Guided Search Technique for LOTOS

By

Mazen Haj-Hussein

Thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Ph.D. in Computer Science

under the auspices of the

Ottawa-Carleton Institute for Computer Science

University of Ottawa

Ottawa, Ontario, Canada

December, 1995

ACKNOWLEDGEMENTS

Completing my Ph.D. is the greatest achievement that I have ever accomplished. However, this would not have been possible without the help of many people that I am so grateful to.

First, I would like express my deepest thanks to my supervisor, Dr. Luigi Logrippo, for his patience and for all the guidance and the advice that he has given me throughout my Masters and Ph.D. studies.

Second, I would like to acknowledge the LOTOS group at the University of Ottawa for their dedication that helped me to develop my ideas in a very knowledgable and friendly environment. In particular, I would like to thank Jacques Sincennes and Antoine Bonavita for their assistance in the implementation and the validation of the techniques discussed in this thesis. Their comments and advice are much appreciated. I am also grateful to Hans van der Schoot for the input that he has given me in the early stages of the thesis.

I would like to express my gratitude to the Natural Sciences and Engineering Research Council and the University of Ottawa for their financial support during my graduate studies.

I would also like to thank my parents for their support. A special thanks to my father, Dr. Mohammed Haj-Hussein, who has dedicated his life to science and creativity. He believed that computer science is the science of the future. I thank him for his advice.

Last, but far from least, I would like to thank my family, my loving wife Allyson and my two children Leyla and Malek. Without Allyson’s patience and support during my long years of study, this would not have been possible.

REMERCIEMENTS

Compléter mon doctorat est le plus grand défi que j'ai relevé. Toutefois, cela aurait été impossible sans l'aide de plusieurs personnes envers lesquelles je suis très reconnaissant.

Je voudrais d'abord spécialement remercier mon directeur de thèse, Professeur Luigi Logrippo, pour sa patience, l'aide et les conseils qu'il m'a prodigués tout au long de ma maîtrise et mes études au doctorat.

Je suis aussi très reconnaissant du dévouement manifesté par le groupe LOTOS de l'Université d'Ottawa quant au développement de mes idées, et cela dans un environnement toujours bien documenté et amical. J'aimerais remercier en particulier Jacques Sincennes et Antoine Bonavita pour leur aide quant à la réalisation et à la validation des techniques discutées dans cette thèse. Leurs commentaires et leurs conseils ont été très appréciés. Les discussions avec Hans van der Schoot au tout début ont aussi contribué à l'élaboration de ma thèse.

J'aimerais exprimer mes sincères remerciements au Conseil de recherches en sciences naturelles et en génie du Canada et à l'Université d'Ottawa pour leur support financier tout au long de mes études du 2e et 3e cycle.

J'aimerais aussi remercier mes parents pour leur appui. Tout spécialement mon père, le docteur Mohammed Haj-Hussein, qui a consacré sa vie à la science et à la créativité et pour qui l'avenir se trouve dans les sciences de l'informatique. Je le remercie pour ses conseils.

Enfin, j'aimerais remercier ma famille, mon épouse Allyson et mes enfants Leyla et Malek. Sans la patience et l'appui d'Allyson durant mes années d'études, ce projet n'aurait pu se réaliser.

Contents

Abstract

The dynamic behaviour of a LOTOS specification can be described as a tree, called *behaviour tree*, where the nodes represent the states of the behaviour, and the branches represent the possible next actions. Unfortunately, the behaviour tree for a realistic size LOTOS specification can be very large and often has no finite representation.This is the major limitation for the existing LOTOS verification techniques.

The main goal of this thesis is to provide a new behaviour tree exploration technique, called *Goal-Oriented Execution*, that can be used to check properties of LOTOS specifications by narrowing exploration to a meaningfully selected subset of the tree. In this execution technique, the system derives *traces* (i.e paths in the behaviour tree) satisfying certain assertions that express temporal ordering of actions and data values properties.

Goal-Oriented Execution is a combination of three techniques. The first technique is an automatically generated ADT *evaluator/narrower* engine. It is capable of evaluating an expression based on a rewriting rule approach, borrowed from functional programming, and deriving solutions to a set of constraints using a narrowing technique, borrowed from logic programming. The second technique is a *static analyzer* that determines where the given assertions are likely to hold, producing static information called *static derivation paths*. The third technique, called *guided-inference system*, involves a new type of inference rules that derive traces using static derivation paths to resolve most non-determinism.

Implementation issues of this technique are also discussed, and examples of its usage are provided. The technique is now included in ELUDO, the University of Ottawa LOTOS interpreter.

Chapter 1 Introduction

1.1 Background and Motivation

A Communication protocol is defined as a set of syntactic and semantic rules that govern the exchange of information between entities in a communication system [95]. In today's communication systems, protocols are quite complex. Experience shows that many protocol design errors are detected late in the development cycle and may cause disastrous outcomes [91]. Therefore, protocol specification models must be suitable for adequate validation in the early phases of the development cycle. This generated the need to define and standardize formal methods for describing protocols. A formal method, with precisely defined semantics, provides an excellent basis for avoiding ambiguity in the interpretation of the protocols's characteristics, as well as a mathematical framework for formal proof methods and automated analysis methods.

The International Standardization Organization (ISO) has developed a model, called the Open System Interconnection Reference Model (OSI/RM) [33], that deals with connecting *open systems* (systems that are open for communication with other systems). ISO has adopted Formal Description Techniques (FDTs) to define protocols and services for OSI/RM. The following, mainly taken from [24], are the main objectives for these FDTs:

- *Expressiveness*: an FDT should be capable of describing both the protocols and the services of the OSI model.
- *Formalism:* in order to perform formal analysis, an FDT should be founded on a strong mathematical model that makes it possible to extract the meaning of a specification unambiguously.
- *Modularity*: an FDT should offer the ability to decompose large and complex protocols into readable and maintainable components.
- *Abstraction*: an FDT should also offer the ability to suppress irrelevant implementation details

from the specification. For example, OSI concepts (e.g. service access points, connection endpoints, service primitives, protocol data units and constraints) should be expressible in a completely implementation-independent manner.
- *Executability*: to allow validation to start at an early stage of the protocol development life cycle, it should be possible to construct a running prototype model on the basis of a formal description.

FDTs should be effectively integrated with design methods. They should have the ability to support not only the specification phase, but the complete protocol development process:

- Specification of requirements as a formal abstract model.
- Verification of the model against the requirements.
- Derivation of implementations from the model using stepwise refinement transformations.
- Generation of test cases from the model.
- Testing an implementation against the model.

The model can also serve as documentation for the system. Vissers in [143] provides an excellent perspective of FDTs.

The application of an FDT to the development of large and complex protocols depends heavily on the availability of supporting computer tools. More and better tools will popularize the use of FDTs. Several classifications of FDT tools have been proposed [15][99][131][132]. Our classification is as follow:

1- *Specification tools*: such as editors and syntax/static semantic checkers.
2- *Validation and Verification tools*: assist in checking the syntactic and semantic protocol properties. These include: theorem provers, model checkers, reachability analysers, simulators, and symbolic executors.
3- *Reduction tools*: such as algebraic simplifiers that reduce expressions into a simpler form.
4- *Transformation tools*: these tools are conceived for translating the specification into a more concrete (refined) one, or into a different equivalent representation such as graphs, Petri-nets, etc.. .
5- *Comparison tools*: compare two specifications with respect to given relations such as equivalence and congruence relations.
6- *Testing tools*: such as test cases generators.

Extensive research is being done worldwide in the area of automated protocol validation and verification tools based on FDTs. A survey on such tools can be found in [99].

The FDTs standardized within ISO are ESTELLE (Extended Finite State Machine Language, [83]) and LOTOS (Language Of Temporal Ordering Specification, [84]). In addition, CCITT (the International Telegraph and Telephone Consultive Committee) has adopted SDL (Specification and Description Language, [7][126]) as standard FDT. A comparative evaluation of these specification languages can be found in [16][131]. A *graphical* version of LOTOS is in advanced stage of standardization within ISO and CCITT.

LOTOS is one of the most precisely defined languages in use today. Its static semantics are defined by an attributed grammar, while its dynamic semantics are based on algebraic concepts. LOTOS is made up of two components: a data type component, which is based on the algebraic specification language ACT ONE [40][41], and a control component, which is based on concepts from Milner's CCS [103] and Hoare's CSP [71]. LOTOS was conceived for the specification of the services and protocols of the Open Systems Interconnection [81][82][128][129]. Soon after its introduction, other uses were found for LOTOS: among others, the specification of telephone systems [43] and the specification of distributed algorithms [67].

The dynamic semantics of LOTOS are defined in terms of axioms and inference rules [84]. The actions (transitions) that a given behaviour expression may perform, and the dynamic behaviour of the resulting states, can be derived systematically by applying the inference rules. Therefore, the dynamic behaviour of a LOTOS specification can be seen as a tree, called *behaviour tree*, where the nodes of the tree represent the states of the behaviour, and the branches represent the possible next actions. Unfortunately, the behaviour tree for a realistic size LOTOS specification can be very large and is often has no finite representation.This is the major limitation for the existing LOTOS verification techniques.

To deal with this limitation, LOTOS interpreters in existence today make it possible to execute specifications (i.e. exploring behaviour trees) using different techniques such as step-by-step execution [63][66][98][138], random-walk execution [67], weighted execution [108], fair execution [154], and symbolic execution [1][115]. These techniques lay restrictions such as: (1) user intervention is needed, (2) only a subset of LOTOS constructs can be handled, (3) LOTOS specification must be written in specific style, (4) counters and limits are needed, or (5) information must be included as special comments in the LOTOS specification. These techniques and other techniques that require changing the syntax and semantics of LOTOS are described in more detail in the next chapter.

The main goal of this research is to provide a new behaviour tree exploration technique, called *Goal-Oriented Execution*, that can be used to verify LOTOS specifications by narrowing exploration to a meaningfully selected subset of the tree. In this execution technique, the system derives *traces* (i.e paths in the behaviour tree) satisfying certain assertions. First, the LOTOS specification is analyzed *statically* determining where these assertions are likely to hold, producing static information called *static derivation paths*. Second, these static derivation paths are fed to the inference rules helping directing the dynamic derivation. To comply with LOTOS semantics, the dynamic trace derivation may require sub-traces with new assertions. Therefore static derivation paths are obtained *on demand* by the inference rules, as we shall see.

For this technique to be applied on full LOTOS, where data part is involved, the assistance of a *narrower* tool is needed. Narrowing [118] is a technique for finding solutions to a set of constraints in abstract data types. We present an algorithm for transforming abstract data type equations into a rewriting rules evaluator and a narrower engine with considerable performance efficiency. We also show that other existing tools can be improved using the narrower.

1.2 Structure of the Thesis

This paper is organized as follows. In chapter 2, we give an overview of the Communicating Finite State Machines model and of the relief strategies for state explosion problems. We also provide a brief introduction to LOTOS, to the existing verification tools for LOTOS, and to the existing relief strategies that have been devised to cope with dynamic state space explosion for LOTOS. Chapter 3 provides an overview of our method. This includes the functionality and the strategy of each component of our system and the algorithm of the overall system. In Chapter 4, the definition of the static analyser and of the guided-inference system is presented. The narrower technique is explained in detail in chapter 5. In chapter 6, the application of our method to verify an alternating bit protocol specification is shown. Also, the scope of application of the method is described in this chapter. Finally, the thesis conclusion is given in Chapter 7, along with remarks on possible future work. Appendix A includes the LOTOS specification of the alternating bit protocol used as an example in chapter 6.

Chapter 2 Literature Review

In the first part of this chapter, we give a brief introduction to a layered network architecture model, an overview of protocol verification techniques, and we illustrate some relief strategies that have been devised to cope with the state explosion problems of communicating finite state machines models. In the second part, we present an introduction to LOTOS and we list some of its existing verification tools.

2.1 Layered Network Architecture Model

A layered network architecture model (e.g. OSI/RM, [33]) is defined as a composition of *n* layers, each built upon its predecessor. The purpose of each layer is to offer certain services to the higher layer. Layer N on one system and layer N on another system are called *peer processes*. Peer processes, at layer N, communicate using so called *layer N protocols*.

In reality, the communication between peer processes is not done directly. Instead, each layer passes data and control information to the layer immediately below it, until the physical layer is reached where the actual communication occurs. The receiving physical layer, then, passes the information to the layer above it until the desired layer is reached. In summary, layer N defines services to layer N+1, using the services provided by layer N-1. In this case, layer N is called the *service provider* for layer N+1.

Services are provided at service access points (SAPs), which are identified by unique addresses. The SAPs of layer N are the places where layer N+1 can have access to the services provided. Figure 2-1 shows the composition of layer N with layers N-1 and N+1.

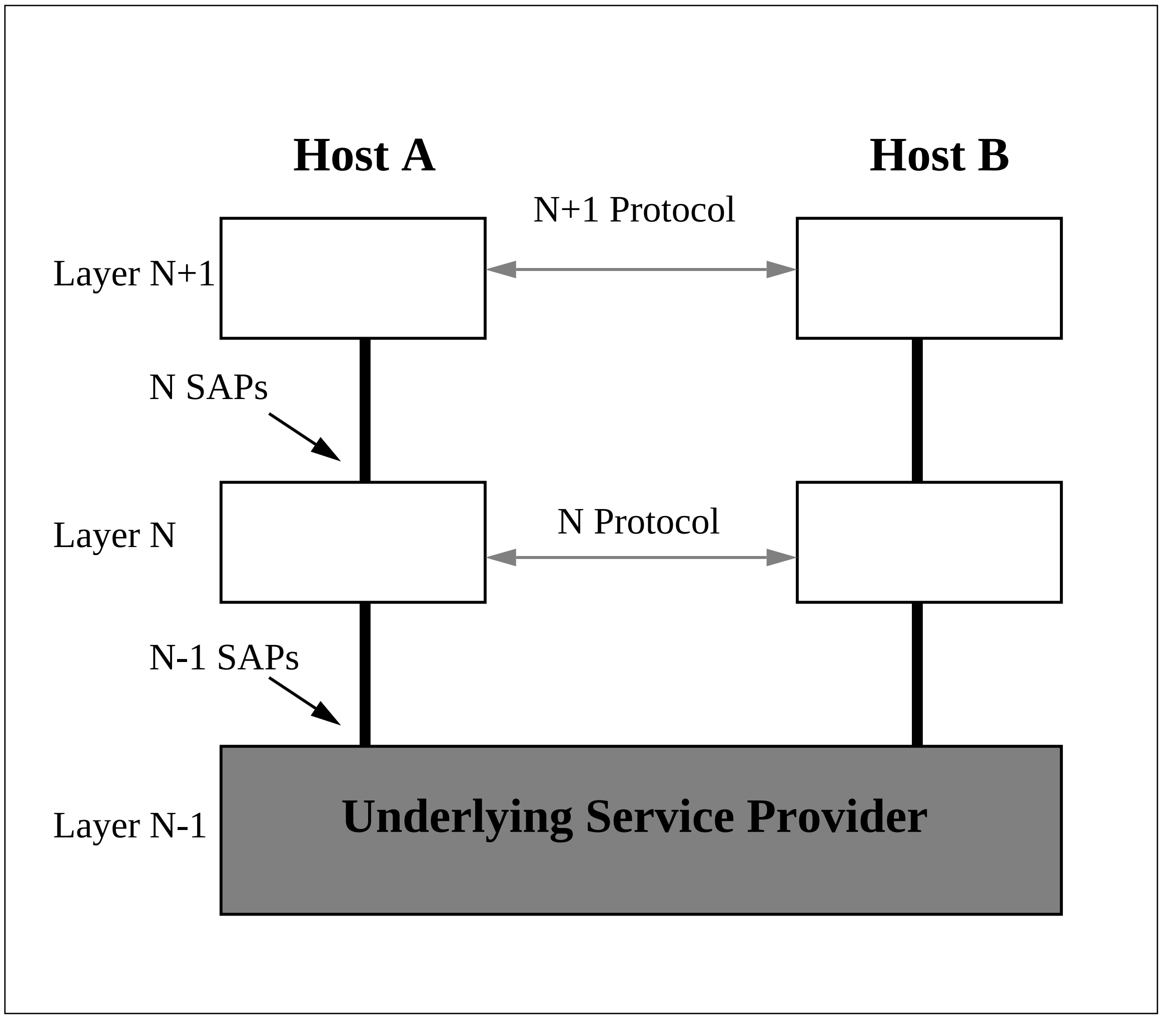


Figure 2-1 A composition of three layers in a Layered model

The user of services is concerned with *what* services are being provided and *where* they are provided, and not *how* they are provided. For this reason, a protocol that defines some specific services can be viewed as a black box whose services are fully described by sequences of messages from and to the users. Such a description constitutes the *service specification* of the protocol, while the description of the exchange of messages among peer processes constitutes the *protocol specification*. The latter specification at a given layer, is the one that should be hidden from the other layers.

2.2 Protocol Verification

In a layered network architecture model, the protocol verification process involves checking for the following properties:

- o **syntactic properties**: These are general design properties of a given protocol such as the absence of the following errors [156]: *state deadlock, unspecified receptions, non-executable interactions, state ambiguity, channel overflow, tempo blocking,* and *unfairness*. The verification of syntactic properties, often called *protocol validation*, does not require knowledge of the provided services.

- o **semantic properties**: These are the intended sets of services that a given protocol needs to provide to the protocol of the layer above. The verification of such properties requires the service specification to be provided, and it is necessary to assume the correctness of the service provided by the layer below. Such properties cannot be classified or generalized since they depend on specific protocol or service specifications. Such verification has proved difficult to automate.

The correctness of syntactic properties does not imply that the semantic properties hold, but the failure of these may prevent the protocol from providing its specified service.Therefore, it is logical to validate a protocol before verifying it.

2.3 Transition-Based Models

The transition-based models are mainly used to describe the control aspects of protocols. For example, the message exchange between entities to establish connection and termination can be best specified by these models. On the other hand, the specification of the data transfer aspects of the protocol can be very complex and often impossible to describe by transition-based models.

These models can be classified into Communicating Finite State Machines models, and Petri-Net models. The latter are not of concern in this paper.

The Communicating Finite-State Machines Model is one of the earliest and simplest methods used for formal verification. Still now, it is the most widely used.

In this model, each communicating process is represented by a FSM. A CFSM can be formally represented by a quadruple $\langle S, E, T, S0 \rangle$, where

- S is a finite set of states
- E is a finite set of events
- T is a function representing the set of transitions T: $S \times E \rightarrow S$
- $S0 \in S$ is the initial state

The coupling between a pair of processes is done by using two implicit FIFO queues connecting the inputs of one process to the outputs of the other process and vice versa.

A transition is either a message transmission (identified by a minus sign (-)) or a reception of a message (identified by a plus sign (+)). Figure 2-2 illustrates a protocol with two processes P1 and P2. In the figure, initially, both processes are in their initial states, namely state 0.

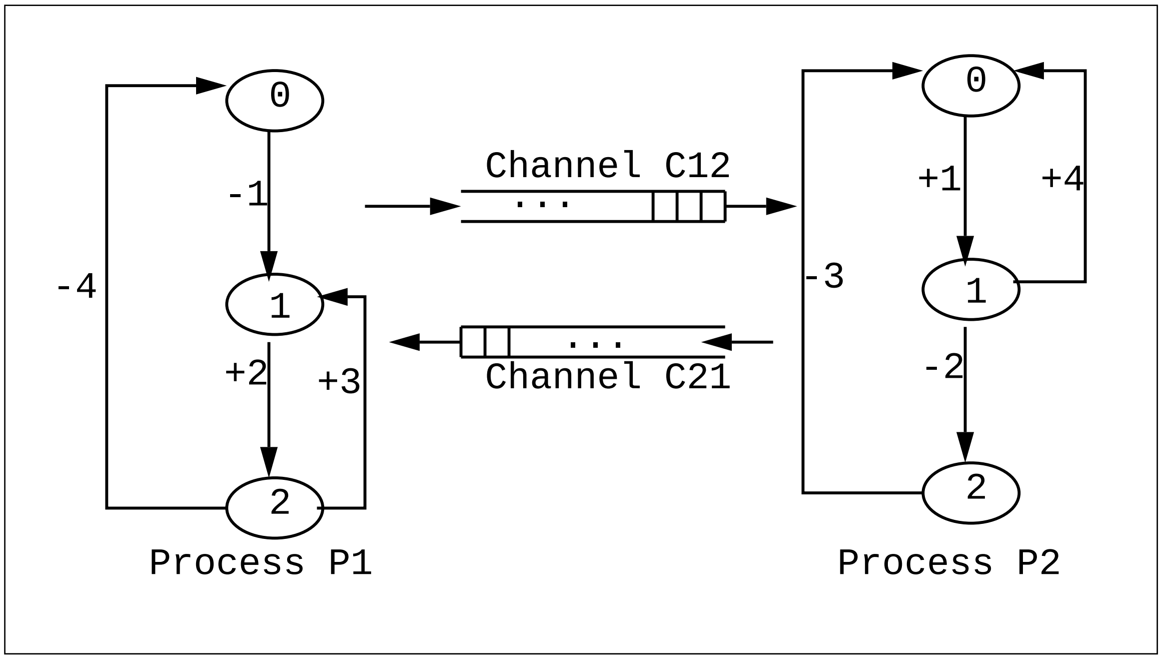


Figure 2-2 A Protocol Specification in CFSMs

The Extended Finite State Machines model [13] was introduced to simplify the representation of the data flow. In this model, the number of states in an FSM can be reduced by the use of variables. (e.g. a message sequence number can be represented by a single variable).

2.3.1 Relief Strategies for the State Space Explosion

Reachability analysis was first introduced by Sunshine in his Ph.D. thesis in 1975 [133] and then developed further by Bochmann [13] and automated by West [125].

The idea behind this approach is to analyse all global states of the protocol which are reachable from the initial state.The global states generated construct what is called a *reachability tree* with the initial global state as the root. The tree is guaranteed to be finite if all channels are bounded.

Reachability analysis has been proven to be one of the most effective methods for the verification of communication protocols based on transition-based models. Both syntactic properties and semantics properties can be verified using reachability analysis. However, the applicability of this method is severely restricted by the so-called *state space explosion* problem. That is, the reachability tree grows very rapidly with the complexity of the protocol, and in many cases it can be unbounded. Therefore, it is often impractical to generate and analyse all reachable global states.

Many researchers have proposed various strategies to overcome this problem. A survey of some existing relief strategies can be found in [96]. In the following we give a brief description of such strategies. It must be observed that none of these strategies resolves completely the problem.

Imposing Limitations and Restrictions

This relief strategy was proposed by West [148] and includes the following techniques:

- Limiting the capacity of the communicating channels.
- Limiting the classes of design errors under consideration.
- Restricting the use of many-valued parameters in the specification such as sequence numbers.

Decomposition/Partition the Protocols

The idea behind such techniques is to decompose/partition protocols into components, which then can be validated separately [30][31][144]. This decreases the complexity of the protocol under validation since the number of states in a protocol component is always smaller than the number of states in the original protocol.

Projections

This strategy was proposed by Lam [92]. Instead of partitioning a protocols into components, it constructs from the given protocol an image protocol for each of the functions that is intended to be verified. The resulting protocol therefore is smaller than the original protocol, implying that the complexity of the problem is reduced.

Transition Choice Rule

The aim of this technique is to control state exploration [12]. This is done by associating a choice rule to each transition. Such a rule is a boolean condition whose value decides whether or not the transition is to be executed during reachability analysis. For example, a rule may specify that no transition can be executed more that once. For instance, appropriate choice rules can eliminate infinite loops that may occur in the analysis.

Simulation

The simulation technique is used to control state exploration by selecting only one transition to fire out of a global state[3]. The transition selection can be done by random choice or by assigning priorities to each of the transitions, where the transition with the highest priority is always the one chosen.

Fair Progress State Exploration

Many strategies have been founded on the fair progress state exploration. This technique was first proposed by Rubin and West [124], then extended by other researchers [59][157]. The idea is to explore only those global state that are reachable when the two protocol entities proceed at the same speed. The limitation of this technique is that it can only be applied on two-process protocols.

Maximal Progress State Exploration

This technique [62] is similar to the fair progress state exploration and is also limited to two-process protocols. The analysis in this case is done in two phases, during each of which a different process is forced to proceed at its maximal speed whenever possible. The advantage of this technique with respect to the previous technique is that channel overflows can be detected.

Simultaneous Execution

This relief strategy is proposed by Itoh and Ichikawa [85]. It is limited to protocols defined by FSMs where all cycles pass through the initial state. In each global state, all admissible transitions of different processes are executed simultaneously to derive the next global state. Moreover, if there are any potentially admissible transitions in the current state of a process P, then the technique forces process P to wait in order that any of its admissible transitions may become executable later. This technique explores a part of the global space. The interaction sequences explored in this technique are called *reduced implementation sequences* and are used to verify the protocol against the given requirement specification.

Tree Protocol Validation

The tree (or acyclic form) protocol validation strategy [23][86] does not explore the global states of a protocol, instead, it grows each process of the protocol into a tree or an acyclic form. During the growing process, protocol design errors such as deadlocks, unspecified receptions, and channel overflows can be detected.

Scatter Search

Holzmann designed a tool called Trace [73][72] that uses a search strategy called scatter search to explore the global space. The search, which is basically a depth first search, is guided by heuristics and is restricted by the depth limit. Examples of the heuristics used are:

- Restrict non-determinism to what is due to local behaviours and remove non-determinism due to concurrency.
- Assign priorities among concurrent events. For example, internal events may have higher priority than observable events.
- Limit the capacity of the communication channels.
- Minimize the FSM models of the protocol processes before verification.

An improved tool, called Supertrace, that uses a better memory management, is described in [75].

Random Walk

West observed from his experience in validating the OSI session layer protocol, that exhaustive validation is redundant, in the sense that the majority of errors detected are found many times in different global states. He concluded that an analysis of a subset of the reachable global state may be sufficient to identify a significant fraction of errors. West then proposed the random walk validation strategy [149] to partially explore the global space.

This strategy can thus be viewed as a modified form of reachability analysis, in which only one transition from the current global state (chosen at random) is executed instead of systematically executing all the transitions in turn. The state exploration is stretched out continuously along a single path without backtracking. As a result, a random walk through a global space does not require a database to prevent multiple traversals of the same state. That is to say that an already explored global state may be explored again. The major disadvantage of such a technique is that it can be used to find errors and not to demonstrate freedom from errors. A systematic reachability analysis is therefore preferable when it is possible.

Generation of Finite Graphs

Vuong et al. [145] have demonstrated how the global states of all non-FIFO protocols and of a certain class of FIFO protocols can be represented as finite graphs, even if these protocols may produce an unbounded number of messages in the transmission media. This approach solves a class of problems which the conventional reachability analysis fails to deal with, due to the infinity of the reachable global states induced by unbounded accumulation of messages in the media.

PROVAT Strategy

The PROVAT strategy [96] uses a heuristic search technique called *error first search* similar to the *best first search* developed in the AI field. Heuristics can be applied at the three points in a search process, namely, the points to decide which global states to expand next, to decide which transition to fire next, and to decide which global states to discard. Heuristics depend on the syntactic design error that needs to be detected.

Symbolic Execution

Symbolic execution techniques [22] are another form of state exploration. The objective of such techniques is to compress the reachability tree, when variables are involved, by constructing a so called *proof tree*. A node in the proof tree represents a large number of nodes in the reachability tree (i.e. a class of global states). The root of the tree represents the initial global state and the leaves of the tree represent all possible final states.

Each state is analyzed to determine whether or not some syntactic properties hold. In addition, the specifier can add his own assertions to express other desired properties such as liveness and timing.

2.4 The Formal Description Technique LOTOS

We recall that LOTOS has two components: the data type component based on algebraic Abstract Data Types (ADT) specification, as in ACT ONE [40][41], and the control component based on Milner’s CCS [103] and Hoare’s CSP [71].

According to [97], the main characteristics of LOTOS are:

1- *Formal definition*: Formally defined syntax, static semantics, and dynamic semantics. In particular, the static semantics are defined by an attributed grammar [84], and the dynamic semantics are described operationally in terms of inference rules [18].

2- *Process algebra*: Following Milner’s ideas, the operational semantics are defined in such a way that it is possible to prove a rich set of algebraic equivalence properties, based on several types of equivalence relations. These properties can be used in order to prove equivalence or correctness of specifications, as well as to transform the structure of a specification.

3- *Interleave concurrency*: Events are considered to be atomic, and thus the parallel execution of two events *a* and *b* is defined as a situation of choice, where *a* can occur before *b*, or vice versa. Therefore, any LOTOS behaviour expression can be rewritten as an expression consisting of a choice between behaviour expressions, each prefixed by a single action (i.e. expansion theorem [103]).

4- *Multiway synchronization*: This concept is borrowed from Hoare’s CSP [71]. Interprocess communication occurs by means of a *rendez-vous* mechanism, called synchronization or interaction. A synchronization will occur on a synchronization point, called *gate*, only if all processes that are committed to synchronize on that gate participate with a matching event.

5- *Nondeterministic synchronization*: Often more than one synchronization is possible. One only will be executed according to a nondeterministic choice.

6- *Executability*: Because LOTOS semantics are defined operationally, it is possible to implement these semantics in an interpreter. Although not every LOTOS specification if finitely executable, those which are provide a *fast prototype* of the entity specified.

7- *Modularity and hiding*: These concepts allow modular system descriptions with different levels of abstraction suitable for stepwise decomposition of processes. By using parameterization, these processes becomes reusable.

Different types of relations among LOTOS specifications are available and provide a framework for determining semantic equivalences between different levels of refinement. Among others, observational equivalence [103], testing equivalence [107], and the implementation relation [25] are of most interest. The testing equivalence represents the black box approach. Two systems are said to be *testing equivalent* if they present the same behaviour to the observer. The implementation relation defines how implementations can be derived from a given specification. Observational equivalence is a stronger relation than the other two.

We do not intend to provide a complete tutorial on the language LOTOS in this thesis. At least two tutorials have been published in journals [18][97], and several other tutorials have enjoyed some degree of distribution. A tutorial is also included in [84].

2.4.1 Data Type Component

Abstract Data Types (ADTs) are used to specify the intended effect of concrete data types by defining their properties as a set of data objects with their manipulating operations. Because of their formal base, the ADT specifications can serve as abstract, correct, and unambiguous references for the implementation.

LOTOS, as an abstract specification language, uses an ADT based on ACT ONE formalism to define its data types [40].

An ADT in ACT ONE consists of a *signature* and a set of *equations*. The signature gives all the syntactic knowledge of a type. It consists of:

a- *Sorts*: names of data carriers. A sort corresponds to the concept of type in most programming languages (e.g. Pascal).

b- *Operations*: functions where each has a domain and a range

$$op: s_1,s_2,...,s_n \rightarrow s_r$$

where *op* is the operator name, $s_1,s_2,...,s_n$ are the sorts of the operation's arguments, and s_r is the sort of its result. An infix operation can be declared as

$$\_op\_ : s_1,s_2 \rightarrow s_r$$

The equations define the semantics of the operators. They have one of the following forms:

a- $l_i = r_i$. An unconditional equation.

b- c => $l = r$. A conditional equation, where *c* has the form $l_1 = r_1,...,l_n = r_n$

In Figure 2-3 we provide a definition for type *nat_bool* that contains two sorts, *nat* and *bool* for natural numbers and booleans respectively and the declaration of some operators and their equations. The natural numbers are represented by using the successor operators *succ* and *0*, where the natural number *n* is represented by $succ^n(0)$, a short-hand for succ repeated *n* times followed by the argument *0*.

```
type nat_bool is
    sorts nat, bool
    opns
          true      :   -> bool
          false     :   -> bool
          0         :   -> nat
          succ      :   nat -> nat
          _or_      :   bool, bool -> bool
          _<_       :   nat, nat -> bool
          _==_      :   nat, nat -> bool
          _>=_      :   nat, nat -> bool
          _mod_     :   nat, nat -> nat;

    eqns
        forall C:bool, M,N:nat
          ofsort bool
               false or C                   = C;
               true or C                    = true;

               succ(M)    < succ(N)         = M < N;
               0          < succ(M)         = true;
               M          < 0               = false;

               succ(M)    == succ(N)        = M == N;
               0          == 0              = true;
               0          == succ(M)        = false;

               M          >= N              = (N < M) or (M==N);
          ofsort nat
               (M >= N) => M mod N = (M - N) mod N;
               (M < N) => M mod N = M;
    endtype;
```

Figure 2-3 An Abstract Data Type

ACT ONE has the following features:

1- Modularization of specifications: This allows the reference to already existing specifications in a library.

2- Combination of Specifications: Related operators and equations can be combined. This concept is based on the fact that complex specifications can be split into smaller parts or vice versa, while simple specifications may be combined (stepwise) to produce a large one.
3- Renaming of Specifications: Sorts and operations can be renamed without changing their semantics.
4- Parameterization and Actualization of specifications: This is the concept of genericity. For example, a general definition of a *Queue* can be specified using the element's sort as a parameter. A specific *Queue* definition (e.g. queue of integers, characters, etc.) can be obtained by suitable actualization of the parameter.

Since ACT ONE allows nonconstructive specifications, the execution may end up into infinite loops or deadlocks. Detecting and/or repairing such specifications may be impossible [44].

2.4.2 Control Component

A Specification in LOTOS can be seen as a process that possibly consists of interacting subprocesses. Each subprocess may in turn consist of other subprocesses. Each process can be imagined as a black box that is capable of synchronizing with other processes (its environment) via common synchronization points called *gates*, or it can perform internal, unobservable actions denoted by **i**. The environment of a process is the other processes, plus an unspecified process which is always ready to interact at any gate. Figure 2-4 shows two LOTOS processes **P** and **Q** synchronizing with each other and the environment at gate *c*.

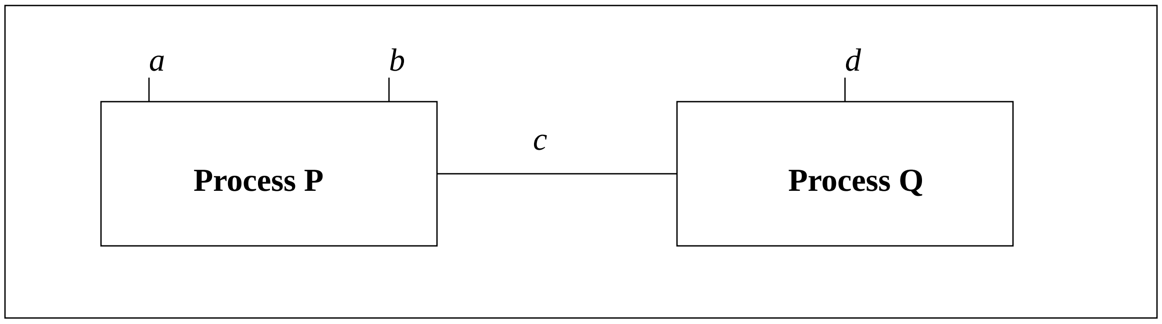


Figure 2-4 Two Synchronizing processes

The basic element of a process behaviour is the *action* which represents a *synchronization* between processes. An action consists of a gate name (interaction point), a list of events, and an optional predicate that restricts the event values to those satisfying the predicate. An event can be either *!E*, denoting the offering of the value *E*, or *?x:s*, denoting that the action is ready to accept any value of sort *s*. For example:

g ?x:int !1 [x > 2]

is an observable LOTOS action which occurs at gate *g* and expects from the environment a value for *x* of sort *int* restricted to be greater than two, while at the same time offering the value one.

As mentioned above, there is another type of action in LOTOS, the internal unobservable action **i**. This action does not interact with the environment.

Interprocess communication in LOTOS occurs when two or more processes, having a *rendez-vous* on a gate, agree on a value (or values) to be established. This is the case of *matching actions*. Table 2.1 shows all possible types of interactions between two processes. When more than two processes are involved, similar rules apply. *eval(E)* indicates the evaluation of the expression *E*.

