' if $B[e_1/X_1,.....e_n/X_n]$ -a-(any)->B

where b is declared as: $b(X_1,....X_n) := B$ or as atomicproc $b(X_1,....X_n) := B$.

$X_1,....X_n$ are formal parameters, and $e_1,.....e_n$ are value expressions.

For every process we assume that there is a declaration called *process abstraction*. A call of this process is called a *process instantiation*. A process instantiation behaves like the body of its abstraction in which the parameters are replaced by value arguments. The "atomicproc" declaration (discussed in Section 4.7.1) is used to declare primitive processes.

We have an additional operator, called **enable** and denoted by >>, that is given in the syntax and not described in this section. We shall define its semantics and use in Section 4.7.1. Briefly, this operator allows to define sequencing between two processes in the same way that ";" defines sequencing between an action and a process. This operator relies on the use of the **exit** action that allows one process to transfer both control and data to another process.

4.5 The Problem of Deadlock in Atomic Sequences

Rule pc-1 (resp. Rule pc-2) states that if a process A (resp. B) derives in an atomic way to another process A' (resp. B'), then the composition of process A (resp. B) with another process B (resp. A) has an atomic derivation and the next action of the composition will be taken from A' (resp. B'). These two rules express the interleaving.

A problem arises in the case of atomic sequences that start by matching and then stop matching after a few steps. This situation may lead to a premature deadlock. The concept of atomic action, as commonly used, does not admit failure in the middle [TAY 86]. This is known as the *all-or-nothing principle*. For example, an atomic action may consist of updating a data base the integrity of which may be damaged by the occurrence of premature deadlocks in the action. The problems of recovery from failure are addressed in Chapter 6. For example, in the expression `a*b;stop || a*c;stop` the

two atomic actions start by matching with action `a`, and the result of this matching is the expression `b;stop<>c;stop`. Then we will have no further action (i.e. a premature deadlock). This type of problem is discussed in Chapter 6 where we provide a mechanism by which we can recover from such a situation if there is another alternative. We may have several execution paths of sequences of actions that start matching. It may happen that we choose one that leads to a premature deadlock. If we knew of the existence of such a deadlock, we would have avoided this path. In general, a good specification should avoid such a situation when possible. An atomic sequence of actions is meant to represent one particular *function* that should be performed as a whole. Therefore we expect that each atomic sequence should match with another sequence with the same atomicity nature and the matching should lead to a termination of both sequences.

Example 4.5

We use this new formalism to derive the behavior of process `E1` (given in the discussion at the end of Section 4.3.1):

```
(a*b;stop + c;g;stop) || a*f;stop

     -a-(at)-> (b;stop |L a*f;stop)
        -b-(int)-> (stop || a*f;stop)
          -a-(at)-> f;stop
             -f-(int)-> stop

     -c-(int)-(g; stop || a*f;stop)
         -g-(int)-> (stop || a*f;stop)
            -a-(at)-> f;stop
              -f-(int)-> stop
```

```
-a-(at)-> (g;stop |R f;stop)
     -f-(int)-> (g;stop || stop)
       -g-(int)-> stop

-a-(at)-(a*b;stop + c;g;stop) |R f;stop
     -f-(int)-> (a*b;stop + c;g;stop) || stop
        -a-(at)-> b; stop
             -b-(int)->stop
        -c-(int)-> g;stop
             -g-(int)-> stop
```

-i-(at)-> (b;stop <> f;stop) and no other possible action (deadlock) !

4.6 Formal Properties of ATCCS

4.6.1 Expansion Theorems

In this section we formulate expansion theorems for ATCCS similar to Milner's expansion theorem for CCS [MIL 80] (see Section 3.3). In these theorems we assume that each process can be written as a choice (i.e. a summation) between terms, also called *summands* in [MIL 80], where each one can be prefixed either by a sequencing operation or by a strong sequencing operation.

Let us consider two processes A and B defined as follows:

$$A = \sum_{i \in IA} a_i;A_i + \sum_{i \in AA} a'_i{}^{*}A'_i \qquad a_i, a'_i \in Act \cup \{i\}$$

and

$$B = \sum_{j \in IB} b_j;B_j + \sum_{j \in AB} b'_j \bullet B'_j \qquad b_j, b'_j \in Act \cup \{i\}$$

Definition 4.2

Each term in the summation is called a **summand**. Summations with no summands are identified with *stop*. ○

For brevity, language elements such as enable were not included. The expansions for these elements are straight forward, and are left to the reader as an exercise.

Expansion theorems will be expressed as equalities between expressions. In fact, these equalities are strong equivalences in the sense of Section 7.3. The theorems given below are stated without proofs. In Section 7 we will give a proof of the first expansion theorem using the notion of *observation equivalence*. The proofs of the other theorems follow the same line.

Theorem 4.1 (Composition).The following holds:

$$A \,||\, B = \sum_{i \in AA} a'_i \bullet (A'_i \,|[\Sigma]|\, B) + \sum_{j \in AB} b'_j \bullet (A \,|R\, B'_j)$$

$$+ \sum_{i \in IA} a_i;(A_i \,||\, B) + \sum_{j \in IB} b_j;(A \,||\, B_j)$$

$$+ \sum_{i \in IA \; j \in IB} \{i;(A_i \,||\, B_j) \;;\; a_i = b_j \text{ and } a_i \in Act \text{ and } b_j \in Act\}$$

$$+ \sum_{i \in AA\, j \in AB} \{i \bullet (A'_i <> B'_j) \ : a'_i = b'_j \text{ and } a'_i \in Act \text{ and } b'_j \in Act\} \qquad \circ$$

This expansion theorem says that the composition of two processes consists of the possible interleavings of their behaviors plus the possibility of their communication. However atomic sequences cannot be interrupted. Once a communication starts, both communicating processes go to their next step.

Theorem 4.2 (Left and right compositions).The following holds:

$$A \,|L\, B = \sum_{i \in IA} a_i : (A_i \,||\, B) + \sum_{i \in AA} a'_i \bullet (A'_i \,|L\, B)$$

$$A \,|R\, B = \sum_{j \in IB} b_j : (A \,||\, B_j) + \sum_{j \in AB} b'_j \bullet (A \,|R\, B'_j) \qquad \circ$$

Theorem 4.3 (Communication).The following holds:

$$A <> B = \sum_{i \in IA\, j \in IB} \{i : (A_i \,||\, B_j) : a_i = b_j \text{and } a_i \in Act \text{ and } b_j \in Act\}$$

$$+ \sum_{i \in AA\, j \in AB} \{i \bullet (A'_i <> B'_j) \ : \ a'_i = b'_j \text{and } a'_i \in Act \text{ and } b'_j \in Act\}$$

$$+ \sum_{a'_i = i} a'_i \bullet (A'_i <> B) + \sum_{b'_j = i} b'_j \bullet (A <> B'_j) \qquad \circ$$

As a result of these theorems we have the following applications: Theorems 4.1, 4.2, and 4.3 and property i-a) of Theorem 4.7 (to be given later) allow us to write expression:

```
E1 =  (a * b ; stop + c ; g ; stop) || a * f ; stop as:
E1 =    a*b;a*f;stop  +       c;(g;a*f;stop + a*f;g;stop)
     + a*f;(a*b;stop + c;g;stop) + i ; (b ; stop <> f ; stop)
```

and expression

```
E2 =        (a*b;stop || a*b;c;stop) as:

E2 =        a*(b;stop |L a*b;c;stop) + a*(a*b;stop |R b;c;stop)
     +      i*(b;stop <> b;c;stop)

   =        a*b;(stop || a*b;c;stop)
     +      a*b;(a*b;stop || c;stop)
     +      i*i; (stop || c;stop)

   =        a*b;a*b;c;stop
     +      a*b;(  a*(b;stop |L c;stop) + c;(a*b;stop || stop) )
     +      i * i ; c ; stop

   =        a*b;a*b;c;stop
     +      a*b;(a*b;c;stop + c;a*b;stop)
     +      i*i;c;stop
```

Notice that the communication operator leads to premature deadlocks in cases like:

```
a*b ; stop || a *c ; stop  and  a*b*c ; stop || a*b;c; stop
```

The following theorems express the effect of the other operators on behavior expressions.

Theorem 4.4 (Disabling). The following holds:

$$A \,[\!>\, B = \sum_{\text{label}(a_i)\neq\text{exit}} a_i;(A_i \,[\!>\, B) + \sum_{\text{label}(a'_i)\neq\text{exit}} a'_i{}^{\bullet}(A'_i \,L\!>\, B)$$

$$+ \sum_{\text{label}(a_i)=\text{exit}} a_i;A_i + \sum_{\text{label}(a'_i)=\text{exit}} a'_i{}^{\bullet}A'_i + \sum_{j\in IB} b_j;B_j + \sum_{j\in AB} b'_j{}^{\bullet}B'_j \qquad \circ$$

Theorem 4.5 (Left disabling). The following holds:

$$A \,L\!>\, B = \sum_{\text{label}(a_i)\neq\text{exit}} a_i;(A_i \,[\!>\, B) + \sum_{\text{label}(a_i)=\text{exit}} a_i;A_i$$

$$+ \sum_{\text{label}(a'_i)=\text{exit}} a'_i{}^{\bullet}A'_i + \sum_{\text{label}(a'_i)\neq\text{exit}} a'_i{}^{\bullet}(A'_i \,L\!>\, B) \qquad \circ$$

Theorem 4.6 (Restriction). Let L be a set of gates, the following holds:

$$A \setminus L = \sum_{a_i \in L} a_i;(A_i \setminus L) + \sum_{a'_i \in L} a'_i{}^{\bullet}(A'_i \setminus L) \qquad \circ$$

For example, using this theorem, we get:

```
(a;b;c;stop + i*a;b;stop + i;d;b;stop + a*b*c*d*e;stop) \ {c,d}

= a;b;stop + i*a;b;stop + i;stop + a*b*stop
```

On should notice, as illustrated by this example, that the restriction operator cuts the atomic sequence `a*b*c*d*e;stop`. This seems natural as this atomic sequence can no longer synchronize on `c` and `d` because they are hidden.

4.6.2 Algebraic Properties of ATCCS

In order to avoid lengthy proofs, in this section we only consider "basic" ATTCS with operators ;, *, ||, |L, |R, and <>.

ATCCS has some useful algebraic properties. The following summarizes some of them:

Theorem 4.7 For all ATCCS behavior expressions A, B, and C, the following holds:

i) a) $A \,||\, stop = stop \,||\, A = A$
   b) $A \,|L\, stop = A$
   c) $stop \,|R\, A = A$
   d) $stop \,|L\, A = stop$
   e) $A \,|R\, stop = stop$
   f) $A <> stop = stop <> A = stop$

ii) a) $A \,||\, B = B \,||\, A$

   b) $A \,|L\, B = B \,|R\, A$

   c) $A <> B = B <> A$

iii) a) $A \,||\, (B \,||\, C) = (A \,||\, B) \,||\, C$
   b) $A \,|R\, (B \,|L\, C) = (A \,|R\, B) \,|L\, C$ ○

Proofs: In Chapter 7 we give proofs of some of these properties by using the notion of observation equivalence. ❑

4.7 Atomic Processes

We use the strong sequencing operator described in Section 2.1 in order to define **atomic processes** (i.e. noninterruptible processes). In our definition we do not impose any restriction on the nature of theses processes. They can be infinitely looping (e.g. `p := a * p )`. We do not ask questions such as "How big an atomic action should be ?". Our atomic actions may take a long time or even be non-ending. A good discussion of this topic can be found in [TAY 86].

4.7.1 The Enable Operator

Until now we have considered only sequencing and strong sequencing between an elementary action and a process. In order to be able to express sequencing from a process to another we use, as in LOTOS, the **enable operator** denoted by >> which allows a process to enable another one upon its termination. In order to do that, an *exit* construct is provided. It uses a gate called *exit* where a process signals its termination. In order for a process B1 to enable another process B2, B1 must reach gate `exit` before B2 can start. More formally, we express this as follows:

Enabling:

e-1: A >> B -a-(any)-> A' >> B if A-a-(any)-> A' and label(a) ≠ exit

e-2: A >> B -a-(any)-> B'
if B-a-(any)->B' and A-exit-(int)->A'

e-3: A >> accept X_1: t_1,....X_n: t_n in B -a-(any)-> A' >> accept X_1: t_1,....X_n: t_n in B
if A-a-(any)-> A' and label(a) ≠ exit

e-4: A >> accept X_1 : t_1,....X_n : t_n in B -a-(any)-> B'
if B(e_1 / X_1,.....e_n / X_n]-a-(any)->B'
and A-exit(e_1,.....e_n)-(int)->A'

where the semantics of the *exit* action is defined as follows:

ex-1: exit -exit-(int)-> stop

ex-2: exit(e_1,..... e_n) -exit(e_1,..... e_n)-(int)-> stop

Rules e-1 and e-3 state that as long as A does not terminate it still continues its execution . If A terminates, then B takes over (Rules e-2 and e-4). Moreover, a terminating process may export value expressions to the second process by the use of the *accept* construct (Rule e-4).

This definition of the enabling operator allows processes `p >>c;stop` and `a*b;c;stop`, where `p` is declared as: `atomicproc p := a*b; exit`, to synchronize.

One notices the difference between this semantics and the corresponding one in CCS* and LOTOS. The *exit* in LOTOS introduces an extra internal event that has to be executed before one can execute the following process. The appearance of such internal event may have undesirable effects when simulating processes because a user will "see" internal events that he didn't explicitly specify. As a consequence of this difference when *exit* is a first action of a process, we may obtain a behavior that is not equivalent (in the sense of ≈ defined in Section 7.4) to the LOTOS one in which the internal event resulting from the exit and enables still exists. To illustrate this problem, let `p` =

(exit>>a;stop) [] b;stop. According to the semantics of CCS* and LOTOS, process p is observation equivalent (in the sense of = as applied to LOTOS and CCS) to:

```
i ; a ; stop [] b ; stop        (E1)
```

According to ATCCS, process p is observation equivalent (in the sense of =) to:

```
a ; stop [] b ; stop            (E2)
```

Note that E1 and E2 are not equivalent in the sense of CCS, LOTOS, and ATCCS.

In the general case, with our definition of the enable operator, we may find three expressions C, B_1, and B_2 such that B_1 and B_2 are observation equivalent but $C >> B_1$ and $C >> B_2$ are not. Here is an example: B_1=a;stop, B_2=i;a;stop, and C=(exit[]i;b; stop).

In order to detect such a situation, we can add semantic checks to see if an expression is well defined in that it never has an *exit* as its first action. This can be done by function *f_exit* is defined as follows:

f_exit(exit(e1,....,en))	= true
f_exit(a:B)	= false
f_exit(a*B)	= false
f_exit(A + B)	= f_exit(A) or f_exit(B)
f_exit(A [S])	= f_exit(A)
f_exit(A \ L)	= f_exit(A)
f_exit([E]-> B)	= f_exit(B)
f_exit(A || B)	= f_exit(A) or f_exit(B)
f_exit(A |R B)	= f_exit(B)
f_exit(A |L B)	= f_exit(A)
f_exit(A [> B)	= f_exit(A) or f_exit(B)
f_exit(A L> B)	= f_exit(A)
f_exit(A >> B)	= f_exit(A)
f_exit(b(E1,....,En))	= f_exit(Bp) if b is declared as $b(X_1,....X_n) := Bp$ or as atomicproc $b(X_1,....X_n) := Bp$

4.7.2 Defining Atomic Processes

In order to use **atomic processes** (which we also call **primitive processes**) we have an **atomic process declaration** syntactically described as:

atomicproc b(X1,....,Xn) := <Behavior Expression>

A primitive process is most likely to be used in the context of the enabling operator because it is the only operator that allows sequencial compositions between two processes. In order to check whether or not an atomic process is well declared, we use the following algorithm that checks the consistency of the atomicity of a process. We want to be able to statically check if a process declaration is consistent with its behavior expression. If a process is declared as atomic then the value of the predicate *atomic* must be true. Function *atomic* is recursively defined on behavior expressions:

atomic(stop)	= false
atomic(exit)	= true
atomic(a;B)	= true if B = exit
	= false otherwise
atomic(a*B)	= atomic(B)
atomic(A + B)	= atomic(A) and atomic(B)
atomic(A [S])	= atomic(A)
atomic(A \ L)	= atomic(A)
atomic([E]-> B)	= atomic(B)
atomic(A || B)	= atomic(A) and atomic(B)
atomic(A |R B)	= atomic(A) and atomic(B)
atomic(A |L B)	= atomic(A) and atomic(B)
atomic(A [> B)	= atomic(A) and atomic(B)
atomic(A L> B)	= atomic(A) and atomic(B)

atomic(A >> B)	= false
atomic(b(E1,....,En))	= true if b is declared as atomic
	= false otherwise

For example the following process has an inconsistent declaration :

```
atomicproc p := a*b;stop + c * d ; p
```

because of the sequencing operation that occurs before the recursive call.

Note also that any process containing *stop* as a part of an atomic sequence is not atomic: a good definition of an atomic process ends with an *exit* preceeded by the sequencing operation ; as in:

```
atomicproc p := a * b ; exit + c * d *p
```

This choice is motivated by the fact that *exit* does not add anything to the atomicity of the sequences in the atomic process. On the other hand it is more likely that specifiers would expect the atomic processes to successfully terminate.

In our definition we do not admit the enabling operator >> in an atomic process because this would have required introducing two enable operators, one that is atomic and another that is not.

4.8 Examples

4.8.1 The Mutual Exclusion Problem

A critical section (C.S. for short) of a process is a sequence of actions that should not overlap with other actions of other processes. C.S. will be represented as atomic

sequences. In the example given below, process A has C.S. $cs_{11}{}^{*}\ cs_{12}$ and process B has C.S. $cs_{21}{}^{*}cs_{22}$.

$$A := \quad a_1 \ ; \ cs_{11} \ ^{*} \ cs_{12} \ ; \ A$$

$$B := \quad b_1 \ ; \ cs_{21} \ ^{*} \ cs_{22} \ ; \ B$$

Using the Expansion Theorems we can find a process equivalent to the composition A || B.

$$A \ || \ B = \quad a_1 \ ; \ (cs_{11}{}^{*}cs_{12}; A \ || \ B) \ + \ b_1 \ ; \ (A \ || \ cs_{21}{}^{*}cs_{22}; B)$$

Let $X = (cs_{11}{}^{*}cs_{12}; A \ || \ B)$ and $Y = (A \ || \ cs_{21}{}^{*}cs_{22}; B)$.

By expansion, we have:

$$X = \quad cs_{11}{}^{*}(cs_{12}; A \ |L \ B) \ + \ b_1; (cs_{11}{}^{*}cs_{12}; A \ || \ cs_{21}{}^{*}cs_{22}; B)$$

$$Y = \quad a_1; (cs_{11}{}^{*}cs_{12}; A \ || \ cs_{21}{}^{*}cs_{22}; B) \ + \ cs_{21}{}^{*}(A \ |R \ cs_{22}; B).$$

Let $U = (cs_{11}{}^{*}cs_{12}; A \ || \ cs_{21}{}^{*}cs_{22}; B)$.

We use the following identities:

$$(cs_{12}; \ A \ ) \ |L \ B = cs_{12}; \ (A \ || \ B)$$

and $\quad A \ |R \ \ (cs_{22}; \ B \ ) = cs_{22}; \ (A \ || \ B)$.

Therefore

$$X = \quad cs_{11}{}^{*}cs_{12}; (A \ || \ B) \ + \ b_1; U$$

$$Y = \quad cs_{21}{}^{*}cs_{22}; (A \ || \ B) \ + \ a_1; U$$

and

$$U = \quad cs_{11}{}^{*}(cs_{12}; A \ |L \ cs_{21}{}^{*}cs_{22}; B)$$

$$+\ \ cs_{21}*(cs_{11}*cs_{12};A\ |R\ cs_{22};B)$$

$$= \quad cs_{11}*cs_{12};\ (A\ ||\ cs_{21}*cs_{22};B)$$
$$+\ cs_{21}*cs_{22};\ (cs_{11}*cs_{12};A\ ||\ B)$$

$$= \quad cs_{11}*cs_{12};(\ \ a_1;(cs_{11}*cs_{12}\ ;\ A\ ||\ cs_{21}*cs_{22};B)$$
$$+\ cs_{21}*(\ A\ |R\ cs_{22};\ B))$$
$$+\ cs_{21}*cs_{22};(\ \ cs_{11}*(cs_{12};\ A\ |L\ B)$$
$$+\ b1;(cs_{11}*cs_{12};A\ ||\ cs_{21}*cs_{22};B))$$

$$= \quad cs_{11}*cs_{12};\ (a1;U + cs_{21}*cs_{22};(A\ ||\ B)\ )$$
$$+\ cs_{21}*cs_{22};\ (b1;U\ \ + cs_{11}*cs_{12};(A\ ||\ B)\ )$$

Finally we get :

$$A\ ||\ B = \quad a_1\ ;\ (\ \ cs_{11}*cs_{12}\ ;\ (A\ ||\ B) + b_1\ ;\ U\ )$$
$$+\ b_1\ ;\ (\ \ cs_{21}*cs_{22}\ ;\ (A\ ||\ B) + a_1\ ;\ U\ )$$

Thus, we can see that the resulting behavior is the interleaving of the behaviors of the two processes without overlapping of the critical sections.

4.8.2 Specifying a Semaphore

Semaphores are one of the most popular methods for defining critical sections. As we have seen in the previous example, our atomicity concept enables one to represent critical sections directly, without using semaphores. Therefore, users of ATCCS should have little use for this construct at the specification level. ATCCS, however, can also be

used as an implementation specification language, and there may be a need to introduce semaphores at that level.

We specify a semaphore by means of its primitives P and V (for *Wait* and *Signal* resp.) (Figure 4.2). They must be atomic. These primitives use a semaphore with an initial value n as a parameter. It represents the number of processes that can be in their critical sections at the same time. The P primitive allows to signal the acquisition of the semaphore and the V primitive allows to liberate it. The semaphore is described by means of a process. The value of the semaphore is assumed to be initially positive and can be decremented and incremented. All semaphore operations are represented as simple actions on gates.

In order to avoid that the two processes p1 and p2 communicate between them using the P and V primitives, we include a mechanism that allows them to identify themselves. Each process will give its identity when calling these primitives. The semaphore on the other hand will accept any process identifier. Each action will hold the identity (represented by parameter i) of the calling process. We assume that the process identifiers are integers. Without this mechanism the two processes may synchronize and enter their critical sections simultaneously.

Process P tests if the semaphore is positive, in which case it decrements it and exits, otherwise, if it is null, it blocks. Process V checks if the semaphore is null in which case it increments it. A system consisting of two processes that uses the semaphore for mutual exclusion to their C.S. is specified below. The C.S. consists of actions cs_{11} and cs_{12} for process p1 and of actions cs_{21} and cs_{22} for process p2 (see Figure 4.3). Process system can be represented using the diagram in Figure 4.4. Gates that don't reach the outside frontier are hidden.

```
atomicproc P(i)  :=   positive!i  * decrement!i ;  exit

atomicproc V(i)  :=   null!i  * increment!i  ;  exit


sem(n) :=      [n>0] -> positive?X:int  * sem_pos(n)

           + [n=0] -> null?X:int  * increment?X:int ; sem(1)

sem_pos(n) :=  increment?X:int ; sem(n+1)

                  + decrement?X:int ; sem(n-1)
```

Figure 4.2: Specification of a semaphore

Note that in CCS and LOTOS one could specify as in [MIL 80], much simpler semaphores than this one. We have chosen this relatively complex specification in order to demonstrate the power and the modularity of our model.

$$p1 := P(1) >> cs_{11}; cs_{12}; V(1) >> p1$$

$$p2 := P(2) >> cs_{21}; cs_{22}; V(2) >> p2$$

```
system := (p1||sem(1)||p2) \ {positive,decrement,null,increment}
```

Figure 4. 3: Specification of mutual exclusion with semaphores

One would like to prove that this system is correct according to the specification of the mutual exclusion. We will consider this problem in Chapter 7.

Before using a formal verification approach, we give a sequence of derivations showing how the mutual exclusion occurs. All actions except cs_{11}, cs_{12}, cs_{21}, and cs_{22} are hidden. Let `L = {positive,decrement,null,increment}` be the set of hidden gates.

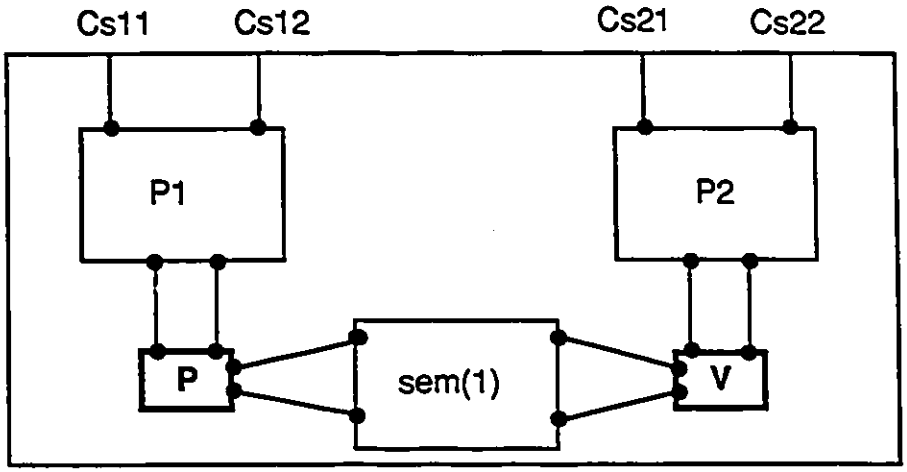


Figure 4.4: System structure for mutual exclusion

```
(p1 || sem(1) || p2 ) \ L

(1)    -i-(at)-> (action positive between P in p1  and  sem)

   ((decrement!1; exit >> cs11;cs12;V(1) >> p1) <> sem_pos(1) || p2 )\ L

(2)    -i-(int)-> (action decrement between P in p1 and sem_pos)

  ((exit >> cs11;cs12 ; V(1) >> p1) || sem(0) || p2)  \ L

(3)    -cs11-(int)->  (p1 enters its critical section)
```

((cs$_{12}$;V(1) >> p1) || sem(0) || p2) \ L

(At this stage p2 cannot enter its C.S. as the semaphore does not offer action positive)

(4) -cs12-(int)-> (p1 is still in its critical section)

((V(1) >> p1) || sem(0) || p2) \ L

(5) -i-(at)-> (action null between V in p1 and sem)

((increment!1 ; exit >> p1) <> (increment?X:int; sem(1)) || p2) \ L

(6) -i-(int)-> (action increment between V in p1 and sem)

((exit >> p1) || sem(1) || p2) \ L

We are back to the initial state. Now p2 will go.

(7) -i-(at)-> (action: positive between sem and P in p2)

((exit>>p1) || sem_pos(1)<>(decrement!2;exit>>cs$_{21}$;cs$_{22}$;V(2)>>p2)) \ L

(8) -i-(int)-> (action decrement between P in p2 and sem_pos)

((exit>>p1) || sem(0) || (exit >> cs$_{21}$;cs$_{22}$;V(2) >> p2)) \ L

(9) -cs$_{21}$-(int)-> (p2 enters its critical section)

((exit >> p1)|| sem(0) || (cs_{22}; V(2) >> p2)) \ L

In the sequence given above we showed a part of the behavior of the system. We could, of course, have considered different alternatives at different points of choice. For example, we could have allowed p1 to enter its critical section a second time (at point 7) and then p2 would have had to wait again.

We would now like to prove that this specification is correct by showing that it is equivalent to a specification of a process that never allows two processes to find themselves simultaneously in their critical sections. For this purpose, we use the expansion theorems on composition and hiding. We also use some formal properties on observation equivalence between processes, which will be discussed later on. A sketch of the proof goes as follows (where the identity represents the strong observation equivalence ≡ described in Chapter 7):

(p1||sem(1)||p2)\L =

i*((decrement!1;exit>>cs_{11};cs_{12};V(1)>>p1)<>sem_pos(1)||p2)\L
+ i*(p1||sem_pos(1)<>(decrement!2;exit>>cs_{21};cs_{22};V(2)>>p2))\L

= i*i;((exit>>cs_{11};cs_{12};V(1)>>p1)||sem(0)||p2)\L
+ i*i;(p1||sem(0)||(exit>>cs_{21};cs_{22};V(2)>>p2))\L

The semantic of the exit statement implies:

(exit >> B) = B for any process behavior expression B (*)

We then have:

$$(p1||sem(1)||p2)\backslash L =$$
$$i*i;((cs_{11};cs_{12};V(1)>>p1)||sem(0)||p2)\backslash L$$
$$+\ \ i*i;((p1||sem(0)||(cs_{21};cs_{22};V(2)>>p2))\backslash L$$

Consider the expression:

$$((cs_{11}\ ;\ cs_{12}\ ;\ V(1)\ >>\ p1)\ ||\ sem(0)\ ||\ p2)\ \backslash\ L.$$

Since actions cs_{11} and cs_{12} are not hidden, they can be offered. This expression corresponds to the state where p1 can enter its critical section. On the other hand p2 cannot enter its critical section since it starts by calling primitive P which starts with action positive. This action is not possible because it is not offered by sem(0). We can say the same thing about the expression:

$$(p1\ ||\ sem(0)\ ||\ (cs_{21}\ ;\ cs_{22}\ ;\ V(2)\ >>\ p2))\ \ \backslash\ L.$$

Therefore we can (informally) state that when one process is in its critical section, the other cannot enter its own critical section. A few steps further in the calculation we get (by expansion):

$$(p1\ ||\ sem(1)\ ||\ p2)\ \backslash\ L =$$
$$i*i;cs_{11};cs_{12};i*i;((exit>>p1)||sem(1)||p2)\backslash L$$
$$+\ i*i;cs_{21};cs_{22};i*i;(p1||sem(1)||(exit>>p2))\backslash L$$

Again, using property (*), we have:

(p1||sem(1)||p2)\L =

i*i; cs_{11};cs_{12};i*i;(p1||sem(1)|| p2)\L

+ i*i; cs_{21};cs_{22};i*i;(p1||sem(1)||p2)\L

In Chapter 7, we give the formal proof of the equivalence of the process above to another process where mutual exclusion is more explicit.