Process A	**Process B**	**Synchronization Condition**	**Interaction sort**	**Effect**
g!E_1	g!E_2	eval(E_1) = eval(E_2)	value matching	Synchronization
g!E	g?x:t	eval(E) ∈ domain(t)	value passing	after Synchronization: x = eval(E)
g?x:t_1	g?y:t_2	t_1 = t_2	value generation	after Synchronization: x = y = v where v ∈ domain(t_1)

Table 2.1: Types of Interactions

For example, if Process **A** is prepared to accept a natural number 4,5,6,7,8, or 9 at gate *g*, as denoted by the following action:

g?X:Nat [(X >= 4) and (X <= 9)]

and at the same time Process **B** is ready to accept a natural number multiple of 3 at the same gate *g*, as denoted by the action

g?X:Nat [(X mod 3) = 0]

then an interaction can occur at gate *g*, if the environment cooperates by offering a natural number satisfying the above conditions, namely 6 or 9.

There exists a simplified version of the language LOTOS, called Basic LOTOS. It employs a finite alphabet of observable actions identified by only the name of the gate where they are offered. Synchronization of actions can be described without value communication. Basic LOTOS is mainly used in the theoretical discussion of the language.

Table 2.4 lists the most fundamental constructs, also called behaviour expressions, for Basic and Full LOTOS. All these constructs are part of LOTOS syntax except the relabelling which appears only in the dynamic semantics of LOTOS..

Table 2.2: LOTOS Behaviour Expressions

Description	Basic LOTOS	Full LOTOS
Inaction: *B* cannot interact with the environment nor execute internal actions	*B* = **stop**	*B* = **stop**
Observable Action Prefix: *B* interacts with the environment on gate *g* then behaves like *B1*	*B* = *g*; *B1*	*B* = *g* $d_1..d_n$**[***P***];** *B1* where $d_i = !t_i$ or **?**x_i**:**s_i
Internal Action Prefix:*B* executes internally action **i** then behaves like *B1*	*B* = **i;** *B1*	*B* = **i;** *B1*
Successful Termination: *B* interacts with the environment on gate δ, with possible data offering, then behaves like **stop**	*B* = **exit**	*B* = **exit**($E_1,..,E_n$) where E_i is a term or E_i = **any** s_i
Choice: *B* can either behave like *B1* or *B2*	*B* = *B1* **[]** *B2*	*B* = *B1* **[]** *B2*
Disable: *B* behaves like *B1* until successful termination unless disabled by *B2*	*B* = *B1* **[>** *B2*	*B* = *B1* **[>** *B2*
Enable: *B* behaves like *B1* unless it terminate successfully then it behaves like *B2*. Data can be accepted from *B1*.	*B* = *B1* >> *B2*	*B* = *B1* >> **accept** $x_1{:}s_1,.., x_n{:}s_n$ **in** *B2*

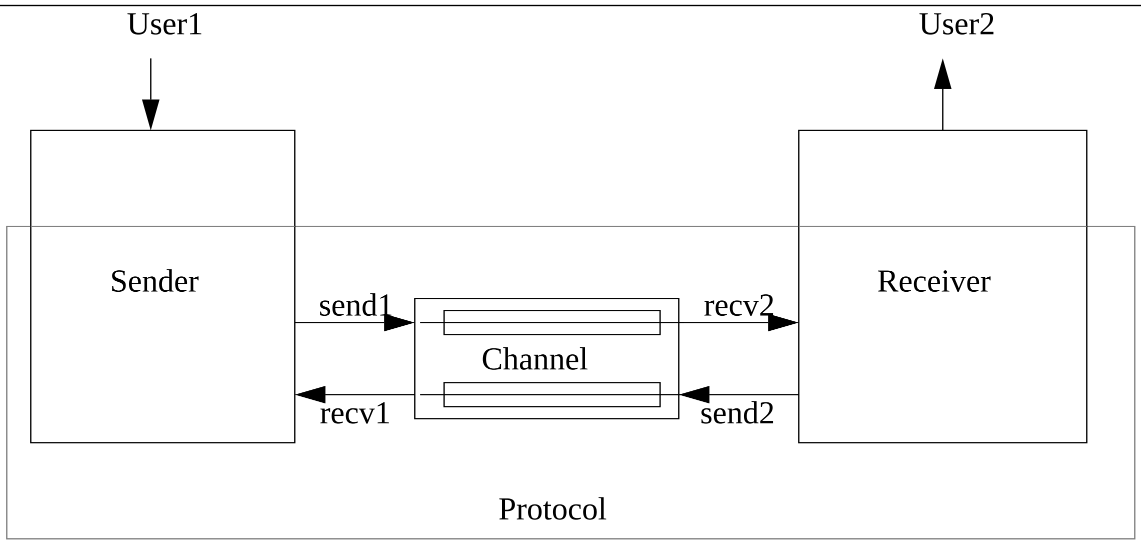
Table 2.2: LOTOS Behaviour Expressions

Description	Basic LOTOS	Full LOTOS
Nested: B behaves like B1. Used to resolve operator priorities.	$B = (B1)$	$B = (B1)$
Hide: B behaves like *B1* by replacing any gate in $\{g_1,..,g_n\}$ offered by *B1* by an internal action **i**.	$B =$ **hide** $g_1,..,g_n$ **in** $B1$	$B =$ **hide** $g_1,..,g_n$ **in** $B1$
Parallel-Selected Synchronization: B behave like *B1* and *B2* simultaneously (in parallel) with synchronization on gates $g_1,..,g_n$.	$B = B1 \mid[g_1,..,g_n]\mid B2$	$B = B1 \mid[g_1,..,g_n]\mid B2$
Parallel-Pure Interleaving: *B* behaves like *B1* and *B2* simultaneously (in parallel) with no synchronization.	$B = B1 \mid\mid\mid B2$	$B = B1 \mid\mid\mid B2$
Parallel-Full Synchronization: *B* behaves like *B1* and *B2* simultaneously (in parallel) with synchronization on any observable action that can be offered by *B1* or *B2*.	$B = B1 \mid\mid B2$	$B = B1 \mid\mid B2$
Relabelling: *B* behaves as *B'* by relabelling every action h_i that *B'* may perform by g_i. This construct is not in the syntax of LOTOS. It is constructed dynamically during the execution of inference rules.	$B =(B')[g_1/h_1,..., g_n/h_n]$	$(B)[g_1/h_1,..., g_n/h_n]$

Table 2.2: LOTOS Behaviour Expressions

Description	Basic LOTOS	Full LOTOS
Process Instantiation and Recursion: *B* behaves like the behaviour definition of process P by replacing (relabelling) any offered formal gate with its correspondent actual gate. In Full LOTOS actual parameters can also be passed.	$B = P[g_1,..,g_n]$	$B = P[g_1,..,g_n](t_1,..,t_n)$
Generalized Choice: *B* behaves like $B1(x = t_1)$ [] .. $B1(x = t_n)$ where $t_i \in s$	*N/A*	*B* = **choice** *x*:*s* **[]** *B1*
Guard: *B* behaves like *B1* if *P* is evaluated to true otherwise it will behave like **stop**	*N/A*	*B* = **[***P***] ->** *B1*
Local Definition: *B* behaves like B by substituting all occurrences of x_i by t_i.	*N/A*	*B* = **let** $x_1=t_1,..,x_n=t_n$ **in** *B*

In Appendix A, a simplified LOTOS version of the Alternating Bit Protocol specification is given [9]. This protocol provides a reliable, uni-directional data transfer service between two users, User1 the source and User2 the sink. It uses an unreliable full duplex one place channel to transfer protocol data units (PDUs) and acknowledgements. To ensure that the messages sent by User1 are received in the correct order by User2, the protocol associates a sequence number, alternating between 0 and 1, with the delivered (PDUs) and acknowledgements. Figure 2-5 illustrates the overall composition of the Sender and the Receiver entities, associated with User1 and User2 respectively, and the unreliable channel. The gates used by the protocol to communicate with the channel are hidden from the environment, i.e. the users.



```
specification  abp_service[User1,User2] : noexit
behavior
    hide send1, recv1, send2, recv2, LOST in
        abp[User1, User2, send1, recv1, send2, recv2, LOST](0 of Bit)
   where
   process abp[User1,User2,send1,recv1,send2,recv2,LOST](s_seq:Bit):noexit:=

      (sender [User1, send1, recv1, LOST] (s_seq)
      |||
      receiver[User2, send2, recv2] (s_seq) )

      |[send1, recv1, send2, recv2, LOST]|
      channel [send1, recv1,  send2, recv2, LOST]

      where
       process sender[User1, send1, recv1, LOST](s_seq:Bit) : noexit := ... endproc
       process receiver[r_user, send, recv](r_seq:Bit) : noexit := ... endproc
       process channel [send1, recv1, send2, recv2, LOST] : noexit := ... endproc
  endproc
endspec
```

Figure 2-5 Alternating Bit Protocol Structure

2.4.3 Inference Rules

The operational semantics of LOTOS behaviour expressions defines the labelled transition relation $B \xrightarrow{a} B'$, which means that the behaviour expression B can perform the action a then behaves as B'.

The relation $\rightarrow$ is defined by means of axioms and inference rules [84], where axioms are statements that are assumed to be valid, and inference rules derive new valid statements depending on the validity of other statements. An inference rule has the form:

$$\frac{S_1, S_2, ..., S_n}{S}$$

which means if S_1 and S_2 ... and S_n are valid then S is valid.

In order to define the inference rules for full LOTOS behaviour expressions, let:

- t_i is a term (ADT value expression)
- $eval(t_i)$ denote the value of the term t_i.
- G denotes the set of the behaviour's formal gates;
- $g, g_i \in$ G;
- **i** denotes an internal action;
- d_i is either
  $!E_i$, denoting the offering of the value $eval(E_i)$
  or
  $?x_i{:}s_i$, denoting that a value for the variable x_i of sort s_i is expected;
- δ denotes the successful termination's action name;
- a, a_i denotes any action;
- *name(a)* denotes the gate identifier of action *a*.
- *card(a)* denotes the number of events offered by action *a*.
- $event_i(a)$ denotes the ith event of action *a*.
- *sort(E)* denotes the sort of event *E*.
- *pred(a)* denotes the associated predicate of action *a*.
- $a_1 \equiv a_2$ denotes a_1 *matches* a_2, defined in Table 2.1.
- $a_1 \uparrow a_2$ denotes the resulting action obtained from matching a_1 and a_2, also defined in Table 2.1.
- $[t_1/x_1,...,t_n/x_n]\ B$ denotes the result of the replacement of all occurrences of $x_1,..,x_n$ in B by $t_1,...,t_n$ respectively.
- $(B)[g_i/g]$ denote that g is relabelled by g_i for every action that may be performed by B on the gate g.

.

Action a_1	Action a_2	$a_1 \equiv a_2$	$a_1 \uparrow a_2$	Effect
$g_1!E_1\ P_1$	$g_2!E_2\ P_2$	$g_1 = g_2$ $sort(E_1)=sort(E_2)$ $eval(E_1)=eval(E_2)$ $eval(P_1)=true$ $eval(P_2)=true$	$g_1!eval(E_1)$	Synchronization
$g_1!E\ P_1$	$g_2?X{:}t\ P_2$	$g_1 = g_2$ $sort(E) = t$ $eval([v/X]P_1)=true$ $eval(P_2)=true$	$g_1!eval(E)$	$X=eval(E)$
$g_1?X{:}t_1\ P_1$	$g_2?Y{:}t_2\ P_2$	$g_1 = g_2$ $t_1=t_2$ $eval([v/X]P_1)=true$ $eval([v/Y]P_2)=true$	$g_1!v$	$X=Y=v$ $v \in$ domain(t_1)

Table 2.3: Matching Actions

Axioms and inference rules that define the labelled transition relation $B \xrightarrow{a} B'$ for full LOTOS behaviour expressions are given in Table 2.4

Description	Axioms/Inference Rules
Inaction: *B* cannot interact with the environment nor execute internal actions	No rules for **stop**
Observable Action Prefix: *B* interacts with the environment on gate *g* then behaves like *B1*	$gd_1...d_n[P];\ B1 \xrightarrow{g!v_1...!v_n} [t_1/y_1,...,t_m/y_m]B1$ if $eval([t_1/y_1,...,t_m/y_m]\ P)$ = true, where $\{(t_1,y_1), ..., (t_m,y_m)\} = \{(t,x) \mid d_i = ?x{:}s,\ t \in domain(s)\}$ $v_i = eval(t)$ if $d_i = !t$, $v_i = eval(t)$ if $d_i = ?x{:}s$ and $t \in domain(s)$
Internal Action Prefix:*B* execute internally action **i** then behaves like *B1*	$\mathbf{i};\ B1 \xrightarrow{\mathbf{i}} B1$
Successful Termination: *B* interacts with the environment on gate δ then behaves like **stop**	$\mathbf{exit} \xrightarrow{\delta} \mathbf{stop}$ $\mathbf{exit}(E_1,...,E_n) \xrightarrow{\delta!v_1...!v_n} \mathbf{stop}$, where $v_i=eval(E_i)$, if E_i is a term $v_i \in domain(s)$, if E_i = **any** *s*

Table 2.4: LOTOS Axioms and Inference Rules

Description	Axioms/Inference Rules
Choice: *B* can either behave like *B1* or *B2*	$\dfrac{B1 \xrightarrow{a} B1'}{B1 \,[]\, B2 \xrightarrow{a} B1'}$ $\dfrac{B2 \xrightarrow{a} B2'}{B1 \,[]\, B2 \xrightarrow{a} B2'}$
Guard: *B* behaves like *B1* only if eval(P) =tr*ue*	$\dfrac{B1 \xrightarrow{a} B1',\ eval(P)=true}{[P] \text{ -> } B1 \xrightarrow{a} B1'}$
Local definition: *B* behaves like *B1* by replacing all occurrences of $x_1,...,x_n$ by $t_1,...,t_n$ respectively	$\dfrac{[t_1/x_1, ..., t_n/x_n]B1 \xrightarrow{a} B1'}{\textbf{let } x_1{:}s_1=t_1, ..\ x_n{:}s_n=t_n \textbf{ in } B1 \xrightarrow{a} B1'}$
Summation on values: *B* behaves like *B1* for any value $t \in domain(s)$	$\dfrac{[t/x]B1 \xrightarrow{a} B1',\ t \in domain(s)}{\textbf{choice } x{:}s\ []\ B1 \xrightarrow{a} B1'}$
Disable: *B* behaves like *B1* unless disabled by *B2*	$\dfrac{B1 \xrightarrow{a} B1',\ name(a) \neq \delta}{B1\ [>\ B2 \xrightarrow{a} B1'\ [>\ B2}$ $\dfrac{B1 \xrightarrow{\delta!v_1...!v_n} B1'}{B1\ [>\ B2 \xrightarrow{\delta} B1'}$ $\dfrac{B2 \xrightarrow{a} B2'}{B1\ [>\ B2 \xrightarrow{a} B2'}$
Enable: *B* behaves like *B1* unless it terminate successfully then it behaves like *B2*	$\dfrac{B1 \xrightarrow{a} B1',\ name(a) \neq \delta}{B1 >> \textbf{accept } x_1{:}s_1..x_n{:}s_n \textbf{ in } B2 \xrightarrow{a} B1' >> \textbf{accept } x_1{:}s_1..x_n{:}s_n \textbf{ in } B2}$ $\dfrac{B1 \xrightarrow{\delta!v_1...!v_n} B1'}{B1 >> \textbf{accept } x_1{:}s_1..x_n{:}s_n \textbf{ in } B2 \xrightarrow{\mathbf{i}} [v_1/x_1,...,v_n/x_n]B2}$
Nested: *B* behaves like *B1*. Used to resolve priorities.	$\dfrac{B \xrightarrow{a} B'}{(B) \xrightarrow{a} B'}$

Table 2.4: LOTOS Axioms and Inference Rules

Description	Axioms/Inference Rules
Hide: *B* behaves like *B1* by replacing any action with a gate in $\{g_1,..,g_n\}$ offered by *B1* by an internal action **i**.	$\dfrac{B1 \xrightarrow{a} B1',\ name(a) \notin \{g_1,..,g_n\}}{\textbf{hide}\ g_1,..,g_n\ \textbf{in}\ B1 \xrightarrow{a} \textbf{hide}\ g_1,..,g_n\ \textbf{in}\ B1'}$ $\dfrac{B1 \xrightarrow{a} B1',\ name(a) \in \{g_1,..,g_n\}}{\textbf{hide}\ g_1,..,g_n\ \textbf{in}\ B1 \xrightarrow{\mathbf{i}} \textbf{hide}\ g_1,..,g_n\ \textbf{in}\ B1'}$
Parallel-Selected Synchronization: B behave like *B1* and *B2* simultaneously (in parallel) with synchronization on gates $g_1,..,g_n$.	$\dfrac{B1 \xrightarrow{a} B1',\ name(a) \notin \{g_1,..,g_n,\ \delta\}}{B1\ \vert[g_1,..,g_n]\vert\ B2 \xrightarrow{a} B1'\ \vert[g_1,..,g_n]\vert\ B2}$ $\dfrac{B2 \xrightarrow{a} B2',\ name(a) \notin \{g_1,..,g_n,\ \delta\}}{B1\ \vert[g_1,..,g_n]\vert\ B2 \xrightarrow{a} B1\ \vert[g_1,..,g_n]\vert\ B2'}$ $\dfrac{\begin{array}{c} B1 \xrightarrow{a_1} B1',\ B2 \xrightarrow{a_2} B2', \\ name(a_1) = name(a_2) \in \{g_1,..,g_n,\ \delta\} \text{ and } a_1 \equiv a_2 \end{array}}{B1\ \vert[g_1,..,g_n]\vert\ B2 \xrightarrow{a_1 \uparrow a_2} B1'\ \vert[g_1,..,g_n]\vert\ B2'}$
Parallel-Pure Interleaving: *B* behaves like *B1* and *B2* simultaneously (in parallel) with no synchronization.	$\dfrac{B1\ \vert[]\vert\ B2 \xrightarrow{a} B'}{B1\ \vert\vert\vert\ B2 \xrightarrow{a} B'}$
Parallel-Full Synchronization: *B* behaves like *B1* and *B2* simultaneously (in parallel) with synchronization on any observable action that can be offered by *B1* or *B2*.	$\dfrac{B1\ \vert[g_1,..,g_n]\vert\ B2 \xrightarrow{a} B'}{B1\ \vert\vert\ B2 \xrightarrow{a} B'}$ where$\{g_1,..,g_n\}$ is the set of all possible gates of *B1* and *B2*.
Relabelling: *B* behaves as *B'* by relabelling every action h_i that *B'* may perform by g_i. This construct is not in the syntax of LOTOS. It is constructed dynamically.	$\dfrac{B \xrightarrow{a} B',\ name(a) \notin \{h_1,..,h_n\}}{(B)[g_1/h_1,...,\ g_n/h_n] \xrightarrow{a} B'[g_1/h_1,...,\ g_n/h_n]}$ $\dfrac{B \xrightarrow{g d_1...d_n} B',\ g = h_i \in \{h_1,..,h_n\}}{(B)[g_1/h_1,...,\ g_n/h_n] \xrightarrow{g_i d_1...d_n} B'[g_1/h_1,...,\ g_n/h_n]}$

Table 2.4: LOTOS Axioms and Inference Rules

Description	Axioms/Inference Rules
Process Instantiation and Recursion:*B* behaves like the behaviour definition of process P by replacing (relabelling) any offered formal gate with its correspondent actual gate. In Full LOTOS actual parameters can also be passed.	$\dfrac{([t_1/x_1,...,t_m/x_m]B)[g_1/h_1,..., g_n/h_n] \xrightarrow{a} B'}{P[g_1,..,g_n](t_1,...,t_m) \xrightarrow{a} B'}$ iff there exists a process definition: $PP[h_1,..,h_n](x_1{:}s_1..x_m{:}s_m)$ **:=** *B*

Table 2.4: LOTOS Axioms and Inference Rules

2.4.4 An Example

Here, we demonstrate how the inference rules can be applied to obtain the possible transitions of a given behaviour expression.

Suppose the following process definition exists:

> **process** P[a,b]:**noexit**:=
> a;b;P[a,b]
> **[]**
> b;a;P[a,b]
> **endproc**

and Let B=

$P[g_1,g_2]\ |[g_2]|\ P[g_2,g_3]$

1–> Find all *trans* and B’ such that

$P[g_1,g_2]\ |[g_2]|\ P[g_2,g_3]\ \text{-trans}\rightarrow B'$

By applying the inference rules of selected synchronization we obtain:

$$\frac{P[g_1,g_2]\ \text{-trans-}{>}\ B_1,\ P[g_2,g_3]\ \text{-trans}\rightarrow B_2\ ,\ \text{if name(trans)} \in \{g_2, \delta\}}{P[g_1,g_2]\ |[g_2]|\ P[g_2,g_3]\ \text{-trans}\rightarrow B_1\ |[g_2]|\ B_2}$$

$$\frac{P[g_1,g_2]\ \text{-trans-}{>}\ B_1,\ \text{if name(trans)} \notin \{g_2, \delta\}}{P[g_1,g_2]\ |[g_2]|\ P[g_2,g_3]\ \text{-trans}\rightarrow B_1\ |[g_2]|\ P[g_2,g_3]}$$

and

$$\frac{P[g_2,g_3]\ \text{-trans-}{>}\ B_2,\ \text{if name(trans)} \notin \{g2, \delta\}}{P[g_1,g_2]\ |[g_2]|\ P[g_2,g_3]\ \text{-trans}\rightarrow P[g_1,g_2]\ |[g_2]|\ B_2}$$

2-> To satisfy 1, we have to find all

P[g1,g2] -trans1→ B1

and P[g2,g3] -trans2→ B2

We have: (a)

$$\frac{\text{(a;b;P[a,b] [] b;a;P[a,b]) } [g_1/a, g_2/b] \text{ -trans}\rightarrow B_1}{P[g_1,g_2] \text{ -trans}\rightarrow B_1}$$

and (b)

$$\frac{\text{(a;b;P[a,b] [] b;a;P[a,b]) } [g_2/a, g_3/b] \text{ -trans}\rightarrow B_2}{P[g_2,g_3] \text{ -trans}\rightarrow B_2}$$

3-> To satisfy 2(a) we have to find all

(a;b;P[a,b] [] b;a;P[a,b]) $[g_1/a,g_2/b]$ -trans→ B_1

We have:

$$\frac{\text{a;b;P[a,b] [] b;a;P[a,b] -g-> } B_1'}{\text{(a;b;P[a,b] [] b;a;P[a,b])}[g_1/a,g_2/b] \text{ -g'-> } (B_1')[g_1/a,g_2/b]}$$

g' = g if g $\notin \{g_1,g_2\}$

g' = g_1 if g = a

g' = g_2 if g = b

4-> Find all

a;b;P[a,b] **[]** b;a;P[a,b] -trans→ B_1'

We have:

$$\frac{\text{a;b;P[a,b] -trans}\rightarrow B_1'}{\text{a;b;P[a,b] [] b;a;P[a,b] -trans}\rightarrow B_1'}$$

and

$$\frac{\text{b;a;P[a,b] -trans}\rightarrow B_1'}{\text{a;b;P[a,b] [] b;a;P[a,b] -trans}\rightarrow B_1'}$$

5-> By the axiom of prefix behaviour we can obtain directly the following transitions:

b;a;P[a,b] -b→ a;P[a,b]

a;b;P[a,b] -a→ b;P[a,b]

4<- Back to step 4, we now can obtain the following transitions:

a;b;P[a,b] **[]** b;a;P[a,b] -a→ b;P[a,b]

a;b;P[a,b] **[]** b;a;P[a,b] -b→ a;P[a,b]

3<- Back to step 3, the following transitions can then be obtained:

(a;b;P[a,b] **[]** b;a;P[a,b]) $[g_1/a,g_2/b]$ -g_1→ (b;P[a,b])$[g_1/a,g_2/b]$

(a;b;P[a,b] [] b;a;P[a,b]) $[g_1/a,g_2/b]$ -g_2→ (a;P[a,b])$[g_1/a,g_2/b]$

2<- In step 2(a) above we have:

$P[g_1,g_2]$ -g_1→ (b;P[a,b])$[g_1/a,g_2/b]$

$P[g_1,g_2]$ -g_2→ (a;P[a,b])$[g_1/a,g_2/b]$

are valid transitions.

Similarly for step 2(b) we can obtain:

$P[g_2,g_3]$ $-g_2\rightarrow$ $(b;P[a,b])[g_2/a,g_3/b]$

$P[g_2,g_3]$ $-g_3\rightarrow$ $(a;P[a,b])[g_2/a,g_3/b]$

1<- Then from the initial behaviour expression we can obtain the following transitions:

$P[g_1,g_2]$ $|[g_2]|$ $P[g_2,g_3]$ $-g_2\rightarrow$

$(a;P[a,b])[g_1/a,g_2/b]$

|[g2]|

$(b;P[a,b])[g_2/a,g_3/b]$

$P[g_1,g_2]$ $|[g_2]|$ $P[g_2,g_3]$ $-g_1\rightarrow$

$(b;P[a,b])[g_1/a,g_2/b]$

$|[g_2]|$

$P[g_2,g_3]$

and

$P[g_1,g_2]$ $|[g_2]|$ $P[g_2,g_3]$ $-g_3\rightarrow$

P[g1,g2]

$|[g_2]|$

$(a;P[a,b])[g_2/a,g_3/b]$

The behaviour tree of B= $P[g_1,g_2]$ $|[g_2]|$ $P[g_2,g_3]$ is infinite. See Figure 2-6 .

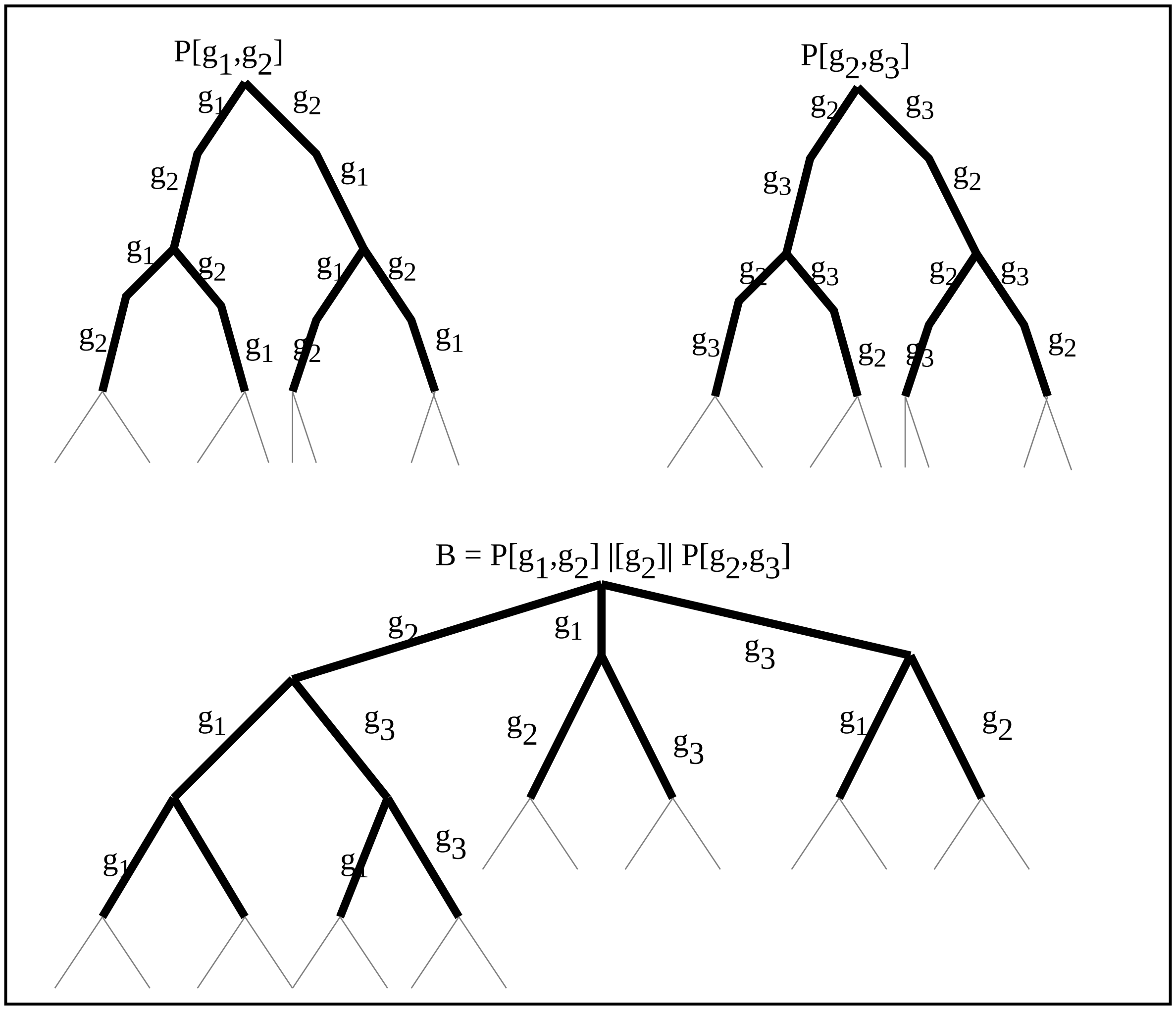


Figure 2-6 An Infinite Behaviour Tree

2.4.5 Relief Strategies for the State Space Explosion

As mentioned earlier, the state space of a LOTOS specification is defined as a behaviour tree that can be very large and often infinite, e.g. the tree in Figure 2-6 . The existing LOTOS verification methodologies use different techniques to cope with LOTOS state space explosion. Unfortunately, there is no survey on such techniques in the literature. The following is a brief description of these techniques.

Step-by-Step Execution

In step-by-step execution [63][66][98][138], the user can explore the behaviour tree by choosing, from the current behaviour, one of the possible next actions and provide values if required by the action. This operation can be repeated under user guidance. Doing so, the user can determine whether or not the exercised branches (sequences of actions) conform to the intended behaviour. Obviously, this technique is tedious if one wishes to execute the specification for more than few dozen steps.

Weighted/Probabilistic Execution

The idea behind this technique is to assign weights to the operators in the given LOTOS specification. The system then traverses the behaviour tree by automatically selecting an action at each level. The selection depends on the weight, accumulated during the derivation, of each offered action, for example, the weight can be interpreted as priority or probability. Ohmaki et al. have adopted this technique in their LOTOS tool environment LIpS [108].

The drawback of this technique is the selection and the representation of these weights. In LIpS, the weights are stated as special comments in the specification.

Fair Execution

In this technique, similar to the previous technique, the system traverses the behaviour tree by automatically selecting an action at each level. The selection, on the other hand, is based on the underlying fairness assumptions. Wu and Bochmann have shown in [154] how fair execution model for Basic LOTOS can be constructed based on three fairness concepts defined for CSP [89], namely process fairness, guard fairness, and channel fairness.

Random Walk Execution

The state exploration is done by randomly selecting an action at each level. This technique can also be considered as a fair execution technique. This technique was used in [67] to obtain execution paths for distributed algorithms specified in LOTOS.

Interleave Expansion

The interleave expansion technique, proposed by Quemada in [116], generates a representation of the transition system of LOTOS where interleaved behaviours are represented in a compressed form. This representation provides a size reduction with respect to the representation of the plain state space. The reduction may be of many orders of magnitude for specifications which make extensive use of interleaving.

The major drawback of this technique is that an extension of LOTOS is necessary in order for the interleaved expansion to be possible, since the compressed form is represented in term of new parallel composition constructs.

Symbolic Execution

The idea behind symbolic execution techniques [1][115] is to produce a compact behaviour tree for a given LOTOS specification by using symbolic behaviour states which represent a large number of explicit behaviour states. In particular, parameters and variables are used instead of actual values. Loops can be found by detecting if a current behaviour state was previously encountered. Symbolic behaviour trees can be used as a base for verification techniques such as model checking [60]. The problems with symbolic execution techniques is that in particular cases the growth of the behaviour tree is still too quick, and the memory consumption is very high, since all encountered behaviour states must be stored in order to detect loops.

ADT Narrowing Techniques

The purpose of ADT narrowing is to find solutions for a given condition (goal) or to determine that there is no solution.This technique is useful to prune branches in the symbolic behaviour trees associated with unsatisfiable predicates [39]. More discussion on this approach is given in the next chapter.

Behaviour Transformation

Since LOTOS is based on rigorous mathematical foundations, behaviour expressions may be transformed into other equivalent expressions using equivalence laws, based on the concept of bisimulation equivalence [84]. Such laws can be used to simplify LOTOS expressions and therefore reduce the state space, or to demonstrate that two behaviour expressions are equivalent. Such proofs can be useful for finding loops during the construction of symbolic behaviour trees[1], i.e. two behaviour expressions are equivalent if both can be transformed, using the same set of laws, into an identical behaviour.

A number of equivalence laws are known. A short list follows:

- B_1 [] B_2 = B_2 [] B_1
- B_1 [] (B_2[] B_3) = (B_1 [] B_2) [] B_3
- B [] **stop** = B
- B_1 | B_2 = B_2 | B_1 — where '|' denotes any parallel operator using the same instance throughout the law
- B_1 | (B_2| B_3) = (B_1 | B_2) | B_3 — where '|' denotes any parallel operator using the same instance throughout the law

- **exit**(...) | **stop** = **stop** where '|' denote any parallel operator
- **stop** >> *B* = **stop**
- *B* [> **stop** = **stop** [> *B* = *B*

Other algebraic laws for weak bisimulation congruence and for testing congruence can be found in [84].

Compilation/Translation Techniques

These techniques translate LOTOS specifications into well other known models, such as Petri nets or Finite State Machines [57][100]. The existing verification tools for the latter can then be applied.

Unfortunately, such techniques may not apply to LOTOS specifications with infinite behaviour trees, due to the fact that all behaviour states need to be captured in order to have a complete translation.

2.4.6 Existing Validation and Verification tools for LOTOS

Many tools exist for LOTOS that assist in specification, simulation, validation, verification, implementation, and testing.

AUTO

AUTO [100] is a verification system for distributed programs. Its functionalities are the following:

- *LOTOS to Automaton Translation*: Translates Basic LOTOS specifications (or other specifications written in an algebraic process algebra such as CCS and SCCS) into an automaton representing the behaviour of the specification.
- *Behaviour Transformation*: An Automaton can be reduced and compared with another automaton with respect to *strong* and *weak* equivalence relations.
- *Step-by-Step Execution*: An Automaton can be explored manually.
- *Graphical representation*: The results are displayed using a tool called AutoGraph.

To guarantee that the produced automaton is finite, AUTO applies certain constraints on the generation of the automaton. For example, the rules avoid dynamic generation of processes inside recursive definitions.

In [21], the authors describe a verification of a point-to-point sliding window protocol with non-acknowledged messages using AUTO.

CÆSAR.ADT

CÆSAR.ADT [56] is a compiler that translates the ADT definitions of LOTOS into a C program. The output of CÆSAR.ADT is a C library containing a C type (*resp*. function) for each sort (*resp*. operation) defined in a LOTOS program.

CÆSAR.ADT allows sorts and operations to be declared “external”, which means that the implementation in C of those sorts and operations is provided by the user, instead of being generated automatically by the tool.

CÆSAR.ADT, however, does not accept all ADT constructs, for example parameterized types are not considered. It also imposes restrictions on the subset of accepted ADT constructs. In addition, special comments must be used in the ADT definitions to identify *constructors* (primitive operations) and to provide correspondence between the names of LOTOS objects and the names of C objects implementing them.

CÆSAR

CÆSAR [55][57] is a tool that uses a *compilation technique* to verify LOTOS specifications. CÆSAR translates the source specification, accompanied by a C implementation of the abstract data types, either written manually or generated by CÆSAR.ADT, into an extended Petri net and a graph. Existing validation and verification tools for Petri nets can then be applied. The graph, on the other hand, describes all possible state transitions. The translation of a LOTOS program into a graph is summarized by the following four steps:

- The *expansion phase* translates the LOTOS program into an equivalent SUBLOTOS program in a bottom-up way. SUBLOTOS is a process algebra which can be viewed as a simplified subset of LOTOS.
- The *generation phase* translates SUBLOTOS behaviour expressions into an intermediate form, called *network*. The network is defined by: a *control part*, represented as a Petri net, and a *data part*, consisting of global and typed variables. These variables are accessed and modified by actions attached to the transitions.
- The *optimization phase* applies transformations to the network to reduce the number of places and transitions. The transformation is applied on the control part using Petri net structural analysis methods, and on the data part using data-flow analysis.
- The *simulation phase* performs reachability analysis and generates a graph corresponding to the given network. The edges of the graph are labelled by *actions*, possibly accompanied by a list of values sent or received during the rendez-vous communication. The states of the graph are labelled by the values of the program (specification) variables.

The drawbacks of CÆSAR is that it disallows recursive process calls in some cases, in order to prevent the generation of infinite graphs. Also, SUBLOTOS dynamic semantics is free from dynamic gate relabelling. This implies that CÆSAR accepts only LOTOS specifications where static relabelling (i.e. substitution of formal gates by actual gates) does not change the dynamic semantics of the specification.

CÆSAR was used for the verification of an atomic multicast protocol [5], a subset of the FIP protocol [4], and an overtaking protocol for cars [42].

ALDÉBARAN

ALDÉBARAN [45][46] is a tool for performing comparison and reduction of graphs according to various bisimulation equivalence relations and preorders, such as observational equivalence [103] and safety equivalence [120].

Graph comparison allows to compare two graphs with respect to one of various equivalences and preorder relations. The result of the comparison (true or false) is obtained as output. In case of failure, ALDÉBARAN provides diagnostic sequences.

Graph reduction, on the other hand, allows to generate the smallest graph which is equivalent to the original graph with respect to a given equivalence relation.

The tool uses two approaches for determining whether two graphs are strongly bisimilar. The first approach computes successive refinements on an initial partition of the states of the graph, until stabilization is reached. The resulting partition coincides exactly with the equivalence classes of strong bisimulation. Two graphs are strongly bisimilar if and only if their stabilized partitions are identical. This approach can be applied to weaker bisimulation-based relations by modifying each graph, taking into account abstraction criteria, and then computing the stabilized partitions with respect to strong bisimulation. The major drawback of this approach is that the application of abstraction criteria is done by adding new transitions to the graph. Therefore, the number of transitions may become very large for the available memory space.

The second approach consists in comparing two graphs “on the fly”. It was used for the verification of Milner’s scheduler, Datalink protocol, and rel/REL$_{fifo}$ protocol[47]. However, this approach only performs comparisons and not reductions.

CLÉOPÂTRE

CLÉOPÂTRE verifies a graph, representing a LOTOS behaviour tree generated by CÆSAR, against a set of formulas expressed in the branching-time temporal logic LTAC[114]. It includes a model checking module [120] and an explanation module [117], which provides diagnostics-based sequences extracted from a graph, generated from the source specification, when a formula is not valid. The subset of LTAC formulas used for the verification of LOTOS programs is described by the following grammar:

$$T \mid init \mid enable(a) \mid after(a) \mid sink \mid f \wedge g \mid \neg f \mid inev[f]g \mid pot[f]g$$

where *f* and *g* are formulas and *a* is a label attached to a transition of the graph. The following is the definition of these formulas:

- any state satisfies *T*;
- the initial state of the program satisfies *init*;
- a state *s* satisfies *enable(a)* if it is possible to execute action a from state *s*;
- a state satisfies *after(a)* if it can only be reached immediately after the execution of action *a*;
- a state satisfies *sink* if it has no outgoing transitions;
- a state satisfies $f \wedge g$ if it satisfies *f* and *g*;
- a state satisfies $\neg f$ if does not satisfy *f*;
- a state *s* satisfies $inev[f]g$ if, for every execution of the program from *s*, *f* is true until *g* becomes true;
- a state *s* satisfies $pot[f]g$ if there exist an execution from *s* such that *f* is true until *g* becomes true.

This tool produces extremely large models, although they are generated and stored efficiently For example the alternating bit protocol results in several thousands of states, depending on the number of messages one wants to consider.

CÆSAR, ALDÉBARAN, and CLÉOPÂTRE were combined in one toolbox [48]. Figure 2-7 illustrates the toolbox architecture: a protocol is checked against its expected service, expressed in LTAC, using the CLÉOPÂTRE tool after the translation of the LOTOS expressions into graphs using CÆSAR tool. ALDÉBARAN is applied on specifications in LOTOS (translated into graphs). In [48], this technique is applied to verify the rel/REL_{fifo} protocol [130] that supports atomic communication between a transmitter and several receivers.

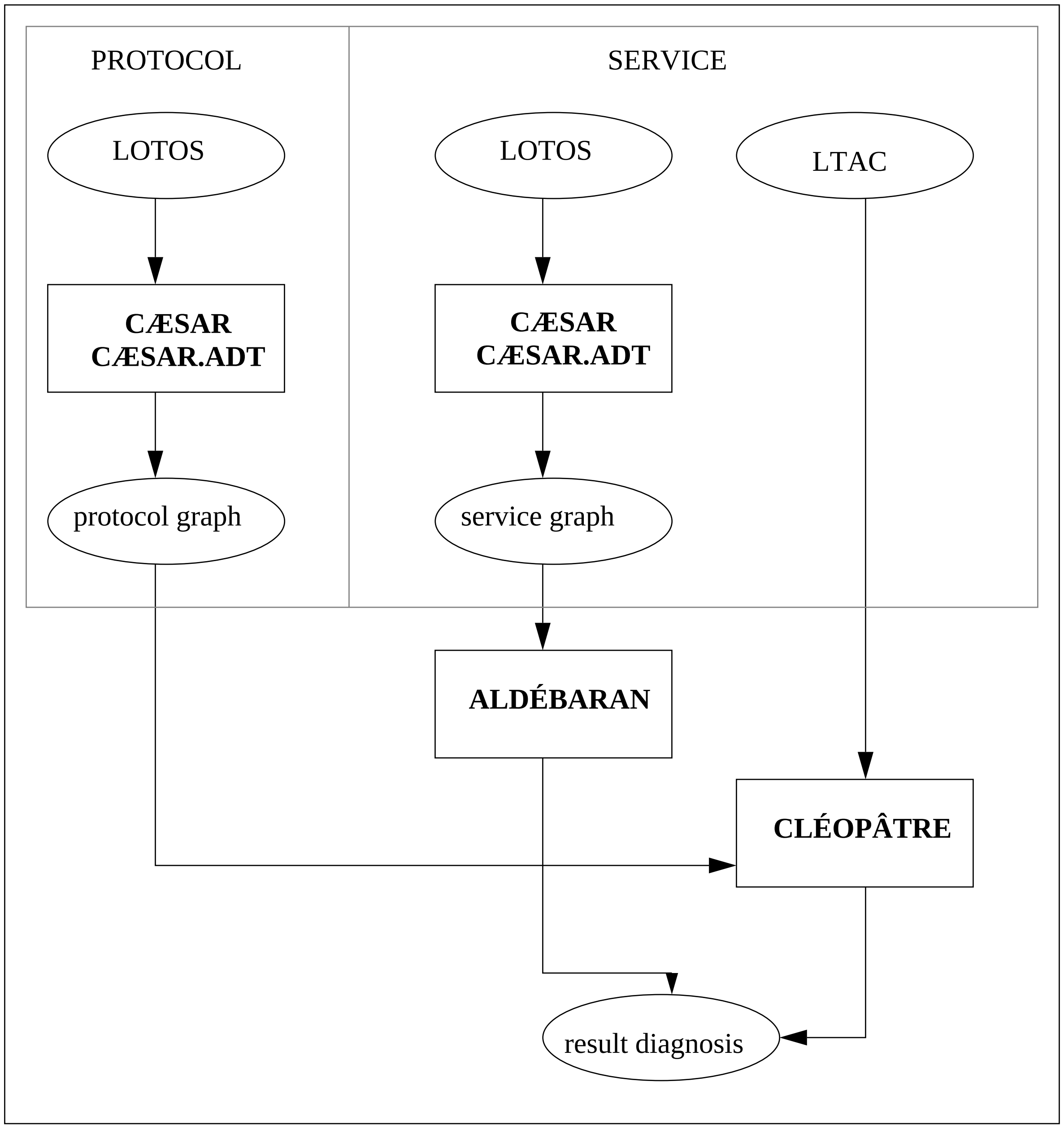


Figure 2-7 Architecture of CÆSAR, ALDÉBARAN, and CLÉOPÂTRE toolbox

LOLA

LOLA (LOtos LAboratory) [115] is a transformation tool used in validation and in design by stepwise refinement. It uses a *symbolic execution technique* to generate a recursive behaviour tree of a given LOTOS specification (i.e. it detects behaviour already encountered) by using the so-called *parameterized expansion*. LOLA also provides the means to translate the symbolic tree into a monolithic style LOTOS specification.

A recent version of the tool also implements the *interleave expansion* discussed in section 2.4.5.

The transformation of an alternating bit protocol specified in LOTOS is given in [115]. [110] describes the testing functionalities of LOLA based on the definition of testing equivalence in [107].

SMILE

SMILE (*S*y*M*bolic *I*nteractive *L*otos *E*xecution) [140][39] is a full LOTOS symbolic simulation tool. It includes an implementation of an Abstract Data Types *narrower* algorithm based on a *lazy evaluation* strategy. Using the narrower, the state space (behaviour tree) exploration can be done symbolically, which means that behaviours are studied without instantiating the variables. The advantage of this symbolic technique with respect to previous existing ones, such as SELA technique described below or LOLA above, is that branches in the tree associated with unsatisfiable predicates can be detected and pruned. Details about the implementation of the narrower are given in [153].

To avoid infinite execution, the implementation of SMILE restricts the number of values for goal variables and the number of axioms applications. Also recently, they added an extra functionality to their system, similar to ours, that uses inference rules directed by static information [39]. A comparison of their technique to ours is given in Chapter 7.

SQUIGGLES

SQUIGGLES [19] is a tool that verifies strong, weak and testing equivalences between Basic LOTOS specifications. [20] describes some applications and the performance of SQUIGGLES.

LIpS

LIpS (*L*OTOS *I*nter*p*retation *S*erver) [108], is a LOTOS interpretation server that treats applications as clients. It provides an automatic simulation of a LOTOS specification with non-determinism.

The simulation is done by implementing the standard LOTOS inference rules including *weight* information. This information is specified by the user in the specification as special comments. Depending on different interpretations of the weights, non-determinism is resolved and the simulation proceeds.

To deal with *fairness*, the weights described in a process definition may change dynamically. For example, if the number of selected events offered by a process, say A, is greater than the number offered by another process, say B, the weights of events offered by A are decreased by 10%. Therefore, the probability to select events from B is increased.

During simulation, LIpS can also assign values to variables specified by the user using, again, special comments.

Lite

Lite (*Lotosphere I*ntegrated *T*ool *E*nvironment) [141], is an integrated tool environment that contains the following functions:

- *Syntax/Static Semantics analysis*.
- *Simulation*: *Lite* has adopted SMILE as a simulation tool, see above. It allows step-by-step and automatic symbolic exploration of the specification's behaviour tree. The simulation includes a narrowing technique to resolve conflicting predicates.
- *Compilation*: this tool translates a subset of LOTOS processes and abstract data types into C language. Certain design decisions that cannot be represented in LOTOS can be added in the specification as special comments called annotations.
- *Transformation*: it supports a number of correctness preserving transformations, such as:
    - Regrouping of parallel processes: is a transformation that takes an expression consisting of a number of processes composed with parallel operators and transforms these into an expression, with strong bisimulation equivalent behaviour, in which the processes are grouped differently.
    - Bipartition of functionality: is a transformation that splits a single process into two processes communicating in a prescribed configuration. The original and the resulting processes are weak bisimulation equivalent.
- *Verification*: *Lite* is able to reduce Basic LOTOS expressions according to a number of equivalences.
- *Testing*: *Lite* contains a tool to derive canonical testers for Basic LOTOS specifications based on the CO-OP method.
- *Graphical Interface*: based on the X-window system.

ELUDO

The university of Ottawa has developed an analysis environment, called ELUDO (*E*nvironment *L*OTOS de l'*U*niversité *D*'*O*ttawa) [61], for LOTOS specifications. ELUDO is made up of the following tools:

- Syntax and Static Semantics Analyzer [98], checks the LOTOS specification source for its syntax and static semantics according to [84], and, if it is found to be correct, an equivalent *internal* Prolog form is generated. The latter form is used by the following tools.
- ISLA [66], an interactive interpreter that simulates the behaviour of a LOTOS specification. This allows the designer to monitor and trace some execution sequences. ISLA provides a wide range of services [63], such as step-by-step execution, symbolic execution, defining constants for repeated values, resuming execution at any check point, and saving the state of the simulation for later execution. Figure 2-8 illustrates a step-by-step execution trace from the Alternating Bit Protocol specification given in Appendix A. Each action is associated with its sequential order in the action menu, and with a list of line numbers in the original specification from which the action was formed. The hidden actions are shown for analysis reason. In this example, three different user data messages are defined as constants, namely, $M1, $M2 and $M3.

```
abp_servive[User1,User2]

 (* First Message with sequence 0. Normal delivery *)
 1  User1 ?$M1:Data  [50]
 1  i (hiding: send1 !makepdu($M1,0):Mess)  [57,91]
 2  i (hiding: recv2 !makepdu($M1,0):Mess [is_pdu(makepdu($M1,0))])  [75,92]
 1  i (hiding: send2 !makeack(0):Mess)  [77,91]
 1  User2 !$M1:Data  [78]
 2  i (hiding: recv1 !makeack(0):Mess [is_ack(makeack(0))])  [65,92]

 (* Second Message with sequence 1. Two timeouts occurred due to the lost of PDU    *)
 (* and ACK.The proper ACK received after sending the same PDU three times.         *)
 1  User1 ?$M2:Data  [50]
 1  i (hiding: send1 !makepdu($M2,1):Mess)  [57,91]
 1  i (specified explicitly)  [94]
 1  i (hiding: TIMEOUT !makepdu($M2,1):Mess)  [67,94]
 1  i (hiding: send1 !makepdu($M2,1):Mess)  [57,91]
 2  i (hiding: recv2 !makepdu($M2,1):Mess [is_pdu(makepdu($M2,1))])  [75,92]
 1  i (hiding: send2 !makeack(1):Mess)  [77,91]
 1  User2 !$M2:Data  [78]
 1  i (specified explicitly)  [94]
 1  i (hiding: TIMEOUT !makeack(1):Mess)  [67,94]
 1  i (hiding: send1 !makepdu($M2,1):Mess)  [57,91]
 2  i (hiding: recv2 !makepdu($M2,1):Mess [is_pdu(makepdu($M2,1))])  [75,92]
 1  i (hiding: send2 !makeack(1):Mess)  [81,91]
 2  i (hiding: recv1 !makeack(1):Mess

 (* Ready to deliver the Third Message *)
 1  User1 ?$M3:Data  [50]
```

Figure 2-8 Step-by-Step Execution sample

- SELA [1], a tool that generates symbolically the behaviour tree of a given LOTOS specification and detects recursion (i.e. if a behaviour is already encountered). SELA also provides the means to translate the symbolic tree into a monolithic style LOTOS specification.
- LMC[60], a model checker for LOTOS that is capable of verifying branching temporal logic properties on the graph model generated by SELA. The properties are expressed in branching temporal logic CTL [32]. The set of CTL formulas used by LMC for the verification of LOTOS specifications is described by the following grammar:

  $$\neg f \mid f \wedge g \mid AX(f) \mid EX(f) \mid A[f \upsilon g] \mid E[f \upsilon g]$$

  where *f* and *g* are formulas and *a* is a label attached to a transition of the graph. The following is the definition of these formulas:

- *AX(f*) means that *f* holds in every immediate successor of the current state.
- *EX(f*) means that *f* holds in some immediate successor of the current state.
- *A*[*f* υ *g*] means that for every computation path, starting at the current state, there exists a sequence of transitions satisfying *g* at last, and *f* for all the other transitions.
- *E*[*f* υ *g*] means that for some computation path, starting at the current state, there exists a sequence of transitions satisfying *g* at last, and *f* for all the other transitions.

ELUDO's tools enable a design methodology involving several phases, see Figure 2-9 :

1- Initial phase: The designer collects the informal requirements.

2- Specification phase: The requirements are specified in LOTOS and by means of temporal logic properties.

3- Checking phase: Once the LOTOS specification is written, the syntax and static semantics analysis is performed. The dynamic semantics are then checked by using the interpreter ISLA.

4- Expansion phase: This phase deals with the generation of the symbolic behaviour tree.

5- Verification phase: At this point the model checker LMC is used to determine if the set of temporal logic properties, provided in the specification phase, is valid for the system specified by exploring the tree generated in the previous phase.

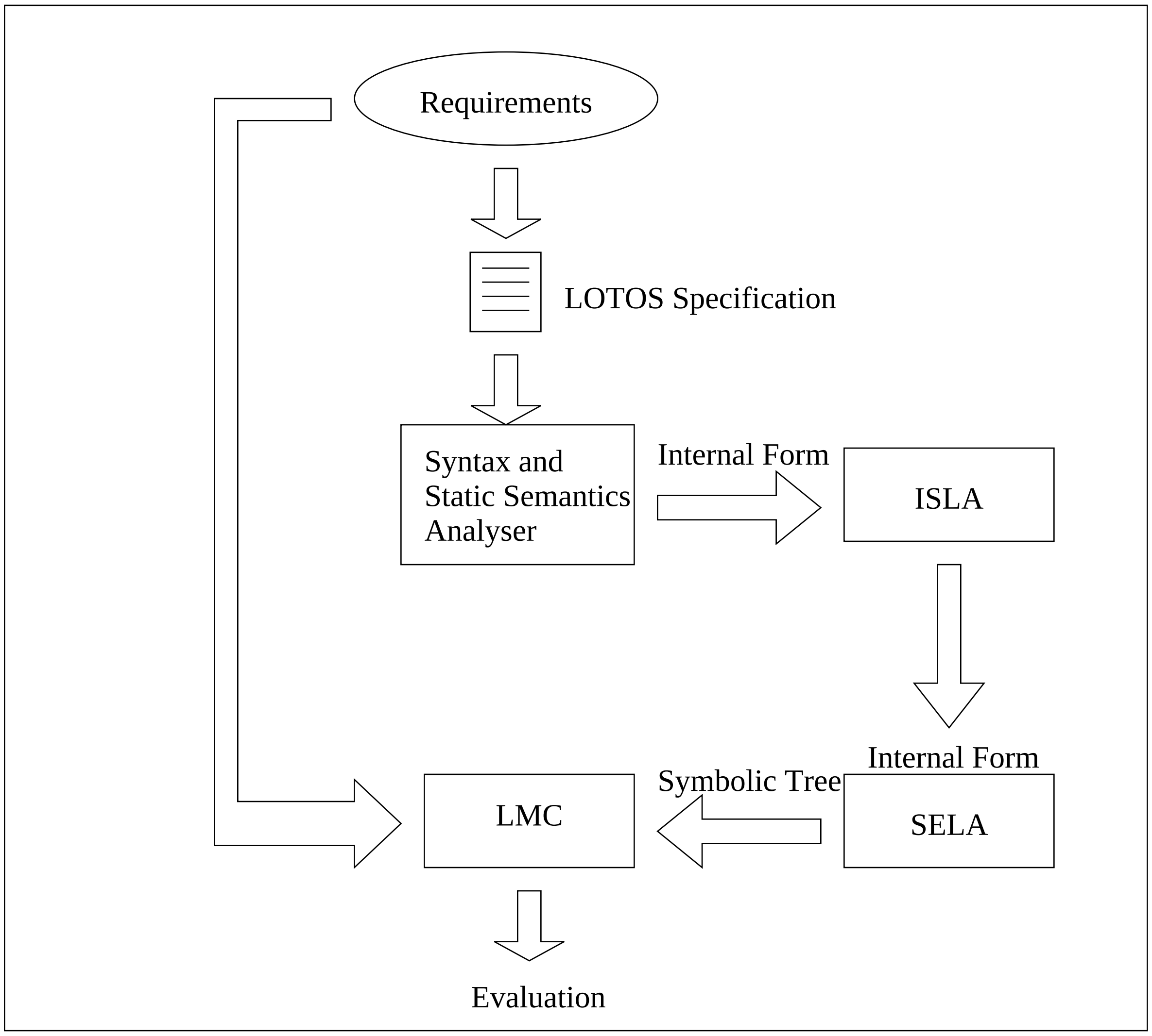


Figure 2-9 ELUDO: A Validation Environment

The environment also provide X-Window graphical interface, called XELUDO.

Chapter 3 Overview Of Goal-Oriented Execution

3.1 Introduction

Verification of protocols and other distributed systems specified in LOTOS requires the analysis of their behaviour trees. As we have discussed in chapter 2, the construction of such trees is restricted by the state space explosion problem. We have also discussed the various techniques used to overcome this problem.

In this chapter, we propose a formal search technique used to explore LOTOS behaviour trees, called *Goal-Oriented Execution*. In this technique, LOTOS specifications can be verified by means of deriving *traces*, i.e paths in the behaviour tree, satisfying certain properties. This is shown in Figure 3-1, where *B* represents the specification under verification, *P* is the trace property to be satisfied, and *t* is a trace derived from *B* and leading to *B'*, i.e. $B{=}t{\Rightarrow}B'$, such that *P(t)* holds.

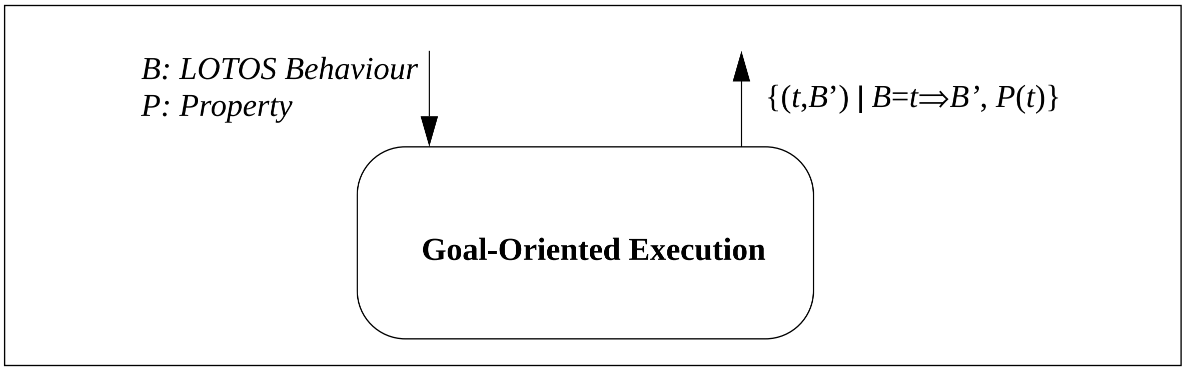


Figure 3-1 A Black Box View of Goal-Oriented Execution

To avoid the derivation of unwanted traces, and therefore to cope with the state explosion problem, the inference system is guided by static information, called *static derivation paths* (SDPs). An SDP locates where, in the abstract syntactic tree of the current behaviour, the given trace property can possibly hold. As a result, this inference system, called *guided-inference system*, executes only parts of the specification to generate the desired traces.

The derived traces are called *variable traces* because they may be associated with free variables and predicates. To assign values to these variables satisfying all predicates, or to determine that some predicates have no solution for any given values, the assistance of a *narrower* tool is needed. Narrowing [118] is a technique for finding solutions to a goal in abstract data types.

Goal-oriented execution is, therefore, composed of:

1- A *static analyzer*: determines the possible static derivation paths.
2- A *guided-inference system*: derives traces guided by SDPs.
3- A *narrower*: a tool to find solutions to a goal expressed as an abstract data type expression.

The overall structure of the goal-oriented execution is illustrated in Figure 3-2.

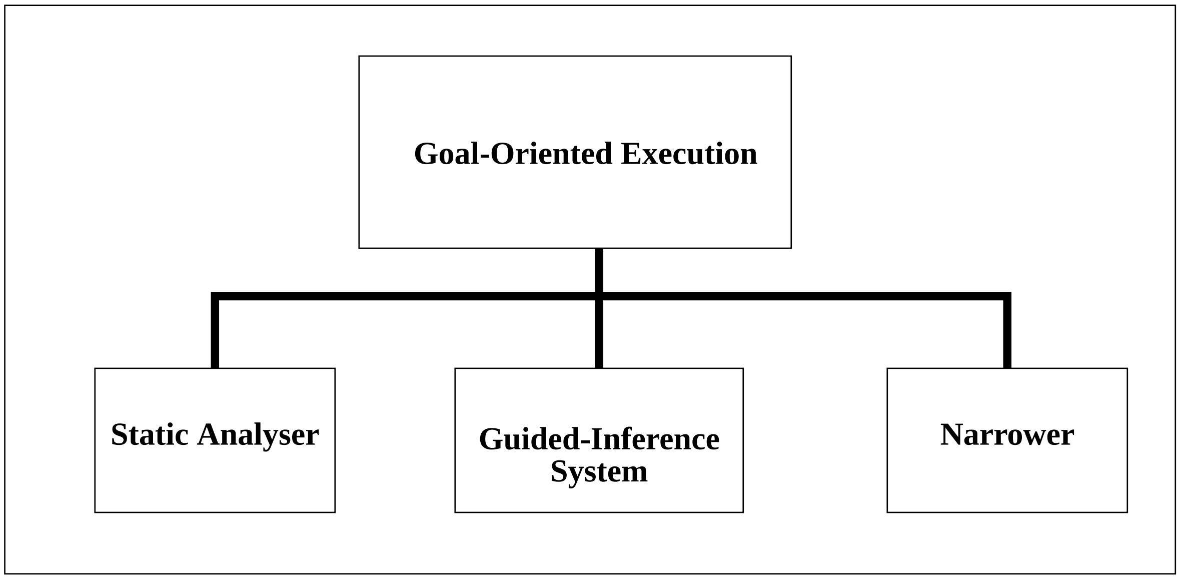


Figure 3-2 A Structural View of Goal-Oriented Execution

The remaining of this chapter is organized as follows. In the next section, we provide general definitions such as conventions and trace operations that will be used throughout the thesis. An inference system that define trace derivation is presented in section 3.2. In the following section,

section 3.4, the definition of the trace properties is given. The components of goal-oriented execution, namely static analyser, guided-inference system and the narrower, are explained in more details in section 3.5, 3.6 and 3.7 respectively. Finally, in section 3.8, we present the goal-oriented execution algorithm.

3.2 General Definitions

3.2.1 Conventions

The following conventions are used in this paper:

- A is the collection of all possible observable LOTOS action denotations.
- B is the collection of all possible LOTOS behaviour expressions.
- G is the collection of all possible LOTOS gates including the gate name of an action performed by an *exit* construct (δ).
- $B, B', B_i \in$ B, stand for LOTOS behaviour expressions.
- $g, g_i, h, h_i \in$ G, stand for gates or basic actions (actions with no events and no predicate).
- Lower case letters *a, b, c*, except **i**, stand for observable or unobservable actions, unless otherwise specified.
- **i** stands for an unobservable action.
- **i**/*a* stands for action *a* when *a* is hidden from (unobservable by) the environment.
- $\alpha(B)$ is the set of gates of all observable actions that appear in behaviour *B*.
- *name(a)* denotes the gate name of action *a*. *name(a)* = **i**, if *a* is an unobservable action.
- *card(a)* denotes the number of events offered by action *a*.
- $event_i(a)$ denotes the ith event of action *a*.
- *sort(E)* denotes the sort of event *E*.
- *pred(a)* denotes the associated predicate of action *a*.
- *rel(g, a)* stands for relabelling the gate name of action *a* by *g*.
- $(B)[g_1/h_1, ..., g_n/h_n]$ stands for a relabeled behaviour expression *B*, where the gate of each action with gate name h_i that *B* can perform is relabelled as g_i.
- $[t_1/x_1,...,t_n/x_n]\ B$ denotes the result of the replacement of all occurrences of variables $x_1,..,x_n$ in B by terms $t_1,...,t_n$ respectively.
- $\overline{V} = V_1,..,V_n$, is a list of free variables.
- $\overline{T} = T_1,..,T_n$, is a list of non-variable terms.
- *eval(*T*)* denotes the evaluation of the ADT expression T.
- $t(\overline{V})$ and $B(\overline{V})$ denote a trace and a behaviour with a list of free variables $\overline{V}$ respectively.
- $t_1 \Lambda t_2$ denotes the condition of matching term t_1 with term t_2, defined below.

- t_1/t_2 denotes the result of matching term t_1 with term t_2
- $a_1 \equiv a_2$ denotes a_1 *matches* a_2, defined in Table 2.1 of the previous chapter.
- $a_1 \uparrow a_2$ denotes the resulting action obtained from matching a_1 and a_2, also defined in Table 2.1.
- $a_1 \equiv_s a_2$ denotes a_1 *statically matches* a_2, defined below.
- $a_1 \uparrow_s a_2$ denotes the resulting action obtained from *statically* matching a_1 and a_2, also defined below.

3.2.2 Variable Actions and Matching

A variable action is an observable or an unobservable LOTOS action possibly prefixed by a guard. The following are some examples:

- *a!0*
- *a?X:nat[X>3]*
- *[Y>4]a!Y?X:nat[X>Y and X < 7]*
- **i**/(*a?X:nat[X>3]*)

Here are some definitions related to matching variable actions:

- $t_1 \Lambda t_2$ denotes the condition of matching term t_1 with term t_2, defined as:

$t_1 \Lambda t_2 = true$	if t_2 is a free variable and t_1/t_2
$t_1 \Lambda t_2 = true$	if t_1 is a free variable and t_2/t_1
$t_1 \Lambda t_2 = t_1 >\!<\!< t_2$	if t_1 and t_2 are not free variables. '>><<' is the narrowing operator that returns true only if there exist values for all free variables in t_1 and t_2 such that $eval(t_1)=eval(t_2)$. This operator is defined in more detail in section 3.7.3.

- $a_1 \equiv_s a_2$ denotes a_1 *statically matches* a_2, defined as follows:

$a_1 \equiv_s a_2$ if $name(a_1) = name(a_2) \neq \mathbf{i}$ and

$card(a_1) = card(a_2)$ and

$sort(event_i(a_1)) = sort(event_i(a_2))$ for $1 \leq i \leq card(a_1)$.

- $a_1 \uparrow_s a_2$ denotes the resulting action obtained from statically matching a_1 and a_2, defined in Table 3-1. For simplicity, the definition considers the case where actions a_1 and a_2 have one event. G_i represents a prefixed list of guards, P_i represents a postfixed list of predicates, and ^

stands for *and*.

Table 3-1 Statically Matching Variable Actions

Action a_1	Action a_2	Condition $a_1 \equiv_s a_2$	$a_1 \uparrow_s a_2$	Effect
$G_1\ g_1!E_1\ P_1$	$G_2\ g_2!E_2\ P_2$	$g_1 = g_2$ *sort*(E_1)=*sort*(E_2)	$G_3\ g_1!E_1\ P_3$	$G_3 = G_1 \wedge G_2$ $P_3 = (E_1\ \Lambda\ E_2) \wedge P_1 \wedge P_2$
$G_1\ g_1!E\ P_1$	$G_2\ g_2?X{:}t\ P_2$	$g_1 = g_2$ *sort*(E) = t	$G_3\ g_1!E\ P_3$	$G_3 = G_1 \wedge G_2$ $P_3 = (X\ \Lambda\ E) \wedge P_1 \wedge P_2$
$G_1\ g_1?X{:}t_1\ P_1$	$G_2\ g_2?Y{:}t_2\ P_2$	$g_1 = g_2$ $t_1 = t_2$	$G_3\ g_1!X\ P_3$	$G_3 = G_1 \wedge G_2$ $P_3 = (X\ \Lambda\ Y) \wedge P_1 \wedge P_2$

Note that a variable X in ?X:t or !X may or may not be free when static matching is applied, as we shall see in the next chapter.

This example demonstrates the above definitions where X_i denote free variables, Y_i denote non-free variables and T_i denote non-variable terms:

$$[Y_1>4]a!Y_1\ ?X_1{:}nat\ !T_1 \uparrow_s a!T_2\ ?Y_2{:}nat\ !T_2\ [Y_2 < T_2] =$$

$$[Y_1>4]a!Y_1\ !X_1\ !T_1\ [(Y_1 >\!\!><\!\!< T_2)\ and\ (true)\ and\ (T_1 >\!\!><\!\!< T_2)\ and\ (Y_2 < T_2)]$$

According to the third case in Table 3-1, the variable X_1 represents the effect of X_1/Y_2 in the result of the above example.

3.2.3 Variable Traces and Operations

A *variable trace* consists of a finite sequence of variable actions. Unlike a normal trace, a variable trace may contain, in addition to variables, unobservable actions. A variable trace is denoted as follows:

• $\langle\rangle$	An empty trace.
• $\langle a \rangle$	A trace containing only one variable action *a*.
• $\langle a_1, a_2 \rangle$	A trace containing two variable actions a_1 followed by a_2.
• $\langle a.t \rangle$	A trace containing the variable action *a* followed by the trace *t*.

For example $\langle a?X{:}nat,\ \mathbf{i}/c,\ [X>3]b?Y{:}bool!X[Y<X] \rangle$ is a variable trace.

The behaviour from which a trace *t* is derived, is denoted by *B(t)*. The definitions of some variable trace operators are given below. A subset of these trace operators for basic LOTOS can be found in [52][71].