4.9 Comparison With Other Work

Concepts of atomic actions related to the ones presented in this thesis were studied in [BER 86a]. They were introduced as a theoretical algebraic framework for the specification of distributed systems with atomicity. As in ATCCS, one can describe atomic sequences of actions. Their semantics are based on an algebraic description in the ACP style (see Section 2.1.4.3). An operator denoted by : is introduced to model what corresponds to our strong sequencing operator. The effect of this operator is described in every context (e.g. parallel composition). For example, there is the equation:

ax |! Y=a:(x |LY)

which is close to rule 6 in Section 4.3. However the strong sequencing operator in our case was described in an operational way.

The model of [BER 86a] has not been used to define any particular language. Moreover, mainly finite processes were studied. Recursive processes were defined as sets of sequences of different (possibly infinite) lengths. There is no notion of communication as it exists in CCS and LOTOS. The notion of matching between actions of processes that

seems natural in behavioral techniques does not exist as such. This makes this theory too general to be used as a framework for specifying communicating systems where the communication requires synchronization at a particular gate and possibly an exchange of values. However, in ACP_τ (a generalization of ACP) [BER 86b] it is possible to describe communication and abstraction in the case of finite processes.

In [ITO 88], there is a model, called *dCCS* for *differential CCS* where each action a is written as a sequence of two actions *a_begin* and *a_end*. The model is meant to refine the notion of actions in a top-down manner. The model tries to map CCS expressions to equivalent dCCS ones using a function called *df*. For example for the sequence operator ";" we have:

df[a ; B] = a_begin ; a_end ; df[B]

Therefore, the model is used as target from CCS and it has its own notion of differential equivalence that allows equivalence between processes only if they accept the same sequences of actions by using a derivation denoted by $-->_d$, from to B to B', defined as follows:

B -a->$_d$ B' if B -a_begin-> B_1 -a_end-> B' for some B_1

Although the author developed a verification method based on the notion of differential equivalence, he does not provide a way of expanding behavior expressions where there is a notion of interleaving of a (composite) action with other actions. In this sense, this model is less general than ours. The notion of equivalence tries to eliminate the interleavings that still exist in the sense of CCS, using the derivation

$-->_d$ given above. One must also say that the model considers a notion of refinement where an action can be replaced by a regular expression using a method described in [MIL 82].

In [GOR 89] two levels of a model called A^2CCS are defined: the *low level* and the *high level*. First we consider the low-level model because it is closer to ATCCS. In this level, a state can be either *visible* or *invisible*. Upon entering an atomic sequence of actions, a process (also called *agent*) enters an *invisible state*. When the sequence terminates, it enters a *visible state*. In an invisible state a sequence of actions cannot be interrupted. There is a sequencing operator defined as: $a;B$ -a-> B where a is an action. On the other hand, an action can be underscored as in $\underline{a}\,;B$ to mean that it starts an atomic sequence of underscored actions. Upon executing this type of actions a process is marked as invisible using symbol *. This type of sequencing is defined as follows:

$\underline{a}\,;B$ -a-> *B

Other rules for the composition operator are defined. In these rules, the visibility of compound processes is considered. Since the rules are rather complicated to present, we illustrate them by means of an example. For example, the expression: a;b;stop||$\underline{c}$;d;stop has the following derivations:

```
a;b;stop||c;d;stop -a-> *(b;stop)||c;d;stop
                       -b->  stop||c;d;stop
                             -c-> stop||*(d;stop)
                                  -d-> stop||stop
                   -c->  a;b;stop||*(d;stop)
```

```
-d->  a;b;stop||stop

      -a-> *(b;stop)||stop

            -b-> stop||stop
```

Where we can see that when a state is marked as invisible it becomes noninterruptible.

In the high-level model, derivations are based on strings of actions rather than simple actions (in the sense of CCS). This type of derivations, also called *transactions*, should lead a process from one visible state, which corresponds to the beginning of the string, to another visible state. The model also uses action underscoring as in the low level model. The sequencing operation is defined by means of the following two rules:

a ; B -a-> B

a; B -aS-> B' if B-S-> B'

The second rule states that we concatenate elementary underscored actions to form one string of actions. The composition operator is defined as follows:

A || B -S-> A' || B if A -S-> A'

B || A -S-> B || A' if A -S-> A'

A || B -S-> A' || B' if A -S1-> A' and B-S2-> B' and S_syn(S1,S2,S)

where S_syn(S1,S2,S) is a *synchronization function* that checks if S1 and S2 synchronize and S is the result of their synchronization. This function is actually the heart of the model. It merges two matching actions one by one in S1 and S2 into the internal event i. For example we have `S_syn(abc,abc,iii)` which reduces the matching of the three actions to three consecutive internal events.

Although the model is general and has some nice properties from the point of view of the notion of abstraction and the mapping from the low level model to the high level model and vice versa, it has some drawbacks.

The low level model is somewhat similar to ATCCS in that the composition does not allow a process to interrupt a sequence of actions if this sequence is executed by a process which is in a "noninterruptible" (i.e. *invisible*) state. This was achieved in our model by means of *, |L and |R operators and attributed inference rules. However in their model, in order to decide whether an expression is in a given state, it is necessary consider the type of the behavior expressions involved. The authors did not (clearly) indicate how this is actually done. This may take some additional control mechanisms. Also there is no recovery from communication deadlock within atomic sequences of actions.

On the other hand, in the high-level model, the synchronization function *S_syn* as it is defined does not always reduce matching sequences into a sequence of internal events. For example in `S_syn(abc,abcdef,iiidef)`, matching occurs in the first three actions only. This corresponds to our premature deadlock. This means that if action labels `d`, `e`, and `f` are hidden using a restriction operator (also included in the model) then the composition leads to a deadlock state. The restriction operator is defined in such a way that it will prevent acceptance of any sequence that contains hidden actions. For example this operator will not allow the sequence `iiidef` to be taken if we hide `d` or `e` or `f`. However if the restriction is not applied this will lead to a premature deadlock. Also this operator checks any sequence to see if an action that is hidden belongs to that sequence no matter how long it is. There is no backward recovery mechanism as in ATTCS (see Chapter 6). Moreover the model does not handle infinite atomic sequences of actions. We know that this type of sequences should not exist in

modelling transactions, but if they exist the semantic rules will never terminate (we have a similar situation in the enhancement to our model described in Chapter 8).

In [BOU 89] an atomic action is introduced with the notion of refinement as a means by which one can replace elementary actions by complex processes. An atomic action transforms the system state without intermediate states. On the other hand, an action also defines operations on system data.

An atomic action is "fetched" in one step indivisible manner and its execution is defined by: p -p-> stop meaning that every atomic action should terminate (*stop* in this model is not necessarily a deadlock, it means termination of a process).

The two notions of fetch and execution are represented by:

(p,s) --> (p',s') if p-u-> p' and s !-u-> s'

where u is a given (complex) sequence of actions, p is process and s is the system state. The derivation !-> defines the effect of an action on data and -> is defined as the execution of the action.

The definition of elementary actions is extended to include behavior expressions. The semantics of the language is defined as follows:

a -a-> stop an action a is seen as a process.

Sequencing between processes is defined by:

p ; q -u-> p'; q if p-u-> p'

p;q -v-> q' if q-v-> q' and p = stop

The first rule expresses regular continuation of a process p. The second rule expresses termination of process p and continuation with next process q.

The composition is defined as:

p || q -u-> p' || q if p-u-> p'

p || q -u-> p || q' if q-u-> q'

p || q -(u || v)-> p' || q' if p-u-> p' and q -v-> q'

Atomic actions are defined by means of operations like [p] defined by:

[p] -p-> stop

The notation `[a;b]` denotes an atomic process composed of subprocesses a and b. For example, if `p = ([a;b] || c )`, we have the only following derivations:

`p -[a;b]-> c -c-> stop` and `p -c->[a;b]-[a;b]-> stop`

In order to define atomic processes the author suggests the syntactic notation:

(let x = [p] in q) meaning the substitution q[[p]/x].

For example the expression `( let x = [a;b;c;stop||f;stop] in ( x; g; stop))` defines a process where x is replaced by the expression `[a;b;c;stop||f;stop]`. Moreover x is defined as atomic.

The model is quite interesting from the point of view of the formalization of the notion of refinement with atomicity and concurrency. It also tackles the problem of interpretation as well as execution. The author also gave an extension of this work in [BOU 88] for a more general and more complex model. In spite of that the model suffers from the lack of constructs that can actually help in the design process, because there is no notion of hiding or synchronization (at least as described in [BOU 89]). Also there is no semantics for either recursion or instantiation.

As a general assessment, we find that, with respect to these models, ATTCS has the following advantages:

1) It is fairly close to CCS, which means that much of CCS theory can be adapted. This is a considerable advantage given the amount of research that has been developed around the CCS model over the years.

2) Its is a complete language, capable of expressing nontrivial examples. This point will be developed in Chapter 5, where it is shown how ATCCS can be used in the system design process.

3) It has a theory of recovery as presented in Chapter 6.

Of course, as it is pointed out repeatedly in our work, ATCCS also has its own limitations, and we are looking forward to further research developments in this area.

As we have seen the concept of atomicity as introduced in ATCCS is modeled by means of built-in operators: *, |L, <>, L> and |R. One could ask whether or not the same result

could be obtained by using other operators capable of imposing some constraints on the allowed sequences of actions of the system. One of the most popular operators of this type is the *full synchronization*operator that was introduced in LOTOS [ISO 87]. This operator is the basis for the now commonly called *constraint oriented specifications* [VIS 88]. The semantics of this operator (noted also | |) in LOTOS is:

A | | B -a-> A' | | B' if A -a-> A' and B -a-> B'

which can be phrased as follows:

> *At any stage, the sequences of actions of the composition A | | B is the intersection of the set of sequences accepted by A with the set of sequence accepted by B.*

Intuitively this means that process A is a constraint to process B and vice-versa.

One could think at first that the fact that an action should always be followed by another one is a type of constraint. However, using the | | operator to specify atomicity will not be enough for two reasons:

1) The constraint to a process is global. That is, we cannot have a constraint that can be used to specify that the sequence `a;b` is atomic sometime and non-atomic some other time. Also if the constraint stops the whole system stops.
2) The constraint of a process in a context does not stop the whole action from being interrupted if placed in another - parallel composition - context.

Therefore, it can be seen that this construct is unable to model atomicity in the general case.

4.10 Behavior Trees and State Diagrams

We now introduce a graphical notation for **behavior trees**. Although not related to the material discussed so far, it is based on the semantic model introduced in this chapter, and it will be used in the following chapters.

The behavior of systems can be expressed by giving their behavior expressions as used all along in this thesis. Another alternative (which is not as general) could be to use nondeterministic finite state automata. In CCS literature, however, a common way to represent such behavior is by means of behavior trees (or **derivation trees**. Behavior trees express the *dynamic aspect* of the behavior of a system. They are constructed using the set of actions that can be performed by the system and an implicit notion of state.

If we have behavior trees of two processes we can obtain by expansion the behavior tree of their composition by any of the composition and communication operators defined previously.

The behavior tree of a process may be infinite if the process has non-finite behavior (e.g. looping). In finite-state automata, infinite behaviors can be represented by loops. In infinite trees we can represent loops by adding naming convention to trees using identifiers.

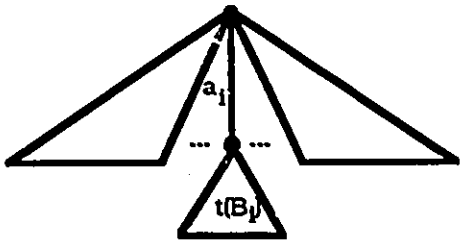
Definition 4.3

Given a behavior expression B we construct a behavior tree of B, denoted by t(B) in the following way:

t(stop) = ● (the tree consisting of one node)

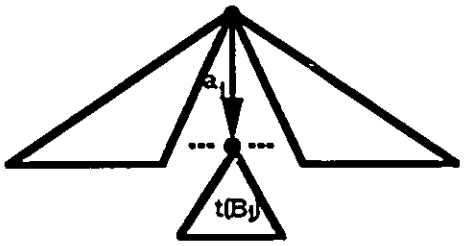
for all a_i and B_i such that B -a_i -(int)-> B_i then :

t(B) =



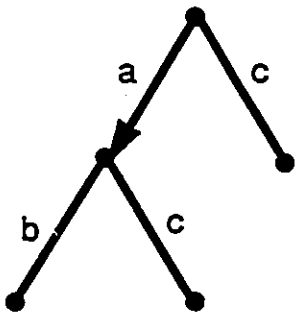
Furthermore if B -a_i -(at) -> B_i for some i then we draw its corresponding branch by adding an arrow to the edge corresponding to a_i in t(B) as shown below:

t(B) =

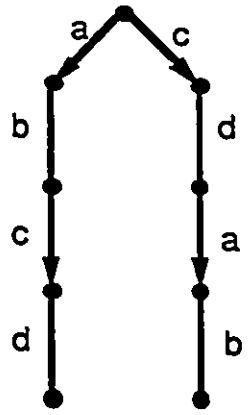


○

For example the expression: `a*(b;stop + c;stop ) + c;stop` has the following tree:

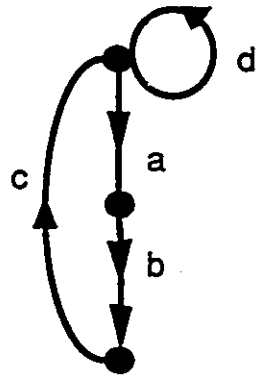


Using the semantics of the calculus, one may construct behavior trees of more general behavior expressions (e.g. behavior expressions involving composition operators) . For example the behavior expression : `a * b ; stop || c * d ;stop` has the following tree:



State Diagrams

Sometimes it is more convenient to draw behavior trees using state diagrams. In these diagrams, states correspond to behavior expressions and edges to actions in a derivation. This representation is mainly helpful for processes that loop. As these diagrams have directed edges we may add an arrow showing their direction. At the end of each transition we may add an extra arrow to show strong sequencing when necessary. For example process p defined as `p := a ; b * c ; p + d ; p` has the following state diagram:



Chapter 5:

Atomicity and System Design

Many papers have been written with relation to the problem of formulating design methodologies for distributed systems (among others [BOC 79],[CHO 83],[SCO 86],[ZAF 80], and [ZAV 85]). Most of them were based on models such as finite-state machines, Petri Nets, or high-level programming languages. The semantics of the model or the language used in these methodologies is of major concern in this design.

In CCS* (and therefore in LOTOS) we have some high-level composition operators that can be used in splitting the system into modules that can be :

- executed sequentially using the enabling operator >>.

- executed concurrently using the composition operator ||.

- executed by having a process disabling another using operator [>.

In doing so, the design process becomes in principle easier [SCO 86]. However, usually it is not easy to split systems into several concurrent components because of the high level of concurrency that is involved. We illustrate these constraints in the following (rather simple) example.

5.1 A Simple Protocol

The first example illustrates a problem with the parallel composition operator of CCS, problem which can be solved by using our strong sequencing operator. Consider a very simple protocol that consists of a sending and a receiving entity. One might think of specifying it as the parallel composition of a sender and a receiver process. Unfortunately, this will normally not be possible because of the interleaving that is involved in the language, namely the sender and the receiver will be able to interleave at each elementary operation and there is no way of preventing them from doing so unless we use some implementation objects, such as semaphores, for synchronization. In other words, the language does not allow to compose the two functions in a simple way in order to obtain a protocol that provides both of them.

The protocol entity that we want to specify should provide a data sending service as well as a data receiving service. The entity will get data from a user via gate `from_user`, send it via a medium at gate `to_medium`, receive data from a medium via gate `from_medium` and deliver it to a user via gate `to_user`. If we write the specification as follows:

```
sender   := from_user ; to_medium ; sender
receiver := from_medium ; to_user ; receiver
protocol := sender || receiver
```

a possible sequence of derivations for the protocol will be as follows :

```
protocol

    -from_user-(int)->       (to_medium ; sender || receiver)

    -from_medium -(int)->    (to_medium ; sender || to_user ; receiver)

    -to_user -(int)->        (to_medium ; sender || receiver)

    -from_medium -(int)->    (to_medium ; sender || to_user ; receiver)

    -to_user -(int)-> ....   (to_medium ; sender || receiver)
```

Therefore, it is possible for the protocol to keep on getting messages without the being able to transmit messages received from the sending user who executes `from_user`. We would like instead the sending function to get messages from the user and transmit them right after, as it is the role of the sender to send these messages upon getting them from the user. In order to have such a possibility, we must prevent some unnecessary interleaving of sending with receiving. By using the strong sequencing, we can specify that execution of action `from_user` must be followed by the execution of the action `to_medium` in an atomic way. In other words, we use the strong sequencing operator to make the sending process send in an atomic manner what it has received from the sending user before receiving any new message from the medium. This ability is provided in FDTs like ESTELLE but unfortunately not in CCS* or LOTOS.

This is described in the following specification:

```
sender := from_user * to_medium ; sender

receiver := from_medium * to_user ; receiver
```

The protocol entity will be:

```
protocol := (sender || receiver )
```

Using the expansion theorem, we have the following equivalences:

```
sender || receiver
      =          from_user * ( to_medium ; sender |L receiver)
           +     from_medium * (sender  |R to_user ; receiver)

      =          from_user * to_medium ; (sender || receiver)
           +     from_medium * to_user ; (sender || receiver)
```

And finally:

```
protocol :=   from_user * to_medium  ; protocol
           +  from_medium * to_user ; protocol
```

In other words, by combining in parallel the two processes, we have obtained a system that provides the two functions, with the desired behavior.

Notice that we could add control to the protocol by the use of guards. This relates to the problem of fairness as well. Without guards one particular function (e.g. the sending function) could always execute, preventing the other function (e.g. the receiving function) from executing. We don't discuss these aspects in this example.

5.2 The Alternating Bit Protocol

In this example we specify the Alternating Bit Protocol (ABP for short)[BAR 69]. The entity that we describe has a **sending view** as well as a **receiving view**. The concept of view was introduced in this context by [ZAV 85].

We design two specifications that both use atomic processes. In the first one, atomic actions are used to model transitions. In the second one, we show how atomicity and parallel composition allow us to compose the specification from several modules, each one representing a view. This shows the usefulness of the concept of atomicity in two different design methodologies.

The protocol has four parameters: the sending buffer, the receiving buffer, the send counter and the receive counter. We assume available functions for handling messages, queues, and so on (e.g. `add`, `remove`, ...) as described in section 5.2.1. We also use variables for the send counter and the receive counter. They are used to identify the outgoing messages and the incoming ones respectively. There is also a timeout mechanism represented by the actions `timer`, `startimer`, and `stoptimer`. Finally, we have functions that test equality when necessary. All these functions can be described in terms of abstract data type equations, for example in the language ACT ONE.

In this example we use the enable operator in conjunction with the accept statement introduced in Chapter 4.

In these specifications we use the concept of **primitive process** as a process that is declared using the atomic process declaration denoted by the keyword *atomicproc*.

5.2.1 The First Specification

This specification is inspired by one written in ESTELLE in [ISO 85]. Our example shows that, by using our calculus, it is easy to translate an ESTELLE specification into an ATCCS specification.

The alternating bit protocol represents the behavior of an entity that is able to send and receive data in a full duplex environment. Each message consists of two parts: the sequence number (the alternating bit), and the data. The entity sends a message with number 0 and waits for its acknowledgement. If the latter is received within a certain period of time, message with number 1 is sent. After the acknowledgement for message 1 is received, a message with number 0 is sent, and so on. If a timeout occurs, the entity sends again the last message sent and previously buffered in a sending buffer. Each entity can also receive messages with their sequence numbers. It uses a receive counter that represents the expected message number of the next message to be received. When the sequence number in the received message matches the expected one, the message is acknowledged and buffered for delivery. Whenever a correct message is received, the receiving counter is incremented (modulo 2). The state diagram of this specification is given in Figure 5.1. In this diagram, each transition (which is an atomic process) is followed by its body which may include assignments of parameter values of the protocol.

We use the following primitives:

`send_data:`	to send data through the network.
`receive_data:`	to receive data through the network.
`receive_ack:`	to receive acknowledgement messages.

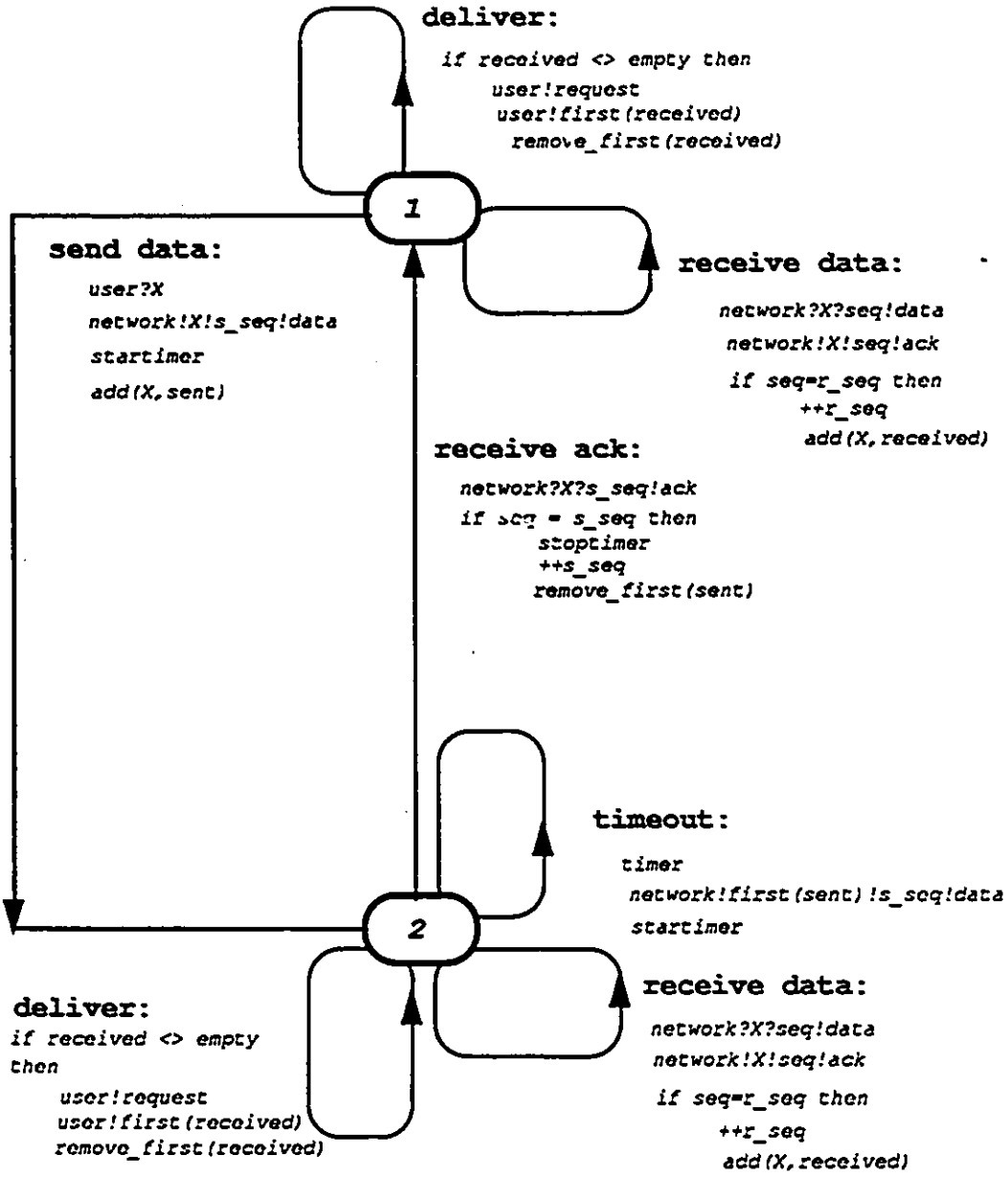


Figure 5.1: A state diagram of the ABP

`deliver:`	to deliver data to the receiving user.
`timeout:`	to handle timeouts.

and the following functions and variables:

`++number:`	to increment a number (modulo 2).
`remove_first(b):`	to remove the first element from b.
`add(M,b):`	to add message M to (fifo) buffer b.
`sent:`	is the buffer of messages sent but not acknowledged yet.
`received:`	is the buffer of messages received but not delivered yet.
`r_seq:`	is the sequence number of the received message.
`s_seq:`	is the sequence number of the sent message.

The model for this specification is given in Figure 5.2. The descripitons of the atomic processes are as follows:

`send_data:`

This process receives data from the sending user, adds it to `sent` buffer, appends to the data the send sequence number and a constant field called `data` to distinguish it from acknowledgement messages, and then sends the whole message through gate `network`. This process also starts a timeout.

`receive_data:`

receives a new message (of type `data`), checks its corresponding sequence number, and adds the message to `received` buffer if required.

`receive_ack:`

receives an acknowledgement message (of type `ack`), increments the sequence number for the next message to be sent, and removes the acknowledged message from `sent` buffer.

`deliver:`

delivers the first message of buffer `received`.

`timeout:`

sends the last message sent but not acknowledged before the timeout.

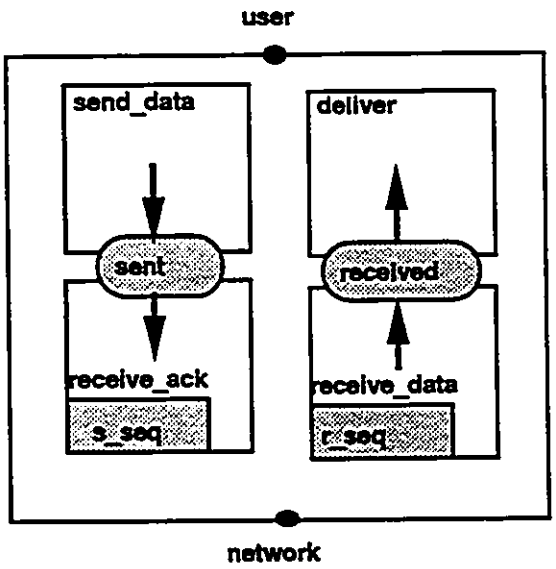


Figure 5.2 A model for the ABP.

In addition, there are two processes corresponding to the two states. They are:

- abp which corresponds to state 1.
- wait_for_ack which corresponds to state 2.This process waits for an acknowledgement message to arrive. During the waiting time messages can be delivered, new data or acknowledgement messages can be received, and timeouts can occur.

The system specification is as follows:

```
abp_entity :=      abp(0,0,empty,empty)

where

abp(s_seq,r_seq,sent,received):=                    (* state 1 in Figure 5.1 *)

        send_data(s_seq)                            (* from state 1 to state 2 *)
                >> accept new_sent
                in wait_for_ack(s_seq,r_seq,new_sent,received)

        + deliver(received)                         (* from state 1 to state 1 *)
              >> accept  new_received
                in  abp(s_seq,r_seq,sent,new_received)

        + receive_data(r_seq,received)              (* from state 1 to state 1 *)
                >> accept new_r_seq, new_received
```

```
            in abp(s_seq,new_r_seq,sent,new_received)

where

  wait_for_ack(s_seq,r_seq,sent,received):=      (* state 2 in Figure 5.1 *)

       timeout(s_seq,sent)                       (* from state 2 to state 2 *)
            >> wait_for_ack(s_seq,r_seq,sent,received)

       + deliver(received)                       (* from state 2 to state 2 *)
            >> accept new_received
            in  wait_for_ack(s_seq,r_seq,sent,new_received)

       + receive_data(r_seq,received)            (* from state 2 to state 2 *)
            >> accept new_r_seq, new_received
            in wait_for_ack(s_seq,new_r_seq,sent,new_received)

       + receive_ack(s_seq,sent)                 (* from state 2 to state 1 *)
            >> accept new_s_seq, new_sent
            in abp(new_s_seq,r_seq,new_sent,received)

 atomicproc send_data(s_seq)  :=

    user?X:data*network!X!s_seq!data*startimer;exit(add(X,sent))

 atomicproc timeout(s_seq,sent) :=
```

```
    timer*network!first(sent)!s_seq!data*startimer;exit

atomicproc deliver(received) :=

 [received ≠ empty ] -> user!request*user!first(received)
                      ; exit(remove_first(received))

atomicproc receive_data(r_seq,received) :=

   network?X:data?seq:int!data * network!X!seq!ack ;
      ( [seq = r_seq] -> exit(++r_seq, add(X,received))
        +
        [seq ≠ r_seq ] -> exit(r_seq,received)
       )

atomicproc receive_ack(s_seq,sent) :=

   network?X:data!s_seq!ack;stoptimer;exit(++s_seq,remove_first(sent))
```

Processes send_data, receive_data, receive_ack, and timeout were declared as atomic processes because we do not want them to be interrupted in any context. Process send_data for example has to perform in an atomic manner. If that process were not atomic, it would have been possible for a clock to send a timeout before the sender actually sends the message through the network. This illustrates the usefulness of the concept of atomicity.