- **Alphabet:** Denoted by $\alpha(t)$, is the set of all gates of the observable actions in t, defined as:
  1. $\alpha(\langle\rangle) = \varnothing$
  2. $\alpha(\langle a.t\rangle) = \alpha(t)$ if $name(a) = \mathbf{i}$
  3. $\alpha(\langle a.t\rangle) = \{name(a)\} \cup \alpha(t)$ if $name(a) \neq \mathbf{i}$

- **Projection:** The projection of a trace t on an alphabet $A \subseteq \alpha(B(t)) \cup \{\delta, \mathbf{i}\}$, denoted by $t\lfloor A$, is the trace t excluding all actions with gate names *not included* in A. It is defined as follows:
  1. $\langle\rangle\lfloor A = \langle\rangle$
  2. $\langle a.t\rangle\lfloor A = t\lfloor A$ if $name(a) \notin A$
  3. $\langle a.t\rangle\lfloor A = \langle a.t\lfloor A\rangle$ if $name(a) \in A$

- **Inverse Projection:** The inverse projection of a trace t on an alphabet $A \subseteq \alpha(B(t)) \cup \{\delta, \mathbf{i}\}$, denoted by $t\lceil A$, is the trace t excluding all actions *included* in A. This operator is defined as follows:
  1. $\langle\rangle\lceil A = \langle\rangle$
  2. $\langle a.t\rangle\lceil A = \langle a.t\lceil A\rangle$ if $name(a) \notin A$
  3. $\langle a.t\rangle\lceil A = t\lceil A$ if $name(a) \in A$

- **Concatenation:** The concatenation of two traces t_1 and t_2, denoted by $t_1\bullet t_2$, is the trace containing the action sequence of t_1 followed by the action sequence of t_2. The concatenation definition is:
  1. $\langle\rangle\bullet t = t$
  2. $\langle a.t_1\rangle\bullet t_2 = \langle a.(t_1\bullet t_2)\rangle$

- **Containment:** Denoted by *g in t*, is used to express the fact that the trace t contains an action with gate name g, and is defined as follows:
  1. g *in* $\langle a.t\rangle$ if $g = name(a)$
  2. g *in* $\langle a.t\rangle = g$ *in* t if $g \neq name(a)$

- **Hiding:** Denoted by $t\downarrow[g_1, ..., g_n]$, where each observable action a in t with $name(a) \in \{g_1, ..., g_n\}$ is hidden by replacing it with $\mathbf{i}/a$. This operator is defined as follows:
  1. $\langle\rangle\downarrow[g_1, ..., g_n] = \langle\rangle$
  2. $\langle a.t\rangle\downarrow[g_1, ..., g_n] = \langle \mathbf{i}/a.t\downarrow[g_1, ..., g_n]\rangle$ if $name(a) \in \{g_1, ..., g_n\}$
  3. $\langle a.t\rangle\downarrow[g_1, ..., g_n] = \langle a.t\downarrow[g_1, ..., g_n]\rangle$ if $name(a) \notin \{g_1, ..., g_n\}$

- **Relabeling:** Denoted by $t[g_1/h_1, ..., g_n/h_n]$, where each observable action a in t with name(a)=h_i, for (1≤i≤n), is replaced by $rel(g_i, a)$. This operator is defined as follows:
  1. $\langle\rangle[g_1/h_1, ..., g_n/h_n] = \langle\rangle$
  2. $\langle a.t\rangle[g_1/h_1, ..., g_n/h_n] = \langle rel(g_i,a).t[g_1/h_1, ..., g_n/h_n]\rangle$ if $name(a)=h_i \in \{h_1, ..., h_n\}$
  3. $\langle a.t\rangle[g_1/h_1, ..., g_n/h_n] = \langle a.t[g_1/h_1, ..., g_n/h_n]\rangle$ if $name(a) \notin \{h_1, ..., h_n\}$

- **Last:** Denoted by t^, is the last action in a nonempty trace t. Is is defined as follows:
  1. $\langle a\rangle\hat{} = a$
  2. $\langle a.t\rangle\hat{} = t\hat{}$ if $t \neq \langle\rangle$

- **Matching:** Denoted by t_1 *match* t_2. Is used to express the fact that t_1 statically matches t_2 with respect to the definitions given in Table 3-1. This operator is defined as:
  1. $\langle\rangle$ *match* $\langle\rangle$
  2. $\langle a_1.t_1\rangle$ *match* $\langle a_2.t_2\rangle = t_1$ *match* $\langle a_2.t_2\rangle$ if $name(a_1) = \mathbf{i}$
  3. $\langle a_1.t_1\rangle$ *match* $\langle a_2.t_2\rangle = \langle a_1.t_1\rangle$ *match* t_2 if $name(a_2) = \mathbf{i}$
  4. $\langle a_1.t_1\rangle$ *match* $\langle a_2.t_2\rangle = a_1 \equiv_s a_2$ and t_1 *match* t_2

- **Merging:** Denoted by $t_1 |\{A\}| t_2$, where $A \subseteq \alpha(B(t_1)) \cup \alpha(B(t_2)) \cup \{\delta\}$. This describes the set of variable traces resulting from composing two LOTOS processes, say P and Q, by means of the parallel composition operator, where t_1 and t_2 are variable traces generated by processes P and Q respectively. For example:

  $\langle \mathbf{i}/a?X{:}nat,\ [X>3]b?Y{:}nat[Y<X],\ c!Y,\ d\rangle$

  $|\{b,c\}|$

  $\langle b?Z{:}nat[Z>1],\ e?X{:}nat[X<Z],\ c!X\rangle =$

  $\{\langle \mathbf{i}/a?X{:}nat,\ [X>3]b?Y{:}nat[Y<X,\ Y>1],\ e?X{:}nat.[X<Y],\ [Y=X]c!Y,\ d\rangle\ \}$

  Note that in the resulting trace, variable Z is represented by variable X. The formal definition of this operator is as follows:
  1. $\langle a_1.t_1\rangle |\{A\}| t_2 = \langle a_1 . (t_1 |\{A\}| t_2) \rangle$ if $name(a_1) = \mathbf{i}$ *or* $name(a_1) \notin A$
  2. $t_1 |\{A\}| \langle a_2.t_2\rangle = \langle a_2 . (t_1 |\{A\}| t_2) \rangle$ if $name(a_2) = \mathbf{i}$ *or* $name(a_2) \notin A$
  3. $\langle a_1.t_1\rangle |\{A\}| \langle a_2.t_2\rangle = \langle a_1 \uparrow_s a_2 . (t_1 |\{A\}| t_2) \rangle$ if $name(a_1) = name(a_2) \in A$ *and* $a_1 \equiv_s a_2$
  4. $t_1 |\{A\}| t_2 = \langle\rangle$ otherwise

Note that non-determinism occurs when rule 1 and rule 2 can both be applied. For example,

$\langle a\rangle |\{c\}| \langle b\rangle = \{\langle a, b\rangle, \langle b, a\rangle\}.$

- **Selecting:** Two selection operators are defined, $\rho(t)$ and $\tau(t)$. $\rho(t)$ describes the ordered set of all guards and predicates in trace t, and $\tau(t)$ describes the trace t excluding all guards, predicates and sorts, also replacing all '?' with '!'. For example:

  $\rho(\langle a?X{:}nat,\ [X>3]b?Y{:}nat[Y<X,\ Y>1],\ e?X{:}nat[X<Y],\ [Y=X]c!Y,\ d\rangle\ ) =$

  $\{X_1>3,\ Y<X_1,\ Y>1,\ X_2<Y,\ Y=X_2\}$

  and

  $\tau(\langle a?X{:}nat,\ [X>3]b?Y{:}nat[Y<X,\ Y>1],\ e?X{:}nat[X<Y],\ [Y=X]c!Y,\ d\rangle\ ) =$

  $\langle a!X_1,\ b!Y,\ e!X_2,\ c!Y,\ d\rangle$

  Variable renaming conventions are used to guarantee uniqueness.

These operations can also be applied on non-variable trace, i.e. normal traces, by replacing $\equiv_s$ by $\equiv$ and $\uparrow_s$ by $\uparrow$. The defnitions of $\equiv$ and $\uparrow$ are given in chapter 2, Table 2.1.

3.3 Inference System for Relation $\rightarrow$

Since we are mainly dealing with trace generation, two basic trace relations are of our concern: relation $\rightarrow$ and relation $\Rightarrow$. They are defined as follows:

Let $a_i \in$ A $\cup$ {**i**} for all $1\leq i \leq n$, then relation $\rightarrow$ is defined as:

$B \xrightarrow{\langle a_1,\ldots,a_n\rangle} B'$ iff $\exists\ B_1, ..., B_n \in$ B with $B = B_1$, $B'=B_n$, and $B_i \xrightarrow{a_i} B_{i+1}$ for all $1\leq i \leq n-1$.

Let $a_i \in$ A for all $1\leq i \leq n$, then relation $\Rightarrow$ is defined as:

$B \overset{\langle a_1,\ldots,a_n\rangle}{\Longrightarrow} B'$ iff $\exists$ natural numbers $k_0,...,k_n$ with $B \xrightarrow{\langle \mathbf{i}^{k0},\ a_1,\ \mathbf{i}^{k1},\ldots,\ a_n,\ \mathbf{i}^{kn}\rangle} B'$.

$B = \langle\rangle \Rightarrow B'$ iff $B = B'$ or $\exists$ natural number n with

$B \text{---}\langle \mathbf{i}^n \rangle\rightarrow B'$.

Therefore, using the trace operations defined above, relation $\Rightarrow$ can be defined as:

$B = t\lceil\{\mathbf{i}\}\Rightarrow B'$ iff $B \text{---} t \rightarrow B'$.

The inference system for relation $B \text{---} a \rightarrow B'$ is given in chapter 2, Table 2.4. Here we provide an inference system that defines the relation $B \text{---}\langle a_1,\ldots,a_n\rangle\rightarrow B'$.

- **Internal Action Prefix**

$$\mathbf{i};\, B \text{---}\langle \mathbf{i} \rangle\rightarrow B \tag{1}$$

$$\frac{B_1 \text{---} t \rightarrow B_{11}}{\mathbf{i};\, B_1 \text{---}\langle \mathbf{i}\,.\, t\rangle\rightarrow B_{11}} \tag{2}$$

- **Observable Action Prefix**

$$g d_1 \ldots d_n [P];\, B \text{---}\langle g!v_1 \ldots !v_n \rangle\rightarrow [r_1/y_1,\ldots,r_m/y_m]B \tag{3}$$

$$\frac{[r_1/y_1,\ldots,r_m/y_m]B \text{---} t \rightarrow B'}{g d_1 \ldots d_n [P];\, B \text{---}\langle g!v_1 \ldots !v_n\,.\, t\rangle\rightarrow B'} \tag{4}$$

if eval($[r_1/y_1,\ldots,r_m/y_m]$ *P) = true, where*

$\{(r_1,y_1), \ldots, (r_m,y_m)\} =$

$\{(r,x) \mid \exists\, d_i \in \{d_1 \,..\, d_m\}$ *with* $d_i = ?x{:}s,\ r \in domain(s)\}$

$v_i = eval(r)$ *if* $d_i = !r$,

$v_i = eval(r)$ *if* $d_i = ?x{:}s$ *and* $r \in domain(s)$

- **Successful Termination**

$$\mathbf{exit}(E_1,\ldots,E_n) \text{---}\langle \delta!v_1 \ldots !v_n\rangle\rightarrow \mathbf{stop} \tag{5}$$

$v_i = eval(E_i)$, *if* E_i *is a term*

$v_i \in domain(s)$, *if* $E_i =$ ***any*** *s*

- **Choice**

$$\frac{B_1 \xrightarrow{t} B_{11}}{B_1 \;[]\; B_2 \xrightarrow{t} B_{11}} \tag{6}$$

$$\frac{B_2 \xrightarrow{t} B_{21}}{B_1 \;[]\; B_2 \xrightarrow{t} B_{21}} \tag{7}$$

- **Nested**

$$\frac{B \xrightarrow{t} B'}{(B) \xrightarrow{t} B'} \tag{8}$$

- **Guard**

$$\frac{B \xrightarrow{t} B',\ eval(P)=true}{[P]\text{->}B \xrightarrow{t} B'} \tag{9}$$

- **Local Definition**

$$\frac{[r_1/x_1, .., r_n/x_n]B \xrightarrow{t} B'}{\textbf{let}\ x_1{:}s_1=r_1, ..\ x_n{:}s_n=r_n\ \textbf{in}\ B \xrightarrow{t} B'} \tag{10}$$

- **Summation on Values**

$$\frac{[r/x]B \xrightarrow{t} B',\ r \in domain(s)}{\textbf{choice}\ x{:}s\ []\ B \xrightarrow{t} B'} \tag{11}$$

o **Hiding**

$$\frac{B \xrightarrow{t} B'}{\textbf{hide } GL \textbf{ in } B \xrightarrow{t\downarrow\{\mathrm{GL}\}} \textbf{hide } GL \textbf{ in } B'} \tag{12}$$

All actions in the trace generated by the hide operator with gate names in the list GL, are hidden.

o **Enabling**

$$\frac{B_1 \xrightarrow{t} B_{11},\ name(t^\wedge) \neq \delta}{B_1 >> \textbf{accept } x_1{:}s_1..x_n{:}s_n \textbf{ in } B_2 \xrightarrow{t} B_{11} >> \textbf{accept } x_1{:}s_1..x_n{:}s_n \textbf{ in } B_2} \tag{13}$$

$$\frac{B_1 \xrightarrow{t} B_{11},\ t^\wedge = \delta!v_1...!v_n}{B_1 >> \textbf{accept } x_1{:}s_1..x_n{:}s_n \textbf{ in } B_2 \xrightarrow{t\downarrow\{\delta\}} [v_1/x_1,...,v_n/x_n]B_2} \tag{14}$$

$$\frac{B_1 \xrightarrow{t_1} B_{11},\ [v_1/x_1,...,v_n/x_n]B_2 \xrightarrow{t_2} B_{21},\ t_1^\wedge = \delta!v_1...!v_n}{B_1 >> \textbf{accept } x_1{:}s_1..x_n{:}s_n \textbf{ in } B_2 \xrightarrow{(t_1\downarrow\{\delta\})\bullet t_2} B_{21}} \tag{15}$$

o **Disabling**

$$\frac{B_1 \xrightarrow{t_1} B_{11},\ name(t_1^\wedge) = \delta}{B_1 \ [> B_2 \xrightarrow{t_1} B_{11}} \tag{16}$$

$$\frac{B_1 \xrightarrow{t_1} B_{11},\ name(t_1^\wedge) \neq \delta}{B_1 \ [> B_2 \xrightarrow{t_1} B_{11} \ [> B_2} \tag{17}$$

$$\frac{B_1 \xrightarrow{t_1} B_{11},\ B_2 \xrightarrow{t_2} B_{21},\ name(t_1^\wedge) \neq \delta}{B_1 \ [> B_2 \xrightarrow{t_1 \bullet t_2} B_{21}} \tag{18}$$

- **Selected Synchronization**

$$\frac{B_1 \xrightarrow{t_1} B_{11},\ t_1\lfloor(\{S\}\cup\{\delta\}) = \langle\rangle}{B_1 \ |[S]|\ B_2 \xrightarrow{t_1} B_{11} \ |[S]|\ B_2} \tag{19}$$

$$\frac{B_2 \xrightarrow{t_2} B_{21},\ t_2\lfloor(\{S\}\cup\{\delta\}) = \langle\rangle}{B_1 \ |[S]|\ B_2 \xrightarrow{t_2} B_1 \ |[S]|\ B_{21}} \tag{20}$$

$$\frac{\begin{array}{c} B_1 \xrightarrow{t_1} B_{11},\ B_2 \xrightarrow{t_2} B_{21},\ t_1\lfloor(\{S\}\cup\{\delta\}) \neq \langle\rangle,\ t_2\lfloor(\{S\}\cup\{\delta\}) \neq \langle\rangle, \text{ and} \\ t_1\lfloor(\{S\}\cup\{\delta\})\ \mathit{match}\ t_2\lfloor(\{S\}\cup\{\delta\}) \end{array}}{B_1 \ |[S]|\ B_2 \xrightarrow{t_1 \ |\{S\}|\ t_2} B_{11} \ |[S]|\ B_{21}} \tag{21}$$

- **Interleave Parallelism**

$$\frac{B_1 \ |[]|\ B_2 \xrightarrow{t} B'}{B_1 \ |||\ B_2 \xrightarrow{t} B'} \tag{22}$$

The interleave operator is treated as the selected synchronization operator with an empty list of synchronization gates.

- **Full Synchronization**

$$\frac{B_1 \ |[\alpha(B_1) \cup \alpha(B_2)]|\ B_2 \xrightarrow{t} B'}{B_1 \ ||\ B_2 \xrightarrow{t} B'} \tag{23}$$

The full synchronization operator is treated as the selected synchronization operator with the list of synchronization gates composed with the alphabet of behaviours B_1 and B_2.

o **Relabeling**

$$\frac{B \xrightarrow{t} B'}{(B)[g_1/h_1, ..., g_n/h_n] \xrightarrow{t[g_1/h_1, ..., g_n/h_n]} (B')[g_1/h_1, ..., g_n/h_n]} \qquad (24)$$

o **Process Instantiation**

$$\frac{\exists p[h_1,\ldots,h_n](x_1{:}s_1..x_m{:}s_m) := B,\ ([r_1/x_1,...,r_m/x_m]B)[g_1/h_1,..., g_n/h_n] \xrightarrow{t} B'}{p[g_1,\ldots,g_n]\ (r_1,...,r_m) \xrightarrow{t} B'} \qquad (25)$$

3.4 Trace Properties

The trace properties allowed by our method are defined by a *targeted action* or an *ordered set of actions*, and a *restricted gate set*. Properties on values can also be expressed by associating constraints to the targeted actions. We have modeled these properties using relations $\Rightarrow^{+}$ and $\Rightarrow^{\times}$ defined below. In the following, *a'* identifies an action that matches action *a*, i.e. $a' \equiv a$:

1- $(a,B)/G =t\Rightarrow^{+} B'$, where $name(a) \in \alpha(B)$ and $G \subseteq \alpha(B) \cup \{\delta\}$, defines the derivation of behaviour *B* on a trace *t* leading to a targeted action *a*' without passing through any other action with gate name in *G*. Note that, if $name(a) \notin G$, then trace *t* may contain any number of actions with gate name *name(a)*.

2- $(\langle a_1,\ldots,a_n\rangle, B)/G =t\Rightarrow^{\times} B'$, $name(a_i) \in \alpha(B)$ and where $G \subseteq \alpha(B) \cup \{\delta\}$, defines the derivation of behaviour *B* on a trace *t*, such that *t* contains a predetermined series of actions $\{a_1',\ldots,a_n'\}$, not necessarily contiguously, without passing by any other action with gate name in $G \cup \alpha(\langle a_1,\ldots,a_n\rangle)$. This implies that trace *t* cannot have any other appearances of actions with gate names in $\alpha(\langle a_1,\ldots,a_n\rangle)$, and does not necessarily terminate with action a_n'.

For example, let *B* be the behaviour given in Figure 3-3, then

$(d?U{:}nat, B)/\{c,d\} =t\Rightarrow^{+} B'$ holds with

$t = \langle e!2, b!2\ !1, d!1\rangle$ $\quad$ $B' = (c!2\ !1\ ;\ \mathbf{stop}\ |[b]|\ f!2\ !1;\ \mathbf{stop})$

$t = \langle e!3, b!3\ !1, d!1\rangle$ $\quad$ $B' = (c!3\ !1\ ;\ \mathbf{stop}\ |[b]|\ f!3\ !1;\ \mathbf{stop})$

$t = \langle e!3, b!3\ !2, d!2\rangle$ $\quad$ $B' = (c!3\ !2\ ;\ \mathbf{stop}\ |[b]|\ f!3\ !2;\ \mathbf{stop})$

and

(⟨e?U1:nat[U1<3], d?U:nat⟩, B)/{c} =t$\Rightarrow^{\times}$ *B'* holds with

t = ⟨e!2, b!2 !1, d!1⟩ *B' = (c!2 !1 ;* **stop** *|[b]| f!2 !1;* **stop***)*

t = ⟨e!2, b!2 !1, d!1, f!2 !1⟩ *B' = (c!2 !1 ;* ***stop*** *|[b]|* ***stop****)*

while

(d!3:nat, B)/{c,d} =t$\Rightarrow^{+}$ *B'* does not hold because the only action that matches *d!3:nat* in the specification without going through an action with gate name *c* is *d!Z* with *Z*=3. For this to hold, action *b!Y?Z:nat[Z>=1]* must match *b!W?X:nat[X<W]* with *X=Z=3*, *W=Y* and *[3 < W]*. This forces a contradiction with the predicate in the (hidden) action *g?Y:nat[Y<4]*, i.e. predicate *[3 < W]* and predicate *[Y<4]* cannot be both true when *W=Y*. Therefore, there is no feasible trace *t* that satisfies the initial relation. And

(d?Y:nat, B)/{b} =t$\Rightarrow^{+}$ *B'* does not hold because there exists no trace *t* from *B* that lead to an action that matches *d?Y:nat* without passing by an action with gate name *b*.

The following abbreviations for action denotations and restricted sets used by relations $\Rightarrow^{+}$ and $\Rightarrow^{\times}$ are also be supported:

- '*' represents any action, any number of events, or a restriction gate set of all possible gates, depending on the context.
- '-' any action gate name or any event sort, also depending on the context.
- a restriction gate set of all possible gates can also be represented as *{*}*.

The following are some examples:

- *'a*'* represents an action on gate *a* with zero or more events
- *'- ?X:- ?Y:-'* represents any action with exactly two events of any sort.
- *'- ?X:Nat ?Y:- *'* represents any action with at least two events where the first event must be of sort *Nat*.

These abbreviations are only an implementation extension and therefore are not considered in the theory.

Note that internal actions are not expressible by the user since they are not part of execution traces. Internal actions involved in derived traces can be viewed before they are removed. See section 3.8.

B=

(a; c?V:nat; b!V!0; d!(V+1); **stop**
[]
e?W:nat; ([W>1] -> b!W?X:nat[X<W]; c!W!X; **stop***))*
|[b]|
hide *g* **in** *(g?Y:nat[Y<4]; b!Y?Z:nat[Z>=1]; d!Z; f!Y!Z;* **stop***)*

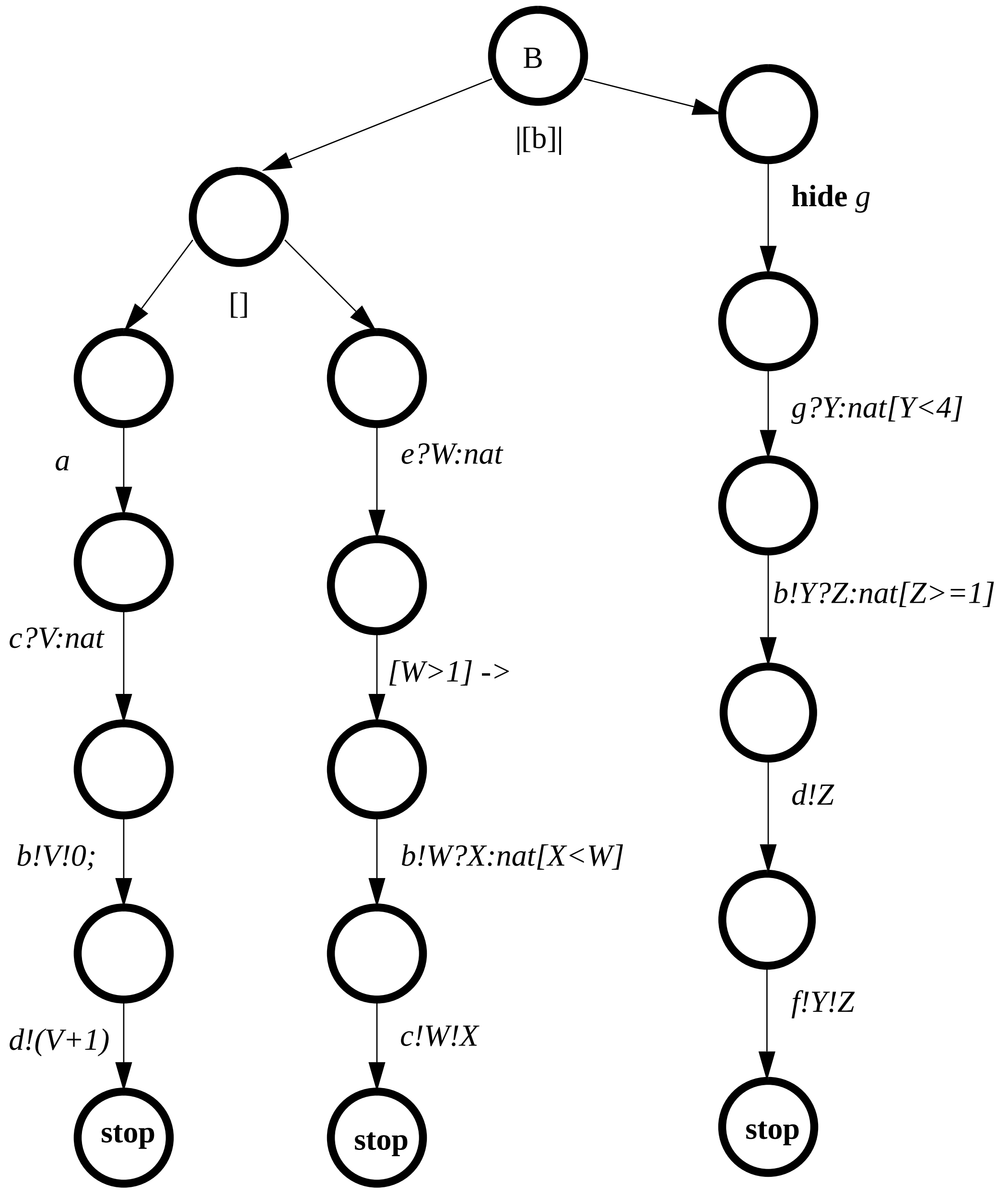


Figure 3-3 A behaviour and its Abstract Syntactic Tree

The formal definition of relations $\Rightarrow^+$ and $\Rightarrow^\times$ is as follows:

1- $(a, B)/G =\langle b_1,\ldots,b_n\rangle\Rightarrow^+ B'$, iff $B =\langle b_1,\ldots,b_n\rangle\Rightarrow B'$ such that $name(b_i) \notin G$ for $1 \le i \le n-1$, and $b_n \equiv a$. This implies that if $name(a) \in G$, then only action b_n in trace $\langle b_1,\ldots,b_n\rangle$ has gate name $name(a)$.

2- $(\langle a_1,\ldots,a_n\rangle, B)/A =\langle b_1,\ldots,b_m\rangle\Rightarrow^\times B'$, iff $B =\langle b_1,\ldots,b_m\rangle\Rightarrow B'$ such that $\langle b_1,\ldots,b_m\rangle\lfloor\alpha(\langle a_1,\ldots,a_n\rangle)$ *match* $\langle a_1,\ldots,a_n\rangle$, and $\forall d \in \alpha(\langle b_1,\ldots,b_m\rangle\lceil\alpha(\langle a_1,\ldots,a_n\rangle)$, $d \notin G$.
An alternative recursive definition of relation $\Rightarrow^\times$ is:
$(\langle a.t_1\rangle, B)/G =t_2\Rightarrow^\times B'$ iff $(a, B)/(G \cup \alpha(\langle a.t_1\rangle))=t_{11}\Rightarrow^+ B_1$, $(t_1, B_1)/(G \cup name(a)) =t_{12}\Rightarrow^\times B'$, and $t_2 = t_{11}\bullet t_{12}$.
$(\langle\rangle, B)/G =t\Rightarrow^\times B'$ iff $B =t\Rightarrow B'$ such that $\forall d \in \alpha(t)$, $d \notin G$. Note that t can be $\langle\rangle$.

Since relation $\Rightarrow^\times$ can be defined recursively using relation $\Rightarrow^+$, the main functionality of goal-oriented execution is then to provide an efficient implementation for relation $\Rightarrow^+$. Following the notation of Figure 3-1, $B=t\Rightarrow B'$, where t leads to a' without passing through actions with gates in G, is written $(a,B)/G =t\Rightarrow^+ B'$. See Figure 3-4.

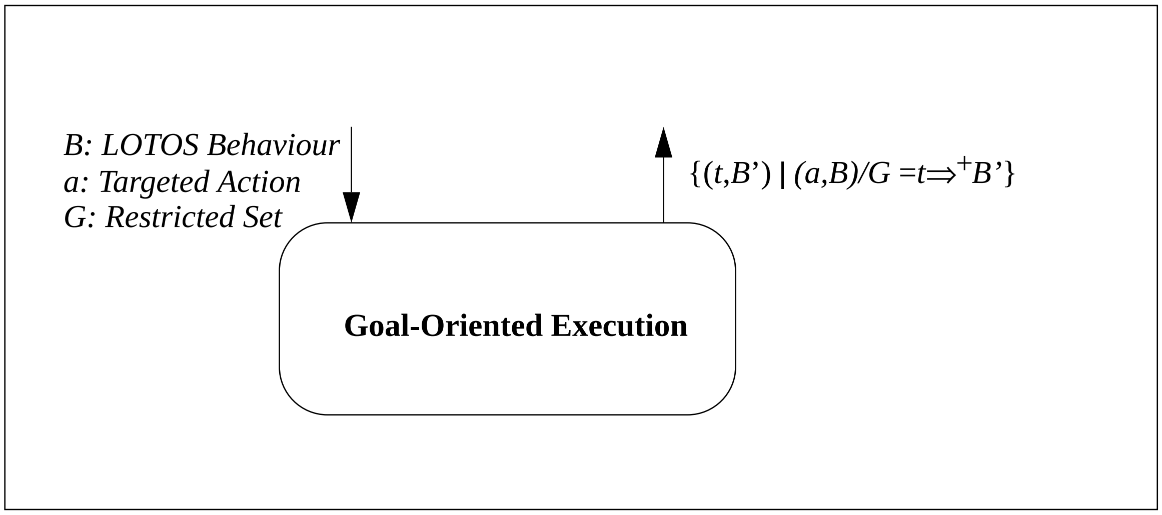


Figure 3-4 Goal-Oriented Execution with relation $\Rightarrow^+$

One way of implementing relation $\Rightarrow^+$ is to derive all possible traces using the inference system defined in section 3.3, and to keep only those satisfying the given property. Computationally, this is usually infeasible. Our aim in this thesis is to provide an efficient implementation for relation $\Rightarrow^+$.

To accomplish our goal, two other relations are defined, namely $\rightarrow^{+}$ and $\rightarrow^{\times}$. These relations, unlike relations $\Rightarrow^{+}$ and $\Rightarrow^{\times}$, are applied on variable traces that may contain unobservable actions. Their informal definition is given below where *a'* identifies an action that matches action *a* using variable matching relation $a' \equiv_s a$:

1- $(a,B)/G \xrightarrow{t(\overline{V})}{}^{+} B'(\overline{V})$, where $name(a) \in \alpha(B)$ and $G \subseteq \alpha(B) \cup \{\delta\}$, defines the derivation of behaviour *B* on a variable trace $t(\overline{V})$ leading to a targeted action $a' \uparrow_s a$ without passing through any other action with gate name in *G*. Note that synchronization between actions is also done using static matching $\uparrow_s$.

2- $(\langle a_1,\ldots,a_n\rangle, B)/G \xrightarrow{t(\overline{V})}{}^{\times} B'(\overline{V})$, $name(a_i) \in \alpha(B)$ and where $G \subseteq \alpha(B) \cup \{\delta\}$, defines the derivation of behaviour *B* on a variable trace *t*, such that *t* contains a predetermined series of actions $\{a_1' \uparrow_s a_2,\ldots,a_n' \uparrow_s a_n\}$, not necessarily contiguously, without passing by any other action with gate name in $G \cup \alpha(\langle a_1,\ldots,a_n\rangle)$.

For example, let B be the behaviour given in Figure 3-3, then

$(d?U{:}nat, B)/\{c,d\} \xrightarrow{t}{}^{+} B'$ holds with

t = ⟨ *e?W/Y:nat,*

i/*g?W/Y:nat[W/Y < 4],*

[W/Y >1]b!W/Y ?X/Z/U:nat[X/Z/U <W/Y, X/Z/U>=1],

d!X/Z/U⟩

B' = (c!W/Y !X/Z/U ; **stop** *|[b]| f!W/Y !X/Z/U;* **stop***)*

and

$(\langle e!4, d?U{:}nat\rangle, B)/\{c,f\} \xrightarrow{t}{}^{\times} B'$ holds with

t = ⟨ *e!W/Y/4,*

i/*g?W/Y/4:nat[W/Y/4 < 4],*

[W/Y/4>1]b!W/Y/4 ?X/Z/U:nat[X/Z/U < W/Y/4, X/Z/U>=1],

d!X/Z/U⟩

B' = (c!W/Y/4 !X/Z/U ; **stop** *|[b]| f!W/Y/4 !X/Z/U;* **stop***)*

while

$(d?Y{:}nat, B)/\{b\} \text{—}t\rightarrow^{+} B'$ does not hold because there exists no variable trace *t* from *B* that leads to an action that statically matches *d?Y:nat* without passing by an action with gate name *b*.

In sections 3.5 and 3.6, we present the derivation methodology for the variable traces under relations $\rightarrow^{+}$ and $\rightarrow^{\times}$. Section 3.8 demonstrates an implementation strategy that uses the narrowing technique, described in section 3.7, to map relations $\rightarrow^{+}$ and $\rightarrow^{\times}$ into relations $\Rightarrow^{+}$ and $\Rightarrow^{\times}$ respectively.

3.5 Static Derivation Paths

A *static derivation path* (SDP) of an action *a* in a given behaviour *B* is a sequence identifying a path in the abstract syntactic tree of *B* leading to an action *a'* where $a' \equiv_S a$. This path reflects the *directed* traversal of the operators composing the behaviour. As mentioned earlier, SDPs guide the inference system towards executing a part of the specification where the desired sequence of actions can be found. They provide a static analysis of the specification, which is necessary to prevent the inference engine from attempting to derive the full behaviour tree of the specification, often leading to the state space explosion problem. Consider for example the following very simple behaviour expression:

B = (a; b; c; **stop** ||| *d; c; f;* **stop***) [] g; h;* **stop**

Obviously, if we are looking for an execution trace leading to action *h*, the left sub-expression of the operator *[]* does not need to be explored. An SDP for such a goal would instruct the inference engine to direct itself immediately to the right-hand-side of the operator *[]*. Thus an SDP encodes structural information of the specification, which yields information on the direction where evaluation must proceed (left or right) for binary operators.

A static derivation path has the following form:

• []	An empty path.
• [*e*]	A path containing only one element *e*.
• [e_1,e_2]	A path containing two elements, e_1 followed by e_2.
• [*e.s*]	A path containing the element *e* followed by the path *s*.

An element of an SDP is a symbol identifying the type of the current behaviour construct in the abstract syntactic tree. The symbols are names chosen after the LOTOS operators they represent (e.g. *choice*, *guard*, *nested*). If the behaviour is involved in a binary operator, i.e. |||, |[G]|, ||, [], [> or >>, then branches for the left and right behaviour of the operator are identified by the symbol *left* and *right* respectively, preceded by the symbol ^. The following is the BNF for an element of an SDP identified by <*sdp_element*>:

<*sdp_element*>	*::=* <*unary_element*> | <*binary_element*>
<*unary_element*>	*::=* <*unary_operator*>
<*binary_element*>	*::=* <*binary_operator*> ^ <*direction*>
<*unary_operator*>	*::= prefix* | *exit* | *guard* | *let* | *chval* | *nested* | *hide* | *relabel* | *instance*
<*binary_operator*>	*::= choice* | *parallel* | *disable* | *enable*
<*direction*>	*::= left* | *right*

The elements corresponding to the *relabel* and the *instance* operators will contain other information related to gate relabeling that is discussed in section 4.1.1 of the next chapter.