5.2.2 Design by Parallel Composition

In this design of the alternating bit protocol we split the service specification into two functions (or views): sending and receiving.

The receiving function does the following:

receives data, stores them in the receiving buffer, and delivers data from the receiving buffer to the receiving user.

The sending function does the following:

gets new message from the sending user, sends them via the network interface, waits for acknowledgements, and times out if necessary.

The model is shown in the diagram given in Figure 5.3.

We have two processes representing the two functions. These processes use the atomic processes defined in the previous section. The specification is:

```
abp(s_seq, r_seq, sent, received) :=

        sending(s_seq, sent)
    || receiving(r_seq, received)
```

where

```
sending(s_seq, sent) :=
```

```
    send_data(s_seq)
     >> accept new_sent  in wait_for_ack(s_seq,new_sent)
where

  wait_for_ack(s_seq,r_seq,sent,received):=

        timeout(s_seq,sent)
           >> wait_for_ack(s_seq,r_seq,sent,received)

     + receive_ack(s_seq,sent)
        >> accept new_s_seq, new_sent in sender(new_s_seq,new_sent).

 receiving(r_seq,rcv_buffer) :=

     deliver(received)
     >> accept  new_received  in receiving(r_seq,new_received)

     + receive_data(r_seq,received)
         >>accept new_r_seq, new_received
              in receiving(new_r_seq,new_received)
```

For each one of the views defined above, there is relevant data (showed in dashed areas in Figure 5.3) associated with it. This is information needed for the view's proper operation. It is represented as parameters in the specification. The relevance is derived from the functions of the processes for that view.

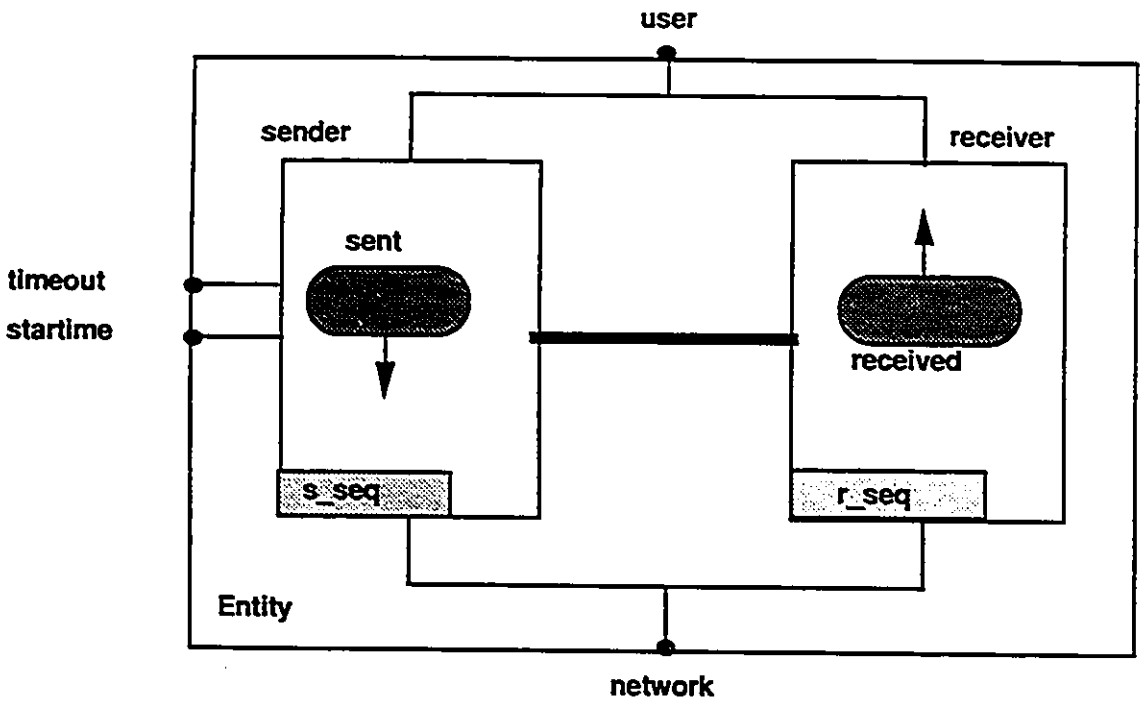


Figure 5.3: A Model of the design of the ABP

In the specification above, sending process has `s_seq` and `sent` as relevant data. `s_seq` is accessed by the sending process to update the sending sequence number.

On the other hand, the receiving process has two relevant data: `received` buffer, that contains the messages received but nor yet delivered, and *r_seq*, the receiving sequence number.

The example given above shows the application of the design method that consists of the parallel composition of two views in order to build a large system. This composition is made possible by the atomicity, because, as discussed in Section 5.1, without the atomicity we could have some unwanted interleavings.

Since each transition in the first specification, given in Section 5.2.1, is modeled by an atomic action in the second one given above, it is not difficult to see that the second specification gives a behavior that is similar to the behavior of the first one. We can see (informally) that the second specification does not add any interleaving that did not exist in the first one.

Unfortunately, at present we are not able to formalize this intuitive concept in a proof. Hopefully, further research in this direction, including the development of computer-assisted verification tools, may eventually make it easier to prove that the two specifications are in some sense equivalent. This might also require changing the specification in order to get rid of parameters (e.g. buffers) and data messages and work with a simplified version.

Chapter 6:

Systems Recovery

6.1 The Concepts of Atomicity and Recoverability

We have seen that an atomic action is an activity, possibly consisting of many steps, that appears primitive and indivisible to any activity outside the atomic action. To other activities, an atomic action is like a primitive operation which transforms the state of the system without having any intermediate states. Atomic actions are generally accepted as a fundamental mechanism for the control of concurrency in both centralized and distributed systems.

In distributed systems, atomic actions must possess some properties based on some criteria among which the following can be identified [TAY 86]:

1) *Atomicity*: in the sense that an action has either its complete desired effect, or no effect. This is what is commonly called *the all-or-nothing principle*. In systems such as data base management systems this is preserved by the run time environment.

2) *Consistency*: in the sense that the action leaves the system in a consistent state if the initial state was consistent. For example, in a distributed database where multiple copies of a record exist, these must be identical before and after the action, although they may be different during the action. The purpose is to enable us to specify systems in which the intermediate steps are not necessarily consistent but the whole action is.

3) *Durability*: once an atomic action has completed, all changes of the system state become permanent.

If these criteria are not respected, then the reason for atomicity is not fulfilled. In order to cope with the failure in respecting these criteria, we may need a mechanism that enables us to reestablish the situation. This is done via a **recovery mechanism** explained in the next sections.

It is also reasonable to assume that nothing is known about the duration of atomic actions, although they should not run forever. Termination is undecidable, of course, however non-termination may sometimes be detected by doing some semantic checks on the specifications. That is, we may be able to tell if an atomic action *may run forever* as it is the case in the following examples:

```
atomicproc p := a * p + b ; stop
or
atomicproc q := a * q
```

Such processes have infinite branches in their behavior trees:

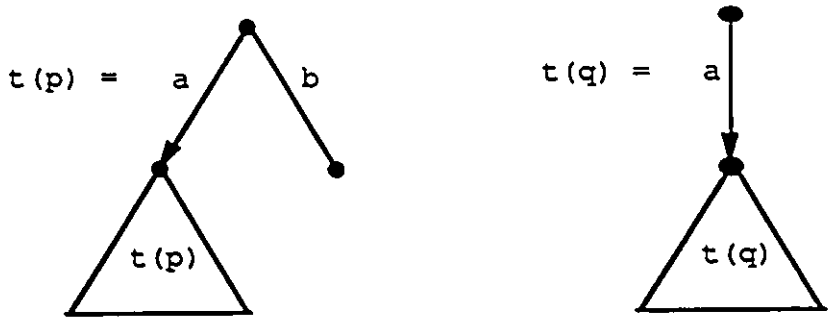


Figure 6.1: Behavior trees for process p and q

6.2 Discussion

In this section we relate the problems discussed above to our model.

The semantics of the parallel operator as specified in the inference rules as well as in the expansion theorems state that synchronization may occur between two processes that offer matching actions. That is, an atomic sequence can match only with an atomic sequence. However, matching sequences may end up in deadlock situations as shown in the following examples:

Example 6.1

Let `p := a * b * c ; p` and

`q := a * b * d ; q + a * b * c ; q`

The composition: (p || q) \ {a,b,c,d} as given by the expansion theorem is:

```
(p || q) \ {a,b,c,d} =

            i * i * (c ;  p <> d  ; q) \ {a,b,c,d}

          + i * i * i ; ( p || q ) \ {a,b,c,d}
```

According to the rule for the communication operator <>, the first sequence of matching leads to a premature deadlock.

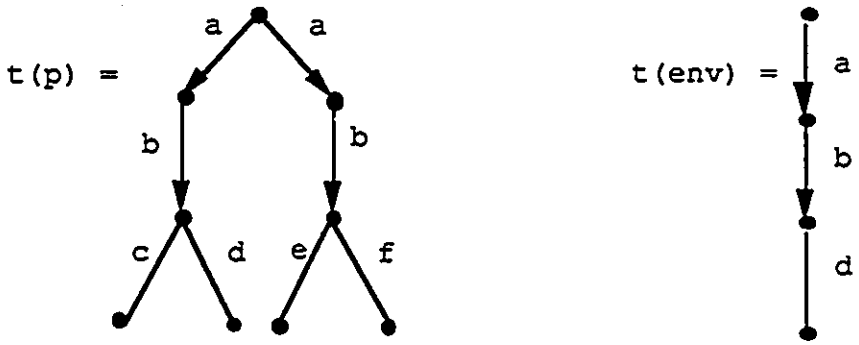
Example 6.2

Let process p and its environment env be defined as follows:

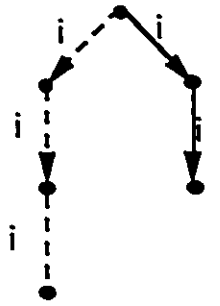
```
p   := a * b * (c ; stop  +  d ; stop)
       + a * b * (e ; stop + f ; stop)

env := a * b * d ; stop
```

These processes have the following behavior trees:



Expression `(p||env)\{a,b,c}` has a tree from which we extract the following initial portion:



We can see that there is a path that leads to a complete sequence of matchings for the atomic sequence: `a*b*d` (as shown by the dashed lines). However, the system can also nondeterministically choose the other alternative in process `p`, which leads to a deadlock.

These two examples show that there is a need to deal with such situations that may occur during the dynamic behavior of a specification. In other words, there is a need for a mechanism of **recovery** from this type of situation.

One possible mechanism involves a **forward recovery** [TAY 86], [GOR 89]. Informally, this consists in going as far as possible in an atomic action until we identify an error and, based on this knowledge, correct the system state containing the error. This approach requires accurate damage assessment and identification of the cause of the error. The technique relies on mechanisms such as signals, exceptions and exception handlers and others which can be considered as implementation dependent features that, in our view, do not belong to specification techniques.

The other alternative, the one we will use in this thesis, involves **backward recovery**. Backward recovery [JAL 86] means that a process is executed until something goes wrong, in which case we return to a previous execution point. This implies the definition of points in the execution to which a process may get back in such cases. These points are called **checkpoints**. We will define two types of checkpoints: **manual checkpoints**, and **automatic checkpoints**.

6.3 Semantics of Manual Checkpoints

A **checkpoint** is a point to which a process may go back during execution. We use the concept of checkpoint in the context of atomicity to allow for recovery in case of deadlock during the communication of this process with its environment. We specify the behavior of a checkpoint by giving its semantics by means of inference rules. The checkpoint concept as applied to CSP has been discussed in [HOA 85]. Recently, a checkpoint operator has also been introduced by Milner [MI L 89] for CCS.

We speak of **manual checkpoints** because we are formalizing the procedure by which a user can set a checkpoint during execution of a specification in a running environment like the University of Ottawa LOTOS interpreter [LOG 88]. At any given stage of execution, a user can cause the interpreter to save the current state of the system. After further steps, execution can be backed up to the checkpoint causing the interpreter to resume execution at a certain point. The notion of checkpoint in that context allows the specifier to investigate the behavior of the system that he has specified. In the context of atomicity, the concept of checkpoint is a necessary mechanism for recovering from errors during the execution of atomic actions.

6.3.1 Simple Manual Checkpoints

First we extend the set of possible action labels Act by adding **ribck** and **commit.** These, however, will not be considered as included in Act.

Definition 6.1

A process *B* is said to be a **simple manual checkpoint** for a process *A* if:

1) It is declared to be so by means of the action denoted by **commit** (for commit to a checkpoint).

2) It is executed by means of the action denoted by **ribck** (for rollback) ○

A checkpoint will exist until another checkpoint is declared by using the *commit* action. Action *ribck* can be seen as the modelling of the behavior of a user who executes the specification step by step and gets back to a previously defined checkpoint, and action *commit* can be considered as a command that allows the user to save an intermediary state that can be used in the *ribck* action.

Semantics

We introduce a **simple manual checkpoint operator** denoted by <->. A <-> B indicates that A is the current behavior expression and that B is the current checkpoint. We use an unary operator called the **start operator**, denoted by ^ that allows any process that

results from the execution of an action to be a (possible) checkpoint. The semantics of the checkpoint operator is defined as follows:

cp-1: A <-> B -a-(any)-> A' <-> B if A -a-(any)-> A'

and a ≠ commit and a ≠ rlbck

cp-2: A <-> B -commit-(int)-> ^ A

cp-3: A <-> B -rlbck-(int)-> ^ B

Note that by the rollback operation, stop is no longer a deadlock in an atomic action because if A gets to a stop it can always execute a rollback. Note that rules cp-2 and cp-3 allow A <-> B to rollback or commit at any time. However for this to be possible, a checkpoint must have been set first.

When checkpoints are required, the initial behavior expression will indicate this by the use of the ^ operator. This operator is specified as follows:

cp-4: ^A -a-(any)- > A' <-> A

if A -a-(any)-> A'

This operator will be mainly used in the context of parallel composition to allow a process entering a communication inside an atomic action to set checkpoints. At the beginning, if checkpoints are desirable for a process A, we write ^A.

Rule cp-1 says that as long as A can execute we can continue its execution, carrying over the checkpoint. Rule cp-2 enables one to edit from a checkpoint and at the same time to immediately establish a new one. Rule cp-3 says that any declared checkpoint can be used as a restarting point. Action *rlbck* allows to resume from that point. Rule cp-2 and cp-3 can be chosen at any point during execution. Rule cp-4 says that A is declared as a possible checkpoint for itself. This rule allows to start the declaration of a *potential checkpoint* at the beginning of a process behavior.

Example 6.3

We give an example of the checkpoint mechanism on process `p` declared below:

```
p := a ; b ; c ; e * f ; stop
```

```
^p -a-(int)-> (b;c;e*f ; stop) <-> p                    by Rule cp-4
```

(p is its own checkpoint)

```
   -b-(int)-> (c;e*f;stop) <-> p                        by Rule cp-1
   -commit-(int)-> ^ (c;e*f;stop)                       by Rule cp-2
   -c-(int)-> (e*f;stop) <-> (c;e*f;stop)               by Rule cp-4
```

(The previous checkpoint has been removed and a new one has been established)

```
   -e-(at)->  (f;stop) <-> (c;e*f;stop)                 by Rule cp-1
   -rlbck-(int)-> ^ (c;e*f;stop)                        by Rule cp-3
```

(We restart from the previously declared checkpoint)

```
   .... etc...
```

6.3.2 Multiple Manual Checkpoints

Using the simple manual checkpoint mechanism described in the previous section we can have at most one checkpoint at any given time. Multiple checkpoinyts can be used in systems where it is desirable to have several intermediary states to be stored for further rollbacks. This is the case for example in testing systems where synchronization points are needed to restart the process of testing from a particular point. In database systems, multiple checkpoints are kept in order to allow to redo some actions that were executed. It is desirable in this case to have multiple checkpoints to restore the system from a particular state. In dealing with atomic actions it is sometimes desirable to have multiple checkpoints as shown in the following example:

Example 6.4

Let p be the process:

```
p := a * (   b * ( c ; stop + d ; stop)  +  b * d ; stop)
     + a * c ; stop
```

Let env be a possible environment to p:

```
env :=    a * b * d ; stop
        + a * c ; stop
```

From the behavior tree of `p || env` we extract the sub-tree of Figure 6.2. Note there are deadlocks within atomic sequences: these are nodes 2, 5, and 7.

Suppose that it is possible to have multiple checkpoints as described below. In the tree of Figure 6.2 we have two different checkpoints: one that corresponds to node `checkpoint 1` and the other to node `checkpoint 2`. These two checkpoints can be used as follows:

- If the environment wants to communicate with sequence `a*c`, then after executing `a`, it can find itself either in `node 3` in order to continue the sequence by executing action `c`; or in the deadlock situation of `node 2` or `node 5`. If it chooses the deadlock nodes 2 or 5 it could decide to return to node `checkpoint 1`.

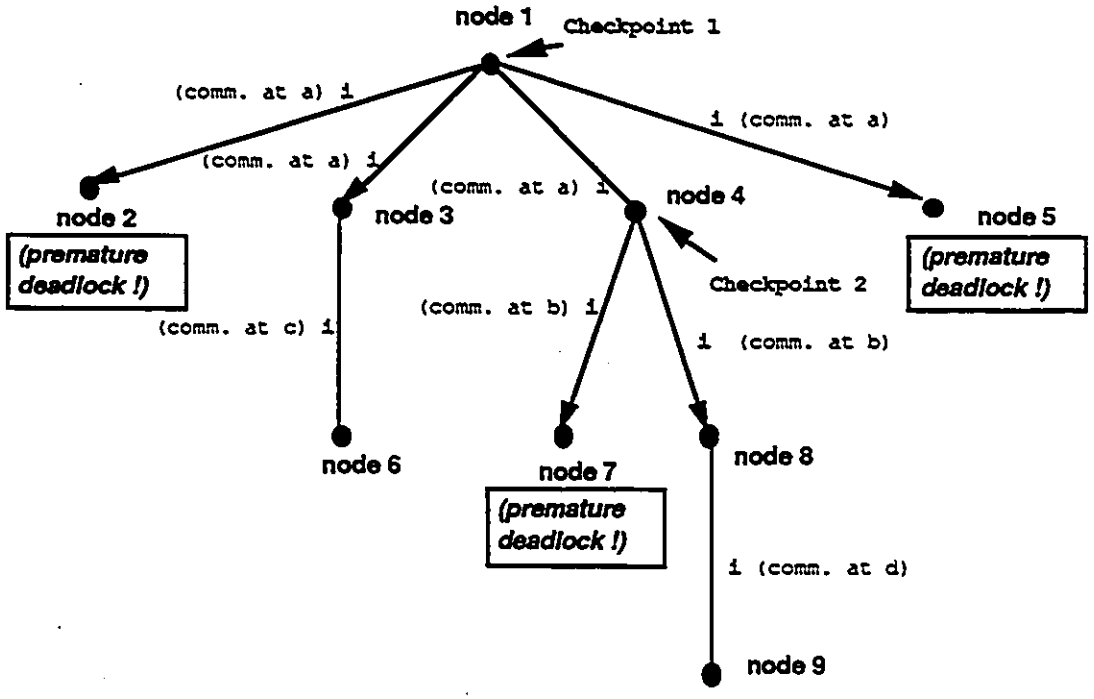


Figure 6.2: `(Sub-)Behavior tree for p||env`

- On the other hand, if it wants to execute sequence a*b*d, then after executing action a it could find itself either in node 4 or in node 3 where it could continue the sequence by executing action b; or in deadlock nodes 2 or 5. If it chooses node 3 then it again finds itself in a deadlock situation in which case it can resume from checkpoint 1. However, in node 4 action b can be executed leading to either node 7 or node 8. In node 8 it can complete its execution of the atomic sequence. But in node 7 it reaches a deadlock. In this case it can resume its execution either from checkpoint 2 because this point was reached after execution of action a; or it can resume right from the beginning of the atomic sequence at checkpoint 1.

Therefore with multiple checkpoints the user has the freedom of choosing which one of the checkpoints he would like to go back to.

Semantics

In order to allow for multiple checkpoint definitions, we use a new operator <<->>, called the **multiple manual checkpoint operator**. We also use another new operator, called the **multiple start operator**, noted ^^. The two operators are jointly defined in the following rules:

mcp-1: A <<->> B -a-(any)-> A' <<->> B if A -a-(any)-> A'
and a ≠ commit and a ≠ rlbck

mcp-2: A <<->> B -commit-(int)-> A ^^ (A <<->> B)

mcp-3: A <<->> B -rlbck-(int)-> B

mcp-4: A ^^ B -a-(any)-> A' <<->> B if A -a-(any)-> A'

and a ≠ commit and a ≠ rlbck

In order to set a checkpoint context for process A, we start with A ^^ A.

Rule mcp-1 says that as long as A can execute, it may continue its execution. Rule mcp-2 enables one to add the current process, A, to the set of previously defined checkpoints which is included in B. Rule mcp-3 allows to restart from the last defined checkpoint. Rule mcp-4 allows to execute an action from any process that can do so. This rule defines the ^^ operator. It is defined in such a way as to prevent setting the same checkpoint consecutively several times by repeatedly using the *commit* action. We force at least one action to be executed before committing to a checkpoint.

This definition allows one to declare several checkpoints and resume from the latest declared one. It also allows, by means of successive rollbacks, to go back through the list of defined checkpoints until the desired one is reached.

Example 6.5

By using process p described earlier in Example 6.3, we give an example of the use of multiple checkpoints:

```
p^^p -a-(int)-> b;c;e*f;stop <<->> a;b;c;e*f;stop          by Rule mcp-4
   -b-(int)-> c;e*f;stop   <<->> a;b;c;e*f;stop            by Rule mcp-1
   -commit-(int)->  c;e*f;stop ^^ (c;e*f;stop <<->> a;b;c;e*f;stop)
                                                           by Rule mcp-2
```

(We commit process P to a set of two checkpoints)

```
-c-(int)-> e*f;stop  <<->> (c;e*f;stop <<->> a,b;c;e*f;stop)
```

by Rule mcp-4

```
-rlbck-(int)->   c;e*f;stop  <<->> a;b;c;e*f;stop
```

by Rule mcp-3

(We get back to the last declared checkpoint)

```
-rlbck-(int)->  a;b;c;e*f;stop
```

by Rule mcp-3

(We get back to the first checkpoint: initial state)

```
... etc ...
```

Discussion

We defined two checkpoint operators with their respective auxiliary operators. Our calculus will allow the use of any one of them according to what the specifier would like from a specification. The question is still open as to whether to use the simple checkpoint or the multiple checkpoint concept. The advantage of the latter is that it allows different states to be a backup for a specification if something "goes wrong" within or even outside an atomic sequence of actions. For example, we may use the notion of checkpoints as synchronization points that will help in designing a test responder in a remote testing environment. Following the example of Figure 2.6, a tester system may wish to test sequence `a,b,d`. In order to be able to recover from the several deadlocks that are possible before completing the sequence, it is desirable to set some checkpoints as discussed above.

There are some drawbacks of the multiple checkpoint mechanism however. The first one is that the amount of information that is needed at one time is large. The second one is that the access to a particular checkpoint can be done only sequentially. A good improvement would be to declare checkpoints as processes each one with its own unique identifier (say `n`) in the form `cp(n)` for instance. Then a checkpoint would be

referenced directly by giving its identifier in the rollback action as in `rlbk!n` which would then call `cp(n)`. This would allow a "faster" access to a checkpoint. This solution requires implementation "tricks" that are beyond our formal specification technique.

6.4 Expansion Theorems for Recovery Operators

The recovery can be integrated into the semantics of the parallel composition operator. For example, for the simple manual checkpoint operator <->, one can write ^(A ||B) in order to set checkpoint for the communications within atomic sequences. In this context the checkpoint can be used to rollback if a deadlock occurs inside a sequence of atomic actions.

The following propositions express the expansions for the two checkpoints operators defined above. In both propositions, as in the case of expansion theorems for the composition operators, we assume that process A can be written as:

$$A = \sum_{i \in IA} a_i; A_i + \sum_{i \in AA} a'_i \bullet A'_i$$

Proposition 6.1 (Checkpoint operators)

$$A <-> B = \sum_{i \in IA} a_i;(A_i <-> B) + \sum_{i \in AA} a'_i \bullet (A'_i <-> B)$$

$$+ \text{ commit; } (\hat{}A) + \text{rlbck; } (\hat{} B)$$

where

$$\hat{}A = \sum_{i \in IA} a_i;(A_i <-> A) + \sum_{i \in IA} a'_i \bullet (A'_i <-> A)$$

O

Proposition 6.2 (Multiple checkpoints operators)

$$A \langle\langle\text{->>}\rangle\rangle B = \sum_{i \in IA} a_i;(A_i \langle\langle\text{->>}\rangle\rangle B) + \sum_{i \in AA} a'_i{}^{*}(A'_i \langle\langle\text{->>}\rangle\rangle B)$$

$$+ \text{ commit; } (A \,\hat{}\hat{} (A \langle\langle\text{->>}\rangle\rangle B)) + \text{rlbck;}B$$

where

$$A \hat{}\hat{} B = \sum_{i \in IA} a_i;(A_i \langle\langle\text{-->>}\rangle\rangle B) + \sum_{i \in AA} a'_i{}^{*}(A'_i \langle\langle\text{->>}\rangle\rangle B)$$ O

6.5 Automatic Recovery Procedures

6.5.1 Introduction

The notion of checkpoint described so far can only be used in a user-directed running environment (e.g. in a simulation procedure). What we really need is a mechanism by which a system could automatically resume execution if something goes wrong. In order to have this mechanism we add rules that allow, during process execution, to mark execution paths that were followed unsuccessfully in a specification. Note however that problems could arise in the case of infinite atomic sequences or infinite number of possible paths from a given node. In the first case, it might be possible to find arbitrarily long atomic paths that would make recovery impossible (unless a deadlock is reached within a finite number of actions). In the second case, an infinite number of recoveries may be required from a given state.

6.5.2 Semantics of Automatic Recovery

We extend the set of actions to include actions **commit**, **rlbck** (as above), and a particular action, called **dead** .

Definition 6.2

We say that a derivative expression B of a behavior A is **dead** if there exists a derivation:

A -dead-(any)-> B ○

Action *dead* is introduced to "mark" paths that were already taken, leading to deadlock. It will be used in order to infer a derivative of a process behavior only if that derivative is not marked as dead. Note that *dead* replaces the first atomic action of a sequence that leads to deadlock.

In order to be able to model automatic recovery, we need to define a new unary operator called the **fault tolerance operator** denoted by @ defined as follows:

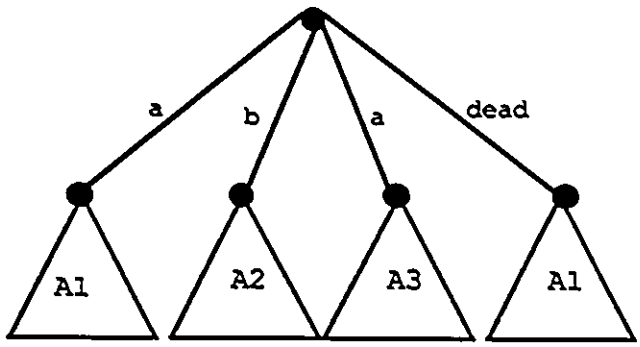
ft-1 : @ A -i-(at)-> <A', A', A> if A -i-(at)-> A' and not (A -dead-(any)-> A')

ft-2 : @ A -i-(int)-> @A' if A -i-(int)-> A'

ft-3: @ A -a-(any)-> @A' if A -a-(any)-> A' a ∈ Act

where operator <_,_,_> is defined below.

The fault tolerance operator allows one to infer a transition from A to A' if there is no dead path from A going through A'. For example in the following case:



@A avoids taking branch a leading to A_1, but allows branch b and a leading respectively to A_2 and A_3.

We will now describe a mechanism that can be used in order to construct and use such trees.

We express the automatic recovery mechanism by defining a ternary operator, called **recovery operator**, denoted by <_,_,_> with three arguments being behavior expressions. The semantics of this operator is:

rec-1: < C, B, R > -i-(at)-> < C', B, R > *(continuation of atomic sequence)*

if C -i-(at)-> C'

rec-2: < C, B, R > -i-(int)-> @C' *(termination of atomic sequence)*

if C -i-(int)-> C'

rec-3: < C, B, R > -rlbck-(int)-> @ (R + dead : B) *(rollback from deadlock)*

if not(C -i-(any)-> C ').

In the definition above the arguments have the following meanings: C is the current behavior. B is the behavior expression denoting the beginning of the sequence that is to be checked for deadlock (such as A1 in the tree given above). R is the behavior expression at the beginning of the atomic action. From expression R we can reach C by a sequence of derivations. The idea is that R will be used to restart from the initial state. B is used to mark the dead derivatives and C is the current behavior expression. Whenever atomic actions start matching (operator <>), this will generate i before an * operator. At this point, we have to check that only internal actions should happen (Rule rec-1) until the sequence of atomic matchings ends. When this happens, the last matching should precede a sequencing operator. This terminates the communication with the atomic action (Rule rec-2).

In order to use the automatic recovery mechanism with the composition operator, we can write: @(A||B) . This way, according to the definition of @, we take the initial behavior expression A||B as a "backup" (for variable R), the behavior expression A'<>B', as the next state (for variable C). The expression A'<>B' is also the branch that was initially taken (for variable B).

Example 6.6

Let `p := a*b;c;stop` and `q := a*b;d;stop + a*c;stop`. The expression `@(p||q)` has the following complete set of derivations:

```
@( p || q)
   -i-(at)->  <b;c;stop<>c;stop, b;c;stop<>c;stop, p||q>          (ft-1)
          ( Here, the only possible action is rlbck !)
      -rlbck-(int)-> @(p||q + dead; (b;c;stop<>c;stop))           (rec-3 )
```

(From now on we cannot choose `b;c;stop<>c;stop` *as a possible outcome of action i since it is dead. Therefore the only possibility now is the matching between* `a*b` *in* `p` *and* `a*b` *in* `q`*)*

```
-i-(at)-><  b;c;stop<>b;d;stop,
            b;c;stop<>b;d;stop,
            p||q +dead; (b;c;stop<>c;stop)>                    (ft-1)
      -i-(int)->  @(c;stop || d;stop)                          (rec-2)
            -c-(int)-> @(stop || d;stop)
                  -d-(int)-> @(stop||stop)
            -d-(int)-> @(c;stop || d;stop)
                  -c-(int)-> @(stop||stop)
- i -(at)->  <b;c;stop<>b;d;stop, b;c;stop<>b;d;stop, p||q>       (ft-1)
   -i-(at)-> @ (b;c;stop<>b;d;stop)                              (rec-2)
         -i-(int)->  @(c;stop || d;stop)
            -c-(int)-> @(stop || d;stop)
                  -d-(int)-> @(stop||stop)
            -d-(int)-> @(c;stop || stop)
                  -c-(int)-> @(stop||stop)
```

(The process stops here after finishing its atomic sequence of actions)

Again it is possible to express the expansion for these operators with recovery.

6.5.3 A Protocol Example

We give an example of a specification in which we use the recovery mechanism. We specify a very simple (and maybe rather artificial) stop-and-wait protocol using a

direct coupling between sender and receiver. The system can be illustrated as shown in Figure 6.3.

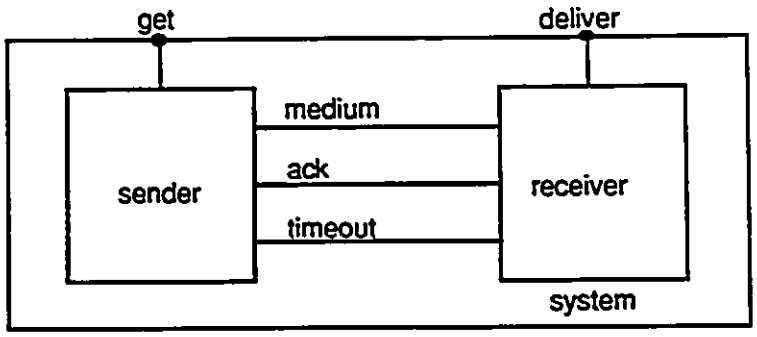


Figure 6.3: Structure of system2

We assume that the receiver initiates a timeout to allow the sender to resend the last message sent and not yet acknowledged.

In this specification, we use atomic actions in order to show the use of the recovery. In fact these atomic actions will introduce a certain nondeterminism at the level of elementary actions in the sense that several atomic sequences of action may start with the same sequence as a prefix. By choosing one particular prefix, we could initiate an atomic sequence that could lead to a deadlock.

```
sender := get ; sender1

where

sender1 :=   medium * ack ; sender
           + medium * timeout ; sender1
```

```
receiver :=   medium * ack ; deliver ; receiver
            + medium * timeout ; deliver ; receiver

system2 := (sender || receiver ) \ { medium,ack,timeout}
```

In chapter 9, we describe a system that implements the expansion theorems. It builds process abstractions and instantiations that correspond to intermediate states in the expansion. These processes are of the form `abst_n` where `n` is an index that identifies the state. A (slightly modified) display of this expansion is given in the example below. Using this system, we find the following specification that is equivalent to the specification of process `system2` given above:

```
abst_0 := get ; abst_1

abst_1 := i * abst_2 + i * stop + i * stop + i * abst_3

abst_2 := i ; abst_4

abst_3 := i ; abst_5

abst_4 := get ; abst_5 + deliver ; abst_0

abst_5 := deliver ; abst_1
```

In this equivalent system, we can see that we have a possible deadlock as shown below:

```
abst_0 -get -(int)->  abst_1  -i-(at)->  stop
```

with no further possible action.

In the derivation given above we have chosen to synchronize at gate `medium`. But this gate prefixes two possible sequences that are: `medium*ack` and `medium*timeout`. The sequence `medium*ack` in the sender can begin matching with sequence `medium*timeout` in the sender and this matching stops, leading to a deadlock. This example illustrates in a simple way what could happen within an atomic action. The recovery mechanism will allow us to avoid such a situation by the use of Rule pc-1 to pc-5 with pc-6-b (given in Section 6.4.2) instead of pc-6. We go through the path that leads to a deadlock and we avoid it the next time. Because of the semantics of recovery we mark this path a dead one as shown below.

We give the output of a session of an execution of the specification by using our ATCCS simulator described in Chapter 9. At each execution step the system gives the menu of actions among which the user may choose one.

Menu of actions:

```
1 : get<int>->
    @((sender1||receiver)\ {medium,ack,timeout})

Which action do you choose (0 : to stop) ? 1.
```

Menu of actions:

```
1 : i<at>->  (* Start communication between medium*ack and medium*ack *)
  <(ack;sender<>ack;deliver;receiver)\ {medium,ack,timeout}
  ,(ack;sender<>ack;deliver;receiver)\ {medium,ack,timeout}
  ,(sender1||receiver)\\ {medium,ack,timeout}>

2 : i<at>->  (* Start communication between medium*ack and medium*timeout *)

 <(ack;sender<>timeout;deliver;receiver)\{medium,ack,timeout}
 ,(ack;sender<>timeout;deliver;receiver)\{medium,ack,timeout}
 ,(sender1||receiver)\{medium,ack,timeout}>

3 : i<at>->

  <(timeout;sender1<>ack;deliver;receiver)\{medium,ack,timeout}
  ,(timeout;sender1<>ack;deliver;receiver)\{medium,ack,timeout}
  ,(sender1||receiver)\{medium,ack,timeout}>


4 : i<at>->

  <(timeout;sender1<>timeout;deliver;receiver)\{medium,ack,timeout}
  ,(timeout;sender1<>timeout;deliver;receiver)\{medium,ack,timeout}
  ,(sender1||receiver)\{medium,ack,timeout}>

Which action do you choose (0 : to stop) ? 2.

Menu of actions:

1 : rlbck<at>-> (* Rollback from a deadlock to the beginning of the atomic action *)

  @ ( (sender1||receiver)\{medium,ack,timeout}
   +dead;(ack;sender<>timeout;deliver;receiver)\{medium,ack,timeout})


Which action do you choose (0 : to stop) ? 1.
```

Menu of actions:

```
1 : i<at>->

  @ ( (ack;sender<>ack;deliver;receiver)\{medium,ack,timeout})


2 : i<at>->

  @ ((timeout;sender1<>ack;deliver;receiver)\{medium,ack,timeout})
 ......
   (* The system avoids the already chosen action that did not succeed *)
```

Chapter 7:

Algebraic Properties of ATCCS Operators

In this chapter, we present some verification aspects of the calculus without recovery . We apply this verification to systems specified in ATCCS. The verification is based on equivalences between processes, similar to Milner's observation equivalence [MIL 80] .

7.1 Introduction

It is desirable for a language to allow description of systems at various levels of abstractions. ATCCS, LOTOS and CCS* allow to describe systems at different descriptive levels, and all descriptions are expressions in the language. Using this feature, one can generally differentiate a service specification which is a high level description from an (abstract) implementation, that is a low level description. A service specification is a description of the desired behavior of a system as seen by a user. An implementation, on the other hand, is a more detailed description of how the system actually works or what are its different components [VIS 85]. The relationship between specifications and implementations can be described in CCS* by using a notion of system equivalence known as **observation equivalence**. For example, assume that we want to prove that an implementation of a service as given in Figure 3.3 is equivalent to a given service specification. This can be informally stated as follows:

A receiving user of the service receives data messages in the same order as they are sent by a sending user.

This can be specified as follows:

```
service_spec := get ; deliver ; service_spec.
```

where `get` is the action of taking a new message from the sending user, and `deliver` is the action for delivering that message to the receiving user. The implementation of the service, also called the protocol in [VIS 86], would require the composition of the protocol entities and the media. This composition implies communication between the different components, which can result in internal events **i. Weak observation equivalence**, as we shall see, does not take into account the internal events. Laws relating expressions with internal events to equivalent ones without internal events can be used to prove that a specification is equivalent to an implementation .

Proving Equivalence Between Systems

In **weak observation equivalence** (denoted by ≈), expressions are considered equivalent if they can offer the same sequences of *observable actions* and transform into equivalent (sub-) expressions. The definition of this equivalence is given in Section 7.3. For example, we can show using this notion that B ≈ i; B for every behavior expression B. That is, if we add a silent move at the beginning of an expression, then what is observed remains the same.

Using this equivalence and other properties, Milner [MIL 80] came up with a set of axioms among which we have:

Axiom 1 : a;i;B ≈a;B

Axiom 2 : i;B +B ≈i;B

By using these axioms we can show, for example, the following equivalence:

```
a1 ; (a2;(i;a3;stop + a3;i;stop ) + a3;a2;i;stop )
=
a1 ; (a2 ; a3 ; stop + a3 ; a2 ; stop)
```

using Axiom1 twice then Axiom2 then Axiom1.

To prove observation equivalence between behavior expressions, Milner [MIL 80] has described an inductive method. It uses a sequence of equivalence relations that has to hold by induction. The observation equivalence in this case is the intersection of all the equivalences in the induction sequence. This proof mechanism is sometimes complex. Fortunately however, Park [PARK 81], has come up with a more practical way of proving these equivalences without using induction. He uses a notion of **bisimulation** as a binary relation that contains the two processes to be proved as equivalent, and based on this bisimulation he applies the conditions that should hold for the equivalence under study. In the next sections we will use Park's technique that is now widely used in verification for CCS* and LOTOS specifications.

7.2 Definitions

In the following, let *s* denote a sequence of actions a_1..... a_n $n \geq 0$. When n=0, *s* is the empty sequence which is denoted by ε.

Definition 7.1 (*Interrupt-Acceptance* =s=>)

Here we define derivations by interruptible sequences of (possibly internal) events.

i) $B =\varepsilon\Rightarrow B'$ if there exist $B_1, \ldots, B_n$, $n \geq 0$ such that:

$$B = B_1 \text{ -i-(int)-> } B_2 \text{ -i-(int)-> } \ldots \text{-i-(int)-> } B_n = B'$$

ii) $B =s\Rightarrow B'$ if there exist $B_1, \ldots, B_n$, $n \geq 0$ and $B'_1, \ldots, B'_n$, $n \geq 0$ such that:

$$B =\varepsilon\Rightarrow B_1 \text{ -}a_1\text{ -(int)-> } B'_1 =\varepsilon\Rightarrow B_2 \ldots =\varepsilon\Rightarrow B_n \text{ -}a_n\text{ -(int)-> } B'_n =\varepsilon\Rightarrow B'$$

In this case, we say that *B* **interrupt-accepts** sequence *s*. ○

Definition 7.2 (*Atomic Acceptance* $=\langle s\rangle\Rightarrow$)

Here we define derivations by atomic sequences of (possibly internal) events.

i) $B=\langle\varepsilon\rangle\Rightarrow B'$ if there exist $B_1, \ldots, B_n$, $n \geq 0$ such that:

$$B= B_1 \text{ -i-(at)-> } B_2 \text{ -i-(at)-> } \ldots \text{-i-(at)-> } B_n= B'$$

ii) $B =\langle s\rangle\Rightarrow B'$ if there exist $B_1, \ldots, B_n$, $n \geq 0$ and $B'_1, \ldots, B'_n$, $n \geq 0$ such that:

$$B =\langle\varepsilon\rangle\Rightarrow B_1 \text{ -}a_1\text{ -(at)-> } B'_1 =\langle\varepsilon\rangle\Rightarrow B_2 \ldots =\langle\varepsilon\rangle\Rightarrow B_n \text{ -}a_n\text{ -(at)-> } B'_n =\langle\varepsilon\rangle\Rightarrow B'$$

In this case, we say that *B* **atom-accepts** sequence *s*. ○

Definition 7.3 (*All-acceptance* $=^*s\Rightarrow$)

Here we define derivations by any sequences of (possibly internal) events.

Let any_i range over the set {int,at} for all i .

i) $B =^*\varepsilon\Rightarrow B'$ if there exist $B_1, \ldots, B_n$ $n \geq 0$ such that

$$B = B_1 \text{ -i-(any}_1\text{)-> } B_2 \text{ -i-(any}_2\text{)-> } \ldots \text{ -i-(any}_{n-1}\text{)-> } B_n =B'$$

ii) $B =^{*}s\Rightarrow B'$ if there exist B_1 ..... B_n, $n \geq 0$ and B'_1 ..... B'_n , $n \geq 0$ such that:

$$B=^{*} \varepsilon\Rightarrow B_1\text{-}a_1\text{-}(any_1)\text{->}B'_1=^{*} \varepsilon\Rightarrow B_2 \ldots =^{*} \varepsilon\Rightarrow B_n\text{-}a_n\text{-}(any_n)\text{->}B'_n=^{*} \varepsilon\Rightarrow B'$$

In this case, we say that *B* **all-accepts sequence** *s*. ○

Example 7.1

1) Atomic acceptance:

Let `p = a * i * ( b * i ; i ; f ; stop + c ; d ; stop).`

1-1)

`p =<a,b>=> i * i ; f ; stop` because

```
p -a-(at)->   i * ( b * i ; i ; f ; stop + c ; d ; stop )
     -i-(at)->   (b * i * i ; f ; stop + c ; d ; stop )
     -b-(at)->   i * i ; f ; stop
```

1-2) Similarly, we can see that `p =<a,b>=>  i ; f ; stop`

2) Interrupt acceptance:

Let `q = a * b ;stop + c ; d ; i ; i; e ; f ; stop`

`q =c,d,e=> f ; stop` because

```
q -c-(int)-> d ; i ; i ; e ; f ; stop
     -d-(int)->  i ; i ; e ; f ; stop
     =i,i=>   e ; f ; stop
     -e-(int)->    f ; stop
```

3) All-acceptance:

Let `r = a * b * ( c ; i * i ; d ; stop  + f * i ; stop)`

`r =*a,b,c=> d;stop` because

```
r -a-(at)-> b * ( c ; i * i ; d ; stop  + f * i ; stop)
     -b-(at)->  ( c ; i * i ; d ; stop  + f * i ; stop)
     -c-(int)->   i * i ; d ; stop
     -i-(at)->    i ; d ; stop
     -i-(int)->   d ; stop
```

Definition 7.4

We define R^{-1}, the converse of a binary relation R, and the composition R.S of two binary relations R and S as:

$R^{-1} = \{ (y,x)$ such that: $(x,y) \in R \}$

$R.S = \{ (x,z)$ such that: for some y, $(x,y) \in R$ and $(y,z) \in S \}$ ○

7.3 Strong Observation Equivalence

The equivalence considered in this section is a relation by which two processes are considered to be equivalent if they are indiscernible by any observer's experiment.
In order to be able to prove equivalences between processes one can use the observation that:

In order for two processes to be equivalent they have to offer the same set of actions that lead to results that are also equivalent.

Or said another way, for two processes A and B to be equivalent, they must be able to **simulate** each other for every action they perform, including the internal actions. This equivalence is a **strong observation equivalence** in the sense that it requires that the two processes be equivalent by considering their internal behavior as well as their external one. In terms of derivations it can be phrased as follows:

whenever a process A can execute an action which, with some atomicity nature, leads to a behavior expression A', B must be able to execute the same action which,

with the same atomicity nature, leads to B', and vice versa. In addition, A' and B' are equivalent.

Using this equivalence we can show some interesting algebraic properties like the ones given in Theorem 7.1 and 7.2.

Definition 7.5 (*Strong Observation Equivalence* ■)
We say that B_1 is **strongly equivalent** to B_2, which we note $B_1 \blacksquare B_2$ if there exists a relation Ψ over the set of behavior expressions, called a **strong bisimulation** which contains the pair $< B_1, B_2 >$ such that:
if $B_1 \ \Psi \ B_2$ then for any action a we have:
i) whenever B_1-a-(at)->B'_1 then there exists B'_2 such that B_2-a-(at)->B'_2 and $B'_1 \ \Psi \ B'_2$
ii) whenever B_1-a-(int)->B'_1 then there exists B'_2 such that B_2-a-(int)->B'_2
and $B'_1 \ \Psi \ B'_2$
iii) whenever B_2-a-(at)->B'_2 then there exists B'_1 such that B_1-a-(at)->B'_1 and $B'_1 \ \Psi \ B'_2$
iv) whenever B_2-a-(int)->B'_2 then there exists B'_1 such that B_1-a-(int)->B'_1
and $B'_1 \ \Psi \ B'_2$ ○

Srong bisimulation obeys the following proposition wich is staraight forward to show that it also applies to ATCCS:

Proposition 7.1 (*[MIL 89] p. 90*)
Assume that R and S are strong bisimulations. Then the following relations are all strong bisimulations:

1) Identity
2) R^{-1}
3) R S

4) R ∪S ○

Example 7.2

We shall illustrate this technique on a simple example. Let us show that A ≡ B

where A = a ; b ; i * c ; stop

and B = a ; (b ; i * c ; stop + b ; i * c ; stop)

In order to do this, we will show that the following relation $\Phi = \Psi.\Psi^{-1}$ is a bisimulation where:

Ψ = { < a;b;i*c;stop, a;(b ;i*c;stop + b;i*c;stop)>,

< b;i*c;stop, b ;i*c;stop + b;i*c;stop>,

< i*c;stop, i*c;stop >,

< c;stop, c;stop >,

< stop, stop > }

Using the definition of bisimulation given above, we verify this equivalence.

Step 1:

Case 1.a:

A -a-(int)-> b;i*c;stop. We also have:

B -a-(int)-> b ; i * c ; stop + b;i*c;stop

and < b;i*c;stop, b;i*c;stop + b;i*c;stop> ∈ Φ

Case 1.b:

B -a-(int)-> b ; i * c ; stop + b;i*c;stop. We also have:

A -a-(int)-> b ; i * c ; stop

and < b;i*c;stop + b;i*c;stop, b;i*c;stop > ∈ Φ

Step 2:

Continuing the process, we have to verify that the conditions of a bisimulation hold for the two resulting behaviors obtained in Step1:

case 2.a:

`b;i*c;stop -b-(int)-> i*c;stop`. We also have:

`b;i*c;stop + b;i*c;stop -b-(int)-> i*c;stop`

and `<i*c;stop,i*c;stop>` $\in \Phi$

case 2.b:

`b;i*c;stop + b;i*c;stop -b-(int)-> i*c;stop`. We also have:

`b;i*c;stop -b-(int)-> i*c;stop`

and `<i*c;stop, i*c;stop>` $\in \Phi$

The process continues until we reach the `stop` in both expressions.

Since $< stop, stop > \in \Phi$, the equivalence is proven.

We could do the same thing by starting from B. The method is the same, so we don't make it explicit.

It is sometimes better to show bisimulations in pictures. Using the behavior trees of the two expressions we can show, as in Figure 7.1, the bisimulation between two nodes corresponding to behavior expressions by linking them with dotted lines.

Using strong equivalence one can prove formal properties (already given in Section 4.6.2) such as commutativity of ATCCS operators, their associativity and idempotence. Some of these properties are listed and proved below. One should notice that they are consistent with the ones of CCS* .

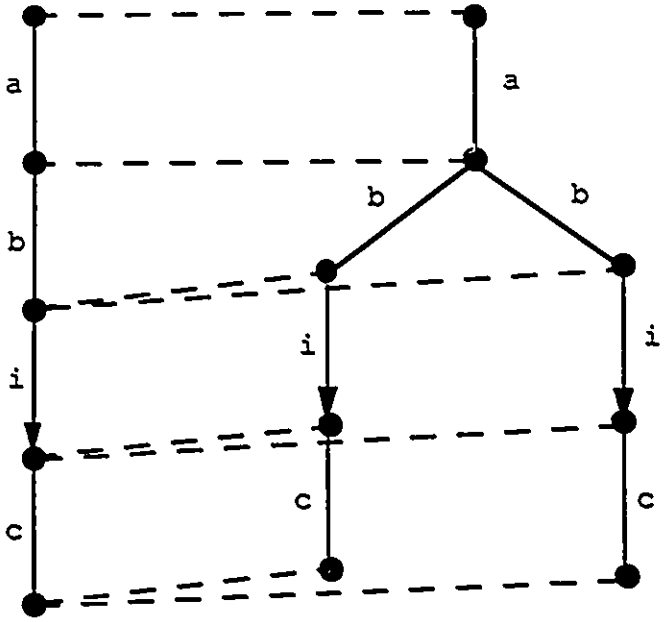


Figure 7.1: A Bisimulation between A and B

As we can see from the definitions, proof techniques based on the notion of bisimulation consist of four parts. In general for brevity we will show less than four parts unless otherwise required.

7.3.1 Proofs of Formal Properties

In this section we give proofs of some formal properties of our calculus. These properties were already stated in Chapter 4.

Theorem 7.1 The following properties hold:

i) $A + A \equiv A$

ii) $A + stop \equiv stop + A \equiv A$

iii) $A \mid\mid stop \equiv A$

iv) a) $A \mid\mid B \equiv B \mid\mid A$

b) $A \mid LB \equiv B \mid RA$

v) $A \mid\mid (B \mid\mid C) \equiv (A \mid\mid B) \mid\mid C$ ○

Proofs: We give proofs for i) and iv-a) only, the other proofs are similar.

Proof of property i): We show that:

$\Psi = \{ <A + A, A>, A \text{ is a behavior expression} \} \cup \text{Identity}$

is a strong bisimulation.

1) If $A + A \text{ -a-(any)-> } A'$ by Rule c-1 or Rule c-2, we have:

$A \text{ -a-(any)-> } A'$. On the other hand $A \text{ -a-(any)-> } A'$ and $<A', A'> \in \Psi$

2) If $A \text{ -a-(any)-> } A'$ by Rule c-1 for example. We then have:

$A + A \text{ -a-(any)-> } A'$ and $<A', A'> \in \Psi$.

The result holds ❑

Proof of iv): We show that:

$\Psi = \{ <A \mid\mid B, B \mid\mid A>, \text{for all A and B} \}$

$\cup \{ <A \mid LB, B \mid RA>, \text{for all A and B} \}$

$\cup \{ <A \mid R\ B, B \mid LA>, \text{for all A and B} \}$

$\cup \{ <A <> B, B <> A>, \text{for all A and B} \}$

is a bisimulation.

1) Let us consider the derivation:

$A \mid\mid B \text{ -a-(int)-> } C$ then:

1-1) either $C = A' \mid\mid B$ by Rule pc-3 which means that

$A \text{ -a-(int)-> } A'$, in which case, according to Rule pc-4, we also have :

$B \mid\mid A \text{ -a-(int)-> } B \mid\mid A'$ where $<A' \mid\mid B, B \mid\mid A'> \in \Psi$

1-2) or $C = A \mid\mid B'$ by Rule pc-4 which means that

B -a-(int)-> B', in which case, according to Rule pc-3, we also have :

B || A -a-(int)-> B' || A where $<A \| B', B' \| A> \in \Psi$

2) Let us consider the derivation:

A || B -a-(at)-> C then :

2-1) either C = A' |L B by Rule pc-1 which means that

A -a-(at)-> A', in which case, according to Rule pc-2, we also have :

B || A -a-(at)-> B |RA' where $<A' |L B, B |R A'> \in \Psi$

2-2) or C = A |R B' by Rule pc-2 which means that

B -a-(at)-> B', in which case, according to Rule pc-1, we also have :

B || A -a-(at)-> B' |LA where $<A |R B', B' |L A> \in \Psi$

3) Let us consider the derivation A||B -i-(int)-> C where C = A' || B' by Rule pc-5, which means that

A -a-(int)-> A', and B -a-(int)-> B', in which case, according to Rule pc-5, we also have:

B || A -i-(int)-> B' || A' where $<A' \| B', B' \| A'> \in \Psi$

4) Let us consider the derivation A||B -i-(at)-> C where C = A' <> B' by Rule pc-6 which means that

A -a-(at)-> A', and B -a-(at)-> B', in which case, according to pc-6, we also have:

B || A -i-(at)-> B' <> A' where $<A' \diamond B', B' \diamond A'> \in \Psi$

The proof for the opposite side, that is condition i) and iii) of Definition 7.5, is similar to the one given above. It is omitted.

The results hold. □

7.3.2 Proof of the Expansion Theorem

We give proof of the expansion theorem (Theorem 4.1) for the composition operator "| |" only. The others follow the same line.

Let us recall this theorem below. We consider two processes A and B defined as follows:

$$A = \sum_{i \in IA} a_i{:}A_i + \sum_{i \in AA} a'_i{*}A'_i$$

and $$B = \sum_{j \in IB} b_j{:}B_j + \sum_{j \in AB} b'_j{*}B'_j$$

Theorem 7.2 *(Composition).*The following holds:

$$\begin{aligned} A \,||\, B = & \sum_{i \in AA} a'_i{*}(A'_i \,|L\, B) + \sum_{i \in IA} a_i{:}(A_i \,||\, B) \\ + & \sum_{j \in AB} b'_j * (A \,|R\, B'_j) + \sum_{j \in IB} b_j : (A \,||\, B_j) \\ + & \sum_{i \in IA\, j \in IB} \{i{:}(A_i \,||\, B_j);\ a_i = b_j \text{ and } a_i \in Act \text{ and } b_j \in Act\} \\ + & \sum_{i \in AA\, j \in AB} \{i{*}(A'_i <> B'_j) \text{ for } a'_i = b'_j \text{ and } a'_i \in Act \text{ and } b'_j \in Act\} \end{aligned}$$ ○

Proof: Surprisingly, the proof can be done using the concept of strong observation equivalence using a very obvious bisimulation, the identity.

For the purpose of the proof, we will use a modified versions of Rule c-1 and Rule c-2. We replace both of them by their generalization, which is:

$$\text{c-1-2: } \sum_{i \in I} B_i \text{ -a-(any)-> } B'_k \quad \text{if } B_k \text{ -a-(any)-> } B'_k \text{ for some } k \in I$$

Let us call the left-hand side of the equation in Theorem 7.2 X and the right hand side Y. The bisimulation is then $\Psi = \{ < A, A > , \text{ where } A \text{ is a behavior expression}\}$.

If X -a-(any)-> C then one of the following is the case:

1) $C = A_i \,||\, B$ for some i, by applying rules s-1 or s-3 on A, c-1-2 on A, and pc-3. This means then we have A -a-(int)-> A_i. Then Y -a-(int)-> $A_i \,||\, B$ for the same i by applying rules s-1 or s-3 and c-1-2 on the second alternative of Y. We then have $<A_i \,||\, B, A_i \,||\, B> \in \Psi$.

2) $C = A \,||\, B_j$ for some j, by applying rules s-1or s-3 on B, c-1-2 on B, and pc-4. This means then we have B -a-(int)-> B_i by Rule s-1 or s-3. Then Y -a-(int)-> $A \,||\, B_i$ for the same i by applying rules s-1 or s-3 and c-1-2 on the fourth alternative of Y. We then have $<A \,||\, B_i \,, A \,||\, B_i> \in \Psi$.

3) $C = A'_j \,|L\, B$ for some j by applying rules s-2 or s-4 on A, c-1-2 on A, and pc-1.

We then have:

Y -a-(at)-> $A'_j \,|L\, B$ for the same j by applying rules s-2 or s-4 on A, and c-1-2 on the first alternative of Y. Again $<A_i \,|L\, B, A_i \,|L\, B> \in \Psi$.

4) $C = A \,|R\, B'_j$ for some j, by applying rules s-2 or s-4 on B, c-1-2 on B, and pc-2:

B -a-(at)-> B'_j. We then have:

Y -a-(at)-> $A \,|R\, B_j$ for the same j, by applying rules s-2 or s-4 on B, and c-1-2 on the third alternative of Y. Again $<A \,|R\, B_j, A \,|R\, B_j> \in \Psi$.