Consider the LOTOS specification and its abstract syntactic tree given in Figure 3-5 and Figure 3-6 respectively. A static derivation path for the action *output* in the behaviour *producer_consumer[input, output]* would be [*instance, relabel, hide, parallel^left, nested, parallel^right, instance, relabel, prefix, prefix*]. It identifies the action *output* to be the action *a* in the process *consumer[c,a]* following the above path. It is indicated by a thick line in Figure 3-6. Process instantiations are done by using *static relabeling*. Although relabeling must be done dynamically from the operational semantics point of view, at this point, the only concern is *where* in a behaviour expression a given action may be found, and not *how* it is derived. More on static and dynamic relabeling is said in the next chapter.

```
specification producer_consumer[g1,g2]:noexit:=
behaviour
    hide g11,g22 in
       (producer[g1,g11] ||| consumer[g22,g2])
       |[g11,g22]|
       channel[g11,g22]
    where
process producer[g,p] :noexit
   g;p;producer[g,p]
endproc
process consumer[c,a] :noexit
c; a; consumer[c,a]
endproc
process channel[r,s] :noexit
   r;s;channel[r,s]
endproc
endspec
```

Figure 3-5 A LOTOS Specification

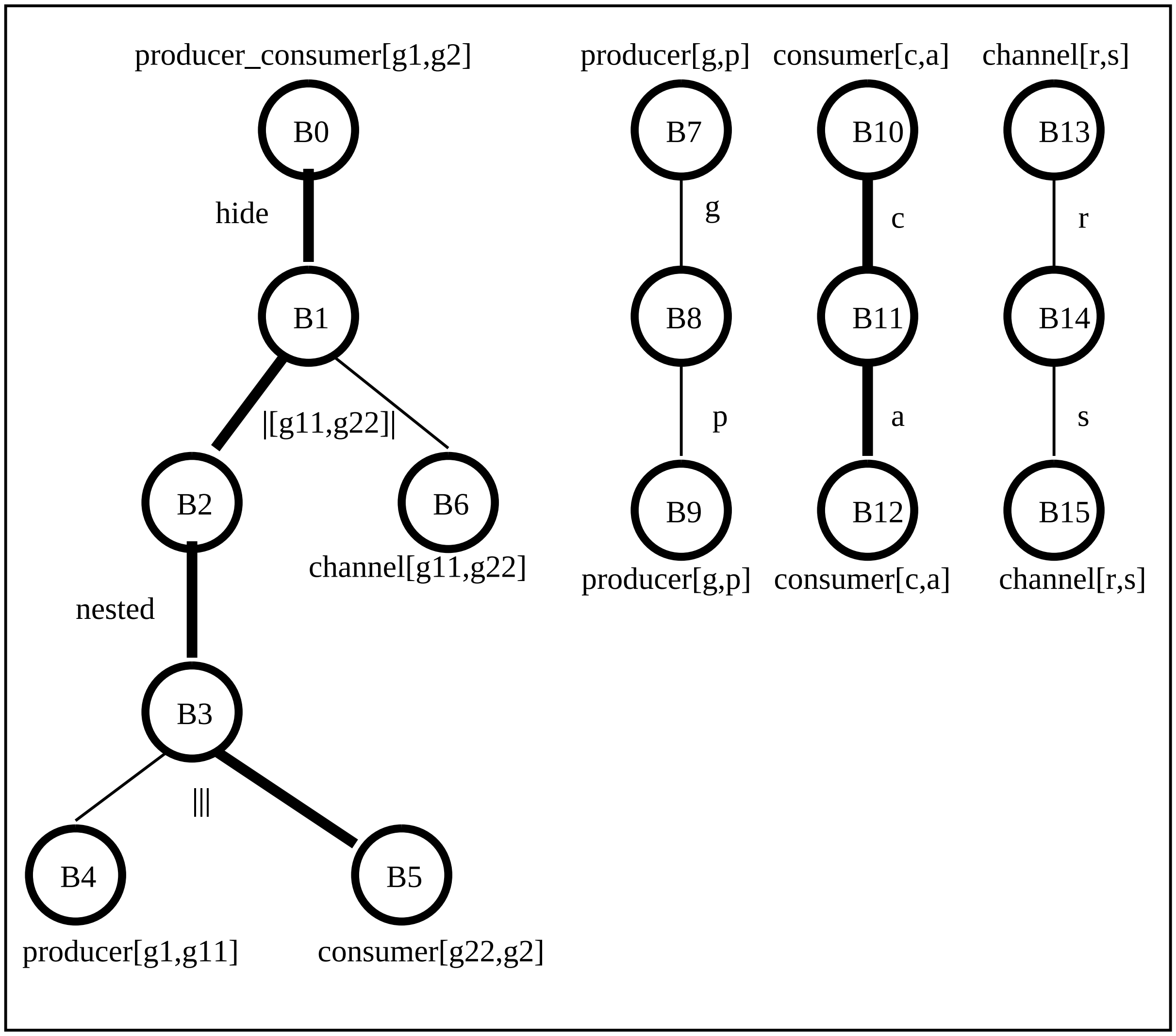


Figure 3-6 An SDP in the Abstract Syntactic Tree of Figure 3-5

Static derivation paths are generated using the function $\Sigma : A \times B \times G^* \rightarrow X$ where

- A is the collection of all possible observable LOTOS action denotations
- B is the collection of all possible LOTOS behaviour expressions
- G is the collection of all possible LOTOS gates including δ
- X is the collection of all possible static derivation path structures

$\Sigma(a,B,G)$, where $name(a) \in \alpha(B)$ and $G \subseteq \alpha(B) \cup \{\delta\}$, derives the set of SDPs in behaviour B leading to all actions a_i, such $a_i \equiv_s a$, without passing through any other action with gate name in G. For example, considering the behaviour B given in Figure 3-3:

$\Sigma(d?U{:}nat, \mathrm{B}, \{c\}) = \{\ [parallel\text{^}right,\ hide,\ nested,\ prefix,\ prefix,\ prefix]\ \}$

which leads to action *d!Z*. Note that the SDP, [*parallel^left*, *nested*, *choice^left*, *prefix*, *prefix*, *prefix*], which leads to action *d!(V+1)*, with $d!(V+1) \equiv_S d?U{:}nat$, is excluded since it passes through action *c?V:nat* that has gate name in {*c*}.

3.6 Guided-Inference System

The next step in goal-oriented execution is to provide an efficiently computable definition for the relation $(a, B)/G \xrightarrow{t(\overline{V})}{}^{+} B'(\overline{V})$. This was accomplished by means of *guided-inference system*, where the variable trace generation is directed by the static derivation paths.

To do so, the relation $(a, B)/G \xrightarrow{t(\overline{V})}{}^{+} B'(\overline{V})$ is redefined as:

$(a, sdp, B)/G \xrightarrow{t'(\overline{V})}{}^{+} B'(\overline{V})$, where $sdp \in \Sigma(a,B,G)$.

The SDPs are generated when needed by the inference system, see Figure 3-11. Consider the following situations:

- If the targeted action was found statically on the right of an enable operator B1 >> B2, namely in B2, then the inference system will be directed to the right, giving a trace from B2. To comply with the semantics of LOTOS, another trace from B1 leading to δ needs to prefix the original trace. In this case another application of $\rightarrow^{+}$ is needed, and yet another SDP.
- A more complicated situation occurs in parallel constructs. Suppose the inference rule were directed to generate a trace t_1 from B_1 in $B_1\ |[S]|\ B_2$. Again, to comply with LOTOS semantics, another trace t_2 from B_2 is needed having the following property: “all actions in t_1 with gate names in S must match actions in t_2 and with identical order.” More formally $t_1 \lfloor (\{S\} \cup \{\delta\})$ *match* $t_2 \lfloor (\{S\} \cup \{\delta\})$. We also have to keep in mind the original characteristics of the resulting trace.

There are other situations where the inference system needs a different application of $\rightarrow^{+}$. These situations are explained more in detail in the chapter 5, where the formal definition of *guided-inference system* is given. Note also that the SDPs may not be executable by the inference system. For instance, although Σ(*a, b;***exit** || *a;***exit**, {}) = {[*parallel^right.prefix*]}, this cannot be executed due to lack of synchronization.

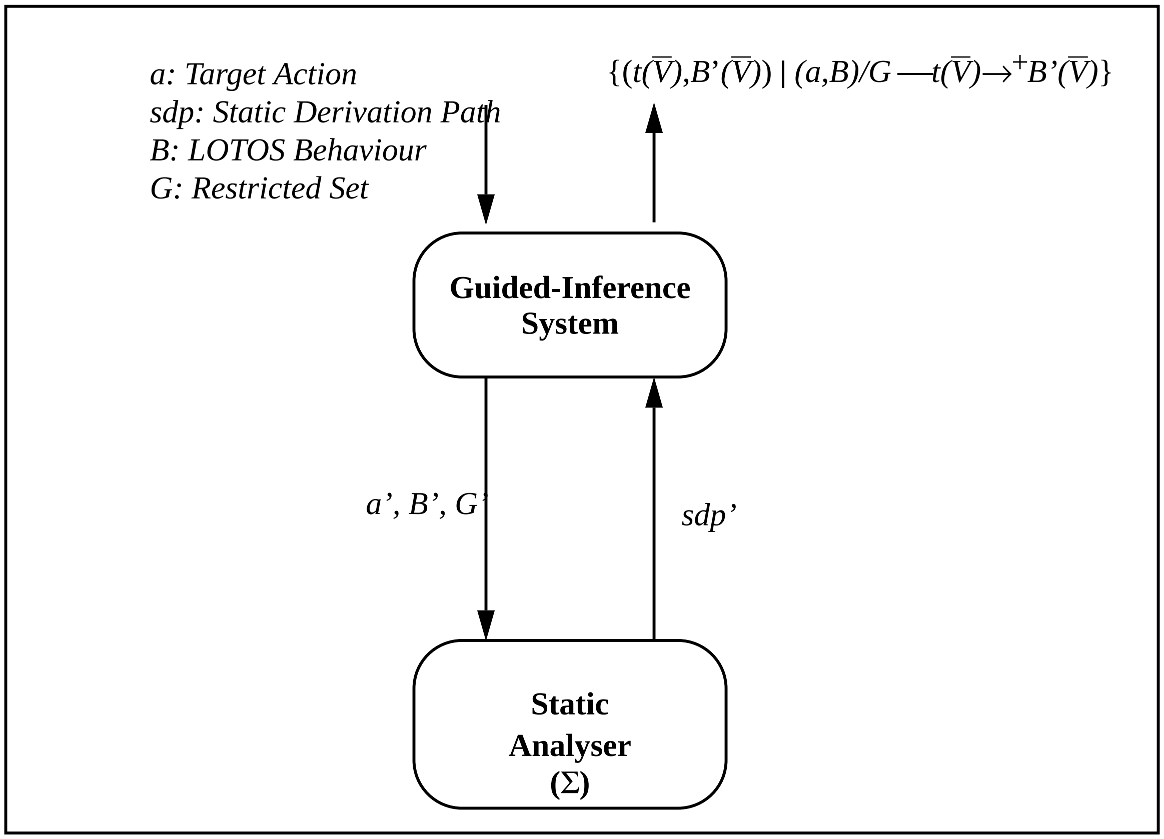

a: Target Action
sdp: Static Derivation Path
B: LOTOS Behaviour
G: Restricted Set
$\{(t(\overline{V}),B'(\overline{V})) \mid (a,B)/G \xrightarrow{t(\overline{V})}{}^{+} B'(\overline{V})\}$
Guided-Inference System
a’, B’, G’
sdp’
Static Analyser (Σ)

Figure 3-7 Guided-Inference System

The guided-inference system is consistent with the usual inference system, but differs from it in the following respects:

1- defines the derivation of a behaviour *B* not only on single actions, but also on traces
2- does not describe all possible derivations of a behaviour *B* in general, but only those satisfying a given property
3- SDPs are generated *on demand* by the inference system
4- derived traces may contain free variables and unobservable actions, i.e. variable traces.

3.7 Rewriting and Narrowing ADT expressions of LOTOS

3.7.1 Introduction

We have developed an algorithm for transforming abstract data type equations into a rewriting rules evaluator and narrower engine with considerable performance efficiency.

A term rewriting rule system is a set of directed equations used as a non-deterministic pattern-directed program that returns as output a simplified term equal to a given input term [77]. For suitably written LOTOS abstract data type equations, an equivalent term rewriting system can be found by simply orienting the equations. Earlier LOTOS interpreters [44] used an inner-most left to right rewriting rule strategy to execute the equations of LOTOS data types. In this section we discuss a new rewriting technique that (1) provides greater performance efficiency and, more importantly, (2) can be used to generate solutions to a goal.

3.7.2 Term Rewriting Systems

In this section we briefly review the basic notations and terminology for term rewriting systems. Surveys of this topic can be found in [34][77].

Let F be a set of operators with fixed arity, V a finite set of variables, A *term* is defined as either a variable from V, which is considered to be a *universal quantifier*, or $f(t_1,...,t_k)$, where $f \in F$ has arity k and t_i are *subterms*. We define $\tau(F,V)$ to be the set of all terms over F and V, and $\nu(t)$ to be the set of variables in term t. A term t with $\nu(t) = \emptyset$ is called *ground*.

A *substitution* σ is a function from V to $\tau(F,V)$. The domain of σ, denoted by $D(\sigma)$, is $\{X| \sigma(X) \neq X\}$. The term $t\sigma$ represents the term obtained by replacing the variables of t by their image under σ. A substitution σ is *as general* as a substitution ρ if there exists a substitution ω such that $\sigma\omega = \rho$.

A term t *matches* (or is an *instance* of) a term s if $t = s\sigma$ for some substitution σ. A term t *unifies* a term s if $t\sigma = s\sigma$ for some substitution σ.

A *term rewriting rule* (TRR), $l \rightarrow r$, is an oriented equation between terms. A *term rewriting system* (TRS) is a finite set of TRRs. For a given TRS R, the rewrite relation $\rightarrow_R$ replaces any subterm that is an instance $l\sigma$ of the left-hand side l by the corresponding instance $r\sigma$ of the right-hand side r of a TRR $l \rightarrow r$ in R. The relation $s \rightarrow_R t$ holds if s rewrites to t in one step under R, and the relation $s \rightarrow^*{}_R t$ holds if s rewrites to t in zero or more steps under R. We also say that t is *derivable* from s. The relation $s{\downarrow_R}t$ holds if s and t *join*; i.e if $s \rightarrow^*_R w$ and $t \rightarrow^*{}_R w$ for some term w. A term s is *irreducible,* or in *normal form,* if there is no term t such that $s \rightarrow_R t$.

A term rewriting relation $\rightarrow_R$ is *terminating* or *noetherian* if there is no infinite chain of rewrites: $t_1 \rightarrow_R t_2 \rightarrow_R ...$, and it is *confluent* if whenever two terms, s and t, are derivable from term u, then a term v is derivable from both s and t, i.e. if $u \rightarrow^*_R s$ and $u \rightarrow^*{}_R t$ then $s \rightarrow^*_R v$ and $t \rightarrow^*{}_R v$ for some term v. If a TRS is both terminating and confluent it is said to be *convergent*. Unfortunately, it is undecidable whether an arbitrary TRS terminates [76]. However, a number of methods have been proposed that prove termination in particular cases [87].

A set of equations E can be transformed into a term rewriting system R using the following technique suggested in [77][87]: for every equation $s = t$ in E, choose non-deterministically one of the following:

1- If $\nu(s) \subseteq \nu(t)$, put $t \rightarrow s$ in R.
2- If $\nu(t) \subseteq \nu(s)$, put $s \rightarrow t$ in R.

As mentioned above, there is no guarantee that R is convergent.

For example, in Figure 3-8 we provide a definition for type *NaturalPlus* that contains one sort *nat* for natural numbers and the declaration of some operators and their equations. A term rewriting system for such equations is given in Figure 3-8.

```
library Boolean endlib

type NaturalPlus is Boolean
      sorts nat
       opns
        0       :        -> nat
        succ    : nat    -> nat
        _++_    : Bool, Bool    -> Bool
        _<_     : nat, nat      -> Bool
        _==_    : nat, nat      -> Bool
        _>=_    : nat, nat      -> Bool
        _-_     : nat, nat      -> nat
        _mod_   : nat, nat      -> nat
       eqns
        forall C:Bool, M,N:nat
        ofsort Bool
                false ++ C                = C;
                true                      = true ++ C;

                succ(M) < succ(N)         = M < N;
                true                      = 0 < succ(N);
                false                     = M < 0;

                succ(M) == succ(N)        = M == N;
                0 == 0                    = true;
                false                     = 0 == succ(M);
                succ(M) == 0              = false;

                M >= N                    = (N<M) ++ (M==N);

        ofsort nat

                M - 0                  = M;
                succ(M) - succ(N)      = M - N;
                0                      = 0 - M;

          (M >= N) =>
                M mod N                = (M-N) mod N;
          (M < N)  =>
                M                      = M mod N;
endtype
```

Figure 3-8 An Abstract Data Type

```
false ++ C                  → C;
true ++ C                   → true;

succ(M)   < succ(N)         → M < N;
0         < succ(M)         → true;
M         < 0               → false;

succ(M)   == succ(N)        → M == N;
0         == 0              → true;
0         == succ(M)        → false;
succ(M)   == 0              → false;

M         >= N              → (N < M) ++ (M==N);


M         - 0               → M;
succ(M)   - succ(N)         → M - N;
0         - M               → 0;

(M >= N) => M mod N         → (M - N) mod N;
(M < N) => M mod N          → M;
```

Figure 3-9 A Term Rewriting System

3.7.3 Narrowing: Equation Solving using Term Rewriting Systems

As mentioned earlier, goal-oriented execution resolves predicates in the derived variable traces by finding proper values to all free variables. The narrowing techniques provide such a facility. In general, a narrower attempts to:

1- find values for the variables in a goal $s = t$ for which equality holds. More formally, finding a substitution σ such that $s\sigma \rightarrow^* u$ and $t\sigma \rightarrow {}^* u$ for some term u.

2- detect when equality is unsatisfiable.

Narrowing can be best implemented using a combination of *logic programming* and *functional programming* [34][35][36][153]. For example, in the case of the ADT of Figure 3-8, a goal of the form:

X = (succ(succ(0)) >= succ(0))

can be solved by *rewriting* the right-hand side of the goal producing a solution:

X = true.

while a goal of the form:

true = (X >= succ(0))

requires *equation solving* to produce values for X that satisfy the equation. Rewriting corresponds to the functional programming capability, while equation solving corresponds to the logic programming capability.

Narrowing Approach

Our current ADT interpreter, called SVELDA [44], evaluates (or rewrites) a given term using the internal form representation of the abstract data type equations. These equations are oriented as rewriting rules where renaming and parametrization are resolved. The new interpreter, which we call ERNAL (An Engine to Rewrite and Narrow the ADTs of LOTOS), transforms the internal representation of the term rewriting rules into an evaluator/narrower engine. See Figure 3-10.

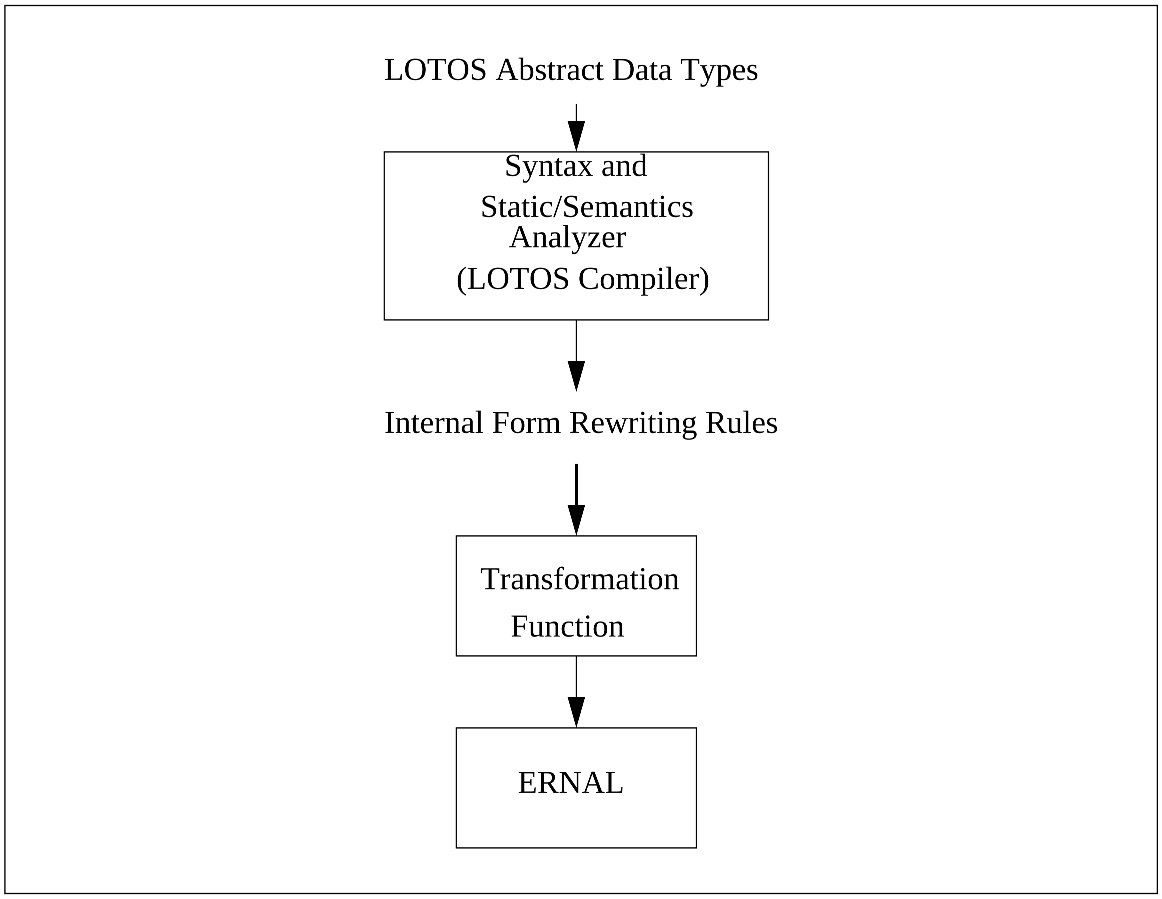

LOTOS Abstract Data Types
Syntax and Static/Semantics Analyzer (LOTOS Compiler)
Internal Form Rewriting Rules
Transformation Function
ERNAL

Figure 3-10 ERNAL’s Structure

ERNAL, the automatically generated engine, is a set of Prolog clauses. Since overloading is allowed in ACT ONE, all arguments are represented in internal form where overloading is resolved by assuring that all operators and sorts have unique internal names. The user interface of ERNAL consists of an infix operator '>><<' of the form:

EXP1:S >><< RES

where *RES* is the resulting evaluation of the ADT expression *EXP* of sort *S*.

The main characteristics of ERNAL are the following:

1- It can be used as a rewriting system. For example:

$((succ^7(0) \ mod \ succ^5(0)):nat >><< X) \qquad \Rightarrow X = succ^2(0)$

This expresses the fact that 7 modulo 5 is 2.

2- It can be used as a narrower. For example:

$((succ^7(0) - X):nat >>\!<< succ^2(0)) \qquad \Rightarrow X = succ^5(0)$

deriving a value of X=5 that satisfies the query 7 minus X is equal 2.

3- In many cases, it produces the most general solution. For example:

$((X >= succ^2(0)):bool >>\!<< true) \qquad \Rightarrow X= succ^2(Y)$

i.e. ‘Y+2’ for any natural number Y which is greater than or equal to 2 .

4- It detects *short-circuit*. That is to say that ERNAL does not evaluate some operands if they do not need to be evaluated in order to obtain the result. For example, among the rewriting rules of the ++ operator in Figure 3-8, the second rule says that if the evaluation of the left operand is *true* then the result is *true*, disregarding the right operand. ERNAL detects such situations and reorders the rewriting rules accordingly

5- Operands are not evaluated more than once. For example, if the rewriting rules are selected sequentially using the order specified, then the left operand of the operator ++ in Figure 3-8 is evaluated twice if it happens to be evaluated to *true*. ERNAL avoids this, see section 5.2.3.

6- ERNAL also provides evaluation traces, enabling the user to step through the execution.

To achieve the above characteristics, the transformation should provide:

1- Outer-most evaluation strategy. This allows the detection of short-circuit situations.

2- Reordering of equations. Equations must be reordered with respect to special criteria in order to achieve a more efficient implementation. As an example, the equations containing short-circuit situations should, whenever possible, be identified and evaluated first.

3- Equations-dependent implementation. For example to achieve point 5 above, evaluated operands of an operator in one rewriting rule should not be re-evaluated in the successor rewriting rules of the same operator.

4- No predefined order of evaluation.The translator determines which operand is to be evaluated first.

Additional details on ERNAL are given in chapter 5.

3.8 Goal-Oriented Execution Algorithm

What has been described so far, were the components of goal-oriented execution, namely the static analyzer, the guided-inference system, and the narrower. The following algorithm uses these components and the trace operators to define relation $\Rightarrow^{+}$.

Giving a behaviour *B*, a targeted action *a*, and a restricted set *G*, find all traces *t* such that $(a, B)/G = t \Rightarrow^{+} B'$ holds. This is done by the following steps:

1. Obtain all static derivation paths satisfying $\Sigma(a, B, G)$.
2. Obtain all variable traces satisfying the relation

   $(a, sdp, B)/G \xrightarrow{t'(\overline{V})}{}^{+} B''(\overline{V})$, where $sdp \in \Sigma(a, B, G)$.
3. For each trace $t'(\overline{V})$, extract the list of all guards and predicates in *t*'. i.e. $\rho(t'(\overline{V}))$.
4. For each trace $t'(\overline{V})$, clean the trace from guards, predicates and sorts. i.e. $\tau(t'(\overline{V}))$.
5. For each trace $\tau(t'(\overline{V}))$ obtained in step 4, clean the trace from unobservable actions. i.e. $\tau(t'(\overline{V}))\lceil\{\mathbf{i}\}$.
6. For each trace $\tau(t'(\overline{V}))\lceil\{\mathbf{i}\}$obtained in step 5, find substitutions σ_i for all free variables $\overline{V}$ using the narrower, such that all conditions in $\rho(t'(\overline{V}))$ are satisfied.
7. For each trace $\tau(t'(\overline{V}))\lceil\{\mathbf{i}\}$obtained in step 5, apply $(\tau(t'(\overline{V}))\lceil\{\mathbf{i}\})\ \sigma_i$ and $B''(\overline{V})\sigma_i$ to obtain the desired trace *t* and its resulting behaviour expression *B'* respectively.

The overall mechanism is shown in Figure 3-11.

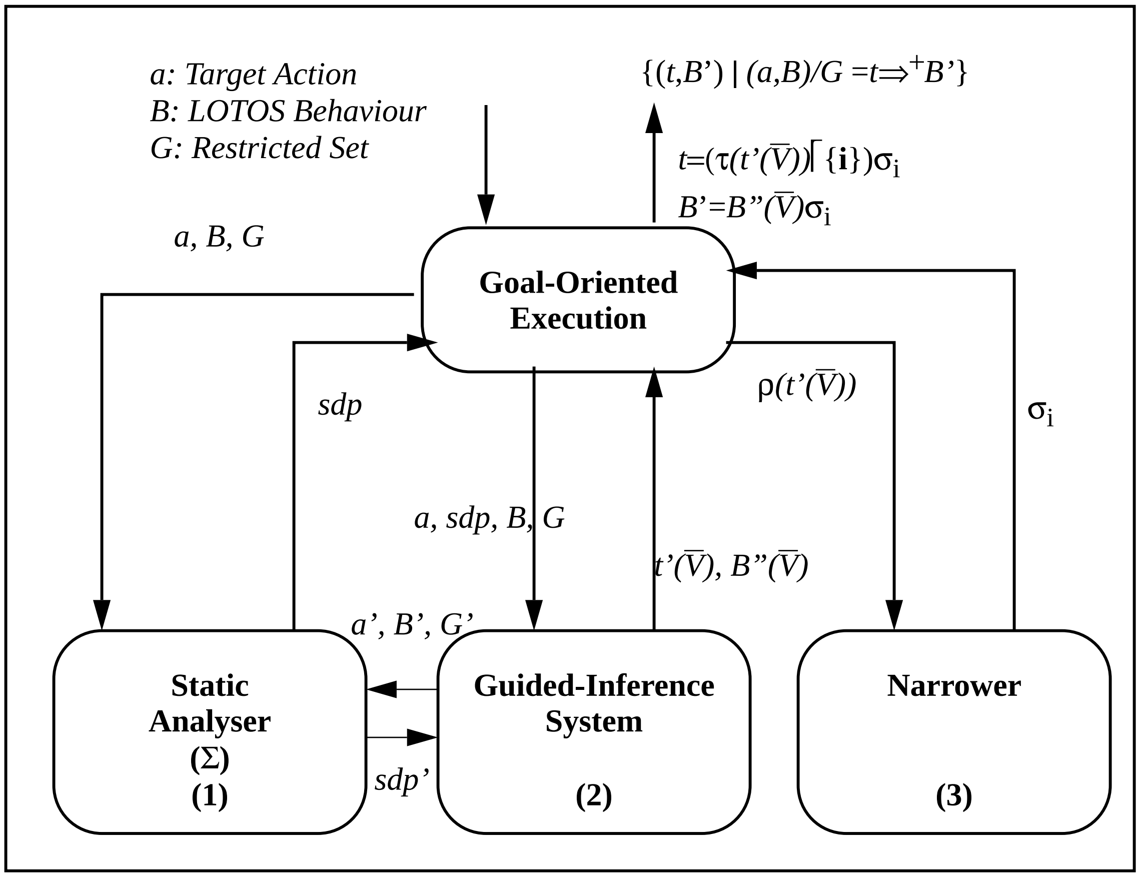


Figure 3-11 Goal-Oriented Execution

Note that, a derived variable trace *t'* obtained in step 2 is considered to be *unfeasible*, if no substitutions are found in step 6 satisfying all conditions in $\rho(t'(\overline{V}))$. This can be avoided by detecting unsatisfiable conditions during the derivation of the variable traces by using the narrower in the inference system of relation $\rightarrow^{+}$. This method is also implemented in our tool and is left as an option to the user.

Here we demonstrate the algorithm using an example. Let *B* be the behaviour given in Figure 3-3, then

(d?U:nat, B)/{c,d} $=t\Rightarrow^{+}$ *B'* can be satisfied by the following steps:

1. Obtain all static derivation paths, satisfying Σ(*d?U:nat*, *B*, *{c,d}*). There is only one such path which is:

   { [*parallel^right, hide, nested, prefix, prefix, prefix*] }

2. Obtain all variable traces satisfying the relation

   (d?U:nat, [*parallel^right, hide, nested, prefix, prefix, prefix*] *,B)/{c,d}* $\xrightarrow{t}{}^{+}$ *B'*

   The following are the only traces (with their resulting behaviour expressions) that can satisfy the above relation:

   t_1 = ⟨ *e?W/Y:nat,*

   i/*g?W/Y:nat[W/Y < 4],*

   [W/Y >1]b!W/Y ?X/Z/U:nat[X/Z/U <W/Y, X/Z/U>=1],

   d!X/Z/U⟩

   B_1' = *(c!W/Y !X/Z/U ;* **stop** *|[b]| f!W/Y !X/Z/U;* **stop***)*

   t_2 = ⟨ **i**/*g?W/Y:nat[W/Y < 4],*

   e?W/Y:nat,

   [W/Y >1]b!W/Y ?X/Z/U:nat[X/Z/U <W/Y, X/Z/U>=1],

   d!X/Z/U⟩

   B_2' = *(c!W/Y !X/Z/U ;* **stop** *|[b]| f!W/Y !X/Z/U;* **stop***)*

3. Extract the list of all guards and predicates in t_1 and t_2:

   $\rho(t_1) = \rho(t_2)$ = { *W/Y < 4, W/Y >1, X/Z/U <W/Y, X/Z/U>=1*}

4. Clean the traces from guards, predicates and sorts using the operator τ:

   $\tau(t_1)$ = ⟨*e!W/Y,* **i**/*g!W/Y, b!W/Y!X/Z/U, d!X/Z/U*⟩

   $\tau(t_2)$ = ⟨**i**/*g!W/Y, e!W/Y, b!W/Y!X/Z/U, d!X/Z/U*⟩

5. Clean the traces from unobservable actions:

   $\tau(t_1)\lceil$ {**i**} = ⟨*e!W/Y,* **i**/*g!W/Y, b!W/Y!X/Z/U, d!X/Z/U*⟩$\lceil$ {**i**}= ⟨*e!W/Y, b!W/Y!X/Z/U, d!X/Z/U*⟩

   $\tau(t_2)\lceil$ {**i**} = $\tau(t_1)\lceil$ {**i**} = ⟨*e!W/Y, b!W/Y!X/Z/U, d!X/Z/U*⟩

6. Using the narrower, find substitutions for *U, W, X, Y* and *Z,* such that all conditions in the set { *W/Y < 4, W/Y >1, X/Z/U <W/Y, X/Z/U>=1*}are satisfied:

$((W/Y < 4) \wedge (W/Y > 1) \wedge (X/Z/U < W/Y) \wedge (X/Z/U >= 1))$:bool >>< < true

=>

$\sigma_1 = U \leftarrow 1, W \leftarrow 2, X \leftarrow 1, Y \leftarrow 2, Z \leftarrow 1$

$\sigma_2 = U \leftarrow 1, W \leftarrow 3, X \leftarrow 1, Y \leftarrow 3, Z \leftarrow 1$

$\sigma_3 = U \leftarrow 2, W \leftarrow 3, X \leftarrow 2, Y \leftarrow 3, Z \leftarrow 2$

7. Apply the substitutions σ_1, σ_2, and σ_3 to the traces obtained in step 5 and to their resulting behaviour expressions:

   Solution 1

   $(\tau(t_1)\lceil\{\mathbf{i}\})\sigma_1 = (\tau(t_2)\lceil\{\mathbf{i}\})\sigma_1 = \langle e!2, b!2\ !1, d!1\rangle$

   $B_1'\ \sigma_1 = B_2'\sigma_1 = (c!2\ !1\ ;\ \mathbf{stop}\ |[b]|\ f!2\ !1;\ \mathbf{stop})$

   Solution 2

   $(\tau(t_1)\lceil\{\mathbf{i}\})\sigma_2 = (\tau(t_2)\lceil\{\mathbf{i}\})\sigma_2 = \langle e!3, b!3\ !1, d!1\rangle$

   $B_1'\ \sigma_2 = B_2'\sigma_2 = (c!3\ !1\ ;\ \mathbf{stop}\ |[b]|\ f!3\ !1;\ \mathbf{stop})$

   Solution 3

   $(\tau(t_1)\lceil\{\mathbf{i}\})\sigma_3 = (\tau(t_2)\lceil\{\mathbf{i}\})\sigma_3 = \langle e!3, b!3\ !2, d!2\rangle$

   $B_1'\ \sigma_3 = B_2'\sigma_3 = (c!3\ !2\ ;\ \mathbf{stop}\ |[b]|\ f!3\ !2;\ \mathbf{stop})$

   giving all the traces satisfying $(d?U{:}nat, B)/\{c,d\} = t\Rightarrow^{+} B'$.

The following is an example where the relation does not hold:

$(d!3{:}nat, B)/\{c,d\} = t\Rightarrow^{+} B'$ is resolved as follows:

1. Obtain all static derivation paths, satisfying $\Sigma(d!3{:}nat, B, \{c,d\})$. There is only one such path which is:

   { [*parallel^right, hide, nested, prefix, prefix, prefix*] }

2. Obtain all variable traces satisfying the relation

   $(d!3{:}nat$, [*parallel^right, hide, nested, prefix, prefix, prefix*] $,B)/\{c,d\} \xrightarrow{t}{}^{+} B'$

The following are the only traces (with their resulting behaviour expressions) that can satisfy the above relation:

$t_1 = \langle$ *e?W/Y:nat,*

i/*g?W/Y:nat[W/Y < 4],*

[W/Y >1]b!W/Y ?X/Z/3:nat[X/Z/3 <W/Y, X/Z/3>=1],

d!X/Z/3$\rangle$

B_1' = *(c!W/Y !X/Z/3 ;* **stop** *|[b]| f!W/Y !X/Z/3;* **stop***)*

$t_2 = \langle$ **i**/*g?W/Y:nat[W/Y < 4],*

e?W/Y:nat,

[W/Y >1]b!W/Y ?X/Z/3:nat[X/Z/3 <W/Y, X/Z/3>=1],

d!X/Z/3$\rangle$

B_2' = *(c!W/Y !X/Z/3 ;* **stop** *|[b]| f!W/Y !X/Z/3;* **stop***)*

3. Extract the list of all guards and predicates in t_1 and t_2:

   $\rho(t_1) = \rho(t_2) = \{$ *W/Y < 4, W/Y >1, X/Z/3 <W/Y, X/Z/3>=1*$\}$

4. Clean the traces from guards, predicates and sorts using the operator τ:

   $\tau(t_1) = \langle$*e!W/Y,* **i**/*g!W/Y, b!W/Y!X/Z/3, d!X/Z/3*$\rangle$

   $\tau(t_2) = \langle$**i**/*g!W/Y, e!W/Y, b!W/Y!X/Z/3, d!X/Z/3*$\rangle$

5. Clean the traces from unobservable actions:

   $\tau(t_1)\lceil\{\mathbf{i}\} = \langle$*e!W/Y,* **i**/*g!W/Y, b!W/Y!X/Z/3, d!X/Z/3*$\rangle\lceil\{\mathbf{i}\} = \langle$*e!W/Y, b!W/Y!X/Z/3, d!X/Z/3*$\rangle$

   $\tau(t_2)\lceil\{\mathbf{i}\} = \tau(t_1)\lceil\{\mathbf{i}\} = \langle$*e!W/Y, b!W/Y!X/Z/3, d!X/Z/3*$\rangle$

6. Using the narrower, find substitutions for *W, X, Y* and *Z*, such that all conditions in the set $\{$ *W/Y < 4, W/Y >1, X/Z/3 <W/Y, X/Z/3>=1*$\}$are all satisfied:

No substitutions can be found since predicate (*X/Z/3 <W/Y*) and predicate (*W/Y < 4*) cannot be both true. Therefore, there is no feasible trace *t* that satisfies the initial relation.