5) $C = A_i \,||\, B_j$ for some i and j, by applying rules s-3 and pc-1-2 twice on A and B.

c-1-2, and pc-5. Then Y -i-(int)-> A_i || B_j for the same i and j by applying rule s-2, and c-1-2 on the fifth alternative of Y. Both X and Y lead to A_i || B_j. We then have $\langle A_i \| B_j, A_i \| B_j \rangle \in \Psi$

6) $C = A'_i <> B'_j$ by applying rules s-4 twice on A and B, and pc-6. Then Y -i-(at)-> A_i || B_j for the same i and j by applying rules s-2 and c-1-2 on the sixth alternative of Y: Both X and Y lead to $A_i <> B_j$.
We then have $\langle A_i <> B_j, A_i <> B_j \rangle \in \Psi$.

The results holds for one part of the proof. The reverse part follows by the same proof mechanism. □

7.4 Weak Observation Equivalence

The strong equivalence studied in the previous section could be seen as "too strong" in that it considers elementary actions only. In communicating systems in general what one may be interested in is the way a system reacts with its environment without any consideration of what is happening inside it, in particular with respect to internal communications. The concept of **weak observation equivalence** (or *observation equivalence* for short) is based on this remark.

Given several specifications of the same system, we may be interested in describing the fact that they are equivalent to each other. One of these two specifications might be an abstract implementation of the other. In behavioral techniques like our calculus, we can describe systems at different levels of details. As mentioned in [BRI 86], the implementation can be considered as a refinement of a specification in that it can describe the way a system is built in terms of smaller concurrent components. In the area of communication protocols the refinement could be the actual protocol specification

while the specification could be the service specification. One might be interested in showing that the protocol implementation is equivalent, in terms of the service it provides to the environment (its users), to the service specification. The service requires the specification of the behaviors of peer protocol entities together with an underlying service provider. By means of composition and hiding operators we may describe this composition as a "black box" with only the interactions that are allowed at the service access points. The equivalence problem would then be to show that the service specification are met. This fits very well within the concept of observation equivalence.

Observation equivalence can be intuitively phrased as follows:

> *two systems are said to be observation equivalent if we cannot distinguish between them by any sequence of observable experiences.*

For this reason in the definitions of the notion of acceptance we do not take into account the internal actions. However, since they play a role in specifying nondeterminism, they are of important concern in the verification process.

The observation equivalence can also be used in order to justify replacing complex systems by simpler equivalent ones within a larger system. This allows one to reduce the size of the specification, thus reducing its complexity of analysis and validation.

Definition 7.6 (*Weak observation Equivalence* $\approx$)

We say that B_1 is **weak observation equivalent** to B_2, which we note $B_1 \approx B_2$ if there exists a relation Φ over the set of behavior expressions called a **weak bisimulation** which contains the pair $<B_1,B_2>$ such that: if $B_1 \Phi B_2$ then for any sequence of actions s we have:

i) whenever $B_1 =^*s\Rightarrow B'_1$ then there exists B'_2 such that $B_2 =^*s\Rightarrow B'_2$ and $B'_1 \Phi B'_2$

ii) whenever $B_2 =^*s\Rightarrow B'_2$ then there exists B'_1 such that $B_1 =^*s\Rightarrow B'_1$ and $B'_1 \Phi B'_2$ O

In this definition, we use the fact the the internal actions i are a non-observable steps in the system and as such are not considered as observed in any environment.

Note that the equivalence $\approx$ is an extension of the observation equivalence (or *weak bisimulation equivalence*) defined by Milner in [MIL 80].

Proposition 7.1 on strong bisimulations applies to weak bisimulations also.

We would like to express some properties of our calculus. We start with simple ones and then we generalize them to more complex constructs. The calculus has all the properties of CCS given in [MIL 80] in the case of absence of atomicity plus some other properties given below. First we give a theorem that shows the relationship between $\equiv$ and $\approx$.

Theorem 7.3

For every behavior expression A and B we have:

$A \equiv B$ implies $A \approx B$ ○

Proof: We show that: is a weak bisimulation.

$$\Phi = \{ <A,B> \mid A \equiv B \}$$

is a weak bisimulation.

Given that $A \equiv B$, and according to the definition of the $=^*s\Rightarrow$ relation, if $A =^*s\Rightarrow A'$ and $B =^*s\Rightarrow B'$ then there exist $A_1, A_2, \ldots, A_n$ and $B_1, B_2, \ldots, B_m$ such that:

$$A = A_1\text{-}a_1\text{-}(any_{A,1})\text{-}> A_2\text{-}a_2\text{-}(any_{A,2})\text{-}> A_3 \ldots A_n\ \text{-}a_n\text{-}(any_{A,n})\text{-}> A'$$

and $$B = B_1\text{-}a_1\text{-}(any_{B,1})\text{-}> B_2\text{-}a_2\text{-}(any_{A,2})\text{-}> B_3 \ldots B_m\text{-}a_m\text{-}(any_{B,m})\text{-}> B'$$

where $a_i \in Act \cup \{i\}$, $any_{A,i} = any_{B,i}$ and $A_i \equiv B_i$ for all i; and n = m. In particular, we have $A_n \equiv B_n$ that is $A' \equiv B'$. The pair $<A', B'> \in \Phi$. The result holds. ❑

Some of the expressions that are equivalent are described in the following theorem.

Theorem 7.4 For every action a and a behavior expression B, we have:

i) $a^* B = a;B \ , a \in Act \cup \{i\}$

ii) $i;B=B$

iii) $i^*B=B$

iv) $a^*i;B = a;B, a \in Act \cup \{i\}$

v) $a^*i^*B=a^*B, a \in Act \cup \{i\}$

vi) $A + i^*(A + B)=i^*(A+B)$ ○

In the sequel, for every equivalence equation we use B to denote the right hand side and B' to denote the left hand side.

Proofs of Theorem 7.4:

We prove i) and iii) . The others are very similar. To prove all these properties we use the bisimulation: Φ = Identity $\cup$ {<B, B'>} where B is the left hand side of an equation and B' is the right hand side.

Property *i)* : To show that Φ is a proper bisimulation, we consider the following cases:

i) case $s \neq \varepsilon$

$a^*B =^*s\Rightarrow C$ iff $a;B =^*s\Rightarrow C$ and in both cases we have the result is C, and $<C,C> \in \Phi$.

ii) case $s= \varepsilon$

$a^*B =^*s\Rightarrow B'$, then B' is a^*B. On the other hand, we have $a;B =^*s\Rightarrow a;B$

and $<a^*B,a;B> \in \Phi$.

Property *iii)* :

i) $s = \varepsilon$. If $B=^*\varepsilon\Rightarrow C$, then $i^*B =^*\varepsilon\Rightarrow C$ because $i^*B \text{ -i-(at)->} B=^*\varepsilon\Rightarrow C$ which is equivalent to say that $i^*B=^*\varepsilon\Rightarrow C$ and clearly $<C,C> \in \Phi$.

ii) Let $s = a_1, a_2, \ldots, a_n$ for $n \geq 1$.

if B =*s=> C then i*B -i-(at)-> B =*s=> C and <C,C> $\in$ Φ. The converse is similar.

Property *iv)* can proven in a similar way. We illustrate this property by proving the following equivalence:

`B = a ; stop + i * (a ; stop + c ; stop)` = `B' = i*(a ; stop+c;stop)` by using the bisimulation Φ = Identity $\cup$ { <B, B'> }.

The set of sequences of actions all-accepted by B is: {a, c, ε }. The set of sequences of actions all-accepted by B' is: {a, c, ε }.

i) We have:B =*a=> stop and B' =*a=> stop and <stop,stop> $\in$ Φ since Φ contains the identity.

ii) We have B=*c=> stop and B' =*c=> stop and <stop,stop> $\in$ Φ since Φ contains the identity.

iii) a) B =*ε=> (a ; stop + c ; stop) and also B' =* ε => (a;stop + c;stop). Both expressions derive to the same expressions and Φ contains the identity.

b) B =*ε=> B and also B' =*ε=> B'. Both sequences lead to expressions that are in Φ.

To show the opposite direction, from B' to B, we also have to consider all possible sequences in the example given above. We will not show this case, which is trivially similar. We only sketched a part of the proof to show how the bisimulation works. ❑

Theorem 7.5 For every behavior expressions B and C we have following:

i) i;B |L C = B || C

ii) C |R i;B = C || B ○

Proof: We show the proof of i) only. Using the expansion theorem and Theorem 7.3. we have : i;B |L C ≡ i ; (B || C). By property ii) in Theorem 7.4 we deduce that i ; (B || C) ≈ (B || C) .The result holds. □

7.5 Atomic Weak Observation Equivalence

Definition 7.6 (*Atomic Observation Equivalence* ><)

We say that B_1 is **atomic observation equivalent** to B_2, which we note $B_1 \asymp B_2$ if there exists a relation Ω over the set of behavior expressions called **atomic weak bisimulation** which contains the pair $< B_1 , B_2 >$ such that if $B_1 \Omega B_2$ then for any sequence of actions s we have:

i) whenever $B_1 =<s>\Rightarrow B'_1$ then there exists B'_2 such that $B_2 =<s>\Rightarrow B'_2$ and $B'_1 \Omega B'_2$

ii) whenever $B_2 =<s>\Rightarrow B'_2$ then there exists B'_1 such that $B_1 =<s>\Rightarrow B'_1$ and $B'_1 \Omega B'_2$

iii) whenever $B_1 =s\Rightarrow B'_1$ then there exists B'_2 such that $B_2 =s\Rightarrow B'_2$ and $B'_1 \Omega B'_2$

iv) whenever $B_2 =s\Rightarrow B'_2$ then there exists B'_1 such that $B_1 =s\Rightarrow B'_1$ and $B'_i \Omega B'_2$ ○

Theorem 7.6 The following holds

i) $i^*B \asymp B$

ii) $a^*i;B \asymp a;B$

iii) $a^*i^*B \asymp a^*B$ ○

Proof: Along the same line as Theorem 7.4. □

Note also that property i) of Theorem 7.4 does not hold.

We also have some implications between ≡, ≃, and ><, as expressed in the next theorem.

Theorem 7.7 For all behavior expressions A and B we have:

$A \equiv B$ implies $A >< B$ implies $A \simeq B$. ○

Proof: The proof of the second implication is straightforward from the definition of >< and ≃, because any sequence that is atom-accepted or interrupt-accepted is also all-accepted. For the first implication we use the bisimulation: $\Omega = \{ < A, B > \mid A \equiv B \}$ and the proof follows the same line as Theorem 7.3 □

7.6 Congruence Equivalences

The equivalence defined above states that two processes are equivalent if we cannot distinguish them by experimentation. However, one may require that two systems that are equivalent stay equivalent *in all contexts*. That is, given two equivalent expressions, we would like to obtain expressions that are still equivalent by replacing some of their sub-expressions with equivalent sub-expressions. This can be used for example to reduce the size of specifications. An equivalence relation that has this property is called a **congruence relation.** Unfortunately not all the equivalences given above are preserved in every context. In the following, we make more precise these statements.

Definition 7.7

An ATTCS **context** C[.] is a behavior expression which contains one or more occurences of a formal parameter '[.]', called a **hole**, which will hold a behavior expression. C[B], for a behavior expression B, is C[.] where all occurrences of [.] have been replaced by B. ○

For example, if C[.] = (A || [.]), then C[B] = (A || B).

Definition 7.8 (*Congruence relation* $\simeq^C$)

A $\simeq^C$B iff for every ATCCS context C [.], we have: C[A] ≃ C[B] ○

In the following theorem, we make precise which contexts preserve equivalences without problems.

Theorem 7.9:

For every behavior expressions B, and C, action a, a substitution S, and set of labels L, we have:

if B ≃ C then

1) a:C ≃ a:B

2) a^*C ≃ a^*B

3) B\L ≃ C\L

4) B [S] ≃ C [S] ○

Proof: The proof is a slight generalization of a similar proof in [MIL 80]. ❑

As pointed in [MIL 80], ≃ is a congruence. However, although we have i : B ≃ B, we don't have i:B + C ≃ B + C. There are, however some contexts that preserve equivalence as expressed in the following propositon:

Proposition 7.1 For every behavior expression B, we have:

a^*i:A + B ≃ a:A + B ○

Proof: Straightforward. We use the identity as a bisimualtion ❑

Moreover, given that we have the atomicity operators, we have the following fact:

Fact:

(B = C implies B || D = C || D) does not hold. ○

This fact is shown by taking `B = a*b;B1`, `C = a;b;C1`, and `D = a*b;D1` for some `B1`, `C1` and `D1`. `B` and `D` may always communicate by executing `a*b`, while `C` and `D` may deadlock.

We therefore see that in our calculus, the composition context || causes additional problems that did not exist in CCS* or LOTOS. In the next chapter we will study two major enhancements of ATCCS that allow us to deal with some of these problems.

7.7 Proving the Mutual Exclusion with Semaphores Correct

Using the properties given above, we are able to show the correctness of the mutual exclusion with semaphores given in Chapter 4.

In Chapter 4 we derived the expansion of the specification of the process `system`:

```
system := (p1 ||  sem(1) ||  p2) \L
```

Using the fact that = is a congruence relation. That is, it preserves the equivalence in any context. Therefore we get:

```
(p1 || sem(1) || p2)\L =
                        i*i; cs11;cs12;i*i;( p1 || sem(1) || p2) \L
                  +     i*i; cs21;cs22;i*i;(p1 ||  sem(1) || p2)) \L
```

By applying Theorem 7.3 we obtain :

```
(p1 || sem(1) || p2) \ L =
                        i*i;cs11;cs12;i*i;(p1 || sem(1) || p2) \ L
                  +     i*i; cs21;cs22;i*i; (p1 || sem(1) ||  p2)) \ L.
```

Let mutex be the process:

```
mutex :=      i ; cs11 ; cs21  ;  mutex
          +   i ; cs21 ; cs22  ;  mutex
```

Which states that mutex either decides to execute the critical section of the first process cs11;cs21, or the one of the second process cs21; cs22 . The choice is nondeterministic as specified by the internal event.

We now show that the two specifications are equivalent.

Property 7.1 The following holds:

```
(p1 ||  sem(1) ||  p2) \ L = mutex
```

○

Proof: In order to prove this property, we use the results of Theorems 7.3, Proposition 7.1, Theorems 7.4 and 7.9. Theorem 7.3 implies the weak equivalence based on the strong

one. Theorems 7.4 and 7.9 allow to eliminate the internal events that are inside the expressions. That is, we have:

```
      i*i;cs11;cs12;i*i;(p1||sem(1)||p2)\L
   = i*i;cs11;cs12;(p1||sem(1)|| p2) \ L
```

and

```
      i*i; cs21;cs22;i*i; (p1 || sem(1) ||  p2))
   = i*i; cs21;cs22; (p1 || sem(1) ||  p2))
```

Proposition 7.1 allows to link both equivalent expressions by the choice operator. Finally, by using name substitution where `p1||sem(1)|| p2` is replaced by `mutex`, the result holds.

By using behavior trees of the two processes given below we can prove the same property by showing that the realtion drawn in Figure 7.2 is a bisimulation. The proof is then straightforward. ❑

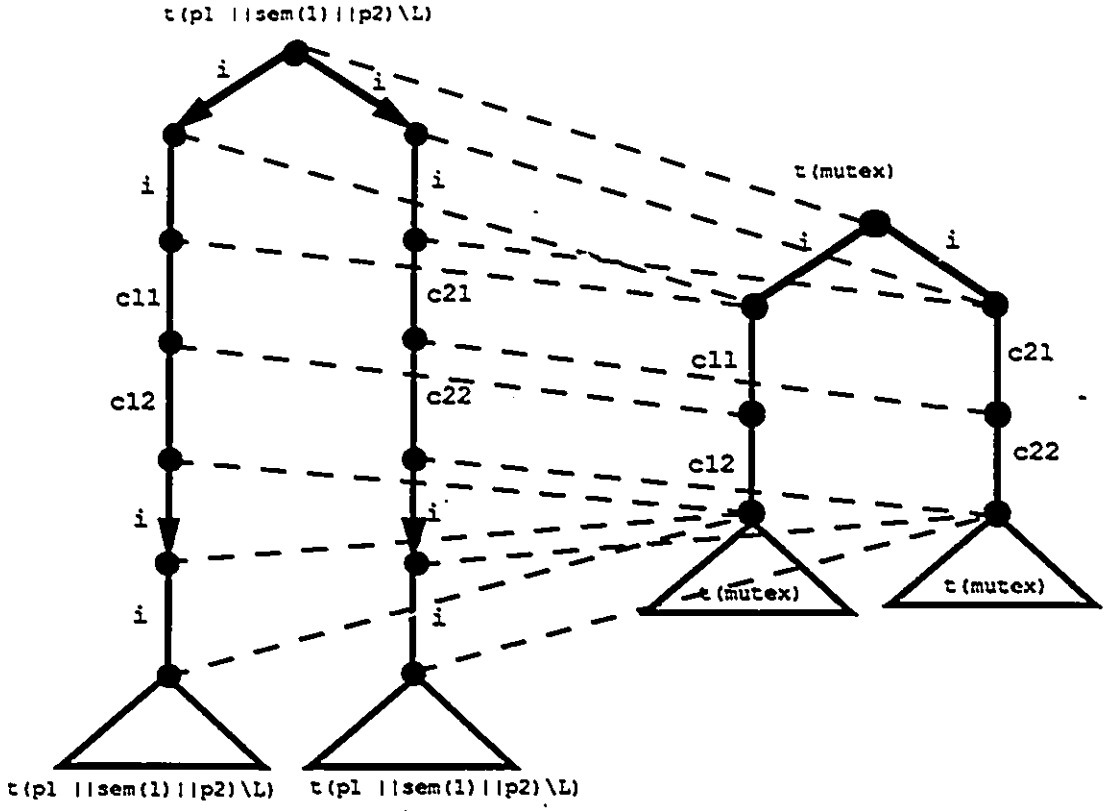


Figure 7.2: A bisimulation between (p1||sem(1)||p2)\L and mutex

Chapter 8:

Enhancements of ATCCS

In this chapter, we introduce an outline of a semantic model that enhances our calculus in order to allow to have some more congruence properties. We include this enhancement for the purpose of having a model that is as complete as possible. The full development of such a semantic model is subject for further research.

8.1 The First Enhancement

As pointed out at the end of Chapter 7, some desirable congruence properties do not hold in our calculus. This difficulty came from the presence of internal events inside atomic actions. We believe that atomic actions should be restricted to observable actions only since internal actions do not participate in interactions with the environment. This is the case, in particular, if these internal events appear as prefixes of behavior expressions. More precisely, if we consider an expression like i•B, we interpret it as if the internal event did not exist before behavior expression B. That is, for example, expression `i*b;stop` should be, in some sense, equivalent to `b;stop` and preferably congruent equivalent in the sense defined in Chapter 7. This also will allow us to keep some nice formal properties of our calculus.. In particular, the following desirable congruence is not true in the calculus described so far:

$$i•B + B \simeq^{C} i•B.$$

For example, we have:

```
a;stop + i*a;stop = i* a ; stop.
```

but if we place these two expressions in the same context, for example, by putting them in parallel with `b;stop`, the two resulting expressions are no longer equivalent. That is:

```
( a;stop + i*a;stop ) ||  a;stop = i*a;stop || a;stop   (*)
```

does not hold because the expression on the right hand side of (*) is equivalent to:

```
i*a;a; stop + a;i*a;stop
```

while the left hand side is equivalent to:

```
a;a;stop + a;(a;stop + i*a;stop) + i;stop + i*a;a;stop + a;i*a;stop
```

which admits `i;stop` as a summand which is immediate deadlock after the internal action.

8.1.1 Semantics

In order to deal with these problems we first define a rule for what we call the **strong leading internal sequencing.** Its purpose is to eliminate internal events that occur as prefixes of expressions:

Strong leading internal sequencing:

sli: $i * B \xrightarrow{a(any)} B'$ if $B \xrightarrow{a(any)} B'$

This rule should replace rule s-2 of Chapter 4. As we can see, this inference rule, whose aim is to absorb leading atomic sequences of internal events, works only if the sequence of internal events so involved does not lead to infinite loops as we have already assumed for atomic sequences. This might at first appear to be an excessively strong limitation. However, it is consistent with the assumption that is always imposed on atomic actions by us, as well as in the literature [GOR 89], that is, that all atomic actions should terminate. It should be assumed that the specifier of the system will be able to guarantee such a property.

By using this rule in the example given in the previous section, matching on a is now possible on both alternatives in the left hand side expression and in the right hand side expression, the two expressions are then equivalent.

One should say, however, that this new semantics modifies somewhat the concept of nondeterminism in case of strong seuqncing with internal events but not in the case of sequencing with internal events. For example we will have:

$$i*a;B + i*b;C = a;B + b;C$$

8.2 The Second Enhancement

8.2.1 Introduction

Up to now we have always considered that atomic sequences can match only with atomic sequences. This is because an atomic sequence is considered as a whole with the meaning that it amounts to *one single action*. This might be seen as too restrictive. In many cases, it could be appropriate to allow nonatomic sequences to match atomic ones if they have the same actions.

In the next section we present a different calculus to eliminate this restriction. Moreover, we obtain a semantics that is consistent with the first enhancement. One should notice that in this case also, if a sequence of actions within the atomic sequence does not match completely after starting to match, we have a deadlock situation, in which case we can apply the recovery procedures given in Chapter 6.

8.2.2 A New Semantics Of Communication

In this enhancement, we generalize the notion of matching in the sense that non atomic sequences of actions can match with atomic ones. For example, using this enhancement we can make expression `a;b;c;stop` and `a*b*c;stop` communicate when they are composed together.

We retain Rule sli and replace Rules pc-5, pc-6 and Rules com-1 to com-4 of Section 4.4 by the following set of rules given below.

We introduce Rule pc-5-a as a first rule for communication:

pc-5-a: A || B -i-(any)-> R

if A <> B -i-(any)-> R

Rule com-a specifies the end of matching and of communication. Notice that this rule expresses the fact that when an atomic sequence finishes we get back to the parallel composition operator:

com-a: A <> B -i-(int)-> A' || B'

if A -a-(int)-> A'

and B -a-(int)-> B' a ∈Act

Rules com-b and com-c reduce the matching between an interruptible sequence and an atomic sequence to one single internal event:

com-b: A <> B -i-(any)-> R

if A -a-(int)-> A'

and B -a-(at)-> B'

and A' <> B' -i-(any)-> R a ∈Act

com-c: A <> B -i-(any)-> R

if A -a-(at)-> A'

and B -a-(int)-> B'

and A' <> B' -i-(any)-> R a ∈Act

Rule com-d reduces the matching between two atomic sequences to one single internal event:

com-d: A <> B -i-(any)-> R

if A -a-(at)-> A'

and B -a-(at)-> B'

and A' <> B' -i-(any)-> R a ∈Act

Rules com-e and com-f eliminate internal events from atomic sequences:

com-e: A <> B -a-(any)-> R

if A' -i-(at)-> A'

and A' <> B -a-(any)-> R a ∈Act

com-f: A <> B -a-(any)-> R

if B -i-(at)-> B'

and A <> B' -a-(any)-> R a ∈Act

Note that these rules are nonstandard in many ways. A first nonstandard aspect is shown in Rule pc-5-a, where the operator <> is used, although is not "generated" by any other rule as it is the case in Rule pc-5. A second aspect is the right recursiveness of the rules, which would cause an infinite loop in the case of infinite atomic actions (such as the ones that one could find in case of recursive instantiations). As in the case of the first enhancement, we require that the specification be written in such a way as not to generate such infinite atomic actions.

This set of rules reduces any sequence of matching to a single internal action. In fact we no longer have atomic derivations with internal actions. This is consistent with the first enhancement which eliminates these actions when they appear as strong prefixes in a behavior expression.

For example, sequence `a;b;c;stop` and `a*b*c; stop` match and their communication reduces to the internal event as shown in the following derivation:

`a;b;c;stop <> a*b*c; stop - i(int)-> stop || stop`

by using Rule com-b 2 times and Rule com-a.

With this new version of the calculus we have more flexibility in specifications and some more formal properties.

8.3 More Congruence Equivalences

These enhancements yield additional formal properties mainly related to congruence equivalences.

Theorem 8.1 For two behavior expressions, A and B, we have:

$$A + i^{*}(A + B) \approx^{C} i^{*}(A + B)$$ ○

This property is interesting in particular when B is *stop* in which case it becomes: $A + i^{*}A \approx^{C} i^{*}A$. In order to prove the theorem, we use the following lemma:

Lemma 8.1 $i^*B = B$ ○

Proof. Follows immediately from rule sli. □

Lemma 8.2 $A + i^*(A + B) = A + B$ ○

Proof : Using Lemma 8.1, we have $i^*(A + B) = A + B$. Since = is a congruence, we also have:

$$A + i^*(A + B) = A + A + B = A + B \text{ since } A + A = A$$ □

Proof of Theorem 8.1:

We will consider the composition context $\| \|$ only since it is the case that causes most of the problems. We use Lemma 8.1. Since = is a congruence, we have:

$A + i^*(A + B) \| C = (A + B) \| C$ for every behavior expression C.

By Theorem 7.3, we then have: $A + i^*(A + B) \| C = (A + B) \| C$. □

Thus A + i*(A+B) and i*(A+B) can be replaced the one with other in the parallel composition context, preserving observation equivalence.

Theorem 8.2 For every behavior expression B and action a we have:

$a^*i:B =^c a:B$ ○

Proof: According to Theorem 7.4, we know that a*i;B ≈ a;B . To show the congruence, we will consider the composition context "|| " only. We show that:

$$\Phi = \{\langle A \| C, B \| C\rangle, A \approx B\} \cup \{\langle A \diamond C, B \diamond C\rangle, A \approx B\}$$
$$\cup \{\langle A \,|L\, C, B \,|L\, C\rangle, A \approx B\} \cup \{\langle A \,|R\, C, B \,|R\, C\rangle, A \approx B\} \cup \approx$$

is a weak bisimulation.

Let a*i;B || C =s=> D. We have several cases:

i) s = ε.

i-1) C -i-(any)-> C' and D is a*i;B | | C', then a;B|| C -i-(any)-> a;B || C'

and <a*i;B || C', a;B || C'> ∈ Φ.

i-2) D is a*i;B || C and a;B || C =ε=> a;B || C and both derivatives belong to Φ.

i-3) C -a-(any)-> C' with D = i;B || C'. We have : a;B || C -i-(any)-> B || C'

and <i;B || C', B || C'> ∈ Φ.

ii) s ≠ ε.

Let a*i;B || C =s=> D, then there exist s1 (such that s1 starts with a) and s2 such that:

a*i;B =s1=> B' and C =s2=> C' where s1 and s2 merge in a way that we don't make explicit here and which is consistent with the semantics, s = merge(s1,s2) and D = B' | C', where the operator '|' stands for || , |R, |L, or <>.

Since a*i; B ≈ a;B, there exists B" such that a;B =s1=> B" and B' ≈ B". We also have a;B || C =s=> B" | C' with s = merge(s1,s2). Since B' ≈ B", <B'|C',B"| C'> ∈ Φ.

The symmetrical case is similar. Therefore the result holds ❑

Chapter 9:

Implementation Issues

9.1. The CCS* Interpreter

In this section we briefly describe our interpreter for CCS* which is a part of the University of Ottawa LOTOS interpreter (version 1) [LOG 88]. The version of LOTOS that was implemented was the one described in the ISO draft proposal [DP 8807] of 1985. The interpreter is a part of a system called SINAPS for (Simulation and INference APplication System). [OBA 87a] contains a more detailed description of the system.
We execute LOTOS specifications by first translating them into CCS* specifications, and then interpreting the latter. The CCS* interpreter is based on CCS* operational semantics.

9.1.1 Implementation

As shown in Chapter 3, the semantics of CCS* is described in terms of an operational semantics by means of inference rules. Some of these rules are:

(Rule 1) a : A -a-> A

(Rule 2) $A + B \text{ -a-> } A'$ if $A \text{ -a-> } A'$

(Rule 3) $A + B \text{ -a-> } B'$ if $B \text{ -a-> } B'$

(Rule 4) $A \mid B \text{ -a-> } A' \mid B$ if $A \text{ -a-> } A'$

(Rule 5) $A \mid B \text{ -a-> } A \mid B'$ if $B \text{ -a-> } B'$

(Rule 6) $A \mid B \text{ -i-> } A' \mid B'$ if $A \text{ -a1-> } A'$

and $B \text{ -a2-> } B'$

and a1 matches a2 and $a1, a2 \neq i$

In our implementation, each inference rule is represented by a clause in PROLOG. We don't present this language in this thesis. Good descriptions can be found in [CLO 80]. The choice of PROLOG as an implementation language was motivated by the fact that its relational nature fits very well with the inference rules of CCS* which also define relations between actions and behavior expressions. This implementation allows us to exercise a sequence of actions for a given process and to generate the set of possible actions that can be performed by a specification [OBA 85]. We also use it in simulating behaviors of systems [OBA 87a]. One should note that similar systems have also been implemented for CCS: the CLARA system [GIA 88] and the Concurrency Workbench used for verification on finite state processes.

We use an internal representation and prefix operators in PROLOG. For example, the sequencing operator is called *seq*, the choice operator is called *choice*, and the parallel composition operator is called *comp* [OBA 87a].

For example, Rule 1 is implemented using the following PROLOG rule:

```
infer(seq(Action,A), Action, A) :-
        isaction(Action).
```

Where *isaction* is a predicate that checks if Action is a legal action.

Rules 2 and 3 for the choice operator are implemented using the following PROLOG rules:

```
infer(choice(A,B),Action,A1):-
        infer(A,Action,A1).

infer(choice(A,B),Action,B1):-
        infer(B,Action,B1).
```

Rules 4 to 6 for the parallel composition operator can be implemented using the following PROLOG rules:

```
infer(comp(A,B),Action,comp(A1,B)):-
        infer(A,Action,A1).

infer(comp(A,B),Action,comp(A,B1)):-
        infer(B,Action,B1).

infer(comp(A,B),i,comp(A1,B1)):-
        infer(A,Action1,A1),
        infer(B,Action2,B1),
        matches(Action1,Action2).
```

where *matches* is a predicate that checks if two actions match according to the definition of the matching rule given in Chap 3.

The set of rules stops when it finds a possible action or when it reaches a deadlock. The *stop* process has no rule associated to it.

From this implementation of inference rules we can perform several kinds of actions:

1) Given a behavior expression we can ask if it accepts a particular action. This procedure is commonly called *validation*. For example, if the initial behavior expression is

```
a ; b ; stop  + b ; c ; stop
```

we can ask in PROLOG the following question:

```
?- infer(choice(seq(a,seq(b,stop)),seq(b,seq(c,stop))), a, X).
```

where X is an uninstantiated variable representing the resulting behavior expression (i.e. the derivative) . The answer in this case would be "yes" since action `a` can be executed by this behavior expression.

2) Given a behavior expression we can ask the system to build the set of all possible next actions. For example for the expression given above, we can ask the following question:

```
?-infer(choice(seq(a,seq(b,stop)),seq(b,seq(c,stop))), Action, X),
      write("Action accepted: "),write(Action),nl,
      fail.
```

Since PROLOG does the backtracking, the `fail` predicate will allow to get back to the `infer` predicate and thus will generate all possible actions from the initial behavior expression. The predicate `write` in PROLOG is not backtrackable. The result in this case would be the set `{a,b}`.

3) Still a third possibility would be to print the set of actions together with their resulting behavior expressions as in:

```
?-infer(choice(seq(a,seq(b,stop)),seq(b,seq(c,stop))),Action,Result),
   write("Action accepted: "),write(Action),nl,
   write("Its resulting behavior is: "),write(Result),nl,
   fail.
```

An interesting aspect is when the set of questions given above is done on sequences of actions instead of simple actions. These aspects are discussed in the next section.

9.1.2 Validating CCS* Specifications

In many cases, the user is interested in exercising all possible (or many of the possible) different orders of execution of the elementary actions of a specified system. Automatic exploration of the global state space of systems is possible by using backtracking in the PROLOG implementation. This technique has already been applied to other specification techniques by several authors [PRO 84], [LOG 83]. In all these cases, PROLOG is asked to enumerate all sets of objects satisfying a certain condition. However unfortunately in most cases efficiency problems would not allow us to go far in this procedure. The question then is how to direct the interpreter to take various "interesting" internal choices.

Our system can be used at least in two different ways: to validate interaction sequences, that is to tell whether an interaction sequence that is submitted to it is a valid behavior for the specified system; and to generate interaction sequences, either exhaustively or at random or (more commonly), under human direction. For a related approach based on a different specification technique, see [URA 86], [PRO 84].

9.1.2.1 Validating interaction sequences

First, we shall see how our system can be used to validate sequences of actions of a process. This procedure is useful whenever one wants to test a specification on some concrete cases, for example to help increasing the confidence that the desired process behavior was specified. One way to do that is by checking if a sequence of interactions is valid for a system by looking at its sequences of derivations. For instance, in the following, a sequence of derivations for process `simple_service` (see Figure 3.8) is given:

```
simple_service
 -get!1-->
    ((out!1;sender)[com1/out])
      | buffer1[com1/in,com2/out]
      |  receiver[com2/in] ) \ {com1,com2}
--i-->   (* Communication at gate comm1 between sender and buffer1 *)
    ( sender[com1/out]
    | (out!1 ; buffer1)[com1/in,com2/out]
    | receiver[com2/in] ) \ {com1,com2}
--i-->   (* Communication at gate comm2 between buffer1 and receiver *)
```

```
    (   sender[com1/out]
     | buffer1[com1/in,com2/out]
     | (deliver!1 ; receiver)[com2/in] ) \ {com1,com2}
--deliver!1-->
   simple_service
```

In order to exercise a sequence of actions the following PROLOG clauses were defined:

```
sequenceOf(A,[],A).
sequenceOf(A,[Action|RestOfActions],All):-
        infer(A,Action,A1),
        sequenceOf(A1,RestOfActions,All).
```

The first clause stops the program with the empty sequence: the behavior expression does not change if no action is applied. The validation process consists of providing an initial behavior expression that will instantiate to A, a list of actions that will instantiate [Action|RestOfActions], and the resulting behavior expression that will instantiate All. The result will be either acceptance or rejection according to whether the sequence is valid or not. By not instantiating one of the three parameters one gets the set of all possible values for that parameter that satisfy the relation "sequenceOf". By not instantiating two of them one gets the set of all possible pairs of values for them that satisfy the same relation, etc. For example by not instantiating the list of actions one gets the set of all possible lists of actions leading from A to All. In Appendix 3, a terminal session giving the validation of a sequence for `simple_service` is shown.

9.1.2.2 Generating Sequences of Interactions

The interpreter can be used in order to systematically generate the tree of all possible sequences of actions for a CCS* specification as defined by the inference rules. When one deals with systems of realistic size, however, random exploration can be practical [WES 86]. To implement such an exploration we used a mechanism that at each step generates a random number and selects the branch of the behavior that corresponds to that number. Another way of selecting consists of directing the execution of a process by giving for each execution step a number indicating a particular choice. The problem is in fact to define a criterion that can be used for selecting these numbers, which must be given before execution starts. This is a problem related to software testing in general, and it is it known that it is hard to define such criteria. We do not address this topic in our thesis.

There also must be a mechanism for controlling the recursion and the actual assignment of values to variables in the specification. In our generation method we use symbolic evaluation where values are replaced by expressions involving variable names and-or actual values. One interesting aspect is that when a variable is to be bound to a particular value and that value is used in a guard as a part of a condition, it is not possible to evaluate such a guard to *true* or *false*. In fact we do not pay any attention to these guards at all. All guards are considered to be *false*. Another solution (which we did not apply) would be to try for each guard to alternatively set it to true and false. Still another possibility (not applied in our system) would be to evaluate all guards to true. Thus we build a symbolic tree that contains a very large set of execution paths which may lead to state explosion. In the approach applied in our system, it seems more natural to provide values in order to be able to actually evaluate guards and predicates that occur in a specification. These values can be provided by an environment (which is a process) to the

specification. The choice of such an environment is a very important aspect. In this case we will do symbolic execution for: `specification | environment`.

9.1.3 SINAPS: a Run Time System for CCS*

SINAPS allows for the execution of CCS* specifications. The main purpose of executing specifications is to increase the confidence of the specifier on what he had or she specified. This allows the testing process to start at the specification level [PRO 84]. For this purpose, the user will choose some execution paths that may allow him to discover some design errors such as deadlocks, livelocks, etc.

This execution is actually done in a step by step manner. The system starts by selecting, at each step, the set of all possible actions that a specification offers. After that, a user may have to select one specific branch that he wants to execute. Generating all possible solutions is done by using a facility of PROLOG, called *bagof*, that allows to generate the set of all possible objects that are linked by a given relationship. The result is a bag of solutions. A bag of objects is like a set except that an object may appear more than once (i.e. a mulitset).

The bagof procedure has three parameters:

- The list of the different objects to be generated.
- The relationship to be used in the generation process.
- The variable that will hold the generated set of solutions.

One possible set of objects would be the set of actions accepted by a process at a given state. Since an action may occur several times in a bag, the bagof facility is suitable to

represent nondeterminism. Note, however, that for this facility to work properly all expressions must be *guardedly well defined* [MIL 80], otherwise infinite bags may be generated and the PROLOG interpreter would go into an infinite loop.

At any given stage, the user may chose an action or issue a command to get back to a previous point, or to get a trace of the executed actions, etc.

The outline of the system is:

```
sinaps(X) :- go(X,X).

 go(X,Y) :-
   bagof([A,At, R],infer(Y,A,R,At),S),
   write('Menu of actions:'), nl,
   display(1,S), selectOne(S,S1),
   go(Y,S1).
 go(X,Y):-                    (* if the previous predicate does not succeed *)
   nl,write('Deadlock !!'),
   whatIsNext(X).             (* We must take some actions *)

 display(_,[]).
 display(N, [[A,At,R] | S]):-
   nl,write(N), write(' : '),
    write(A), write('<'),write(At),write('>'), write('->'),nl, print(R),
   N1 is N + 1,
   display(N1,S).
```

```
selectOne(S,S1):-
  nl,write('Which one (0 : to stop) ? '),
  read(N),  N > 0,  nth(N,S,[_,_,S1]).
```

The `go` procedure implements the run-time system. Procedure `display` displays all the actions offered by the current behavior expression. Each action in this display is followed by the behavior expression to which it leads. `selectOne` prompts the user to select a branch among those displayed. After that, the one selected becomes the current behavior expression. The execution process starts from a CCS* specification, prompts the user with a menu of actions and then the user has different ways of proceeding:

1) If the process is ready to communicate with the outside, the user of the system will have to play the role of the environment by providing matching offers to the process. Also the user will specify a particular branch if he or she is manually resolving the nondeterminism. If the process offers internal actions then the user is able to specify which internal action to execute. The process will then evolve to its next "state". This way we "direct" the execution process. Optionally, the user may ask the system to do random selection of the action to be executed. This way we simulate the nondeterminism.

2) The system provides a set of commands that allow to generate test trees (or execution trees). These commands are those which allow a user to set traces of its specification, to store an execution environment, to set user checkpoints, to resume execution starting from a certain checkpoint, to load test runs from a file, to store test trees in a file, etc..

9.1.3.1 A Specification Example

In this example we specify a simple service in CCS*. We have three processes:

1) A sender process which sends via gate `out` integers in sequence each time it receives a signal from a user at gate `get`, then waits for an acknowledgement at gate `ack`.

2) A half-duplex channel which transfers messages from `sender` at gate `d_in` to `receiver` at gate `d_out`. It may lose a data message as specified by the internal event `i`. It also transfers acknowledgements (without loss) from `receiver` at gate `a_in` to `sender` at gate `a_out`.

3) A receiver which checks if the message received at gate `in` is out of sequence, in which case it discards it, otherwise it delivers it at gate `deliver`.

The whole system specification is given in Figure 9.1. The data type definition of `eq`, `succ` (for successor), and `not` functions are not included in the specification for the sake of brevity. A pictorial description of the service is shown in Figure 9.2. One notices the renamings of gates: `out` of `sender` and `d_in` of `channel` are renamed `com1`: `d_out` of `channel` and `in` of `receiver` are renamed `com3` ; `a_in` of `channel` and `ack` of `sender` are renamed `com2` ; and `a_out` of channel and `ack` of `sender` are renamed `com4`.

```
simple_service :=
    (   sender[com1/out,com4/ack](zero)
     | channel[com1/d_in,com2/d_out,com3/a_in,com4/a_out]
     | receiver[com2/in,com3/ack](zero)
    )  \ {com1,com2}
sender(S) :=    get!; out!S;  ack!S; service(succ(S))
```

```
channel :=      d_in?X:int ; (  d_out!X ;channel +  i ; channel)
          +     a_in?Y:int ; a_out!Y ; channel
receiver(R:int) := in?X:int ;  ack!X ;
                  ( [eq(X,R)] -> deliver!X; receiver(succ(R))
                  + [not(eq(X,R))]-> receiver(succ(X)))
```

Figure 9.1: simple_service specification

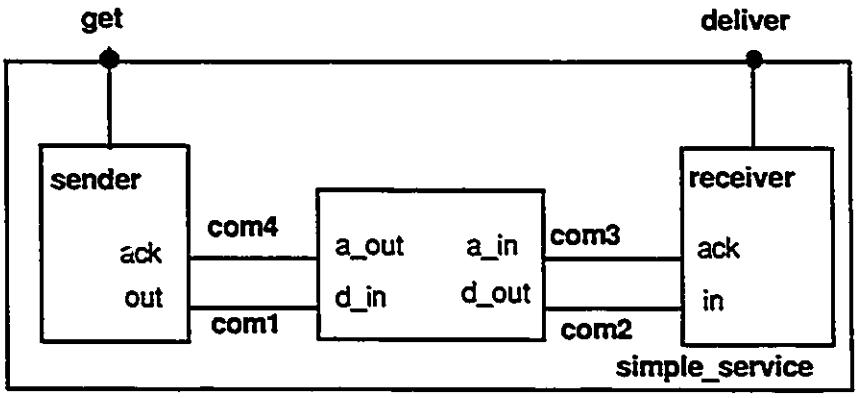


Figure 9.2: Structure of Simple_service

A simulation of this specification is shown in Appendix 2.

9.1.3.2 Use of SINAPS as an Interactive Tester

SINAPS can be used to execute specifications. This capability will allow a user to investigate some aspects of his specification. In order to do that a selection mechanism is provided that allows a user to select a particular action to be executed at any stage of the execution. This is what we call a simulation of the behavior of the system. An example of the use of SINAPS on a protocol can be found in [OBA 87a]. SINAPS has capabilities to allow for:

1) **Interacting with the system**

The user will play the role of the environment by providing and/or accepting values. Also, he or she has to provide information used to resolve the nondeterminism, and issue system commands. The interactions with the system could be loaded from a file as well using the *from* command. This command allows the user to include his interactions and commands in a file that can be loaded to direct the simulation. This facility is useful if one executes a specification and wants to store his interactions so that he can continue with another session without having to do the execution all over again. Also, if a user has sufficient knowledge about his specification and knows a priori the steps he is going through, the command *from* allows him to start at a certain step after these steps are executed.

2) **Using checkpoints**

After having explored an execution path, the user may want to investigate other aspects of his protocol specification. This is made possible by the capability of getting back to a previous checkpoint and resuming execution from it. A checkpoint is defined as a behavior expression. Each checkpoint is identified by a unique identifier. The user can specify that checkpoints be taken either automatically at every execution step, or manually by his request. The second alternative gives to the user full responsibility in setting his checkpoints. In addition, setting checkpoints only when required obviously takes less space than saving a checkpoint at each execution step.

3) **Tracing process executions**

After executing a sequence of interactions with a specification, the user may want to see a summary of the execution paths that were traversed. In order to be able to build these traces of an execution we must have all its intermediary steps. This is the case if we chose the automatic checkpoint option for the execution. Tracing includes the action sequences that were chosen during execution as well as the intermediary states through which the execution went.

4) **Finding errors**

The system can be used for finding design errors. If a deadlock or other error exists, one may be able to find it during simulation. `simple-service` specification contains a potential deadlock, which can be discovered as shown in the session given in Appendix 2.

The deadlock is due to the fact that process `data_channel` can lose a message while the sender is waiting for an ack. But the receiver cannot send this ack unless it receives a message. The solution would be to add a timeout control mechanism. The expression:

```
i ; sender1(S)
```

can be added in order to model such timeout by the internal event **i**. Process `sender` should be defined as follows:

```
sender(S) :=     get!sign ;   sender1 (S)
```

where `sender1` is a new process declared as follows:

```
sender1(S) := out!S;(ack!S;sender(succ(S)) + i;sender1(S))
```

In the corrected specification, if an ack is received a call to receiver will lead to a transmission of the next message. If a timeout occurs, then the current message has to be sent again.

5) **Simulating nondeterministic behaviors**

We also could do the execution by simulating nondeterministic behaviors at any execution step. The nondeterminism that we are concerned with is the *internal nondeterminism* that results from the occurrence of internal events as possible next actions in a specification. Nondeterminism in our system is simulated by letting the system randomly chose its next action among these internal actions if any. The user can let the system go by typing "ok", in which case there is a random selection among the branches that correspond to internal events. If the user participates in an interaction, the system will make a choice on the set of all actions including the ones that corresponds to this interaction.

9.2 A Run Time System for ATCCS

For implementing a run-time system for ATCCS, we used the same ideas as for SINAPS. The PROLOG rules in this case have an extra parameter that will hold the atomicity. We give an outline of the implementations of some the ATCCS operators.

For instance, Rules s-1 to s-4 are implemented using the following PROLOG rules where the operator • is called *s_seq*:

```
infer(seq(i, B), Action, B, int).          (* rule s-1*)
```

```
infer(s_seq(i, B), Action, B, at).                    (* rule s-2 *)
infer(seq(Action,B), Action, B, int):-                (* rule s-3 *)
        isaction(Action).
infer(s_seq(Action, B), Action, B, at) :-             (* rule s-4*)
        isaction(Action).
```

According to the instantiation rules of PROLOG, "at" and "int" are atoms. They cannot be instantiated. In fact they will be used to instantiate variable "Any" in the other rules.

Rule c-1 and Rule c-2 can be implemented as follows:

```
infer(choice(A,B), Action, A1, Any):-          (* rule c-1*)
        infer(A, Action,A1,Any).
infer(choice(A,B), Action, B1, Any):-          (* rule lc-2 *)
        infer(B, Action,B1,Any).
```

Rules pc-1 to pc-4 are implemented as follows (the |L operator is called *lcomp*, |R is called *rcomp*, | is called *comp*, and <> is called *comm*):

```
infer(comp(A, B), Action, lcomp(A1, B), at):-  (* rule pc-1 *)
        infer(A, Action, A1,at).
infer(comp(A,B), Action,rcomp(A, B1),at):-     (* rule pc-2 *)
        infer(B, Action, B1,at).
infer(comp(A, B), Action, comp(A1,B), int):-   (* rule pc-3 *)
        infer(A, Action, A1,int).
infer(comp(A,B), Action,comp(A,B1),int):-      (* rule pc-4*)
        infer(B, Action, B1,int).
```

```
infer(comp(A,B). i .comm(A1,B1),at):-          (* rule pc-5 *)
        infer(A. Action1. A1,at).
        infer(B. Action2. B1,at).
        matches(Action1,Action2).

infer(comp(A,B),i .comp(A1,B1),int):-          (* rule pc-6 *)
        infer(A. Action1. A1,int).
        infer(B. Action2. B1,int).
        matches(Action1,Action2).
```

Rules lc-1 to lc-2 can be implemented as follows:

```
infer(lcomp(A,B). Action. comp(A1,B). int):-   (* rule lc-1 *)
        infer(A. Action. A1. int).
infer(lcomp(A,B ). Action. clomp(A1,B). at):-  (* rule lc-2 *)
        infer(A. Action. A1. at).
```

For the disable and left disable operations we use the following clauses:

```
infer(disa(A,B ). Action. ldisa(A1,B). at) :-  (* rule d-1 *)
        infer(A. Action. A1. at).
        label(Action,L).
        not(equal(L,exit)).
infer(disa(A,B ). Action. disa(A1,B). int) :-  (* rule d-2 *)
        infer(A. Action. A1. int).
        label(Action,L).
```

```
                not(equal(L,exit)).
infer(disa(A,B), Action, A1, Any) :-          (* rule d-3 *)
                infer(A, Action, A1, Any),
                label(Action,exit).

infer(disa(A,B), Action, B1, Any) :-          (* rule d-4*)
                infer(B, Action, B1, Any).
infer(ldisa(A,B), Action, ldisa(A1,B), at) :-  (* rule ld-1 *)
                infer(A, Action, A1, at)
infer(ldisa(A,B), Action, disa(A1,B), int) :-  (* rule ld-2 *)
                infer(A, Action, A1, int).
infer(ldisa(A,B), Action, A1,B, Any) :-        (* rule ld-3 *)
                infer(A, Action, A1, Any),
                label(A,exit).
```

Rules cp-1 to cp-3 for the simple manual checkpoint operator are implemented using the following rules where *ckp* implements the <-> operator:

```
infer(ckp(A,B), action, ckp(A1,B), Any) :-    (* rule cp-1 *)
                infer(A, Action, A1, Any),
                label(Action,L),
                not(in(L,[rlbck,commit ])).
infer(ckp(A,B), commit, ckp1(A ), Any).       (* rule cp-2 *)
infer(ckp(A,B), rlbk, ckp1(B ) ).             (* rule cp-3 *)
```

in(Element,List) is a predicates that tests if *Element* belongs to the list *List*.

Rule cp-4 is implemented as shown below. *ckp1* implements the ^ operator:

```
infer(ckp1(A),Action,ckp(A1,B),Any):-          (* rule cp-5 *)
        infer(A, Action, A1, Any).
```

In the following rules *mkcp* represents the <<->> operator and *start* represents the ^^ operator. Rules mcp-1 to mcp-4 for multiple checkpoint operator are implemented as follows:

```
infer(mckp(A,B), Action, mckp(A1,B), Any) :-   (* rule mcp-1 *)
        infer(A, Action, A1, Any),
        label(Action,L),
        not(in(L,[rlbck,commit] )).
infer(mckp(A,B), commit, start(A,mckp (A,B)), int).   (* rule mcp-2*)
infer(mckp(A,B), rlbck, B, int).                      (* rule mcp-3*)
infer(start(A,B),Action,mkcp(A1,B),Any):-             (* rule mcp-5*)
        infer(A,Action,A1,Any),
        label(Action,L),
        not(in(L,[rlbck,commit] )).
```

The other inference rules are implemented in a similar manner. The complete set of PROLOG rules is given in Appendix 5.

The design of a run time system for ATCCS is very similar to the design of SINAPS.

9.3 Implementation of the Expansion Theorem

In Section 4.6.1 we described the expansion theorem as a means by which one can describe the behavior of systems in terms of the behaviors of their components.

Having implemented the inference rules as shown in Section 9.2, we can also implement the expansion theorem. This implementation takes as input a behavior expression and gives as output an expression, with some necessary intermediate process declarations, of a behavior that is equivalent (in the sense of =) to the initial one which includes only the basic operators: sequencing, strong sequencing, choice and recursion. One must notice that in some cases where recursion is involved (e.g. `p := a;stop | p`), this procedure does not terminate in which case we must stop it by using techniques such as level counters or others.

The transformation relies on the construction, for each behavior expression, of all the possible actions together with their derivatives. That is, for a behavior expression *B*, we construct the set:

$$P(B) = \{ (a_i, B_i) \mid B \text{ -}a_i\text{-(any)-> } B_i \}$$

Each action a_i leads to a resulting behavior expression B_i. The set *P(B)* gives for a given behavior expression *B* all the possible choices that we have starting with *B*. We can link these choices by means of the choice operator. Each one corresponds to an action a_i and leads to a next state represented by behavior expression B_i (i.e. $a_i;B_i$ or a_i*B_i). From that next state, we construct its corresponding $P(B_i)$ and the process continues. If the initial behavior has a finite number of possible actions, the procedure terminates after encountering a next state that is equal (i.e. identical as a behavior expression) to a state

that has already been considered. Of course this procedure may not stop. For example, if data is involved, one may keep finding new values for the variables.

One should note that the procedure described above was implemented for all operators that admit expansions: *enables*, *disable*, *composition* and the *checkpoints* operators.

Example 9.1

Process p described as:

```
            p := p1 | p2
where       p1 := a ; p1
and         p2 := b ; p2
```

is equivalent to `q := a ; q + b ; q` since both p and q describe all the possible interleavings of an arbitrarily long sequence of a's with an arbitrarily long sequence of b's. Process q can be obtained by the expansion of p using the procedure described above for which we give the following algorithm:

Algorithm

The algorithm uses three sets:

a) Cp is the set of nodes that were already visited and not expanded yet. This set will hold behavior expressions indexed by an integer n.

b) V is the set of process behavior expressions that were already visited and already expanded. For each element B in this set, we construct a process declaration (also called *a process abstraction*) indexed by an integer, in the form $abst_n := B$. These process declarations, contained in V, can be used in process instantiations when,

during the expansion, we encounter a behavior expression for which a process abstraction exists and belongs to V.

c) PB contains the set of resulting behavior expressions that are built at each step. These behavior expressions together with their corresponding actions that lead to them, are used to construct "summands" by means of the choice operator. This set actually implements the set P(B) for a given B.

In the algorithm (written in a Pascal-like pseudo-code) we use a procedure called *expand* and a function called *name*. *expand* does the actual behavior expansions and *name* allows to create process instantiations. We also use global variables Cp, V, and n.

```
procedure expansion(P)
(* expansion of process P *)
begin
    n:= 0; V={};
    Cp := { <P,n>} ;
    while not(empty(Cp)) do
    begin
        for all <e,n> in Cp do
        begin
            expand(e,n);
            remove <e,n> from Cp;
            add <e,n> to V ;
        end
    end
end (* expansion*)

procedure expand(e,n)
(* Construct a process instantiation numbered n for behavior expression e *).
begin
    PB:= {};
    For all e' such that e-a-(at)-> e' do
```

```
            add summand "e • name(e') " to PB;
        For all e' such that e-a-(int)-> e' do
            add summand "e ; name(e') " to PB;
        create process "abst_n := PB" ;
    end (* expand *)

    function name(e)
    (* Get the name of expression e if it exists. Otherwise create a new name *)
    begin
        if <e,i> is in V ∪ Cp for some i then
            return(abst_i)
        else
            begin
                n := n+1;
                add <e,n> to Cp;
                return(abst_n);
            end
    end (* name *)
```

We implemented this algorithm in PROLOG as listed in Appendix 4.

An example of the use of the algorithm

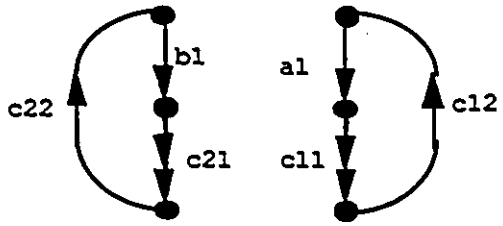
In Section 4.8.1 we modeled two concurrent processes that want to enter their critical sections. These critical sections are mutually exclusive. We describe this fact by means of the strong sequencing operation. The two processes were composed using the | operator. Using the expansion, we construct an equivalent process where we can see that mutual exclusion is guaranteed. Using the algorithm, we derive this expansion. Let us recall the specification:

```
p1 :=  a ; c11 * c12 ; p1

p2 :=  b ; c21 * c22 ; p2

mutex := p1 | p2
```

The two processes can be represented using the following state diagrams:



By running the program, we obtained the following specification:

```
abst0 := a1 ; abst1 + b1 ; abst2

abst1 := b1 ; abst3 + c11 * abst4

abst2 := a1 ; abst3 + c21 * abst5

abst3 := c11 * abst6 + c21 * abst7

abst4 := c12 ; abst0

abst5 := c22 ; abst0

abst6 := c12 ; abst2

abst7 := c22 ; abst1
```

`abst0` has the state diagram given in Figure 9.3. We can observe that the two critical sections `c11*c12` and `c21*c22` never overlap in time, in the sense that after `c11` (resp. `c21`) occurs, only `c12` (resp. `c22`) can occur.

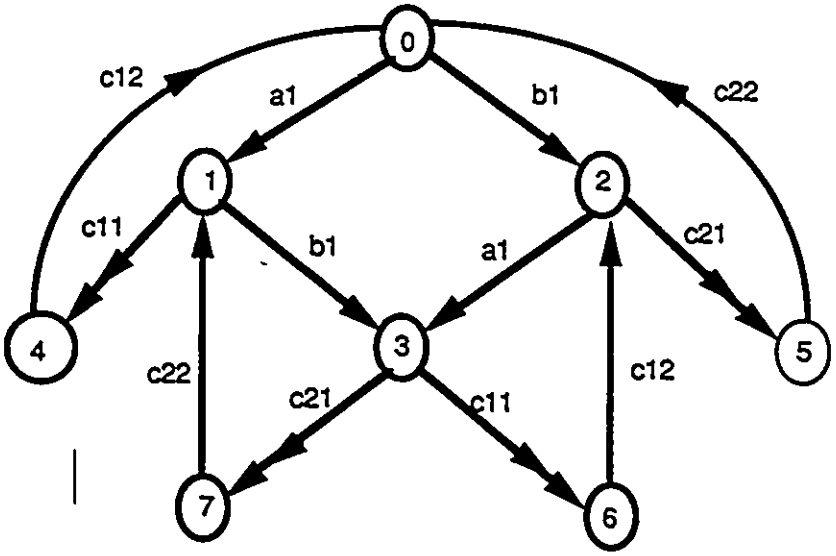


Figure 9.3: State diagram for abst0.

Chapter 10 :

Summary and Conclusions

Atomicity is an important concept in the description of concurrent systems. Algebraic specification techniques such as CCS, CSP and LOTOS cannot describe complex atomic actions directly. We have shown in this thesis that the concept of atomicity can be included in the semantics of the ATCCS calculus. This was illustrated by means of several examples. Our model could be considered as an important extension of CCS by the introduction of several new operators and the concept of recovery. The semantics of this extension is described by means of operational semantics. As we wanted to preserve this type of semantics, we introduced a set of attributed inference rules to model atomicity. This also led to the definition of additional composition and disabling operators (Chapter 4) peculiar to ATCCS.