Chapter 4 Variable Traces Derivation

In the previous chapter, we have discussed the goal-oriented execution technique to derive characterized traces from LOTOS specifications, expressed by relations $\Rightarrow^{+}$ and $\Rightarrow^{\times}$. The components used in this technique are (1) *static analyser*: explores the given LOTOS specification to determine where possibly, and not how, a desired trace can be found, (2) *guided-inference system*: uses static information, generated by the static analyser, to derive variable traces defined by relations $\rightarrow^{+}$ and $\rightarrow^{\times}$, and (3) *ADT narrower*: used to resolve all guards and predicates in the variable traces by assigning values to all free variables.

In this chapter, the formal definitions of the static analyser and the guided-inference system are presented. Their limitations and the heuristics used to overcome these limitations are also discussed.

4.1 Static Derivation Paths

As described in the previous chapter, static derivation paths are generated by the function Σ : $\mathrm{A} \times \mathrm{B} \times \mathrm{G}^{*} \rightarrow \mathrm{X}$. $\Sigma(a,B,G)$ derives the set of all possible static derivation paths from behaviour B leading to each action a_{i} that statically matches action a, i.e. by $a_{\mathrm{i}} \equiv_{\mathrm{S}} a$, and not passing through any prefixed action with gate name in G. This implies that if the gate name of the target action is in G, then the SDPs cannot go through any prefixed action with target action's gate name other than the terminating actions. The formal definition of the function $\Sigma : \mathrm{A} \times \mathrm{B} \times \mathrm{G}^{*} \rightarrow \mathrm{X}$ is provided in the following section.

4.1.1 Formal Definition of Σ

Successful Termination of Σ

The following describes the successful ending of the traversal of the static behaviour, namely when a target action is found:

$\Sigma(a_1, a_2;B, G) = \{[\text{prefix}]\}$,	if $a_1 \equiv_s a_2$
$\Sigma(a, \textbf{exit}(E_1,..,E_n), G) = \{[\text{exit}]\}$,	if $a \equiv_s \delta d_1..d_n$, where
	$d_i = !E_i$ if E_i is a term or
	$d_i = \textbf{?}x_i\textbf{:}s_i$ if $E_i = \textbf{any}\ s_i$

Unsuccessful Termination of Σ

Unsuccessful termination may result from:

1- reaching a stop;
2- the action encountered does not statically match the target action and its gate belongs to *G*;
3- encountering an exit construct that does not statically match the target action;
4- encountering a relabeled behaviour whose actual gate list does not contain the gate name of the target action;
5- encountering an instantiation of a process whose actual gate list does not contain the gate name of the target action; or
6- encountering a list of hidden gates including the gate name of the target action.

More formally:

$\Sigma(a, \textbf{stop}, G) = \varnothing$	
$\Sigma(a_1, a_2;B, G) = \varnothing$	if not$(a_1 \equiv_s a_2)$ and
	$name(a_2) \in G$
$\Sigma(a, \textbf{exit}(E_1,..,E_n), G) = \varnothing$	if not$(a \equiv_s \delta d_1..d_n)$, where
	$d_i = !E_i$ if E_i is a term or
	$d_i = \textbf{?}x_i\textbf{:}s_i$ if $E_i = \textbf{any}\ s_i$
$\Sigma(a, (B)[g_1/h_1, ..., g_n/h_n], G) = \varnothing$	if $name(a) \notin \{g_1, \ldots, g_n\}$

$\Sigma(a, p[g_1,..,g_n], G) = \emptyset$ if $name(a) \notin \{g_1, \ldots, g_n\}$

$\Sigma(a,$ **hide** GL **in** $B, G) = \emptyset$ if $name(a) \in GL$

Recursion

In all other situations, the behaviour has to be analyzed further. This is done by carrying the evaluation of Σ to the sub-behaviour(s), according to the specific rules for each type of construct, as described below. Informally, the recursive generation of the set of SDPs from the current behaviour has one of the 2 forms:

1- *unary operators*: $\Sigma(a, op\ B, G)$ is a composition of the elements of $\Sigma(a, B, G)$, prefixing each element with the symbol representing *op* with the exception of *instance* and *relabel* operators where a new target action gate name and a new restricted gate set are carried out. If $\Sigma(a, B, G) = \emptyset$, then $\Sigma(a, op\ B, G) = \emptyset$.

2- *binary operators*: $\Sigma(a, B_1\ op\ B_2, G)$ is a composition of the elements of $\Sigma(a, B_1, G)$ and $\Sigma(a, B_2, G)$, prefixing each element with the symbol representing *op* followed by ^*direction*, with direction being *left* for elements from $\Sigma(a, B_1, G)$ and *right* for elements from $\Sigma(a, B_2, G)$.

o **Prefix**

$\Sigma(a, \mathbf{i};B, G) =$ $\{[prefix.s] \mid s \in \Sigma(a, B, G)\}$

$\Sigma(a_1, a_2;B, G) =$ $\{[prefix.s] \mid s \in \Sigma(a_1, B, G)\}$

if $name(a_2) \notin G$

Note that even if $a_1 \equiv_s a_2$ in the second rule, recursion is carried out to find another path as long as the gate name of the encountered action a_2 is not restricted, i.e. $name(a_2) \notin G$.

o **Choice**

$\Sigma(a, B_1[]B_2, G) =$ $\{[choice\text{^}left.s] \mid s \in \Sigma(a, B_1, G)\} \cup$

$\{[choice\text{^}right.s] \mid s \in \Sigma(a, B_2, G)\}$

o **Guard**

$\Sigma(a, [P] \text{->} B, G) =$ $\{[guard.s] \mid s \in \Sigma(a, B, G)\}$

- **Local Definition**

  $\Sigma(a,$ **let** $x_1{:}s_1{=}t_1, .. \; x_n{:}s_n{=}t_n$ **in** $B, G) =$

  $\{[let.s] \mid s \in \Sigma(a, B, G)\}$

- **Summation on Values**

  $\Sigma(a,$ **choice** $x{:}s$ [] $B, G) =$

  $\{[chval.s] \mid s \in \Sigma(a, B, G)\}$

- **Nested**

  $\Sigma(a, (B), G) =$ $\{[nested.s] \mid s \in \Sigma(a, B, G)\}$

- **Hiding**

  $\Sigma(a,$ **hide** GL **in** $B, G) =$ $\{[hide.s] \mid s \in \Sigma(a, B, G)\}$

  if $name(a) \notin GL$

- **Enabling**

  $\Sigma(a, B_1{>>}B_2, G) =$ $\{[enable\text{^}left.s] \mid s \in \Sigma(a, B_1, G)\} \cup$

  $\{[enable\text{^}right.s] \mid s \in \Sigma(a, B_2, G)\}$

  if $name(a) \neq \delta$

  $\Sigma(a, B_1{>>}B_2, G) =$ $\{[enable\text{^}right.s] \mid s \in \Sigma(a, B_2, G)\}$

  if $name(a) = \delta$

In this case, if δ action exists in some trace, it will be found at the end of the execution of B_2. All other δ actions in B_1 are transformed into internal actions by the enable operator.

- **Disabling**

$\Sigma(a, B_1[>B_2, G) =$ $\{[disable\text{^}left.s] \mid s \in \Sigma(a, B_1, G)\} \cup$

$\{[disable\text{^}right.s] \mid s \in \Sigma(a, B_2, G)\}$

- **Selected Synchronization**

$\Sigma(a, B_1 \,|[GL]|\, B_2, G) =$ $\{[parallel\text{^}left.s] \mid s \in \Sigma(a, B_1, G)\} \cup$

$\{[parallel\text{^}right.s] \mid s \in \Sigma(a, B_2, G)\}$,

if $name(a) \notin (\{\text{GL}\} \cup \{\delta\})$

$\Sigma(a, B_1 \,|[GL]|\, B_2, G) =$ $\{[parallel\text{^}left.s] \mid s \in \Sigma(a, B_1, G)\}$

if $name(a) \in (\{\text{GL}\} \cup \{\delta\})$

The second rule states the fact that if the target action *a* is a synchronization action, denoted by $name(a) \in (\{\text{GL}\} \cup \{\delta\})$, then only one side is explored. As we shall see in section 4.2, the other side will be explored while generating the desired trace.

- **Interleave Parallelism**

$\Sigma(a, B_1 \,|||\, B_2, G) =$ $\{[parallel\text{^}left.s] \mid s \in \Sigma(a, B_1, G)\} \cup$

$\{[parallel\text{^}right.s] \mid s \in \Sigma(a, B_2, G)\}$,

if $name(a) \neq \delta$

$\Sigma(a, B_1 \,|||\, B_2, G) =$ $\{[parallel\text{^}left.s] \mid s \in \Sigma(a, B_1, G)\}$

if $name(a) = \delta$

An alternative definition of $\Sigma(a, B_1 \,|||\, B_2, G)$ is:

$\Sigma(a, B_1 \,|||\, B_2, G) =$ $\Sigma(a, B_1 \,|[]|\, B_2, G)$

- **Full Synchronization**

$\Sigma(a, B_1 \,||\, B_2, G) =$ $\{[parallel\text{^}left.s] \mid s \in \Sigma(a, B_1, G)\}$

In this case, only one side is explored since the target action *a* must be a synchronization action.

o **Relabeling**

$\Sigma(g\ d_1 ... d_n[P], (B)[g_1/h_1, ..., g_n/h_n], G) =$

$\{[relabel(h, G').s] \mid s \in \Sigma(h\ d_1 ... d_n[P], B, G')\}$

if $g \in \{g_1, \ldots, g_n\}$

where

$h \in$ TargetSet $= \{h_k \mid g_k = g\}$ and $G' = \{h_k \mid g_k \in \mathrm{G}\}$

$\Sigma(\delta\ d_1 ... d_n[P], (B)[g_1/h_1, ..., g_n/h_n], G) =$

$\{[relabel(\delta, G').s] \mid s \in \Sigma(\delta\ d_1 ... d_n[P], B, G')\}$

where $G' = \{h_k \mid g_k \in \mathrm{G}\}$

In these rules, the gate name *g* of the actual target action (when g ≠ δ) and the actual restricted gate set *G* are replaced by the corresponding formal gates of behaviour *B*, namely *h* and *G'* respectively. *h* and *G'* may need to be referenced by the guided-inference system, defined in section 4.2. For that reason, they are saved in the static derivation paths with the *relabel* element.

The following is an example that demonstrates the above rule:

$\Sigma(a?X{:}Nat, \underline{p[h_1,h_2,h_3,h_4][a/h_1, b/h_2, a/h_3, c/h_4]}, \{a,b\}) =$

$\{[relabel(h_1, \{h_1,h_2,h_3\}).s] \mid s \in (\Sigma(h_1?X{:}Nat, \underline{p[h_1,h_2,h_3,h_4]}, \{h_1,h_2,h_3\})\ \} \cup$

$\{[relabel(h_3, \{h_1,h_2,h_3\}).s] \mid s \in (\Sigma(h_3?X{:}Nat, \underline{p[h_1,h_2,h_3,h_4]}, \{h_1,h_2,h_3\})\ \}$

i.e. the static derivation paths of action *a?X:Nat* with restricted set $\{a,b\}$ are those of $h_1?X{:}Nat$ and $h_3?X{:}Nat$ with restricted set $\{h_1,h_2,h_3\}$, since h_1 and h_3 are to be relabeled by *a*, and h_2 is to be relabeled by *b*.

o **Process Instantiation**

$\Sigma(g\ d_1 ... d_n[P],\ p[g_1,..,g_n](t_1,...,t_m),\ G) = \{[instance(h, G').s] \mid s \in \Sigma(h\ d_1...d_n[P], B, G)'\}$

if $g \in \{g_1,\ldots,g_n\}$ and $\exists P[h_1,..,h_n](x_1{:}s_1..x_m{:}s_m)$ **:=** B,

where

$h \in$ TargetSet $= \{h_k \mid g_k = g\}$ and $G' = \{h_k \mid g_k \in G\}$

$\Sigma(\delta\ d_1...d_n,\ p[g_1,..,g_n](t_1,...,t_m),\ G) = \{[instance(\delta, G').s] \mid s \in \Sigma(\delta\ d_1...d_n[P], B, G)'\}$

if $\exists P[h_1,..,h_n](x_1{:}s_1, ..., x_m{:}s_m)$**:exit**$(S_1,...,S_n)$ **:=** B and

$sort(d_i) = S_i$ for $1 \le i \le n$,

where $G' = \{h_k \mid g_k \in G\}$

Similar to the relabeling rules, the gate name *g* (when $g \neq \delta$) of the actual target action and the actual restricted gate set *G* are replaced by the corresponding formal gates. The target gate name and restriction set replacements are also saved in the static derivation paths with the *instance* element.

4.1.2 Observations

Lemma 4-1: The number of possible distinct process instantiations encountered during the derivation of static derivation paths is finite.

Proof: Two process instantiations are said to be distinct if (1) they have different process names, or (2) they have same process name but different actual gates. Since the number of processes in any LOTOS specification is finite then the number of process instantiations with different process names is also finite. The number of possible distinct instantiations for processes having the same process name is also finite, since the number of gates of any LOTOS behaviour *B*, $|\alpha(B)|$, is finite.

o

Lemma 4-2: If the number of occurrences of identical process instantiations encountered during the derivation of static derivation paths is finite, then $\Sigma(a, B, G)$ is a finite set.

Proof: If the number of possible distinct process instantiations encountered during the derivation of static derivation paths is finite (lemma 4-1), and the number of possible occurrences of each of the distinct process instantiations is also finite, then the number of all process instantiation occurrences in deriving the static derivation paths is finite. This implies that the search space for deriving the set $\Sigma(a, B, G)$ is finite.

o

Lemma 4-3: If $\Sigma(a, B, G) = \varnothing$ then there exists no trace $\langle a_1, ..., a_n\rangle$ such that $(a, B)/G = \langle a_1, ..., a_n\rangle \Rightarrow^+ B_r$ holds.

Proof: $\Sigma(a, B, G) = \varnothing$ only if one the following is true:

1- no action *a'* is found that statically matches action *a* using static relabeling. In this case, there will be no action that can be derived from B that matches action *a*, and therefore $(B, a)/G = \langle a_1, ..., a_n\rangle \Rightarrow^+ B_r$ does not hold;

2- an action *a'* is found that matches action *a* using static relabeling, but there is an intermediate action *b* on the same static derivation path that has a gate in *G*. In this case, action *b* must also appear as an intermediate action in the trace $\langle a_1, ..., a_n\rangle$, and therefore contradict the definition of $(a, B)/G = \langle a_1, ..., a_n\rangle \Rightarrow^+ B_r$.

o

Lemma 4-4: $\Sigma(a,B, G) \neq \varnothing$ does not imply that $(a, B)/G = \langle a_1, ..., a_n\rangle \Rightarrow^+ B_r$ holds.

Proof: This can be proven by a counterexample. For example, Σ*(a, (a;* **stop** || *b;* **stop***), {})* = {[*nested, parallel^left, prefix*]}, but *(a, (b;* **stop** || *a;* **stop***))/{} =t*$\Rightarrow^+ B_r$ does not hold for any *t*, because of lack of synchronization.

o

4.1.3 Limitations

The limitations of static derivation paths definition are:

1. The derivation of an SDP may not terminate. For example, consider the definition of *p1* given in Figure 4-4 , the derivation of the first SDP ∈ Σ*(b?X:Nat, p1[a,b](0), {b})* will cycle indefinitely searching for an action that statically matches *b?X:Nat* as follows:

   [instance, choice^left, nested, choice^right, instance, choice^left, nested, choice^right, instance, choice^left, nested, choice^right, ...],

2. The set $\Sigma(a, B, G)$ can be infinite. Again, considering the specification in Figure 4-4 , we have:

Σ*(a, p1[a,b](0), {a, b}) =*

{[instance, choice^left, nested, choice^left, prefix],

[instance, choice^left, nested, choice^right, instance, choice^left, nested, choice^left, prefix],

[instance, choice^left, nested, choice^right, instance, choice^left, nested, choice^right, instance, choice^left,nested, choice^left, prefix],

...},

that can be represented as

{[instance, (choice^left, nested, choice^right, instance)^n, choice^left, nested, choice^left, prefix]},

for $n \geq 0$

```
specification testing[a,b]:noexit
library NaturalNumber, Boolean endlib
behavior

        p1[a,b](0)
        |[a,b]|
        b?X1:Nat;
        b?X2:Nat;
        a?X3:Nat;
        stop
       where
        process p1[a,b](X:Nat):noexit :=
                ( a!X[X ge Succ(Succ(0))]; stop
                 []
                  p1[a,b](Succ(X)) )
                []
                b!X; p1[a,b](Succ(X))
        endproc
 endspec
```

Figure 4-4 Recursive Process Definition

Both problems can be solved by adding appropriate heuristics to the implementation. The first problem occurs when a process instantiation is re-encountered before the target action is found. The second problem occurs when a process instantiation is re-encountered in an alternative, after the target action has been found. Therefore, these limitations are due to recursive process definitions. To avoid such problems, the search space for deriving the static derivation paths can be made finite by limiting the number of identical process instantiations occurrences, see lemma 4-2. We recall that identical process instantiations are those having the same process name and actual gates, i.e. actual value parameters are not considered. Now returning to the previous example, if the number of identical process instantiations is constrained to be at most 3, then:

1. the derivation of the first SDP $\in$ $\Sigma(b, p1[a,b](0), \{b\})$ will terminate with the following path:
   [instance, choice^left, nested, choice^right, instance, choice^left, nested, choice^right, instance, choice^left, nested, choice^left, prefix], and
2. $\Sigma(a, p1[a,b](0), \{a, b\})$ =

   {[instance, choice^left, nested, choice^left, prefix],

   [instance, choice^left, nested, choice^right, instance, choice^left, nested, choice^left, prefix],

   [instance, choice^left, nested, choice^right, instance, choice^left, nested, choice^right, instance, choice^left, nested, choice^left, prefix]}

To reflect the above heuristic, the definition of $\Sigma(a, B, G)$ will carry a list *PI_LIST* of process instantiations (not including actual value parameters) with their number of occurrences. Therefore, $\Sigma(a, B, G)$ is defined as $\Sigma(a, B, G, PI\_LIST)$, where *PI_LIST* is initially empty.

The following is the definition of $\Sigma(a, B, G, PI\_LIST)$ when *B* is a process instantiation.

$\Sigma(g\ d_1...d_n[P], p[g_1,..,g_n](t_1,...,t_m), G, PI\_LIST)$ =$\{[instance.s] \mid s \in \Sigma(h\ d_1...d_n[P], B, G', PI\_LIST2)\}$

<same conditions as in the definition of $\Sigma(a, B, G)$ when *B* is a process instantiation> and

$PI\_LIST2 = PI\_LIST \cup (p[g_1,..,g_n],1)$, if $(p[g_1,..,g_n],N) \notin PI\_LIST$, i.e. first occurrence of $p[g_1,..,g_n]$

$PI\_LIST2 = PI\_LIST-(p[g_1,..,g_n],N) \cup (p[g_1,..,g_n],N+1),$

if $(p[g_1,..,g_n],N) \in PI\_LIST$, i.e. N+1 occurrences of $p[g_1,..,g_n]$, and $N \leq PI\_Limit$

PI_LIST is not affected in the definition of $\Sigma(a, B, G, PI\_LIST)$ when *B* is not a process instantiation.

4.2 Guided-Inference System

As described in section 3.5 of the previous chapter, the algorithm to implement the relation $(a, B)/G =t\Rightarrow^+ B'$ is defined by first generating variable traces satisfying the relation:

$(a, sdp, B)/G \xrightarrow{t(\overline{V})}{}^+ B'(\overline{V})$, where $sdp \in \Sigma(a, B, G)$.

Relation $\rightarrow^+$ is defined using guided-inference system where the derivation is guided by static derivation paths.

To demonstrate some key points in our definition of guided-inference system, the references to behaviour B_i used in our examples refer to the behaviour tree in Figure 4-6 . For example, *B3* identifies the behaviour *a;b;***stop** *[] b;c;***stop**.

```
specification testing[a,b,c,d,e]:noexit:=
behaviour
   ( a;b;stop
     []
     b;c;stop
       [>
          a;b;stop
          []
          d;c;stop )
|[a,c]|
   b;c;stop
   []
   a;e;c;stop
endspec
```

Figure 4-5 A LOTOS Specification

.

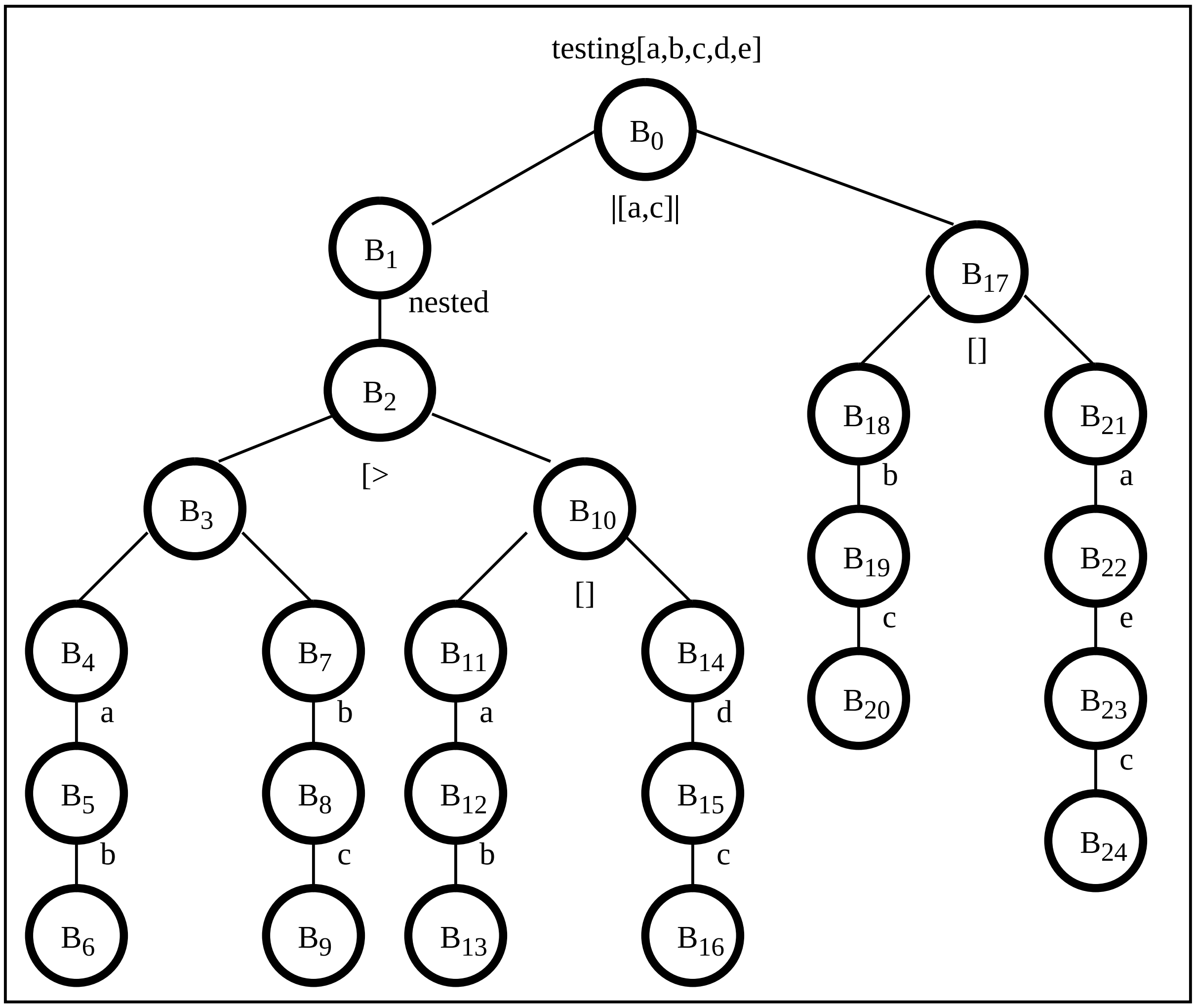


Figure 4-6 Abstract Syntactic Tree of Figure 4-5

4.2.1 Formal Definition

The formal definition of $(a, sdp, B)/G \xrightarrow{t(\overline{V})}{}^{+} B'(\overline{V})$, where $sdp \in \Sigma(a, B, G)$, is given below.

Target action is reached

A target action is reached when the current behaviour expression is an *exit* or an *action prefix*, and the static derivation path has only one element identifying the behaviour. In this case, the accumulated action will be the result of statically matching the current derived action a_2 and the actual target action a_1, precisely $a_1 \uparrow_s a_2$ defined in Table 3-1 of the previous chapter.

- **Successful Termination**

$$(a, [\text{exit}], \mathbf{exit}(E_1,..,E_n))/G \longrightarrow\langle a \uparrow_s \delta d_1..d_n\rangle\rightarrow^+ \mathbf{stop} \qquad (1)$$

$$d_i = !E_i \text{ if } E_i \text{ is a term or}$$

$$d_i = \mathbf{?}x_i\mathbf{:}s_i \text{ if } E_i = \mathbf{any}\ s_i$$

The action $a \uparrow \delta d_1..d_n$ is accumulated.

Example

The relation $(\delta\ !0, [\text{exit}], \mathbf{exit}(\mathbf{any}\ Nat))/\{\} \longrightarrow t \rightarrow^+ B'$ will be satisfied with $t = \langle\delta\ !0\rangle$ and B' = **stop**.

- **Action Prefix**

$$(a_1, [\text{prefix}], a_2;B)/G \longrightarrow\langle a_1\uparrow_s a_2\rangle\rightarrow^+ B \qquad (2)$$

When the end of the SDP is reached, the prefixed action is matched with the target action producing the desired action. If the matching causes variables in a_2 to be substituted by terms in a_1, then this substitution is carried to the resulting behaviour B.

Example

Let $B = c?Y:Nat\ ?Z:Nat[Y<Succ(0)];\ d\ !\ Y;$ **stop**,

then $(c!0\ ?X:Nat[X>0], [\text{prefix}], B)/\{\} \longrightarrow t \rightarrow^+ B'$ will be satisfied with

$t = \langle c!0\ ?Z:Nat[0<Succ(0)\ \wedge\ [Z>0]]\rangle$ and $B' = d\ !\ 0;$ **stop**.

Target action is not reached

This is the case where the static derivation path has more than one element.

- **Action Prefix (Target action is not reached)**

$$\frac{s \neq [],\ (a, s, B)/G \longrightarrow t \rightarrow^+ B'}{(a, [\text{prefix}.s], a';B)/G \longrightarrow\langle a'.t\rangle\rightarrow^+ B'} \qquad (3)$$

These rules adds the internal and external prefixed actions to the derived trace.

Example

The relation *(c,* [prefix, prefix], **i***; c;* **stop** *)*/{} $\xrightarrow{t}^{+}$ *B'* will be satisfied with *t* = ⟨**i**, *c*⟩ and *B'* = **stop**.

o **Choice**

$$\frac{(a, s, B_1)/G \xrightarrow{t}^{+} B'}{(a, [\text{choice\^{}left}.s], B_1 \; [] \; B_2)/G \xrightarrow{t}^{+} B'} \qquad (4)$$

$$\frac{(a, s, B_2)/G \xrightarrow{t}^{+} B'}{(a, [\text{choice\^{}right}.s], B_1 \; [] \; B_2)/G \xrightarrow{t}^{+} B'} \qquad (5)$$

The trace generated for the choice operator is either the trace of B_1 (in the case of *choice^left*) or B_2 (in the case of *choice^right*).

o **Nested**

$$\frac{(a, s, B)/G \xrightarrow{t}^{+} B'}{(a, [\text{nested}.s], (B))/G \xrightarrow{t}^{+} B'} \qquad (6)$$

Nesting has no effect on the derived trace.

o **Guard**

$$\frac{(a, s, B)/G \xrightarrow{\langle b_1.t \rangle}^{+} B'}{(a, [\text{guard}.s], ([P]\text{->}B))/G \xrightarrow{\langle b_2.t \rangle}^{+} B'} \qquad (7)$$

where $b_2 = [P]b_1$, if b_1 has no associated guard,
$b_2 = [P\text{\^{}}P_1]b_{11}$, if b_1 has the form: $[P_1]b_{11}$

The guard *P* in [*P*]-> *B* is associated with the first action of every trace generated from *B*.

Example

Let *B* = [*X>Succ(0)*]-> (*c!X;* **stop** [] [*Y<Succ(0)*] -> *d!X!Y; a?Z:Bool;* **stop**), then (*a?Z:Bool*, [guard, nested, choice^right, guard, prefix,prefix], *B*)/{}—*t*→$^+$ *B'*, will be satisfied with

t = ⟨*[(X>Succ(0)) ^ (Y<Succ(0))]d!X!Y, a?Z:Bool*⟩ and *B'* = **stop**.

o **Local Definition**

$$\frac{(a,\ s,\ [t_1/x_1,\ ..,\ t_n/x_n]B)/G \xrightarrow{t}{}^{+} B'}{(a,\ [\text{let}.s],\ \textbf{let}\ x_1{:}s_1{=}t_1,\ ..\ x_n{:}s_n{=}t_n\ \textbf{in}\ B)/G \xrightarrow{t}{}^{+} B'} \qquad (8)$$

The trace generated from the behaviour **let** x_1*:*s_1=t_1, .. x_n:s_n=t_n **in** *B* is the same trace generated from *B* after replacing all occurrences of x_1 .. x_n in *B* by t_1 .. t_n respectively.

Example

Let *B* = **let** *X:Nat=Succ(0)* **in** *c?Y:Bool; a!X; d!X,* **stop**,

then (*a?Z:Nat*, [let, prefix, prefix], *B*)/{}—*t*→$^+$ *B'*, will be satisfied with

t = ⟨*c?Y:Bool, a!Succ(0)*⟩ and *B'* = *d!Succ(0);* **stop**.

o **Summation on Values**

$$\frac{(a,\ s,\ B)/G \xrightarrow{\langle b_1.t\rangle}{}^{+} B'}{(a,\ [\text{chval}.s],\ \textbf{choice}\ x{:}s\ []\ B)/G \xrightarrow{\langle b_2.t\rangle}{}^{+} B'} \qquad (9)$$

where b_2 = [*x*=*x*]a_1, if b_1 has no associated guard,
b_2 = [(*x*=*x*)^P_1]a_{11}, if b_1 has the form: [P_1]b_{11}

For behaviour **choice** *x:s* [] *B*, The guard *x*=*x* is associated with the first action of every trace generated from *B*. The narrower will generate values for *x* when applied on the predicate *x*=*x*.

Example

Let *B* = **choice** *X:Nat* [] (*c!X;* **stop** [] [*Y<Succ(0)*] -> *d!X!Y; a!X?Z:Bool;* **stop**),

then (*d?Z1:Nat?Z2:Nat*, [chval, nested, choice^right, guard, prefix], *B*)/{}=*t*→$^+$ *B'*, will be satisfied with

t = ⟨*[(X=X) ^ (Y<Succ(0))]d!X!Y*⟩ and *B'* = *a!X?Z:Bool***; stop**.

- **Hiding**

$$\frac{(a, s, B)/G \xrightarrow{t}{}^{+} B'}{(a, [\text{hide}.s], \textbf{hide } GL \textbf{ in } B)/G \xrightarrow{t\downarrow\{\text{GL}\}}{}^{+} \textbf{hide } GL \textbf{ in } B'} \tag{10}$$

All actions in the trace generated by the hide operator with gate names in the list GL, are hidden.

Example

Let B = **hide** *a,b* **in** *c;a;d;***stop**,

then (c, [hide, prefix, prefix], B)/{}$\xrightarrow{t}{}^{+}$ B', will be satisfied with

$t = \langle c, a\rangle\downarrow\{a,b\} = \langle c, \mathbf{i}/a\rangle$ and B' = **hide** *a,b* **in** *d;***stop**.

- **Enabling**

$$\frac{(a, s, B_1)/G \xrightarrow{t_1}{}^{+} B_{11}}{\begin{array}{c}(a, [\text{enable\^{}left}.s], B_1 >> \textbf{accept } x_1{:}s_1,.., x_n{:}s_n \textbf{ in } B_2)/G \xrightarrow{t_1}{}^{+} \\ B_{11} >> \textbf{accept } x_1{:}s_1,.., x_n{:}s_n \textbf{ in } B_2\end{array}} \tag{11}$$

$$\frac{\begin{array}{c}\exists\, r \in \Sigma(\delta\ ?X_1{:}s_1 \ldots ?X_n{:}s_n, B_1), \\ (\delta\ ?X_1{:}s_1 \ldots ?X_n{:}s_n, r, B_1)/G \xrightarrow{t_1}{}^{+} B_{11} \\ (a, \text{s}, [v_1/x_1, .., v_n/x_n]B_2)/G \xrightarrow{t_2}{}^{+} B_{21}\end{array}}{\begin{array}{c}(a, [\text{enable\^{}right}.s], B_1 >> \textbf{accept } x_1{:}s_1,.., x_n{:}s_n \textbf{ in } B_2)/G \xrightarrow{t_1\downarrow\{\delta\} \bullet t_2}{}^{+} B_{21} \\ \text{where } t_1\text{\^{}} = \delta\ d_1,.., d_n, \\ v_i = E \text{ if } d_i = !E \text{ or } v_i = x \text{ if } d_i = ?x{:}s\end{array}} \tag{12}$$

The first rule (11) states that if the goal action is in B_1 then the resulting trace will be a trace t_1 generated from B_1 guided by the remainder SDP s, and the resulting behaviour will be B_{11} >> **accept** $x_1{:}s_1,.., x_n{:}s_n$ **in** B_2 where B_{11} is the behaviour B_1 after trace t_1.

The second rule (12) states that if the goal action is in B_2 then the resulting trace will be the concatenation of two traces:

1. Trace $t_1\downarrow\{\delta\}$: where t_1 is a trace from B_1 leading to an action on gate δ with events matching the variable definition list in the **accept** clause, i.e. δ $?X_1{:}s_1 \ldots ?X_n{:}s_n$, and not

including any action with the gate name in G.

2. Trace t_2: is a trace from B_2 guided by the remainder SDP s, where all variables defined in the **accept** clause are substituted in B_2 by the terms offered by action δ above.

The resulting behaviour B_{21} in the second rule will simply be behaviour B_2 after trace t_2.

Example

Let behaviour *B = a?Z:Nat;* **exit***(Succ(Z))>>* **accept** *X:Nat* **in** *c!X;***exit***(Succ(X)),*

then the set of static derivation paths Σ*(c?Y:Nat, B)*/{} = {[enable^right, prefix]}. Therefore the relation *(c?Y:Nat, B)*/{} $\text{—}t\!\rightarrow^{+}$ *B'* is defined as *(c?Y:Nat,* [enable^right, prefix], *B)/*{} $\text{—}t\!\rightarrow^{+}$ *B'* which will match rule (12) where the following relations must hold:

1. (δ*?X*$_1$*:Nat*, <u>*a?Z:Nat;* **exit***(Succ(Z))*</u>)/{} $\text{—}t_1\!\rightarrow^{+}$ B_{11}
2. *(c?Y:Nat*, [prefix], <u>*c!X;***exit***(Succ(X))*</u> *)*/{} $\text{—}t_2\!\rightarrow^{+}$ B_{21}

The first relation (δ*?X*$_1$*:Nat*, <u>*a?Z:Nat;* **exit***(Succ(Z))*</u>)/{} $\text{—}t_1\!\rightarrow^{+}$ B_{11} will be defined as (δ*?X*$_1$*:Nat*, [prefix, exit], <u>*a?Z:Nat;* **exit***(Succ(Z))*</u>)/{} $\text{—}t_1\!\rightarrow^{+}$ B_{11},where [prefix, exit] ∈ Σ*(*δ*?X*$_1$*:Nat*, <u>*a?Z:Nat;* **exit***(Succ(Z))*</u>*)*/{}, and will be satisfied with t_1 = ⟨*a?Z:Nat*, δ*!Succ(Z)*⟩ and B_{11} = ***stop***.