The model shows a set of interesting formal properties mainly with respect to the composition operators (associativity, commutativity, ...). Moreover, as in CCS, we have defined expansion theorems for our calculus. These theorems help to derive expressions from which composition operators may be eliminated (Chapter 4).

Applications of our calculus were given and the usefulness of modular design by system decomposition was shown in ATCCS versus other formalisms. For example in calculi such as CCS or LOTOS, in order to refine atomic actions it is necessary to include mutual exclusion mechanisms such as semaphores. By using ATCCS, instead, it is possible to decompose atomic actions into atomic sequences of actions, which appears to be simpler and more natural (Chapter 5).

As shown in several examples in this thesis, we had to address the concept of recoverability in the case of failures within atomic actions. This concept was modelled by means of inference rule semantics in the same way that ATCCS as a whole was defined. We introduced checkpoint concepts and generalized them: simple, multiple and automatic. These concepts also exist in several applications related to distributed systems (e.g. database systems,...) (Chapter 6).

Applications for our model, on which we are working currently , include the specification of transaction systems on distributed data bases.

Formal models such as ATCCS and CCS allow for formal verification. One of the most common ways to achieve this is by using the concept of observation equivalence which allows to decide whether two descriptions can be distinguished by any experiment. This concept also allows to prove formal algebraic properties. We generalized and adapted to our model this equivalence concept by using the concept of bisimulation. Several formal properties were given and proved (Chapter 7). Since our model modified somewhat the CCS semantics of communication and interleaving, these modifications led to certain problems with respect to the notion of congruence equivalence. In order to overcome these problems, we introduced two enhancements to the semantic model, namely eliminations of internal events from strong sequences of actions, and the notion of

communication between atomic actions that become reduced to one internal event, instead of a sequence of them (Chapter 8).

A formal description technique such as ATCCS allows to describe systems at different levels of abstraction. The process of writing specifications may be lengthy and error prone. One way to increase the confidence of the specifier on what he or she specified is by means of simulation, which allows exercising the specification and hopefully discovering errors. We have implemented a run time system for ATCCS, similar to another system we implemented for CCS at an early stage of our work. This system allows to execute specifications in a step by step manner. A specifier follows his or her specification by selecting the observable behavior and by directing the choice of internal actions. Also we gave an implementation of the expansion theorems which helps in the verification process. All the implementations were carried out the PROLOG language (Chapter 9).

Atomicity has been studied by several authors (Chapter 4). All of them agree on the importance of this concept in system design and specification. Most of these authors either use transformations of behavior expressions in order to eliminate some unnecessary interleavings, or directly introduce atomicity in a semantic model. For example, ACP uses an algebraic theory for the description of the semantics whereas we use an operational model. In our model, however, we are closer to the way CCS and LOTOS are described. Moreover our model allows executability. This has the advantage of allowing us to build run time systems as prototypes as shown throughout this thesis.

An interesting extension of our model would be to allow for several levels of atomicity. This would, for example, allow modeling processes where actions (and may be even processes) may have priorities associated with them and therefore formalizing the

notion of scheduling between processes. Some authors have already added these aspects to specification languages but without atomicity [BOC 88].

Another interesting extension would be to consider atomicity in a model where time is involved to actually implement the notion of temporal checkpoints. Some authors have already added time to process algebra [QUE 87] [GRO 90].

BIBLIOGRAPHY

[AZE 85] Azema P. and Papanagiotakis G., Protocol Analysis by using Predicate Nets. Proceedings of the 5th International Workshop on Protocol Specification, Verification and Testing, Toulouse, France, June 1985.

[AZE 78] Azema P., Ayache J. and Berthomieu B. Design and verification of communication procedures: a bottom-up approach. 3rd International Conference on Software Engineering, 1978., pp. 168-174.

[BAR 69] Bartlett K.A. , Scautelberry R.A. and Wilkinson P.T., A Note on Reliable Full-duplex Transmission over Half-duplex Links, Communications of the ACM, vol 12, #3, May1969, pp. 260-261.

[BER 86a] Bergstra J.A. and Klop W., Process Algebra for Synchronous Communication, Information and Control 60, 1986, pp. 109-137.

[BER 86b] Bergstra J.A. and Klop W., Process Algebra: Specification and Verification in Bisimulation Semantics. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra Editors, Mathematics and Computer Science II, CWI Monographs 4, North-Holland, Amsterdam, 1986, pp. 61-94.

[BIL 82] Billington J., Specification of the Transport Service using the Numerical Petri Nets, Proc. of the 2nd IFIP International Workshop on Protocol Specification, Testing and Verification, May 82, pp.77-100.

[BLO 87] Bloom B., Istrail S. and Meyer A.R., Bisimulation can't be traced, Technical Report, MIT/LCS/TM-345, Massachusetts Institute of Technology, Nov. 1987.

[BOC 75] Bochmann G.V., Logical Verification and Implementation of Protocols, Proceedings of the 4th data communication symposium, IEEE, 1975.

[BOC 77a] Bochmann G.V and Gecsei J., A Unified Method for the Specification and Verification of Protocols, Proceedings, Information Processing 77, pp. 230-234.

[BOC 78] Bochmann G.V., Finite State Description of Communication Protocols, Computer Networks 2, (4/5), Sept.-Oct. 78.

[BOC 80] Bochmann G.V., Merlin P., On the Construction of Communicating Protocols, Proceedings of the international Conference on Computer Communications, 1980, pp. 371-379.

[BOC 88] Bochmann G.V., and Vaucher J., Adding Performance Aspects to Specification Languages, Proc. of the 6th IFIP International Workshop on Protocol Specification, Testing and Verification, May 86, pp.19-32.

[BOL 87a] Bolognesi T. & Brinksma E., "Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN systems, 14: 25-59, 1987.

[BOL 87b] Bolognesi T. and Smolka S. A., Fundamental Results for the Verifiocation of Observation Equivalence: a Survey, Proc. of the 7th IFIP International Workshop on Protocol Specification, Testing and Verification, May 87, pp. 165-179.

[BOY 79] Boyer R.S. and Moore J.S., **A Computational Logic**, Academic Press, New York, 1979.

[BOU 88] Boudol G. and Castellani I., Concurrency and Atomicity, Theoretical Computer Science 59, 1988, pp. 1-60.

[BOU 89] Boudol G. and Castellani I., Atomic Actions, Rapport de Recherche N1026, INRIA Sophia- Antipolis, France, Mai 1989.

[BOU 85] Boudol G., Notes on Algebraic Calculi for Processes, Logics and Models of Concurrent Systems, NATO ASI Series f13, K. Apt ed., 1985.

[BRA 78] Brand D. and Joyner W.H., Verification of Protocols using Symbolic Execution, Computer Networks 2 4/5), Sept.-Oct. 78.

[BRA 83] Brand D. and Zafiropulo P., On Communicating Finite State Machines, JACM, Vol 30, #2, Apr.83, pp. 323-342.

[BRE 88] Bretchneider M., Duque Anton M., and Fink B., Constructing and Verifying Protocols using TCSP, Proceedings of the FORTE Conference on Formal Description Techniques 1988.

[BRI 84] Brinksma E. and Karjoth G., A Specification of the OSI Transport Service in LOTOS, Proc. 4th IFIP International Workshop on Protocol Specification, Testing and Verification, 1984, pp. 227-251.

[BRI 86] Brinksma E., Scollo G., and Steenbergen C., LOTOS Specifications, their Implementations, and their Tests, in Protocol Specification, Testing, and Verification, Bochmann G.V., and Sarikaya B. (eds.), IFIP 6, North Holland, 1986, pp. 349-360.

[BRIA 86a] Briand J.P, Fehri M.C., Logrippo L. Obaid A. . Executing LOTOS specifications, in Protocol Specification, Testing, and Verification, Bochmann G.V., and Sarikaya B. (eds.), IFIP 6, North Holland, 1986.

[BRIA 86b] Briand J.P. , Fehri M.C., Logrippo L., and Obaid A., Stercuture of a LOTOS Interpreter, Proc. ACM-SIGCOM Conference, Stowe, Vermont, 1986, pp. 167-171.

[BRO 84] Brookes, S.D, Hoare C.A.R., and Roscoe A.W., A Theory of Communicating Sequential Processes, JACM, Vol. 31, # 3, July 1984, pp. 560-599.

[BUH 83] Buhr R.J.A., **System Design with ADA**, Prentice-Hall, 1983.

[CAR 84] Carchiolo V., Faro A. and Scollo G., Temporal Ordering Specification of Some Session Services, ACM SIGCOM Symposium on Communication Architectures and Protocols, Montreal, 1984, pp. 107-114.

[CAR 85] Carchiolo V., Le Moli G., Palazzo S. and Pappalordo G., Modelling and Specifying a Presentation Protocol by Temporal Ordering, Proc. 5th IFIP International Workshop on Protocol Specification,Testing and Verification, 1985, pp. 423-440.

[CAS 88] Castellani G. and Hennessy M., Distributed Bisimulations, Rapport de Recherche N875, INRIA Sophia- Antipolis, France, Juillet 1988.

[CLE 89] Cleveland R., Parrow J. and Steffen B., A Semantics Based Verification Tool for Finite State System, Proc. 9th IFIP International Workshop on Protocol Specification, Testing and Verification, 1989.

[CLO 84] Clocksin W.F. and Mellish C.S.,**Programming in PROLOG**, Springer-Verlag, Berlin 1984.

[CHO 83] Chow C.H. and Gouda M.G., A Discipline for Constructing Multiphase Communicating Protocols, Research Report TR-233, Oct. 83, Dept. of Comp. Science, University of Texas at Austin.

[CLA 86] Clark E.M., Emersion E.A. and Sistla P., Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, ACM Transactions on Programming Languages and Systems, 8, #2, April 86, pp. 244-263.

[COU 84] Courtiat J.P., Ayache J.A. and Algayrs B., Petri Nets are Good for Protocols, ACM SIGCOM Symposium on Communication Architectures and Protocols, Montreal 84, pp. 66-76.

[DAN 77] Danthine A. and Bremer R.J., Modelling and Verification of End-to-end Transport Protocols, Computer Networks 2 (4/5), Oct. 78, pp. 381-395.

[DAN 77a] Danthine A., Petri Nets for Protocol Modelling, IFIP-T66, COMNET Symposium, Budapest, Hungary, Act. 77.

[DAN 80] Danthine A., Protocol Representation with Finite State Models, IEEE Trans. on Communications, COM-28, #4, April 80, pp. 632-643.

[DIA 82] Diaz M., Modelling an Analysis of Communication and Cooperation Protocols Using Petri Nets Based Models, Proc. 2nd IFIP International Workshop on Protocol Specification, Testing and Verification, May 82, pp. 465-519.

[DIJ 65] Dijsktra E.W., **Co-operating Sequential Processes**, in Programming Languages, Genuys, F. (ed.), London: Academic Press, 1965.

[DIJ 76] Dijsktra E.W., **A Discipline of Programming**, Engelwood Cliffs, NJ., Prentice Hall, 1976.

[DOE 83] Doeppner T.W. and Giacalone A., A formal description of the UNIX* Operating System, Proceedings of 2nd Annual ACM Symposium on Principle of Distributed Computing, Montreal, Quebec, Aug 83, pp. 241-253.

[EHR 85] Ehrig H. and Mahr B., **Fundamentals of Algebraic Specifications 1**, Springer-Verlag, Berlin 1985.

[FEH 86] Fehri M.C., An Abstract Data Type Interpreter, Master Thesis, University of Ottawa, 1987.

[FLO 67] Floyd R.W., Assigning Meanings to Programs, Proceedings of Symposium in Applied Mathematics, American Mathemetical Society, 1967, pp. 19-32.

[GER 80] Gerhart S.L and all., An Overview of AFFIRM : a Specification and Verification System, Proc. IFIP congress, Austria, Oct. 80, pp. 343-348.

[GIA 88] Giacalone A. and Smolka S.A., Integrated Environments for Formally Well-Founded Design and Simulation of Concurrent Systems, IEEE Transactions on Software Engineering, Vol 14-6, June 1988, pp. 787-802.

[GOG 77] Goguen J.A and all., Initial Algebra Semantics and Continuous Algebra, JACM, 24(1977), pp. 68-95.

[GOR 89] Gorrieri R., Marchetti S., and Montanari U., A^2CCS: Atomic Actions for CCS, Personal communication. To appear in Theoretical Computer Science, 1989.

[GOT 86] Gotzhein R., Specifying Abstract Data Types with LOTOS, Proc. of the 6th IFIP International Workshop on Protocol Specification, Testing and Validation, Montreal, July 1986, pp. 1-13.

[GOU 83] Gouda M.G. An Example for Constructing Communicating Machines by Step-wise Refinement, Proc. 3rd IFIP International Workshop on Protocol Specification, Testing and Verification, pp. 63-74, 1983.

[GOU 84] Gouda M., Closed Covers: to Verify Progress for Communicating Finite State Machines, IEEE Trans. on Software Engineering, Vol SE-10, #6, Nov. 84, pp. 846-855.

[GRO 90] Groote J.F., Specifying Real-Time Systems in ACP, Proc. 10th IFIP International Workshop on Protocol Specification, Testing and Verification, Ottawa, Canada, 1990, pp. 259-268.

[GUT 78] Guttag J.V. and Horning J.J., The Algebraic Specification of the Abstract Data Types, Acta informatica 10, 1978, pp. 27-52.

[GUT 79] Guttag J.V. and Horning J.J., Formal Specification as Design Tool, Proceeding of the 5th. Conference on the Specification of Reliable Software, ACM SIGPLAN/SIGSOFT, Cambridge, Mass., 1979.

[HAL 80] Halpern B.J., Verifying Concurrent Processes Using Temporal Logic, Ph.D thesis, Computer Science Laboratory, Dept. of Electical Engineering, Stanford University, Aug.80.

[HAA 85] Haas O. Formal Protocol Specification Based on Attribute Grammars, Proc. 5th IFIP International Workshop on Protocol Specification, Testing and Verification, 1985, pp. 39-48.

[HAR 78] Harazango J., Protocol Definition with Formal Grammars, Proceedings of the Symposium of Computer Communication Protocols, IFIP T.C.6, Liege, Belgium, Feb. 79.

[HAL 83] Halpern B.J. and Owicki S., Modular Verification of Communication Protocols, IEEE Trans. on Communication, Vol. COM-31, #1, Jan. 83, pp. 56-68.

[HEN 85] Hennessy M. Acceptance tree, Journal of the ACM, Vol. 32, #4, Oct. 85, pp. 896-928.

[HOA 69] Hoare C.A.R., An Axiomatic Basis for Computer Programming, Comm. of the ACM, Vol 12, #10,1983, pp. 576-80.

[HOA 78] Hoare C.A.R., Communicating sequential processes, Comm. of the ACM, Vol 21, #8, Aug. 78, pp. 666-677.

[HOA 85] Hoare C.A.R., **Communicating Sequential Processes**, Prentice-Hall, 1985.

[ISO 83] ISO TC97/SC16 N1347, A FDT based on an Extended State Transition Model, Technical Report, ISO, July 1983.

[ISO 84] Information Processing Systems- Open System Interconnection, LOTOS, A formal Description Technique, Ref. ISO/TC97/WG16-1N99.

[ISO 85] Information Processing Systems- Open System Interconnection, ESTELLE, A Formal Description Technique Based On Extended State Transition Model.Draft ISO/DP 9074.

[ISO 86] Working Document on LOTOS- Draft Proposal. DP8007- 1986

[ITO 88] Itoh M., Calculus of Communicating systems for Refined Obervation of Actions, Publication #60, Université de Montréal, Département d' Informatique et de Recherche Opérationnelle, Janvier 1988.

[JAC 75] Jackson M.A., **Principles of Program Design**, Academic Press, N.Y., 1979

[KNU 68] Knuth D.E., Semantics of Context-free Languages, Mathematical System Theory 2, 1968 , pp. 127-145.

[KAN 90] Kanelakis P.C. and Smolka S.A., CCS expressions, Finite State Processsors, and Three problems of Equivalence, Journal of Information Computing , vol. 86, #1, 1990, pp. 43-68.

[KEL 76] Keller R.M., Formal Verification of Parallel Programs, Communications of the ACM, Vol 19, #7, July76.

[LAM 80] Lamport L., Sometime is Sometimes Not Never, Proceeding of the 7th ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, June 1980, pp. 174-185.

[LAR 88] Larsen K. and Skou A., Bisimulation Trough Probabilistic Testing, Technical Report RR-88-29, Institut For Elektroniske Systemer, Afdeling For Matematik og Datalogi, Aalborg, Danmark, Oct. 1988.

[LIS 74] Liskov B. and Zilles S.N., Specification Techniques for Data Abstractions, IEEE Trans. on Software Engineering, March 75, pp. 7-19.

[LOG 83] Logrippo L., Constructive and Executable Specifications of Protocol Services by Using Abstract Data Types and Finite State Transducers, Proc. 3rd IFIP International Workshop on Protocol specification, Testing and Verification, 1983, pp. 111-124.

[LOG 88] Logrippo L., Obaid A., Briand J.P. and Fehri M.C., LOTOS: An Exeeeutable Specification Language for Distributed Systems, Software: Practice and Experience, Vol. 122, #4, 1988, pp. 365-385.

[LAM 84] Lam S.S. and Shankar A.U., Protocol Verification via Projections, IEEE Trans. on Software Engineering, Vol. SE-10, #4, July 84.

[MAN 81] Manna Z. and Pnuelli A. Temporal logic, in The Correctness Problem in Computer Science, Boyer and Moore ed., Academic Press 1981, pp. 215-273.

[MER 76] Merlin P.M., A Methodology for the Design and Implementation of Communication Protocols, IEEE Trans. on Communications, Vol COM-24, #6, June 76.

[MER 76b] Merlin P.M. and Farber D.J., Recoverability of Communication Protocols-Implications of Theoretical Study, IEEE Trans. on Communications, Vol. COM-24, Sept.1976.

[MER 79] Merlin P.M., Specification and validation of Protocols, IEEE Trans. on Communications, Vol COM-27,#11, Nov. 79, pp. 1671-1679.

[MER 83] Merlin P. and Bochmann G.V., On the Construction of Sub-module Specifications and Communication Protocols, ACM TOPLAS, Vol. 15, #1,Jan 83, pp. 1-25.

[MIL 80] Milner R., A Calculus of Communicating Systems, Lecture Notes in Computer Science 92, 1980.

[MIL 83] Milner R., Calculi for Synchrony and Asynchrony, Theoratical Computer Science, 25(1983), pp. 267-310.

[MIL 89] Milner R., **Comunication and Concurrency**, Prentice Hall, 1989.

[MIL 84] Milner R, A Complete Inference System for a Class of Regular expressions, Journal of Computing and System Science, 28, 1984, pp. 439-460.1984.

[MIN 83] Milne G. CIRCAL a Calculus for Circuit Description, INTEGRATION the VLSI journal, 1,2,3(1983), pp. 121-160

[MIN 85] Milne G., CIRCAL and the Representation of Communication, Concurrency and Time, in ACM Trans. on Programming Languages and Systems, Vol 7,#2, Apr. 85, pp. 270-298.

[MYE 79] Myers G. L., **The Art of Software Testing**, Willey- Interscience, New York, 1979.

[OBA 85] Obaid A., Applications of the Inference Rules of CCS and LOTOS, Technical report, TR-14-85, Computer Science Dept., University of Ottawa, 1985.

[OBA 87a] Obaid A. and Logrippo L., SINAPS : An Interactive Simulator for CSS* , Proceedings of the IASTED Conference on Applied Simulation and Modelling, Santa Barbara, California, U.S.A., May 1987, pp. 160-163.

[OBA 87b] Obaid A. and Logrippo L., An Atomic Calculus of Communicating Systems, Protocol Specification, Testing, and Verification 7, in H. Rudin and C.H. West (eds.), North Holland, 1987, pp. 91-104.

[OCC 86] OCCAM programming manual, Prentice-Hall international series in computer science, 1986.

[OWI 82] Owicki S. and Lmport L., Proving Liveness Properties of Concurrent Systems, ACM Transactions On Programming Languages and Systems , Vol. 4 , 1982, pp. 455-495.

[PAR 77] Parnas D.L., The Use of Precise Specification in the Development of Software, IFIP 77, 1977, pp. 861-867.

[PARK 81] Park D., Concurrency and Automata on Infinite Sequences, Theoretical Computer Science, 5th GI-Confernce.,LNCS vol. 104, Springer Verlag, 1981.

[PARR 87] Parrow J.G., Verifying a CSMA/CD-Protocol with CCS, Report ECS-LFCS-87-18, Computer Science Department, Unviversity of Edinburg, 19887.

[PET 62] Petri C.A., Communication with Automata, Thesis, Darmstat institute of technology, Bonn, Germany, 1962.

[PET 77] Peterson J.L, Petri nets, ACM Computing Surveys, Vol 9, #3, Sept. 77.

[PLO 81] Plotkin G., A Structural Approach to Operational Semantics, Daimi FN -19, Aarhus university, 1981.

[PNU 77] Pnuelli A. The Temporal Logic of Programs, IEEE,18th Annual Symposium on Foundations of Computer Science, Providence, Rhodes island, 1977, pp. 46-57.

[PRO 84] Probert R.L and Ural H., High-level Testing and Example-Directed Development of Software Specifications, Journal of Systems and Software,Vol 4, #3, 1984, pp. 317-325.

[PRO 88] Probert R.L, Ural H., and Hornbeek M.W.A., An Integrated Software Environment for Developing and Validating Standardized Conformance Tests, Proc. 8th IFIP International Workshop on Protocol Specification, Testing and Verification,1988, pp. 87-100.

[QUE 87] Quemada J., Fernandez A., Introducing Quantitative Relative Time into LOTOS, Proc. 7th IFIP International Workshop on Protocol Specification, Testing and Verification, Rudin H. and West C.H. Editors, North Holland, 1987, pp. 105-121.

[RAZ 80] Razouk R.R. and Estrie G., Modeling and Verification of Communication Protocols in SARA : The X.21 interface, IEEE Transactions on Computers, COM-29, #12, Dec.80, pp. 1038-1051.

[RAF 83] Rafiq O. and Ansart J.P., A Protocol Validator and its Applications, Proc. 3rd IFIP International Workshop onProtocol Specification, Testing and Verification , 1983, pp. 189-212.

[ROS 85] Rosin C. Protocol Specification with the OCCAM Language, Proc. 5th IFIP International Workshop on Protocol Specification, Testing and Verification, pp. 235-246, 1985.

[RUD 82] Rudin J. and West C.H., An Improved Protocol Validation Technique, Computer Networks 6(1982), vol 2, pp. 65-73.

[SAR 82] Sarikaya B. and Bochmann G., Some Experience with Test Sequence Generation for Protocols, Proc. 2nd IFIP Int.. Workshop on Protocol Specification, Testing and Verification, 1982.

[SCO 86] Scollo G. and Van Sinderen C., On the Architectural Design of the Formal Specification of the Session Standard in LOTOS, Proc. of the 6th IFIP International Workshop on Protocol Specification,Testing and Verification, Gray Rocks Inn Montreal, June 86, pp. 1.1-1.12

[SCH 81] Schwartz R.L and Melliar-Smith P.M., Temporal logic Specifications of Distributed Systems, 2nd Conf. on Distributed Computing Systems, Paris, France, Apr. 81.

[SCH 82] Schwartz R.L and Melliar-Smith P.M., From State Machine to Temporal Logic: Specifications Methods for Protocol Standards, IEEE Trans. on Communications, Vol. COM-30, #12, Dec. 82, pp. 2486-2496.

[SED 85] Scollo G., Draft proposal in LOTOS of the OSI Transport Protocol, SEDOS publication, Ref SEDOS/C1/22,Apr. 85.

[SOR 85] Soriano A. and Voiron J., Conception et mise en oeuvre d'un système de preuve pour un sous-ensemble de CCS sans transmission de valeurs, Rapport de Recherche, RR #530 , IRISA, France, mai 1985.

[SHI 83] Shids M.W. and Wray M.J., A CCS Specification of the OSI Network Service, Internal report, CSR-136-183, 1983, University of Edimburg, England.

[STE 76] Stenning N.V., A Data Transfer Protocol, Computer Networks, Vol 12, Sept. 76, pp. 99-110.

[SUN 82] Sunshine C.A. and all. Specification and Verification of Communicating Protocols in AFFIRM Using State Transition Models, IEEE Trans. on Software Engineering, Vol SE-8, #5, Sept. 82, pp. 460-489.

[SYM 80] Symons F.J.W., Introduction to Numerical Petri Nets: A General Graphical Model of Concurrent Processing Systems, Australian Telecommunication Research, Vol. 14, #1, 1980, pp. 28-32.

[TAY 86] Taylor D.J., How Big Can an Atomic Action Be?, Technical Report-CS-85-20, Dept. of Comp. Sci., University of Waterloo, July 86.

[TEN 78] Teng A.Y. and Liu M.T., A Formal Approach to the Design and Implementation of Network Communication Protocol, Proceedings of the 2nd international Computer Software and Applications Conference, IEEE Computer Society, Chicago, Illinois, Nov. 78, pp. 722-727.

[URA 86] Ural H. and Probert R.L., Step-wise Validation of Communication Protocols and Services, Computer Networks and ISDN systems, Vol. 11, #3, March 86, pp.183-202.

[VIS 85] Vissers C. and Logrippo L., The importance of the Service Conception the Design of Data Communication Protocols, Proc. 5th IFIP International Workshop on Protocol Specification, Testing and Verification,1985, pp. 3-18.

[VIS 88] Vissers C. , Specification Styles in LOTOS, Protocol Specification, Testing, and Verification 8, S. Aggarwal and K. Sebnani (eds.), North Holland, 1988, pp. 189-204.

[WES 78a] West C.H., A General Technique for Communication Protocol Validation, IBM Journal for Research and Development, Vol 22, #1, July 78, pp. 393-403.

[WES 78b] West C.H., An Automated Technique for Communication Protocol Validation, IEEE Trans. on Communications, Vol. COM-26, #8, 1978.

[WES 78c] West C.H. and Zafiropulo P., Automated Validation of a Communication Protocol: the CCITT X.21 Recommendations, IBM journal of Research development, Vol. 22, Jan. 78, pp. 60-71.

[WES 86] West C.H., Protocol Validation by Random State Exploration, Proc. 6th IFIP International Workshop on Protocol Specification, Testing and Verification,1986, pp. 233-242.

[ZAV 85] Zave P., A Distributed Alternative to Finite Machine Specifications, ACM Trans. on Programming Languages and Systems, Vol 7, #1, Jan. 85, pp. 10-36.

[ZAF 78] Zafiropulo P., Protocol Validation by Duologue Matrix Analysis, IEEE Trans. on Communications, Vol. COM-26, #8, Aug. 78.

[ZAF 80] Zafiropulo P. and all., Towards Analysing and Synthetising Protocols, IEEE Trans. on Communications, COM-28, #4, April 80, pp. 651-661.

[ZIM 80] Zimmermann H., OSI Reference Model- The ISO Model of Architecture for Open Systems Interconnections, IEEE Trans. on Communications, Vol. COM-28, #4, Apr. 1980, pp. 651-661.

APPENDIX 1

SINAPS: A RUN TIME SYSTEM FOR CCS* IN PROLOG

In this appendix we give the complete listing of the SINAPS code in Prolog. An outline of this system was given in Section 9.2

```
%
%definitons  of  ccs*  operators
%

?-op(127,yfx,'hide').
?-op(127,yfx,'rell').
?-op(128,xfy,'seq').
?-op(129,xfy,'guar').
?-op(130,fy,'choice').
?-op(131,xfy,'comp').
?-op(132,xfy,'disa').
?-op(136,xfx,':=').


%
%inference  rules
%

infer(i seq B,i,B,[i]).
infer([G|A] seq B,[G|A],B,X):-
      is_action([G|A]).
infer(choice B,G,B11,X):-
     member(B1,B)
    ,infer(B1,G,B11,X).
infer(B1 comp B2,G,B11 comp B2,X):-
     infer(B1,G,B11,X).
infer(B1 comp B2,G,B1 comp B21,X):-
     infer(B2,G,B21,X).
infer(B1 comp B2,i,B11s comp B21s,[G,V]):-
     infer(B1,[G|A1],B11,_)
    ,infer(B2,[G|A2],B21,_)
    ,matching(A1,A2,V,X)
    ,substl([V],[X],B11,B11s)
    ,substl([V],[X],B21,B21s).
infer(B hide A,[G|Act],B1 hide A,X):-
      infer(B,[G|Act],B1,X)
     ,not(member(G,A)).
infer(B hide A,i,B1 hide A,X):-
     infer(B,i,B1,X).
infer(B rell S,Gs,B1 rell S,X):-
     infer(B,G,B1,X)
    ,relabels(G,Gs,S).
infer([E] guar B,G,B1,X):-
      eval([E],[true])
     ,infer(B,G,B1,X).
```

```
infer(B1 disa B2,[d|A],B11,X):-
     infer(B1,[d|A],B11,X).
infer(B1 disa B2,[G|A],B11 disa B2,X):-
     infer(B1,[G|A],B11,X)
    ,G \== d.
infer(B1 disa B2,G,B21,X):-
     infer(B2,G,B21,X).
infer(subst(X,V,B),G,B1,Y):-
      subst1(X,V,B,Bs)
     ,infer(Bs,G,B1,Y).
infer(ProcI,G,B1,X):-
     ProcI=..[ProcN|Value_args]
    ,Term := Bb
    ,Term=..[ProcN|Actual_values]
    ,eval(Value_args,Actual_values)
    ,infer(Bb,G,B1,X).


%
%simulation  procedures
%

sinaps:-erase,home,midprint('Welcome to sinaps version 1.'),nl,
        print('Type in the process to be '),nl,
        print('simulated followed  by a dot'),nl,
        abolish(alt,2),
        abolish(evaluated,2),
        read(X),go(X).

go(X):-go(X,[]).


go(B1,[]):-
     posit(6,1),eraseend,
     oprint(B1,'Current process : ')
    ,midprint(' Wait  ... '),nl
    ,bagof([Z,Br,M],infer(B1,Z,Br,M),S1)
    ,interface(S1,T)
    ,nl,next(B1,S1,T).
go(B1,S):-
     posit(6,1),eraseend,
     oprint(B1,' (same) Current process : ')
    ,infer(B1,G,U,I)
    ,interface(S,T)
    ,nl,next(B1,S,T).
go(B1,S):-
     posit(6,1),eraseend,
     nl,midprint(' Deadlock ! '),!
    ,posit(22,1)
    ,nl,midprint('Choose a command or type in help : ')
    ,nl,next(B1).


interface(S,T):-
     nl,write_1_by_1(S,T,1)
    ,posit(22,1)
    ,nl,midprint('Choose an action,  ')
    ,nl,midprint('Or choose a command or type in help : ').
```

```
next(B1):-input(X),( next3(B1,[],X,[])
                   ; nl,write(' Wrong command ')
                    ,nl,display_help
                    ,posit(22,1)
                    ,midprint('Choose a command or type in help : ')
                    ,next(B1)
                   ).

next(B1,S,T):-input(X),
                 ( X = [N],number(N),next1(B1,S,X,T)
                 ; X = [N|_],number(N),next2(B1,S,X,T)
                 ; next3(B1,S,X,T)
                 ; posit(22,1),midprint(' Wrong command or mismatch ')
                  ,nl,display_help,go(B1,S)
                 ).

next1(B1,S,[N],T):-
        n_th(T,N,X)
       ,n_th(S,N,[X,B11,_])
       ,up_date(B1)
       ,go(B11,[]).
next2(B1,S,[N|X],T):-
       user_input([N|X],[N,G|Y],Values)
      ,write(' Ok for the syntax !'),nl
      ,n_th(T,N,X1)
      ,n_th(S,N,[X1,B11,_])
      ,listOfVariables(X1,Variables)
      ,match_actions([G|Y],X1,_,_)
      ,subst1(Values,Variables,B11,B11s)
      ,up_date(B1)
      ,go(B11s,[]).
next3(B1,S,[X],T):-
        (  X=sch(Y)
           ,alt(Y, B)
           ,go(B,[])
         ; X=lch
           ,alt(_,B)
           ,go(B,[])
         ; X=delc(Y)
           ,alt(Y,B)
           ,retract(alt(Y,choice B))
           ,go(B1,S)
         ; X=new(Y)
           ,go(Y)
         ; X=scp(Y)
           ,asserta(cp(Y,B1))
           ,go(B1,S)
         ; X=gcp(Y)
           ,get_checkpoint(Y,B1,B2,S,S2)
           ,go(B2,S2)
         ; X=help
           ,display_help
           ,go(B1,S)
         ; X=la
           ,list(alt)
           ,go(B1,S)
         ; X=lcp
```

```
          ,list(cp)
          ,go(B1,S)
        ; X=ppr
          ,print(B1)
          ,go(B1,S)
        ; X=p_on
          ,assert(print_on)
          ,go(B1,S)
        ; X=p_off
          ,retract(print_on)
          ,go(B1,S)
        ; X=rma
          ,abolish(alt,2)
          ,go(B1,S)
        ; X=rmcp
          ,abolish(cp,2)
          ,go(B1,S)
        ; X=ht
          ,!,midprint(' End of Simulation ')
        ).


writel([]).
writel([X|Y]):-write(X),nl,writel(Y).

display_help:-
      nl,write('The meaning of the commands is as follows :')
     ,nl,write('  lch    : gives the last choice made. ')
     ,nl,write('  sch(n) : gives the choice with number n. ')
     ,nl,write('  delc(n): deletes the clause with number n.')
     ,nl,write('  scp(n) : set a checkpoint with number n.')
     ,nl,write('  gcp(n) : gives the checkpoint with number n. ')
     ,nl,write('  la     : lists all the constructed alternatives.')
     ,nl,write('  lcp    : list all the user checpoints.')
     ,nl,write('  ppr    : print the curent process.')
     ,nl,write('  p_on   : print option on. ')
     ,nl,write('  p_off  : print option off. ')
     ,nl,write('  ht     : stops the simulation process.')
     ,nl.


up_date(stop):-!.
up_date(A seq B ):-!.
up_date(X):-add_to_alt(X).

get_checkpoint(Y,B,B1,S,[]):-cp(Y,B1),!.
get_checkpoint(Y,B,B,S,S):-nl,write('Non existing checkpoint '),nl,!.


n_th([X|Y],1,X):-!.
n_th([X|Y],N,Z):-N1 is N-1,n_th(Y,N1,Z).

member(X,[X|_]).
member(X,[Y|Z]):-member(X,Z).


write_l_by_l([],[],_).
write_l_by_l([[i,Y,[M1,M2]]|Z],[i|T],N):-
        nl
```

```
       ,midprint(N),write(' : '),write(i)
       ,write(' {'),write(M1),write('?!')
       ,eval([M2],[M21]),write(M21),write('}')
       ,write('-->'),nl
       ,oprint(Y,'   ')
       ,N1 is N+1
       ,write_1_by_1(Z,T,N1).

write_1_by_1([[X,Y,M]|Z],[X|T],N):-
        nl
       ,midprint(N),write(' : '),printa(X),write('-->'),nl
       ,oprint(Y,'   ')
       ,N1 is N+1
       ,write_1_by_1(Z,T,N1).

list(alt):-
       write(' The choices you went thru are : ')
      ,nl,not(list1(alt))
      ,write(' End').
list(cp):-
       write(' Your checkpoints are : ')
      ,nl,not(list1(cp))
      ,write(' End ').

list1(alt):-alt(X,Y),write(X),write(' : '),print(Y),nl,fail.
list1(cp):-cp(X,Y),write(X),write(' : '),print(Y),nl,fail.


add_to_alt(B):-alt(_,B),!.
add_to_alt(B):-get_counter(X),asserta(alt(X,B)).

count(0).

get_counter(N1):-retract(count(N)),!,N1 is N+1,assert(count(N1)).
get_counter(0):-assert(count(0)),!.


%
%  inference  applications
%

call_validator(B1,[G|X],B11):-
        print(' Initial behaviour expression :')
       ,nl,print(B1),nl
       ,exerce(B1,[G|X],B11)
       ,nl,print(' The resulting behaviour expression is :')
       ,nl,print(B11)
       ,nl,nl,print(' Valid sequence '),fail.

   call_generator(B1):-
        print(' Initial behaviour expression :')
       ,nl,print(B1)
       ,nl,nl,write(' Generation starts :')
       ,repeat
       ,nl
       ,exerce(B1,[G|X],B11)
       ,nl,nl,print(' Applied action sequence :')
       ,nl,print1([G|X]),fail.
```

```
   call_r_generator(B1):-
        print(' Initial behaviour expression :')
       ,nl,print(B1)
       ,nl,nl,write(' Generation starts :')
       ,repeat
       ,nl
       ,r_exerce(B1,[G|X],B11)
       ,nl,nl,print(' Applied action sequence :')
       ,nl,printl([G|X]),fail.

   exerce(B1,[],B1).
   exerce(B1,[Action|RestOfActions],B2):-
        infer(B1,Action,B11,I)
       ,nl,write(' Interaction : '),print(I)
       ,exerce(B11,RestOfActions,B2).

%
%     random sequence generation
%

   r_exerce(B1,[],B1).
   r_exerce(B1,[Action|RestOfActions],B2):-
       bagof(G,Br^infer(B1,G,Br,I),S)
      ,select_a_branch(S,Action)
      ,infer(B1,Action,B11,J)
      ,nl,write(' Interaction : '), print(J)
      ,r_exerce(B11,RestOfActions,B2).

%
%  random selection procedure
%
     select_a_branch(S,A):-
        length(S,N)
       ,random(N,X)
       ,n_th(S,X,A).

%
%  portray facility for (nice) printing
%


portray(A comp B):-write('('),print(A),write(')')
                  ,write(' || '),write('('),print(B),write(')').
portray(A seq B):-printa(A),write(';'),print(B).
portray(A hide B):-write('('),print(A),write(')'),write('\'),write(B).
portray(choice B):-write('('),print_choices(B),write(')').
portray(E guar B):-write(E),write('->'),print(B).
portray(A rell B):-write('('),print(A),write(')'),write(B).
portray(ProcI):-ProcI=..[ProcN|Val_args],eval([Val_args],[Act_vals])
               ,ProcC=..[ProcN|Act_vals],write(ProcC).

printa([G|E]):-gate(G),write(G),print_events(E).
printa([#,X,T]):-write('!'),eval1(X,Y),write(Y),write(':'),write(T).
printa([$,X,T]):-write('?'),write(X),write(':'),write(T).
printa([$]):-write('?').
printa([#]):-write('!').

print_choices([]).
```

```
print_choices([X]):-print(X).
print_choices([X|Y]):-print(X),write(' + '),print_choices(Y).

printl([]).
printl([G1|G2]):-tab(10),write('< '),printa(G1),write('
>'),nl,printl(G2).

oprint(X,M):-
    print_on,
    nl,midprint(' Current process : '),nl
   ,print(B1),nl.
oprint(_,_).

print_events([]):-!.
print_events([E1|E2]):-printa(E1),print_events(E2).

%
%utility  procedures
%

append([],X,X).
append([A|X],Y,[A|R]):-append(X,Y,R).

delete_from([],L,L):-!.
delete_from([A|X],Y,Z):-delete(A,Y,Y1),delete_from(X,Y1,Z).

delete(A,[],[]).
delete(A,[A|X],Y):-delete(A,X,Y).
delete(A,[B|X],[B|Y]):-delete(A,X,Y).

select_nth(1,[X|_],X).
select_nth(N,[X|Y],Z):-
        select_nth(N1,Y,Z)
       , N is N1 + 1.


seed(13).

random(R,N):-
    retract(seed(S)),
    N is (S mod R)+1,
    NewSeed is (125*S + 1) mod 4096,
    asserta(seed(NewSeed)),!.

g_random(0,M,[]).
g_random(N,M,[X|Y]):-
     random(M,X),
     N1 is N - 1,
     g_random(N1,M,Y).



%     .
% action denotations and and matching rules
%

gate(d).
gate(G):-gates(X),member(G,X).
```

```
is action([G|A]):-gate(G),actions(A).

actions([[$]]).
actions([[#]]).
actions(E):-event_list(E).



event_list([]):-!.
event_list([X|Y]):-input_action(X),event_list(Y).
event_list([X|Y]):-output_action(X),event_list(Y).

input_action([$,X,T]).
output_action([#,X,T]).



match_actions([G|X],[G|Y],V,X1):-
     matching(X,Y,V,X1).
match_actions(_,_,_,_):-
     write(' Matching error '),!,fail.


matching([[$]],[[#]]).
matching([[#]],[[$]]).

matching(E1,E2,V,X):-
     matching_list(E1,E2,V,X).


matching_list([],[],[],[]):-!.
matching_list([E1|X],[E2|Y],[V1|R],[X1|S]):-
     matches(E1,E2,V1,X1),matching_list(X,Y,R,S).
matching_list([E1|X],[E2|Y],[V1|R],[X1|S]):-
     matches(E2,E1,V1,X1),matching_list(X,Y,R,S).

matches([$,X,T],[#,V,T],V,X).

%
%  miscellaneous  functions
%


eval([],[]).
eval([X|Y],[V|U]):-eval1(X,V),eval(Y,U).

eval1(X,Y):-rw(X,Y),!.

relabelling(G,Gs,[X,Y]):-select_nth(N,X,G),select_nth(N,Y,Gs).

relabels([G|E],[Gs|E],S):-relabelling(G,Gs,S),!.
relabels(G,G,S):-!.


%
%    rewriting system (written by M. Fehri )
%
```

```
rw(V,V)   :- atomic(V),!.
rw(V,VN)  :- !,
             V =.. [Fn|Args],
             rwl(Args,NewArgs),
             V1 =.. [Fn|NewArgs],
             rw2(V1,VN).


rwl([],[]) :- !.

rwl([H|T],[B|R]) :- rw(H,B), rwl(T,R).

rw2(X,Y) :- not(var(X)).X ==> Z,
            !,
            rw(Z,Y).

rw2(X,X) :- !.

infer1(ProcI,G,B):-
   ProcI =..[ProcN|Values],
   Term := B,
   Term =..[ProcN|Late_values],
   is(Late_values),
   write(Term).
```

APPENDIX 2

SIMULATION OF `simple-service` SPECIFICATION

This appendix contains an annotated script of a simulation session (using SINAPS) of a simple service specification (see Section 9.1.3.1).

Interacting with The System

```
| ?- sinaps.
Welcome to sinaps version 2.0
(C) University of Ottawa-1986
Enter a process !
|: service.
Ok ...
Choose an action or a command:

1 : get!sign:int-->

|: 1>get!sign:int.                (user provides a signal message)
Ok ...
Choose an action or a command:

1 : i {com1?![succ(zero)]}-->   (sender sends message 1 to channel)

|: 1.
Ok ...
Choose an action or a command:

1 : i-->                         (loss of message)
2 : i {com2?![succ(zero)]}-->   (receiver gets message 1)

|: 2.
Ok ...
Choose an action or a command:

1 : i {com3?![succ(zero)]}-->   (receiver sends an ack to channel)

|: 1.
Ok ...
Choose an action or a command:

1 : deliver!sign:int-->          (message delivery signal)
2 : i {com4?![]}-->              (ack received by sender)

|: 1>deliver!sign:int.           (user decides delivery first)
Ok ...
Choose an action or a command:

1 : i {com4?![]}-->

|: 1.                            (ack is now received)
Ok ...
```

```
Choose an action or a command:

1 : get!sign:int-->          (next message signal)

|: 1>get!sign:int.
Ok ...

Choose an action or a command:

1 : i {com1?![succ(succ(zero))]}-->
                              (sender sends message 2 to channel)
```

Using Checkpoints

```
|: la.                       ( list all checkpoints )
The choices you went thru are :

13 :
(   sender(succ(succ(succ(zero)))) [[out,ack],[com1,com4]]
 || channel[[d_in,d_out,a_in,a_out],[com1,com2,com3,com4]]
 || receiver(succ(succ(succ(zero))))[[d_in,ack],[com2,com3]]
 )  [com1,com2,com3,com4]

     ...

3 :
(     (ack!succ(zero):int;sender(succ(succ(zero))))
           [[out,ack],[com1,com4]]
   || (d_out!succ(zero):int;channel + i;channel)
           [[d_in,d_out,a_in,a_out],[com1,com2,com3,com4]]
   || receiver(succ(zero))[[d_in,ack],[com2,com3]]
 )  [com1,com2,com3,com4]

1 :
service


|: sch 3.                   (get back to checkpoint 3)
Ok ...

Choose an action or a command:

1 : i-->
2 : i {com2?![succ(zero)]}-->

|: 2.
Ok ...

Choose an action or a command:

1 : i {com3?![succ(zero)]}-->

|: 1.
```

Tracing Process Executions

```
| ?- tr service.          (user asks to trace back the whole service)

service

--get!sign-->

  (  (out!succ(zero):int;ack!succ(zero):int;sender(succ(succ(zero)))
     ) [[out,ack],[com1,com4]])
   || channel[[d_in,d_out,a_in,a_out],[com1,com2,com3,com4]]
   || receiver(succ(zero))
         [[d_in,ack],[com2,com3]]
  )  [com1,com2,com3,com4]

--i-->

  (     (ack!succ(zero):int;sender(succ(succ(zero))))
           [[cut,ack],[com1,com4]]
    || (d_out!succ(zero):int;channel + i;channel)
           [[d_in,d_out,a_in,a_out],[com1,com2,com3,com4]]
    || receiver(succ(zero))
           [[d_in,ack],[com2,com3]]
   )  [com1,com2,com3,com4]

      ....
      ....

--deliver!sign-->

   (sender(succ(succ(succ(zero))))[[out,ack],[com1,com4]]
    ||  channel[[d_in,d_out,a_in,a_out],[com1,com2,com3,com4]]
    ||  receiver(succ(succ(succ(zero))))[[d_in,ack],[com2,com3]]
   )  [com1,com2,com3,com4]

--get!sign-->
```

Finding Errors

```
|: sch 1.                 (user starts from the beginning)
Ok ...
Choose an action or a command:

1 : get!sign:int-->

|: 1>get!sign:int.
Ok ...
Choose an action or a command:

1 : i (com1?![succ(zero)])-->
```

```
|: 1.
Ok ...

Choose an action or a command:

1 : i-->                    (loss of message)

2 : i {com2?![succ(zero)]}-->

|: 1.                       (user decides loss of message)
Ok ...

Deadlock !

Choose a command or type in help :
|: ht.
 End of Simulation
```

Nondeterministic Behaviors

```
Choose an action or a command:

1 : get!sign:int-->

|: rand.                   (user chooses random execution option)

Choose an action or a command:

1 : get!sign:int-->

Random  selection option !
|: get!sign:int.
i {get?![sign]}           (system chooses the only possible action)
Ok ...

Choose an action or a command:

1 : i {com1?![succ(zero)]}-->

Random  selection option !
|: ok.                     ( no interaction is possible.Ok allows to proceed)

i {com1?![succ(zero)]}    (system takes the only choice)
Ok ...

Choose an action or a command:

1 : i-->

2 : i {com2?![succ(zero)]}-->

Random  selection option !
|: ok.
i {com2?![succ(zero)]}    (system chooses message delivery)
Ok ...
```

APPENDIX 3

AN EXAMPLE OF VALIDATION

We give an example of a validation process using SINAPS (see Section 9.1.3.1). Note that the sequence to be validated is shown in an internal CCS* representation, in which ! is represented by '#'.

Validation

```
|?- call_validator(simple_service,
                    [[get,[#,1,int]],
                     i,i,
                    [deiver [#,1,int]]
                    ],B11).

Initial behavior expression:
simple_service

The resulting behavior expression is:

      (   sender[com1/out]
       || buffer1[com1/in,com2/out]
       || receiver[com2/in]
      ) \ {com1,com2}


  Valid Sequence !!
```

Test sequence generation

```
       user1 :=  get!1 ; user1

       user2 :=  deliver?X:int ; user2
```

The generation process is applied to the following composition of the three processes:

```
      protocol := user1 || simple_service || user2


|?- call_generator(protocol).

Initial Behavior:

       protocol

Generation starts:
```

```
Applied action sequence:

            <get!1>
            <i>

Applied action sequence:

            <get?X:int>
            <i>
            <i>

Applied action sequence:

            <get?X:int>
            <i>
            <i>
            < deliver?X:int>
```

APPENDIX 4

EXPANSION OF ATCCS BEHAVIOR EXPRESSIONS IN PROLOG

This appendix is a listing of the PROLOG code that implements the expansion theorem for ATCCS (see Section 9.3).

```
expand          :-  write(' Enter process to be expanded:'),nl,
                    read(P),
                    count(N),
                    transf(P,P1),
                    assert(visited(undone,proc(toexp(N),P1))),
                    expansionLoop.

expansionLoop :-
      visited(undone,proc(toexp(N) ,P)),
      nl,write('Trying : '),write(N),nl,
      expanding(P,SP),
      assert(proc(abst(N) , SP)),
      retract(visited(undone,proc(toexp(N) ,P))) ,
      assertz(visited(done,proc(toexp(N) ,P))),
      expansionLoop.
expansionLoop.

expanding(P,SP) :- bagof(S,summand(P,S),Sol)
                  ,sumconstruction(Sol,SP).

summand(P, seq(A,call(abst(Nh)))) :- infer(P,A,Q,int), get_id(Q,Nh).
summand(P, sseq(A,call(abst(Nh)))) :- infer(P,A,Q,at), get_id(Q,Nh).

get_id(Q,Nh) :- visited(_,proc(toexp(Nh),Q)),!.
get_id(Q,Nh) :- increment
                , count(Nh)
                ,nl, write(' New index created :'),write(Nh),nl
                , assertz(visited(undone,proc(toexp(Nh),Q))),!.

sumconstruction([],[]):-!.
sumconstruction([X],X):-!.
sumconstruction([X|R],choice(X,R1)):- sumconstruction(R,R1).

count(0).
increment :-  retract(count(N)) , N1 is N + 1 , assert(count(N1)).

transf(call(P),Q):- proc(P,Q),!.
transf(P,P).


%
%  Display  facility
%
```

```
dispall :-
      proc(abst(N),B),
      write('('),write(N),write(')'),
      nl, tab(3),disp(B), nl,nl,
      fail.

disp(sseq(A,B)):-       write('--'),write(A),
                        write('-(at)-->'),disp(B).
disp(seq(A,B)):-        write('--'),write(A),
                        write('-(int)-->'),disp(B).
disp(choice(A,B)):-disp(A),nl,tab(3),write('+'),nl,tab(3),disp(B).
disp(call(abst(N))):-write('('),write(N),write(')').
```

APPENDIX 5

INFERENCE RULES OF ATCCS IN PROLOG

The inference rules for ATCCS as implemented in PROLOG are given in the following listing (see Section 9.2). Note that in these rules, the fourth parameter represents the atomicity of the derivations.

```
%
%  atccs  operators
%
infer(exit,exit,stop,int).
infer(seq(i,B), i, B, int).
infer(sseq(i, B) , i , B , at).
infer(seq(A,B), A, B, int):-
     isaction(A).
infer(sseq(A, B) , A , B , at):-
     isaction(A).
infer(choice(B1,B2),A,B11,Any):-
    infer(B1,A,B11,Any).
infer(choice(B1,B2),A,B21,Any):-
    infer(B2,A,B21,Any).
infer(comp(B1, B2), A , comp(B11 , B2),int):-
    infer(B1,A,B11,int).
infer(comp(B1, B2), A , comp(B1, B21),int):-
    infer(B2,A,B21,int).
infer(comp(B1,B2), A , lcomp(B11 ,B2),at):-
    infer(B1,A,B11,at).
infer(comp(B1, B2), A ,rcomp(B1, B21),at):-
    infer(B2,A,B21,at).
infer(comp(B1,B2), i , comp(B11,B21), int):-
    infer(B1, A1, B11, int),
    infer(B2, A2, B21, int),
    matches(A1,A2).
infer(comp(B1,B2), i , comm(B11,B21), at):-
    infer(B1, A1, B11, at),
    infer(B2, A2, B21, at),
    matches(A1,A2).
infer(comm(B1,B2),i  , comp(B11,B21),int):-
    infer(B1, A1, B11, int),
    infer(B2, A2, B21, int),
    matches(A1,A2).
infer(comm(B1,B2),i , comm(B11,B21),at):-
    infer(B1, A1, B11, at),
    infer(B2, A2, B21, at),
    matches(A1,A2).
infer(comm(B1,B2),i , comm(B1,B21),at):-
    infer(B2, i, B21, at).
infer(comm(B1,B2),i , comm(B11,B2),at):-
    infer(B1, i, B11, at).
infer(lcomp(B1,B2), A , comp(B11,B2),int):-
    infer(B1,A,B11,int).
infer(lcomp(B1,B2), A , lcomp(B11,B2),at):-
    infer(B1,A,B11,at).
```

```
infer(rcomp(B1,B2), A , comp(B1, B21),int):-
    infer(B2,A,B21,int).
infer(rcomp(B1,B2), A , rcomp(B1,B21),at):-
    infer(B2,A,B21,at).
infer(enabl(B1,B2),A,enabl(B11,B2),Any):-
    infer(B1,A,B11,Any)
   , A \== exit.
infer(enabl(B1,B2),A,B21,Any):-
     infer(B1,exit,B11,_),
     infer(B1,A,B21,Any).
infer(hide(L,B),A,hide(L,B11),Any):-
     infer(B,A,B11,Any),
     not(member(A,L)).
infer(call(P),A,B11,Any):-
    proc(P,B),
    infer(B,A,B11,Any).

%
% manual  recovery  operators
%

infer(ckp(A ,B) , action , ckp(A1 ,B), Any) :-
      infer(A , Action , A1, Any),
      label(Action,L),
      not(member(L,[rlbck,commit ])).
infer(ckp(A ,B) , commit , ckpl(A ) , Any).
infer(ckp(A ,B) , rlbk , ckpl(B ) ).
infer(ckpl(A),Action,ckp(A1,B),Any):-
       infer(A , Action , A1 , Any).
infer(mckp(A ,B) , Action , mckp(A1 ,B), Any) :-
      infer(A , Action , A1, Any),
      label(Action,L),
      not(member(L,[rlbck,commit] )).
infer(mckp(A ,B) , commit , start(A ,mckp (A ,B)), int).
infer(mckp(A ,B) , rlbck, B,  int).
infer(start(A,B),Action,mkcp(A1,B),Any):-
      infer(A,Action,A1,Any),
      label(Action,L),
      not(member(L,[rlbck,commit] )).

%
%  automatic  recovery  operators
%

infer(theta(B1),i,recv(B2,B2,B),at):-          /* operator @ */
      infer(B1,i,B2,at),
      not( infer(B1,dead,B2,Any)).
infer(theta(B1),i,theta(B2),int):-
      infer(B1,i,B2,int).
infer(theta(B1),A,theta(B2),any):-
      infer(B1,A,B2,any),
        isaction(A).
infer(recv(C,B,R),i,recv(C1,B,R),at):-
     infer(C,i,C1,at).
infer(recv(C,B,R),i,theta(C1),int):-
     infer(C,i,C1,int).
infer(recv(C,B,R),rlbck,theta(choice(R,seq(dead,B))),at):-
     not(infer(C,i,C1,Any)).
```

SYMBOLS AND NOTATIONS INDEX

Below are the notations used in this thesis for important entities and constructors together with (in the middle column) their meanings and (in the right column) the number of the page in which each notation is defined or first appears:

Operators

Derivations

B =*s=> B'	B all- accepts sequence s with B' as a result	142
B =<s>=> B'	B atomic -accepts sequence s with B' as a result	142
B =s=> B'	B interrupt -accepts sequence s with B' as a result	142

Actions

g$\alpha_1\alpha_2$...	Action denotation: α_1 ... are input/output events	59
!v	Output event: output value expression v	60
?X:t	Input event: input for X of type t	60
commit	Commit to a checkpoint	120
dead	Mark a dead branch	130
exit(e_1,..... e_n)	Successful termination	74
i	Internal event	48
rlbck	Roll back to a previous checkpoint	120

Substitutions

[a_1/a'_1,.....a_n/a'_n]	Substitute gate name a'_i by a_i for $1 \le i \le n$	49
[e_1/X_1,.....e_m/X_m]	Substitute variable X_i by expression e_i for $1 \le i \le m$	49

Behavior expressions

a; B	Sequencing: a then B	59
a*B	Strong sequencing: a then B	59
A+B	A or B	60
A [> B	A disabled by B	40
A \ L	A where all elements of L are hidden	60
A | B	Parallel composition of A and B (in CCS)	40
A |L B	Left parallel composition of A and B	62
A |R B	Right parallel composition of A and B	62
[G] -> B	B if condition G is true	61
A [S]	Relabelling of A using substitution S	61
A || B	Parallel composition of A and B	61

A <> B	A communicates with B	63
$b(e_1,...,e_n)$	Call process b with the value parameters e_1, ...	64
A <-> B	A with B as a unique checkpoint	121
^A	Start A with A as a unique checkpoint	121
A <<->> B	A with B as a set of checkpoints	125
A ^^A	Start A with A as a set of checkpoints	126
@A	Fault tolerance in A	130
<A,B,C>	Execute A with checkpoint B, avoid C	131
*B	Invisible B	88
[p]	Use process p as an atomic action	92
$\sum B_i$	Summation (or choices) over an indexing set	42

Sets

Act	Set of observable actions	39
Act $\cup$ { exit, i }	Set of actions	58

Functions

atomic	Tests if an atomic declaration is consistent	76
f_exit	Tests if *exit* appears as a first action in an expression	75
label(a)	Label of action a	49
t(B)	Behavior tree of expression B	95
S_syn	Synchronization function	89

Relations

$R \cup S$	Union of relations R and S	144
R^{-1}	Converse of relation R	144
R.S	Product of relations R and S	144

Equivalences

$\asymp$	Atomic weak observation equivalence	158
$\blacksquare$	Strong observation equivalence	145
$-$	Weak observation equivalence	154
$-_C$	Observation congruence	161

Special notations

any	To denote any type of derivation	58
at	To denote an atomic derivation	58
int	To denote an interruptible derivation	58
C[.]	Context for behavior expressions	159
Dom(t)	Domain of type t	49
atomicproc	To define an atomic process	76

INDEX

A

B

C

D

E

F

G

S

T

U

V

W