Substituting the variable X by *Succ(Z)* in B_2, the second relation will become:
2'. *(c?Y:Nat*, [prefix], <u>*c!Succ(Z);***exit***(Succ(Succ(Z)))*</u> *)*/{} $=t_2\!\rightarrow^{+}$ B_{21}

and will be satisfied with t_2 = ⟨*c!Succ(Z)*⟩ and B_{21}= **exit***(Succ(Succ(Z)))*. Therefore the original relation *(c?Y:Nat*, [enable^right, prefix], *B)/*{} $\text{—}t\!\rightarrow^{+}$ *B'* will be satisfied with

$t = t_1 \downarrow \{\delta\} \bullet t_2$ = ⟨*a?Z:Nat*, δ*!Succ(Z)*⟩↓{δ} • ⟨*c!Succ(Z)*⟩ = ⟨*a?Z:Nat*, **i**/δ*!Succ(Z), c!Succ(Z)*⟩ and

B' = B_{21} = **exit***(Succ(Succ(Z)))*.

o **Disabling**

$$\frac{(a, s, B_1)/G \text{—}t_1\!\rightarrow^{+} B_{11},\ name(a) = \delta}{(a, [\text{disable\^{}left}.s], B_1 \,[\!> B_2)/G \text{—}t_1\!\rightarrow^{+} B_{11}} \qquad (13)$$

$$\frac{(a, s, B_1)/G \overset{t_1}{\longrightarrow}{}^{+} B_{11},\ name(a) \neq \delta}{(a, [\text{disable\^{}left}.s], B_1 \text{ [> } B_2)/G \overset{t_1}{\longrightarrow}{}^{+} B_{11} \text{ [> } B_2} \qquad (14)$$

$$\frac{(a, \text{s}, B_2)/G \overset{t_2}{\longrightarrow}{}^{+} B_{21},\quad (\langle\rangle, B_1)/(G\cup\{\delta\}) \overset{t_1}{\longrightarrow}{}^{\times} B_{11}}{(a, [\text{disable\^{}right}.s], B_1 \text{ [> } B_2)/G \overset{t_1 \bullet t_2}{\longrightarrow}{}^{+} B_{21}} \qquad (15)$$

Rule (13) and (14) handle the case where the goal action is in B_1. In this case, the resulting behaviour expression will be constructed depending if the gate name of the target action is δ or not.

Rule (15) states that if the search is guided to the right behaviour B_2, then the resulting trace will be the concatenation of two traces:

1. Trace t_1: is any trace derived from B_1 with length ≥ *0* and not including actions with gate names in $G\cup\{\delta\}$,
2. Trace t_2: is a trace from B_2 guided by the remainder SDP *s*.

The resulting behaviour in the third rule will be B_2 after trace t_2, namely B_{21}.

Example

Considering behaviour B_2 in Figure 4-6 where $(c, B_2)/\{b, c\}$ = {[disable^right, choice^right, prefix, prefix]} then the relation $(c, B_2)/\{b, c\} \overset{t}{\longrightarrow}{}^{+} B'$, is defined as (*c*, [disable^right, choice^right, prefix, prefix], $B_2)/\{b, c\} \overset{t}{\longrightarrow}{}^{+} B'$. This matches rule (15) where the following relations must be satisfied:

1. (*c*, [choice^right, prefix, prefix], $B_{10})/\{b\} \overset{t_2}{\longrightarrow}{}^{+} B_{\text{r2}}$
2. $(\langle\rangle, B_3)/\{b, c, \delta\} \overset{t_1}{\longrightarrow}{}^{\times} B_{\text{r1}}$

The first relation will be satisfied with $t_2 = \langle d, c\rangle$ and B_{r2} = **stop**; the second relation then becomes $(\langle\rangle, B_3)/\{b, c, \delta\} \overset{t_1}{\longrightarrow}{}^{\times} B_{\text{r1}}$, and will be satisfied with two results:

1. $t_1 = \langle\rangle$, $B_{\text{r1}} = B_3$ = *a; b;* **stop** *[] b; c;* **stop**,
2. $t_1 = \langle a\rangle$, $B_{\text{r1}} = B_5$ = *b;* **stop**.

Therefore the original relation (*c*, [disable^right, choice^right, prefix, prefix], $B_2)/\{b, c\} \overset{t}{\longrightarrow}{}^{+} B'$ will be satisfied with $t = t_1 \bullet t_2$, where $t_1 \in \{\langle\rangle, \langle a\rangle\}$, $t_2 = \langle d, c\rangle$ and $B' = B_{\text{r2}}$ = **stop**.

o **Selected Synchronization**

$$\frac{\begin{array}{c} name(a) \in (\{S\}\cup\{\delta\}) \\ (a, \text{s}, B_1)/G \xrightarrow{t_1}{}^{+} B_{11}, \\ (t_1 \lfloor (\{S\}\cup\{\delta\}), B_2)/(G\cup\{S\}\cup\{\delta\}) \xrightarrow{t_2}{}^{\times} B_{21}, \end{array}}{(a, [\text{parallel\^{}left}.s], B_1 \,|[S]|\, B_2)/G \xrightarrow{t_1|\{S\}|\, t_2}{}^{+} B_{11} \,|[S]|\, B_{21}} \quad (16)$$

$$\frac{\begin{array}{c} name(a) \notin (\{S\}\cup\{\delta\}) \\ (a, \text{s}, B_1)/G \xrightarrow{t_1}{}^{+} B_{11}, \\ (t_1 \lfloor (\{S\}\cup\{\delta\}), B_2)/(G\cup\{S\}\cup\{\delta\}) \xrightarrow{t_2}{}^{\times} B_{21}, \end{array}}{(a, [\text{parallel\^{}left}.s], B_1 \,|[S]|\, B_2)/G \xrightarrow{(t_1 \lfloor \{name(a)\} \,|\{S\}|\, t_2) \bullet \langle t_1\text{\^{}}\rangle}{}^{+} B_{11} \,|[S]|\, B_{21}} \quad (17)$$

$$\frac{\begin{array}{c} name(a) \in (\{S\}\cup\{\delta\}) \\ (a, \text{s}, B_2)/G \xrightarrow{t_2}{}^{+} B_{21}, \\ (t_2 \lfloor (\{S\}\cup\{\delta\}), B_1)/(G\cup\{S\}\cup\{\delta\}) \xrightarrow{t_1}{}^{\times} B_{11}, \end{array}}{(a, [\text{parallel\^{}right}.s], B_1 \,|[S]|\, B_2)/G \xrightarrow{t_1|\{S\}|\, t_2}{}^{+} B_{11} \,|[S]|\, B_{21}} \quad (18)$$

$$\frac{\begin{array}{c} name(a) \notin (\{S\}\cup\{\delta\}) \\ (a, \text{s}, B_2)/G \xrightarrow{t_2}{}^{+} B_{21}, \\ (t_2 \lfloor (\{S\}\cup\{\delta\}), B_1)/(G\cup\{S\}\cup\{\delta\}) \xrightarrow{t_1}{}^{\times} B_{11}, \end{array}}{(a, [\text{parallel\^{}right}.s], B_1 \,|[S]|\, B_2)/G \xrightarrow{(t_1|\{S\}|\, t_2 \lfloor (\{name(a)\}) \bullet \langle t_2\text{\^{}}\rangle}{}^{+} B_{11} \,|[S]|\, B_{21}} \quad (19)$$

Inference rules (16) and (17) state the fact that the desired action is in B_1 (i.e. parallel^left), where the remainder of the SDP, namely *s*, will guide the inference rules to generate a trace t_1 such that $(a, \text{s}, B_1)/G \xrightarrow{t_1}{}^{+} B_{11}$. Depending whether the gate name of the target action is a member of the synchronization list gate or not, the resulting trace *t* will be $t_1|\{S\}|\, t_2$ or $(t_1 \lfloor \{name(a)\} \,|\{S\}|\, t_2) \bullet \langle t_1\text{\^{}}\rangle$ respectively. This fact guarantees that the target action *a* (or t_1^) will be at the end of the resulting trace. For either case, the resulting trace will be valid only if we can generate a trace t_2 from B_2 such that $t_2 \lfloor (\{S\}\cup\{\delta\})$ *match* $t_1 \lfloor (\{S\}\cup\{\delta\})$ and t_2 does not contain any elements in the restricted set *G*. This can be done efficiently using the relation:

$$(t_1 \lfloor (\{S\}\cup\{\delta\}), B_2)/G \xrightarrow{t_2}{}^{\times} B_{21},$$

where $(\{S\}\cup\{\delta\})$ is added to the restricted set to prevent the generation of unwanted synchronization actions.

Inference rules (18) and (19) are similar to rules (16) and (17) where the inference rules are guided to the right, namely B_2.

Example

Consider behaviour B_0 in Figure 4-6 . We have

$\Sigma(c, B_0)/\{b, c\}$ = {[parallel^left, nested, disable^right, choice^right, prefix, prefix]}

Then the relation $(c, B_0)/\{b, c\} \text{—}t{\rightarrow}^{+} B'$ is defined as:

(c, [parallel^left, nested, disable^right, choice^right, prefix, prefix],B_0)/$\{b, c\}$)$\text{—}t{\rightarrow}^{+} B'$.

This matches rule (16) where the following relations must hold:

1. (c, [nested, disable^right, choice^right, prefix, prefix], B_1)/$\{b, c\}\text{—}t_1{\rightarrow}^{+} B_{r1}$
2. $(t_1 \lfloor \{a, c, \delta\}, B_{17})/(\{b, c\} \cup \{a, c, \delta\}) \text{—}t_2{\rightarrow}^{\times} B_{r2}$

The first relation will be satisfied with two results from the previous example:

1. $t_1 = \langle d,c \rangle$, B_{r1} = **stop**,
2. $t_1 = \langle a, d, c \rangle$, B_{r1} = **stop**.

The second relation then becomes

$(\langle d,c \rangle \lfloor \{a, c, \delta\}, B_{17})/\{a, b, c, \delta\}\text{—}t_2{\rightarrow}^{\times} B_{r2}$, or

$(\langle a, d, c \rangle \lfloor \{a, c, \delta\}, B_{17})/\{a, b, c, \delta\} \text{—}t_2{\rightarrow}^{\times} B_{r2}$

The first relation will not hold since $\Sigma(c, B_{17})/\{a, b, c, \delta\} = \emptyset$, and the second rule will succeed with $t_2 = \langle a, e, c \rangle$, B_{r2} = **stop**. And as a conclusion, the original relation

(c, [parallel^left, nested, disable^right, choice^right, prefix, prefix],B_0)/$\{b, c\}$)$\text{—}t{\rightarrow}^{+} B'$

will hold with:

$t = t_1|\{a, c\}|\, t_2 = \langle a, d, c \rangle|\{a, c\}|\, \langle a, e, c \rangle$, $B' = B_{r1}|[a, c]|B_{r2}$ = **stop**|[a, c]|**stop**. Therefore:

$t \in \{\langle a, d, e, c \rangle, \langle a, e, d, c \rangle\}$.

- **Interleave Parallelism**

$$\frac{(a, s, B_1 \,|[]|\, B_2)/G \xrightarrow{t}{}^{+} B'}{(a, s, B_1 \,|||\, B_2)/G \xrightarrow{t}{}^{+} B'} \qquad (20)$$

The interleave operator is treated as the selected synchronization operator with an empty list of synchronization gates.

- **Full Synchronization**

$$\frac{(a, s, B_1 \,|[\alpha(B_1) \cup \alpha(B_2)]|\, B_2)/G \xrightarrow{t}{}^{+} B'}{(a, s, B_1 \,||\, B_2)/G \xrightarrow{t}{}^{+} B'} \qquad (21)$$

The full synchronization operator is treated as the selected synchronization operator with the list of synchronization gates composed with the alphabet of behaviours B_1 and B_2.

- **Relabeling**

$$\frac{(h\ d_1 ... d_n[P], s, B)/G' \xrightarrow{t}{}^{+} B'}{(g\ d_1 ... d_n[P], [\text{relabel}(h, G').s], (B)[RL])/G \xrightarrow{t[RL]}{}^{+} (B')[RL]} \qquad (22)$$
$$\text{where } RL = g_1/h_1, ..., g_n/h_n$$

In this rule, the gate name *g* of the actual target action and the actual restricted gate set *G* are replaced by *h* and *G'* respectively which are the corresponding formal gates of behaviour *B*. These new elements are found by the static analysis. See section 4.1.

Example

Consider behaviour B_0 of Figure 4-6 . Let

s = [parallel^left, nested, disable^right, choice^right, prefix, prefix]

the relation $(g_3, [\text{relabel}(c, \{b\}).s, B_0[g_1/a, g_2/b, g_3/c, g_4/d, g_5/e])/\{g_2\}) \xrightarrow{t}{}^{+} B'$ matches rule (22) where the relation $(c, s, B_0)/\{b\} \xrightarrow{t_1}{}^{+} B_{r1}$ must hold, as in the previous example, with:

1. $t_1 = \langle a, d, e, c\rangle$, $B' = \textbf{stop}|[a, c]|\textbf{stop}$,
2. $t_1 = \langle a, e, d, c\rangle$, $B' = \textbf{stop}|[a, c]|\textbf{stop}$.

Therefore the original relation

$$(g_3, [\text{relabel}.s], B_0[g_1/a, g_2/b, g_3/c, g_4/d, g_5/e])/\{g_2\}) \xrightarrow{t}^{+} B'$$

will be satisfied with

$$t = t_1[g_1/a, g_2/b, g_3/c, g_4/d, g_5/e], B' = (\textbf{stop}|[a, c]|\textbf{stop})[g_1/a, g_2/b, g_3/c, g_4/d, g_5/e]$$

This implies

$$t \in \{\langle g_1, g_4, g_5, g_3\rangle, \langle g_1, g_5, g_4, g_3\rangle\}.$$

o **Process Instantiation**

$$\frac{\exists p[h_1,\ldots,h_n] := B, (a, [\text{relabel}(h, G').s], (B)[\text{RL}])/G \xrightarrow{t}^{+} B'}{(a, [\text{instance}(h, G').s], p[g_1,\ldots,g_n])/G \xrightarrow{t}^{+} B'} \quad (23)$$

$$\text{where RL} = g_1/h_1, ..., g_n/h_n$$

In this rule the element *instance*(h, G') in replaced by *relabel*(h, G'), since a process instantiation relabels the behaviour of the process and the elements h and G' are the new gate name of the target action and the new restricted gate set found during static analysis.

Example

Given the specification in Figure 4-6 , suppose we want to reach action g_3 without passing by g_2 from the behaviour *testing*$[g_1, g_2, g_3, g_4, g_5]$. This can be specified by the relation

$$(g_3, testing[g_1, g_2, g_3, g_4, g_5])/\{g_2\} \xrightarrow{t}^{+} B'$$

which is defined as

$$(g_3, sdp, testing[g_1, g_2, g_3, g_4, g_5])/\{g_2\} \xrightarrow{t}^{+} B',\text{ where}$$

$sdp \in \Sigma(g_3, testing[g_1, g_2, g_3, g_4, g_5])/\{g_2, g_3\}$

For *sdp* = [instance(*c, {b}*), parallel^left, nested, disable^right, choice^right, prefix, prefix] we have the relation $([\text{instance}(c, \{b\}).s], testing[g_1, g_2, g_3, g_4, g_5])/\{g_2\} \xrightarrow{t}{}^{+} B'$ that matches rule (23) and yields to

$(c, \text{relabel}(c, \{b\}).s, (B_0)[g_1/a, g_2/b, g_3/c, g_4/d, g_5/e])/\{g_2\}\ ) \xrightarrow{t}{}^{+} B'$, where

s = [parallel^left, nested, disable^right, choice^right, prefix, prefix]

This is the same relation we had in the previous section that resulted in:

$t = t_1[g_1/a, g_2/b, g_3/c, g_4/d, g_5/e]$, $B' = (\textbf{stop}|[a, c]|\textbf{stop})[g_1/a, g_2/b, g_3/c, g_4/d, g_5/e]$

This implies

$t \in \{\langle g_1, g_4, g_5, g_3\rangle, \langle g_1, g_5, g_4, g_3\rangle\}$.

The following lemmas sketch the correctness of the guided-inference system definition given above and the goal-oriented execution algorithm presented in section 3.8.

4.2.2 Observations

In the following proofs, we refer to the inference rules of relation $\rightarrow^{+}$ defined in the previous section.

Lemma 4-5: If $(a, sdp, B)/G \xrightarrow{t'}{}^{+} B''$, where $sdp \in \Sigma(a, B, G)$, holds, then $t'^{\wedge} \equiv_s a$.This lemma identifies the fact that the derived traces by the inference rules of relation $\rightarrow^{+}$, do in fact terminate with an action that matches the goal action.

Proof: Static derivation path $sdp \in \Sigma(a, B, G)$ always leads to an action a' that statically matches action a, denoted by $a' \equiv_s a$, using static relabeling. Looking at the guided-inference rules definition, a given $sdp \in \Sigma(a, B, G)$ will direct the execution to derive a sub-trace t'' where $t''^{\wedge} \equiv_s a$, see termination rules (1) and (2). For the other rules, we have the following:

Rule (10): sdp = [hide.s] - states that the desired trace $t' = t''\downarrow\{GL\}$. But since static derivation paths definition guarantees the fact that $name(a) \notin \{GL\}$, and since $t''^{\wedge} \equiv_s$

a, therefore $t'^\wedge \equiv_s a$.

Rule (12): *sdp* = [enable^right.*s*] - in this rule the desired trace $t' = t_1 \downarrow\{\delta\} \bullet t''$, and therefore, $t'^\wedge = t''^\wedge \equiv_s a$.

Rule (15): *sdp* = [disable^right.*s*] - similarly, in this rule the desired trace $t' = t_1 \bullet t''$, and therefore, $t'^\wedge = t''^\wedge \equiv_s a$.

Rule (16): *sdp* = [parallel^left.*s*] and $name(a) \in (\{S\} \cup \{\delta\})$, where *S* is the list of synchronization gates - the desired trace $t' = t''|\{S\}|\, t_2$, where $(t'' \lfloor (\{S\} \cup \{\delta\}), B_2)/G \xrightarrow{t_2}{}^{\times} B_{21}$. Since $t''^\wedge \equiv_s a$ and $name(a) \in (\{S\} \cup \{\delta\})$, then $(t'' \lfloor (\{S\} \cup \{\delta\}))^\wedge \equiv_s a$, and therefore from the definition of $\rightarrow^{\times}$, we will have $t_2^\wedge \equiv_s a$ and $t_2 \lfloor (\{S\} \cup \{\delta\}$ *match* $t'' \lfloor (\{S\} \cup \{\delta\}$. This implies $t'^\wedge \equiv_s (t''|\{S\}|\, t_2)^\wedge \equiv_s a$.

Rule (17): *sdp* = [parallel^left.*s*] and $name(a) \notin (\{S\} \cup \{\delta\})$, where *S* is the list of synchronization gates - the desired trace in this rule is $t' = (t'' \lfloor \{name(a)\}\ |\{S\}|\, t_2) \bullet \langle t''^\wedge \rangle$. Since $t''^\wedge \equiv_s a$, therefore $t'^\wedge \equiv_s a$.

Rule (18) and rule (19) are similar to rule (16) and rule (17) respectively where *sdp* = [parallel^right.*s*].

All the other rules state the fact that the desired trace $t' = t''$, and therefore $t'^\wedge \equiv_s a$.

o

Lemma 4-6: If $(a, sdp, B)/G \xrightarrow{t' \bullet \langle a' \rangle}{}^{+} B''$, where $sdp \in \Sigma(a, B, G)$, holds, then $\forall b \in G$, *not(b in t')*. This lemma identifies the fact that the derived traces by the inference rules of relation $\rightarrow^{+}$, do not contain any action, other than the last action $t'^\wedge$, having a gate name in *G*.

Proof: A given $sdp \in \Sigma(a, B, G)$ will direct the execution of the guided-inference system definition to derive a sub-trace $t_1 \bullet \langle a_1 \rangle$. Static derivation paths definition guarantees the fact that $\forall b \in G$, *not(b in* t_1*)*. When other sub-trace t_2 is needed to derive the desired trace $t' \bullet \langle a' \rangle$, the restricted set is carried out to guarantee the fact that $\forall b \in G$, *not(b in* t_2*)*, except for rule (16) and rule (18) where *name(a)* is a member of the synchronization set. In these rules the sub-trace t_2 will have the form $t_2' \bullet \langle a_2' \rangle$ where $\forall b \in G$, *not(b in* t_2'*)*. Therefore, the desired trace $t' \bullet \langle a' \rangle$ will be equal to

$t_1 \bullet \langle a_1 \rangle |\{S\}| t_2' \bullet \langle a_2' \rangle$, with $name(a) = name(a') = name(a_2') \in \{S\}$ and which is equivalent to:

$t' \bullet \langle a' \rangle = (t_1|\{S\}|t_2') \bullet \langle a_1 \uparrow_s a_2' \rangle$.

Since $\forall b \in G$, *not(b in* t_1*)* and *not(b in* t_2'*)*, then $\forall b \in G$, *not(b in* $t_1|\{S\}|t_2'$*)*. Which implies $\forall b \in G$, *not(b in t')*

o

Lemma 4-7: If $(a, sdp, B)/G \longrightarrow\!\langle a_1',\ldots,a_n'\rangle\!\rightarrow^+ B''$ holds, where $sdp \in \Sigma(a, B, G)$, and there exists a substitution σ such that $eval(\rho(\langle a_1',\ldots,a_n'\rangle)\sigma) = true$, then there exists a trace $\langle a_1,\ldots,a_n\rangle$ such that $B \longrightarrow\!\langle a_1,\ldots,a_n\rangle \rightarrow B'$ holds with $name(a_i) \notin$ G for 1≤i≤n-1 and $a_n \equiv a$, $\langle a_1,\ldots,a_n\rangle = \tau(\langle a_1',\ldots,a_n'\rangle)\sigma$, and $B' = B''\sigma$.

Proof: Here we only give a sketch of the proof. When $(a, sdp, B)/G \longrightarrow\!\langle a_1',\ldots,a_n'\rangle\!\rightarrow^+ B''$ holds using the definition in section 4.2.1, and a substitution σ is found by the narrower such that $eval(\rho(\langle a_1',\ldots,a_n'\rangle)\sigma) = true$, then all encountered guards and predicates that are accumulated in the variable trace $\langle a_1',\ldots,a_n'\rangle$, plus the action synchronization conditions, see the definition of Λ in section 3.2.2, are satisfied. Now, if the definition of $\rightarrow$, defined in section 3.3, follows the same derivation path of the variable trace $\langle a_1',\ldots,a_n'\rangle$ and uses the values in the substitution σ when needed, then a trace $\langle a_1,\ldots,a_n\rangle$ will be generated and, obviously, $a_i \equiv_s a_i'$ for 1≤i≤n. This implies from lemma 4-5 and lemma 4-6 that $name(a_i) \notin$ G for 1≤i≤n-1 and $a_n \equiv a$.

Therefore, a detailed proof of this lemma would imply a comparison between the inference rules of relation $\rightarrow^+$ and those of relation $\rightarrow$. We leave to the reader.

o

Lemma 4-8: If $(a, sdp, B)/G \longrightarrow t' \rightarrow^+ B''$ holds, where $sdp \in \Sigma(a, B, G)$, and there exists a substitution σ such that $eval(\rho(t')\sigma) = true$, then there exist a trace t such that $(a, B)/G = t \Rightarrow^+ B'$ holds with $t = (\tau(t')\sigma)\lceil\{\mathbf{i}\}$ and $B' = B''\sigma$.

Proof: If $(a, sdp, B)/G \longrightarrow t' \rightarrow^+ B''$ holds, where $sdp \in \Sigma(a, B, G)$, and there exists a substitution σ such that $eval(\rho(\langle a_1',\ldots,a_m'\rangle)\sigma) = true$, then there exist a trace t_2 such that $B \longrightarrow t_2 \rightarrow B'$ holds with $t_2 = \tau(t')\sigma$ and $B' = B''\sigma$, see lemma 4-7. As defined in section 3.4, $(a, B)/G = t \Rightarrow^+ B'$, iff $B = t \Rightarrow B'$ such that $name(b) \notin$ G for all *b in t*. And we have $B = t \Rightarrow B'$ iff $B \longrightarrow t_2 \rightarrow B'$ such that $t = t_2\lceil\{\mathbf{i}\} = (\tau(t')\sigma)\lceil\{\mathbf{i}\}$. Therefore this lemma holds.

o

4.2.3 Limitations

Guided-inference system execution may not terminate for two reasons:

1- When static derivation paths cycle in an unfeasible path. This problem was resolved by limiting the number of identical process instantiations involved in deriving the static derivation paths, see section 4.1.3.

2- When relation $(\langle\rangle, B)/G \text{—}t\rightarrow^{\times} B'$ is involved in deriving the traces. This relation is defined as:

$$(\langle\rangle, B)/G \text{—}\langle\rangle\rightarrow^{\times} B \qquad (1)$$

$$\frac{(*, B)/G \text{—}t_1\rightarrow^{+} B_{2,} \quad (\langle\rangle, B_2)/G \text{—}t_2\rightarrow^{\times} B'}{(\langle\rangle, B)/G \text{—}t_1\bullet\, t_2\rightarrow^{\times} B'} \qquad (2)$$

where ‘*’ stands for any action

The above definition of relation $(\langle\rangle, B)/G \text{—}t\rightarrow^{\times} B'$ may derive infinite number of undesirable sub-traces. For example, let

A[a, b] := a; A[a, b] [] b; ***exit***

D[a, b, c] := (A[a, b] [> c; ***exit*** *)|[a,b,c]| a; b; c;* ***exit***

The relation $(c, D[a,b,c])/\{\} \text{—}t\rightarrow^{+} B'$ has an obvious solution $t = \langle a, b, c\rangle$. But since the first static derivation path in $\Sigma(c, D[a,b,c])/\{c\}$ will guide the inference rules through parallel^left then through disable^right, the following rules will be executed:

1. Rule (16), for parallel^left, states that the trace t_1 derived from the left behaviour (leading to action c) must synchronize with a trace t_2 derived from the right behaviour on gates *[a,b,c]*. More formally, this rule tries to satisfy $t= t_1|\{a,b,c\}|t_2$, such that $t_1\lfloor\{a,b,c,\delta\} = t_2\lfloor\{a,b,c,\delta\}$, and $t_1{}^\wedge = c$. This can succeed only if $t_1= \langle a, b, c\rangle$.
2. Rule (16), for disable^right, is responsible of deriving the sub-trace t_1 needed by rule (16) above. Rule (15) states that $t_1= t_{11}\bullet t_{12}$, where $t_{12} = \langle c\rangle$ derived from the right behaviour, and t_{11} is derived from the relation $(\langle\rangle, A[a,b])/\{\delta\} \text{—}t\rightarrow^{\times} B'$. This relation produces an infinite number of traces, namely,

$t_{11} \in \{\langle\rangle, \langle a\rangle, \langle a, a\rangle, \langle a, a, a\rangle, \langle a, a, a, a\rangle, \langle a, a, a, a, a\rangle,...\}$, this implies

$t_1 \in \{\langle c\rangle, \langle a, c\rangle, \langle a, a, c\rangle, \langle a, a, a, c\rangle, \langle a, a, a, a, c\rangle, \langle a, a, a, a, a, c\rangle, ...\}$,

and therefore the desired sub-trace $t_1 = \langle a, b, c\rangle$ cannot be derived, since the $(\langle\rangle, A[a,b])/\{\delta\} \xrightarrow{t}{}^{\times} B'$, failed to produce the sub-trace $\langle a, b\rangle$.

We have limited the number of solutions for relation $(\langle\rangle, B)/G \xrightarrow{t}{}^{\times} B'$ by removing the recursion in its definition as follows:

$$(\langle\rangle, B)/G \xrightarrow{\langle\rangle}{}^{\times} B \tag{1}$$

$$\frac{(*, B)/G \xrightarrow{t}{}^{+} B'}{(\langle\rangle, B)/G \xrightarrow{t}{}^{\times} B'} \tag{2}$$

where ‘*’ stands for any action

Rule (1) simply states that $(\langle\rangle, B)/G \xrightarrow{t}{}^{\times} B'$ will always hold with $t=\langle\rangle$, and $B'=B$. The second rule states that $(\langle\rangle, B)/G \xrightarrow{t}{}^{\times} B'$ will hold if $(*, B)/G \xrightarrow{t}{}^{+} B'$ holds, where ‘*’ stands for any action. The number of solutions for $(*, B)/G \xrightarrow{t}{}^{+} B'$ is finite due to the fact that $\Sigma(*, B, G)$ is finite with the added heuristics defined in section 4.1.3.

Now consider the above relation $(c, D[a,b,c])/\{\} \xrightarrow{t}{}^{+} B'$ where the number of identical process instantiations for static derivation paths is limited to 2. The first static derivation path in $\Sigma(c, D[a,b,c])/\{c\}$ will guide the inference rules through parallel^left then through disable^right, so the following rules will be executed:

1. Rule (16), for parallel^left, states that the trace t_1 derived from the left behaviour (leading to action c) must synchronize with a trace t_2 derived from the right behaviour on gates $[a,b,c]$. More formally, this rule tries to satisfy $t = t_1|\{a,b,c\}|t_2$, such that $t_1 \lfloor \{a,b,c,\delta\} = t_2 \lfloor \{a,b,c,\delta\}$, and $t_1{}^\wedge = c$.
2. Rule (15), for disable^right, is responsible of deriving the sub-trace t_1 needed by rule (16) above. Rule (15) states that $t_1 = t_{11} \bullet t_{12}$, where $t_{12} = \langle c\rangle$ derived from the right behaviour, and t_{11} is derived from the relation $(\langle\rangle, A[a,b])/\{\delta\} \xrightarrow{t}{}^{\times} B'$. This relation produces the following traces:

   $t_{11} \in \{\langle\rangle, \langle a\rangle, \langle a, a\rangle, \langle a, b\rangle\}$,

   this implies

$t_1 \in \{\langle c\rangle, \langle a, c\rangle, \langle a, a, c\rangle, \langle a, b, c\rangle\}$,

and therefore, back to step 1 above, $t = t_1|\{a,b,c\}|t_2$, such that $t_1 \lfloor \{a,b,c,\delta\} = t_2 \lfloor \{a,b,c,\delta\}$ and $t_1{}^\wedge = c$, can be satisfied with $t_1 = \langle a, b, c\rangle$ producing the desired solution $t = \langle a, b, c\rangle$.

Note that any other bound on the number of identical process instantiations would have given the same final result.

Chapter 5 Narrowing Technique

5.1 Introduction

This chapter is devoted to the implementation technique of ERNAL (An Engine to Rewrite and Narrow the ADTs of LOTOS). As mentioned in chapter 3, ERNAL is an automatically generated Prolog implementation tool capable of efficiently evaluating and narrowing equations specified in LOTOS data types. The implementation is obtained by applying transformation techniques to a given term rewriting system.

Efficient narrowing techniques have been studied extensively in the literature [34][35][36][153], and an analysis of this subject would be a doctoral thesis in itself. Here we discuss a new approach, which we do not claim to be optimal, but which provides good results with relatively simple transformations.

SVELDA, our current ADT interpreter, uses an inner-most left-to-right evaluation strategy and it can only be used to rewrite terms and not to solve equations. ERNAL, on the other hand, will adopt outer-most *best-order* evaluation strategy. We call the operands evaluation order "best-order" since it is not specific and it depends on factors described later in this chapter.

Figure 5-1 shows the order in which the operators are evaluated using inner-most left-to-right strategy and outer-most left-to-right strategy respectively for the expression

(succ(0) > succ(succ(0))) and (succ(0) < 0)

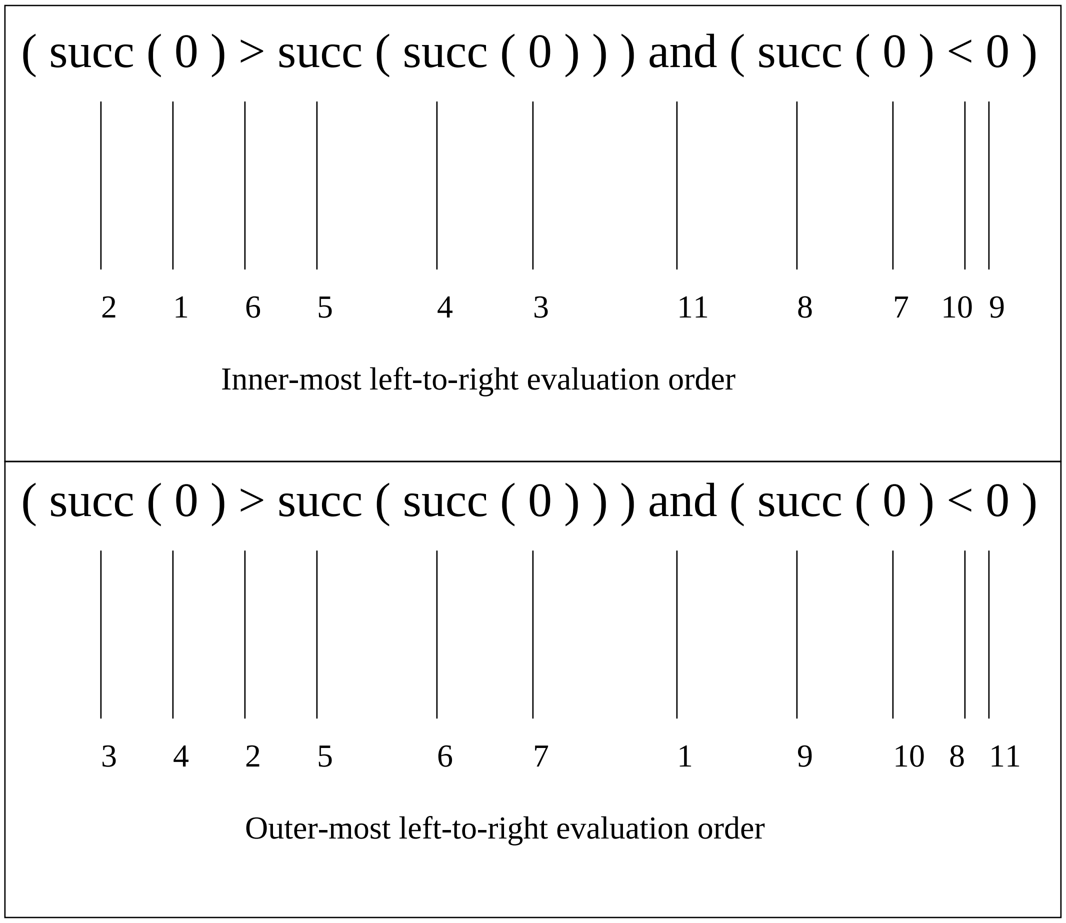

(succ (0) > succ (succ (0))) and (succ (0) < 0)
2 1 6 5 4 3 11 8 7 10 9
Inner-most left-to-right evaluation order
(succ (0) > succ (succ (0))) and (succ (0) < 0)
3 4 2 5 6 7 1 9 10 8 11
Outer-most left-to-right evaluation order

Figure 5-1 Orders of evaluation

We chose the outer-most evaluation for the following reasons (see section 3.7.3):

a- It detects *short-circuit*. That is to say that it does not evaluate some operands if they do not need to be evaluated in order to obtain the result, i.e. best-order evaluation.

b- Unlike the inner-most evaluation, it does not require the use of de-structuring and re-structuring Prolog operator '=..' to ungroup and re-group ADT expressions.

c- It is capable of producing the most general substitution, called the most general incomplete structure in Prolog. Such a substitution is useful for narrowing. For example, the most general solution of the goal *X > succ(0)* will be *X = succ(succ(Y))*, where *Y* is a variable representing any natural number.

ERNAL is a set of *eval/2* Prolog clauses of the form:

eval(EXP:S, RES)

where *RES* is the resulting evaluation of the ADT expression EXP of sort *S*. For readability, we use external names in our examples.

The transformation techniques used to obtain the desired ERNAL implementation are given in the next section. In section 5.3, an evaluation of these techniques is reported. Finally, section 5.4 lists the limitations of ERNAL's implementation.

5.2 Transformation Phases

The transformation of a TRS into an evaluator/narrower implementation consists of the following phases, see Figure 5-2 :

1- *Reordering Phase*: For a given TRS $R \in RS$, reorder the underlying TRRs of each operator $f \in F$, where *F* is a set of defined operators in *R*, using a function $ORD : RS \rightarrow RS$ discussed below.

2- *One-to-One Mapping Phase*: Transform the reordered TRRs into *eval/2* Prolog clauses using a function $TR : RS \rightarrow E$.

3- *Merging Phase*: For each operator $f \in F$, merge all its *eval/2* clauses into one *eval/2* clause using a function $MRG : E \rightarrow E$. Note that *MRG* is a *many-to-one* mapping function.

Note that, as explained in section 3.7.2 of chapter 3, we assume that the ADT equations have already been properly oriented as rewriting rules.

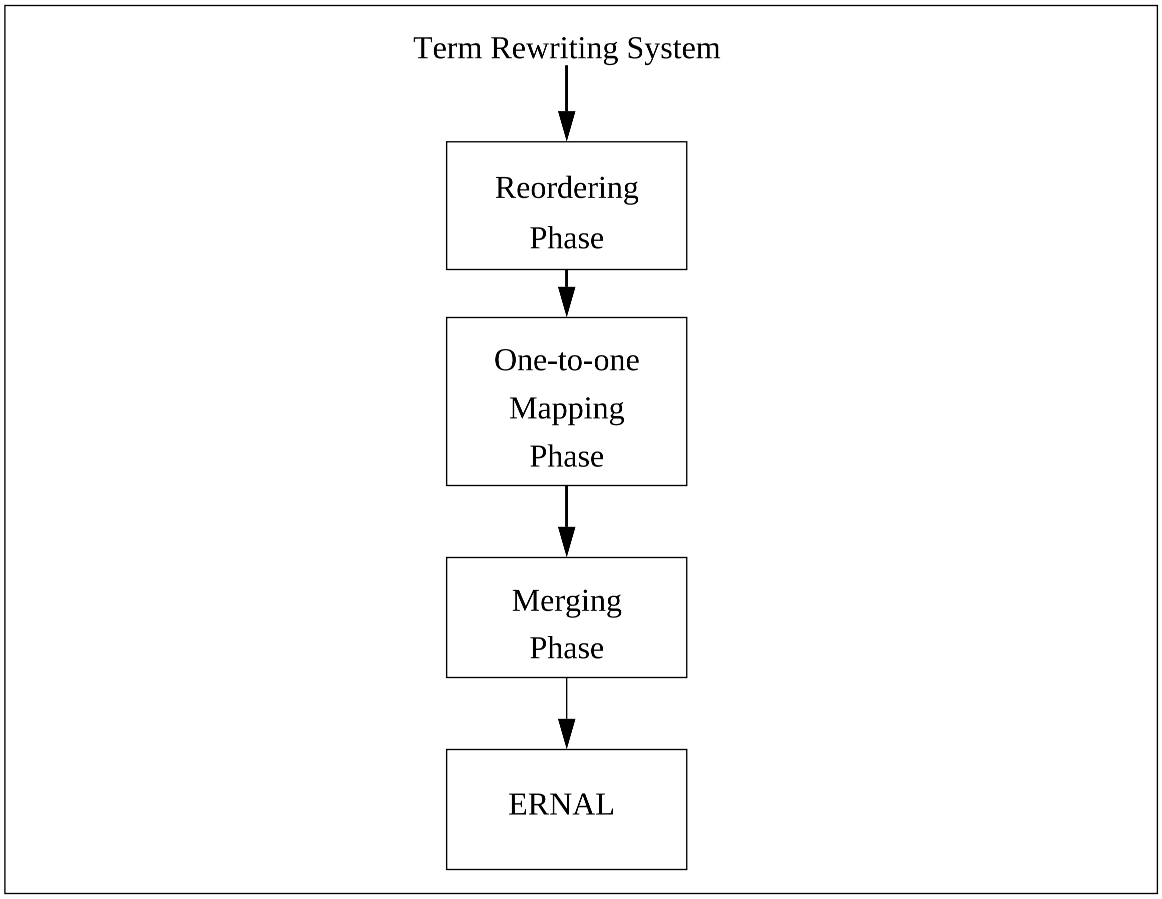


Figure 5-2 Transformation Phases

5.2.1 Reordering Phase

The order of rewriting rules may affect the execution performance. For example, looking at the TRRs of the ++ operator in Figure 3-8 given in previous chapter, one sees that a short-circuit can be detected by the second rule, therefore it's more reasonable to evaluate this rule first. Non-termination may also be caused by the order of rewriting rules. To take into consideration these factors, we have devised a sorting criterion, defined by a function $ORD : RS \rightarrow RS$, to sort the TRRs for each operator using three numerical keys that will be assigned to each rule. These numerical keys are defined by the following functions:

- $SC : R \rightarrow N$, called *short-circuit* function, that returns the number of variable operands of the defined operator occurring on the left hand side of the rule $r \in$ R but which do not occur on its right hand side nor in its condition part (for conditional rewriting rules). These operands are those that do not need to be evaluated to obtain the result. For example, the values of this function on the two rules of the ++ operator defined in Figure 3-8 , are 0 and 1 respectively.

This because the first rule has the variable C which appears on both sides, while in the second rule the variable C appears only on the left hand side. For readability, the values of a sorting key function for a given operator with *k* rewriting rules is represented as (n_1, n_2, ..., n_k), where n_i is the value assigned to the *i*th rule. The values of short-circuit function applied on the rules of the ++, <, ==, >=, -, and the *mod* operators defined in Figure 3-8 are (0, 1), (0, 0, 1), (0, 0, 0, 0), (0), (0, 0, 1), and (0, 0) respectively. We believe that this is the most important sorting key, since it forces to detect short-circuits early, and thus, evaluation will be more efficient.

- $GT : R \rightarrow N$, called *ground-terms* function that returns the number of ground term operands of the left hand part of the rule $r \in$ R. This is useful for narrowing when free variables can be instantiated with ground terms. In Figure 3-8 , the values of ground-terms function applied on the rules of the ++, <, ==, >=, -, and the *mod* operators are (1,1), (0, 1, 1), (0, 2, 1, 1), (0), (1, 0, 1), and (0,0) respectively.
- $XT : R \rightarrow I$, where $r \in$ R, called *complexity* function, that returns the complexity of a rule with respect to the number of operators and operands in its condition and right-hand part. By using this function as a sorting key, we will evaluate the most complex rule last. In Figure 3-8 , the values of complexity function applied on the rules of the ++, <, ==, >=, -, and the *mod* operators are (1,1), (3, 1, 1), (3, 1, 1, 1), (7), (1, 3, 1), and (8, 4) respectively.

	SC	*GT*	*XT*
false ++ C → *C;*	*0*	*1*	*1*
true ++ C → *true;*	*1*	*1*	*1*
succ(M) < succ(N) → *M < N;*	*0*	*0*	*3*
0 < succ(M) → *true;*	*0*	*1*	*1*
M < 0 → *false;*	*1*	*1*	*1*
succ(M) == succ(N) → *M == N;*	*0*	*0*	*3*
0 == 0 → *true;*	*0*	*2*	*1*
0 == succ(M) → *false;*	*0*	*1*	*1*
succ(M) == 0 → *false;*	*0*	*1*	*1*
M >= N → *(N < M) or (M==N);*	*0*	*0*	*7*
M - 0 → *M;*	*0*	*1*	*1*
succ(M) - succ(N) → *M - N;*	*0*	*0*	*3*
0 - M → *0;*	*1*	*1*	*1*
(M >= N) => M mod N → *(M - N) mod N;*	*0*	*0*	*8*
(M < N) => M mod N → *M;*	*0*	*0*	*4*

Figure 5-3 A Term Rewriting System With Sorting Keys Assignment

ORD: *RS*→*RS* is defined as follows:

1- For each set of rules RS $\subseteq$ R of an operator *f*, reorder them in <u>descending</u> order with respect to key defined by the function *SC*, giving a TRS R1.

2- For each set of rules RS2 $\subseteq$ R1 of an operator *f* with an identical *SC* value, reorder them in <u>descending</u> order with respect to key defined by the function *GT*, giving a TRS R2.

3- For each set of rules RS3 $\subseteq$ R2 of an operator *f* with identical *SC* and *GT* values, reorder them in <u>ascending</u> order with respect to key defined by the function *XT*, giving a TRS R3.

R3 is the required ordering of TRS R. Figure 5-4 illustrates the reordering of rules in Figure 5-3 .

			SC	GT	XT
true ++ C		$\rightarrow$ *true;*	1	1	1
false ++ C		$\rightarrow$ *C;*	0	1	1
M	*< 0*	$\rightarrow$ *false;*	1	1	1
0	*< succ(M)*	$\rightarrow$ *true;*	0	1	1
succ(M)	*< succ(N)*	$\rightarrow$ *M < N;*	0	0	3
0	*== 0*	$\rightarrow$ *true;*	0	2	1
succ(M)	*== 0*	$\rightarrow$ *false;*	0	1	1
0	*== succ(M)*	$\rightarrow$ *false;*	0	1	1
succ(M)	*== succ(N)*	$\rightarrow$ *M == N;*	0	0	3
M	*>= N*	$\rightarrow$ *(N < M) ++ (M==N);*	0	0	7
0	*- M*	$\rightarrow$ *0;*	1	1	1
M	*- 0*	$\rightarrow$ *M;*	0	1	1
succ(M)	*- succ(N)*	$\rightarrow$ *M - N;*	0	0	3
(M < N) => M mod N		$\rightarrow$ *M;*	0	0	4
(M >= N) => M mod N		$\rightarrow$ *(M - N) mod N;*	0	0	8

Figure 5-4 Reordering of rewriting rules of Figure 5-3

5.2.2 One-to-One Mapping Phase

This phase translates each rule in the reordered TRS into a Prolog *eval/2* clause. We refer the reader to section 3.7.2 for basic definitions to be used below.

We have chosen the logic programming language Prolog for the implementation of our evaluator/narrower since it has the expressive power of combining conditional rewriting (or evaluation), to perform functional simplification, and conditional narrowing, to generate solutions to goals [35].

Let:

a- *r* be a rewriting rule of the form $p \Rightarrow f(x_1,...,x_m) \rightarrow g$, where *p* is a term of sort *bool* and, $f(\bar{x})$ and *g* are terms of the same sort, $\bar{x} = x_1,...,x_n$

b- $y_1,...,y_k$ for $k \leq n$ denote the non-variable arguments of f in $x_1,...,x_n$, in their given order. We call “non-variables” all terms that are not simple variables.

c- $\overline{X}= X_1,...,X_n$ be a set of unique Prolog variables associated with $x_1,...,x_n$ respectively.

d- *RES* denote a unique Prolog variable that will be used to hold the result of an evaluation.

e- $ST{:}\tau(F,V) \to S$ be a function that maps terms into their associated sorts.

f- $V{:}\overline{x} \to \overline{X}$ be a function that maps an argument x_i into its associated Prolog variable X_i as follows:

- if x_i is a non-variable then X_i is a unique Prolog variable.

- if x_i is a variable then $X_i=x_i$.

The transformation function $TR{:}\ R \to E$, where R is a TRS and E is a set of *eval/2* clauses, is defined as follows:

TR(*r*), where $r \in R$ of the form $p \Rightarrow f(x_1,...,x_m) \to g$, is equal to

eval($f(X_1,...,X_m)$:*ST(f)*, *RES*) :-

eval(*p:bool, true*), -- for the condition

eval($V(y_1)$:$ST(y_1)$, y_1), -- for non-variable arguments

...

eval($V(y_k)$:$ST(y_k)$, y_k),

eval(*g:ST(f)*, *RES*). -- for the right hand side

The variable arguments are left as such. The transformation of the rewriting rules in Figure 5-4 is given in Figure 5-5 .

```
eval(A ++ B : 'Bool',C) :-
    eval(A : 'Bool',true),
    eval(true : 'Bool',C).
eval(A ++ B : 'Bool',C) :-
    eval(A : 'Bool',false),
    eval(B : 'Bool',C).
%
eval(A < B : 'Bool',C) :-
    eval(B : nat,0),
    eval(false : 'Bool',C).
eval(A < B : 'Bool',C) :-
    eval(A : nat,0),
    eval(B : nat,succ(D)),
    eval(true : 'Bool',C).
eval(A < B : 'Bool',C) :-
    eval(A : nat,succ(D)),
    eval(B : nat,succ(E)),
    eval(D < E : 'Bool',C).
%
eval((A == B) : 'Bool',C) :-
    eval(A : nat,0),
    eval(B : nat,0),
    eval(true : 'Bool',C).
eval((A == B) : 'Bool',C) :-
    eval(A : nat,succ(D)),
    eval(B : nat,0),
    eval(false : 'Bool',C).
eval((A == B) : 'Bool',C) :-
    eval(A : nat,0),
    eval(B : nat,succ(D)),
    eval(false : 'Bool',C).
eval((A == B) : 'Bool',C) :-
    eval(A : nat,succ(D)),
    eval(B : nat,succ(E)),
    eval((D == E) : 'Bool',C).
%
eval(A >= B : 'Bool',C) :-
    eval((B < A) ++ (A == B) : 'Bool',C).
%
eval((A - B) : nat,C) :-
    eval(A : nat,0),
    eval(0 : nat,C).
eval((A - B) : nat,C) :-
    eval(B : nat,0),
    eval(A : nat,C).
eval((A - B) : nat,C) :-
    eval(A : nat,succ(D)),
    eval(B : nat,succ(E)),
    eval((D - E) : nat,C).
%
eval(A mod B : nat,C) :-
   eval(A < B : 'Bool',true),
   eval(A : nat,C).
eval(A mod B : nat,C) :-
   eval(A >= B : 'Bool',true),
   eval((A - B) mod B : nat,C).
```

Figure 5-5 eval/2 transformation of rewriting rules of Figure 5-4

5.2.3 Merging Phase

One-to-one mapping leads to poor performance since some operands may be evaluated many times before the result is achieved. For example the left operand *A* of the operator ++ in Figure 5-5 is evaluated twice if it happens to be evaluated to *false*. For this reason, as well as for narrowing purposes, this phase is needed. It resolves the problem of re-evaluation of operands by detecting such operands and by evaluating them before applying the rules. The rules then use the evaluated operands.

Let $Eop \subseteq E$ be the ordered set of all *eval/2* clauses of an operator $f \in F$, and let *n* denote the number of such clauses. *MRG(Eop)* is defined as follows:

1- The head of the resulting *eval/2* clause is the unification of the heads of all *eval/2* clauses in *Eop*. This provides association of variables between the bodies.
2- The body of the resulting *eval/2* clause will have the following form:

```
    $Evaluated\_Operands_1$,
            ($Sub\_Results\_Unification_1$, $Resulting\_Clause_1$
            | $Evaluated\_Operands_2$,
                    ($Sub\_Results\_Unification_2$, $Resulting\_Clause_2$
            ...
            ...
                    | $Evaluated\_Operands_n$,
                            ($Sub\_Results\_Unification_n$, $Resulting\_Clause_n$
                            | RES = $f(Y_1,...,Y_m)$
                            )
            ...
            ...
                    )
            ).
```

Where, informally, $Evaluated\_Operands_i$ is the evaluation of the operands that are in $eval/2_i \in Eop$ and not in $eval/2_j$ for j < i. The evaluation results in $Evaluated\_Operands_i$ will be replaced by unique Prolog variables and are checked (or instantiated in case of narrowing) in $Sub\_Results\_Unification_n$ using '=' Prolog Operator. $Resulting\_Clause_i$ is identical to the last clause in *eval/2*$_i \in$ *Eop*. RES = $f(Y_1,...,Y_m)$ where Y_k is equal to the evaluation of the operand X_k if done by *Evaluated_Operands*$_i$, and is equal to X_k otherwise. This term is added in case of incomplete definition of the operator *f*.

Merged eval/2 clauses for the rewriting rules presented in Figure 5-5 , are given in Figure 5-6 .

```
eval(A ++ B : 'Bool',C) :-
    eval(A : 'Bool',D),
    ( D = true,
      eval(true : 'Bool',C)
    ; D = false,
      eval(B : 'Bool',C)
    ; C = D ++ B
    ).
eval(A < B : 'Bool',C) :-
    eval(B : nat,D),
    ( D = 0,
      eval(false : 'Bool',C)
    ; eval(A : nat,E),
      ( E = 0,
        D = succ(F),
        eval(true : 'Bool',C)
      ; E = succ(G),
        D = succ(H),
        eval(G < H : 'Bool',C)
      ; C = E < D
      )
    ).
eval((A == B) : 'Bool',C) :-
    eval(A : nat,D),
    eval(B : nat,E),
    ( D = 0,
      E = 0,
      eval(true : 'Bool',C)
    ; D = succ(F),
      E = 0,
      eval(false : 'Bool',C)
    ; D = 0,
      E = succ(G),
      eval(false : 'Bool',C)
    ; D = succ(H),
      E = succ(I),
      eval((H == I) : 'Bool',C)
    ; C = (D == E)
    ).
eval(A >= B : 'Bool',C) :-
    ( eval((B < A) ++ (A == B) : 'Bool',C)
    ; C = A >= B
    ).
eval((A - B) : nat,C) :-
    eval(A : nat,D),
    ( D = 0,
      eval(0 : nat,C)
    ; eval(B : nat,E),
      ( E = 0,
        C = D
      ; D = succ(F),
        E = succ(G),
        eval((F - G) : nat,C)
      ; C = D - E
      )
    ).
eval(A mod B : nat,C) :-
    ( eval(A < B : 'Bool',true),
      eval(A : nat,C)
    ; eval(A >= B : 'Bool',true),
      eval((A - B) mod B : nat,C)
    ; C = A mod B
    ).
```

Figure 5-6 Merged eval/2 defined in Figure 5-5

Clauses of *eval/2* are also created for the operators that do not have any rewriting rules (i.e. constructors). These clauses are included prior to the previous clauses. They are created as follows:

1- For every sort *s*, reorder all its operators that are not yet translated (i.e. the ones that do not have any TRRs) in increasing order with respect to their arity.
2- Create an *eval/2* clause for each operator, and add such clauses in the given order before the eval/2 clauses created for the operators with TRRs.

Figure 5-7 shows the eval/2 clauses for the constructors *0* and *succ* of sort *nat*, and *true* and *false* of sort *Bool*.

```
eval(0:nat, 0).
eval(succ(X):nat, succ(Y)) :- eval(X, Y).

eval(true:’Bool’, true).
eval(false:’Bool’, false).
```

Figure 5-7 eval/2 clauses for Constructors

Figure 5-8 shows systematically the order in which eval/2 predicate will resolve the goal:

$$eval((succ(0) >= succ^2(X) ++ (succ^2(0) < X)):bool, true)$$

using the automatically derived implementation given in Figure 5-6 and Figure 5-7 .

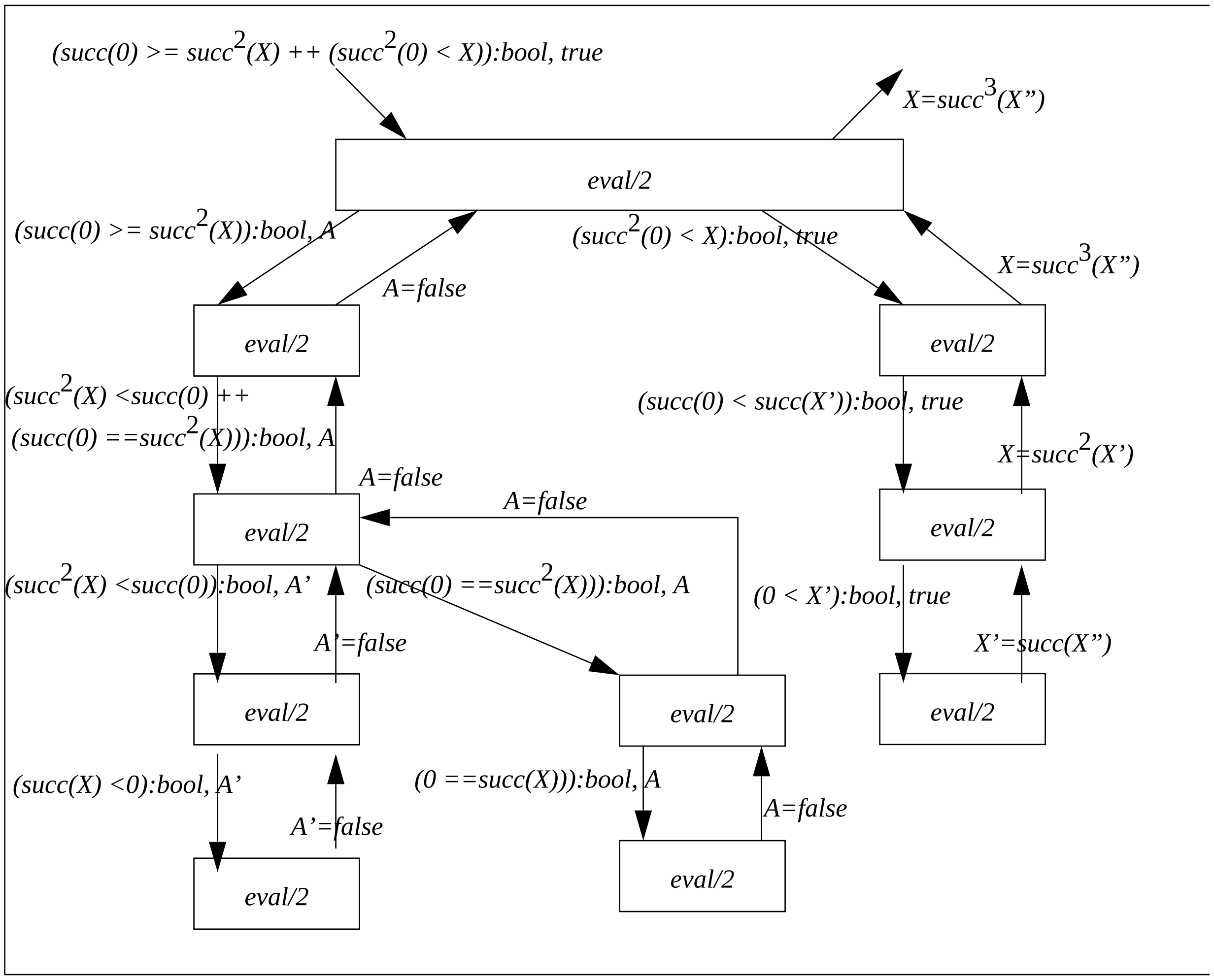


Figure 5-8 Instantiation of Free Variables

As a result, the free variable *X* in the above goal is instantiated to the general solution succ^3(*X”*) for any natural number *X”*, i.e. *X* = *Y* for *Y* >= 3. ERNAL provides textual derivation traces showing the execution steps leading to a conclusion (or a non-conclusion) for a given goal.

5.3 Comparison

Another observation about the transformation defined in the previous section, is that evaluated sub-terms may be re-evaluated when carried out recursively to another *eval/2*. For example, *eval(X-Y,Res)* resolves the rule:

$$succ(M) \quad - succ(N) \qquad \rightarrow M - N;$$

by first evaluating *X* and *Y* and then matching them with *succ(M)* and *succ(N)* respectively. If the matching succeeds then *M* and *N* will be unified to already evaluated terms. So, to avoid re-evaluation of *M* and *N* in recursive *eval/2* call, i.e. *eval(M-N, Res)*, *M* and *N* will be tagged to identify that they are already evaluated.

Different transformation versions were evaluated by applying them on the same set of ADT expressions. The evaluation is judged by the number of *eval/2* invocations. The transformation versions are:

version 1: is One-to-One mapping without reordering.

version 2: is N-to-One mapping without reordering. i.e. Applying the merging phase on version 1.

version 3: is N-to-One mapping with reordering. i.e. eval/2 clauses in Figure 5-6 .

version 4: N-to-One mapping with tagging the evaluated sub-terms.

The following table summarizes such comparison.

Table 3: Transformation Comparison

Expression	Version 1	Version 2	Version 3	Version 4
$(succ^{2}(0) >= succ(0)):Bool$	30	14	14	13
$((succ(0) >= succ^{2}(0))++ (succ^{3}(0) < succ^{4}(0))):Bool$	60	54	52	42
$((succ^{3}(0) < succ^{4}(0)) ++ (succ(0) >= succ^{2}(0))):Bool$	69	31	31	22
$((succ^{7}(0)\ mod\ succ^{3}(0)) - succ(0)):nat$	1576	456	365	219
$(((succ^{10}(0)\ mod\ succ^{6}(0)) - succ(0)) == succ^{3}(0):Bool$	1505	677	444	199

The above table demonstrates that Merging phase, Reordering phase and tagging do indeed increase the overall evaluation efficiency.

Version 4 is the one used for ERNAL.

5.4 Limitations

For our purposes of automatic LOTOS behaviour validation, we are mainly concerned about validating guards and selection predicates accumulated during trace derivations. They are usually conjunctions of conditions resulting from matching a number of actions together. So, the general form of the narrower that will be used in our validation is:

C1 ^ C2 ^ ... Cn >><< *true*

which may lead us to one of the following cases:
a- instantiating free variables in C1, C2, .., and Cn such that their evaluation is *true*
b- determining that there is no solution
c- or, unfortunately, inability to find a solution.

The third case is when the evaluation does not terminate. This is due to the infinite search space that can be caused by one of two reasons:

1- Encountering an infinite branch in the search tree when looking for a solution.

2- Encountering a sub-term that generates an infinite number of non-desirable solutions.

An example of non-terminating evaluation causes by reason 2 is

(X > succ(0)) and (X < succ(succ(0))):bool >><< true,

assuming that the evaluation order of the *and* operator happened to be left-to-right. This is because the left condition has an infinite number of solutions, represented initially by the most general solution *X=succ(succ(Y)).* The right condition cannot be satisfied with any natural number value of *Y*.

To deal with this limitation, ERNAL’s transformation includes controls to limit the backtracking, i.e. limiting the number of solutions, and to limit the number of invocation, i.e. limiting the length of search paths.

We expect that experienced tool users will understand these limitations of the tool, and will use an appropriate specification style. For example, the problem discussed above will not occur if the two arguments of the *and* operator are reversed.

Chapter 6 Goal-Oriented Execution Applications

An arbitrary LOTOS specification cannot be fully verified by using formal methods due to the fact that the dynamic behaviour of a given specification is often infinite. For this reason, many semi-automated tools were developed to verify LOTOS specifications using a variety of different methods, see 2.4.6 of chapter 2.

The goal-oriented execution technique is capable of constructing execution traces satisfying certain assertions. Using this technique, the specification under verification (SUV) is seen as a black box. The verifier needs only the knowledge of the interaction point structures (or action denotations) that include the formal gate names and the possible associated event sorts. Queries are then constructed, using relations $\Rightarrow^{+}$ and $\Rightarrow^{\times}$, and then submitted for execution. See Figure 6-1.

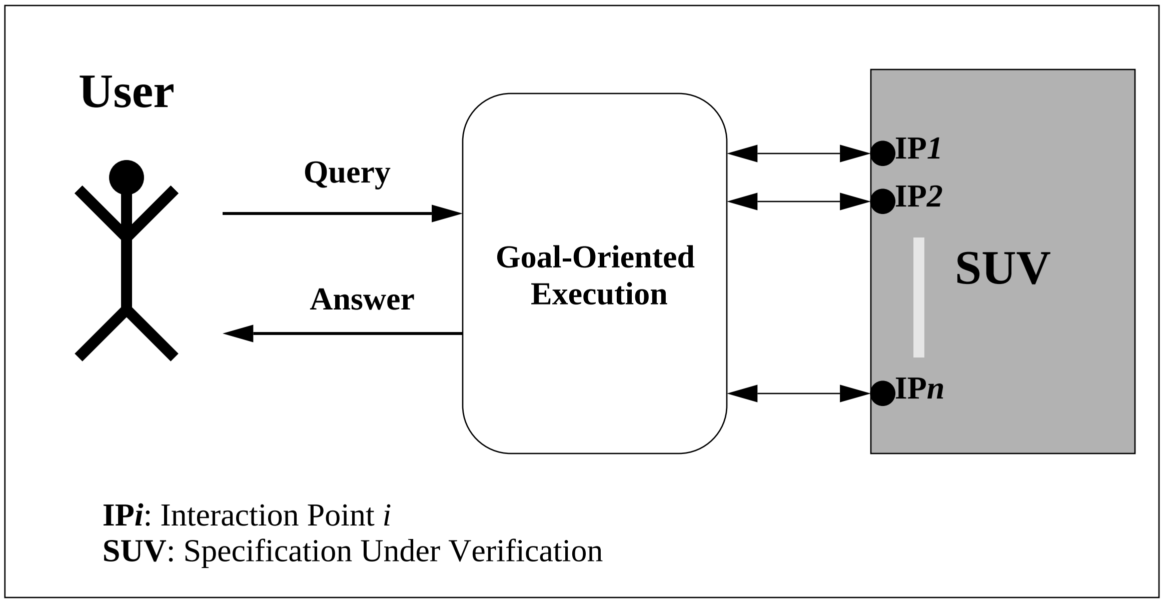

User
Query
Answer
Goal-Oriented Execution
IP*1*
IP*2*
IP*n*
SUV
IP*i*: Interaction Point *i*
SUV: Specification Under Verification

Figure 6-1 Black Box Verification

Two types of properties can be expressed as queries:

1. The existence of execution traces satisfying valid assertions. The user can construct queries on traces that should be possible by the specification. If such queries hold by goal-oriented execution, the possible execution traces are returned.
2. The absence of execution traces satisfying invalid assertions. The user may verify that the specification does not accept traces with invalid assertions. In this case, the related queries should not hold by goal-oriented execution.

Properties such as absence of deadlocks or livelocks are not handled by our method. The absence of a deadlock at a given point of execution can, on the other hand, be determined. For example, from a given behaviour B_1, we can determine by the following queries that deadlock does not occur immediately after action a is executed:

if $(a, B_1)/\{\} =t_1\Rightarrow^+ B_2$ holds then $(*, B_2)/\{\} =t_2\Rightarrow^+ B_3$ should also hold,

where ‘*’ identifies any action. See the next section for other action denotation abbreviations.

In this chapter, we demonstrate how goal-oriented execution can be applied for verification, and how it can be used to enhance existing verification methods.

This chapter is divided as follows: in section 6.1, we provide guidelines for the use of goal-oriented execution to verify LOTOS specifications describing a protocol and the provided services. The application of goal-oriented execution to verify an Alternating Bit Protocol specification is demonstrated in section 6.2. Section 6.3 lists some existing verification techniques that can be improved using our method.

6.1 Verification Guidelines

This section provides guidelines for the application of goal-oriented execution to verify a LOTOS protocol specification for a layered network model. (See section 2.1).

The protocol verification process involves checking for the following properties:

- **syntactic properties**: These are general design properties of a given protocol such as the absence of the following errors [155]: *state deadlock, unspecified receptions, non-executable interactions, state ambiguity, channel overflow, tempo blocking,* and *unfairness*. The verification of syntactic properties, often called *protocol validation*, does not require knowledge of the provided services.

- **semantic properties**: These are the intended sets of services that a given protocol needs to provide to the protocol of the layer above. The verification of such properties requires the service specification to be provided, and it is necessary to assume the correctness of the service provided by the layer below. Such properties cannot be classified or generalized since they depend on a specific protocol or service specifications. Such verification has proved difficult to automate.

- **protocol behaviour properties**: These properties describe the intended protocol behaviour, i.e. the exchange of messages among peer processes, that provides the intended services. As mentioned above, these properties are hidden from the user of the services. We believe that the verification of such properties is very important, since the verification of semantic properties may not necessarily exercise all protocol behaviour properties. An example of a protocol behaviour property is error recovery when a message is lost in an unreliable channel.

Here, we are only concerned about the verification of semantic and protocol behaviour properties. Note that the correctness of protocol behaviour properties does not imply that the semantic properties hold, but the failure of these may prevent the protocol from providing its specified services.Therefore, it is logical to verify protocol behaviour properties first.

The following rules enable an efficient verification of semantic properties and protocol behaviour properties using goal-oriented execution:

Rule 1: **Specification Style** - In order to verify protocol behaviour properties, it is important to specify the protocol behaviour in a process definition that can be tested separately. The overall specification will then have the form:

specification *<service_provider>[SAP1,...,SAPn]* :**noexit**

behaviour

hide *g1,..,gn* **in**

<protocol>[SAP1,..., SAPn, g1,.., gn](<actual parameters>)

where *SAP1,...,SAPn* are the logical service access points provided to the user of the services, and *g1,.., gn* are the logical peer-to-peer protocol communication channels. *g1,.., gn* are obviously hidden from the user of the services. In this form, the service is provided by the specification *<service_provider>[SAP1,...,SAPn]* and the protocol behaviour is expressed by the process *<protocol>[SAP1,..., SAPn, g1,.., gn](<actual parameters>)*. Note that the latter will show the interactions between the protocol entities, but not their internal behaviour.

Rule 2: **Properties Definition** - Define the protocol behaviour and the semantic properties using relations $\Rightarrow^{+}$ and $\Rightarrow^{\times}$.

Rule 3: **Protocol Behaviour Properties Application** - Apply the protocol behaviour properties on the protocol behaviour, i.e. the behaviour of process <protocol>. Hidden actions will be visible at this stage.

Rule 4: **Protocol Behaviour Properties Analysis** - Analyse the results of the protocol behaviour properties resulting from application of Rule 3. If errors are detected in the specification, then modify it and return to Rule 3.

Rule 5: **Semantic Properties Application** - Apply the semantic properties on the overall behaviour, i.e. the behaviour of specification <*service_provider*>. Actions involved in protocol communication will be hidden at this stage.

Rule 6: **Semantic Properties Analysis** - Analyse the results of the semantic properties resulting from application of Rule 5. If errors are detected in the specification, then modify it and return to Rule 3.

6.2 Verification of the Alternating Bit Protocol

The objective of this section is to apply the rules stated in the previous section for the verification of an Alternating Bit Protocol LOTOS specification. The following are the informal service and protocol specifications.

Informal Service Specification

This protocol provides a reliable, uni-directional data transfer service between two users, User1 the source and User2 the sink.

Informal Protocol Behaviour Specification

The protocol uses an unreliable full duplex one place channel to transfer protocol data units (PDUs) and acknowledgments. To ensure that the messages sent by User1 are received in the correct order by User2, the protocol associates a sequence number, alternating between 0 and 1, with the delivered PDUs and acknowledgments. Figure 6-2 illustrates the overall composition of the Sender and the Receiver entities, associated with User1 and User2 respectively, and the unreliable channel. The gates used by the protocol to communicate with the channel are hidden from the environment, i.e. the users.

LOTOS Specification

The following is the top level structure of the Alternating Bit Protocol LOTOS specification to be verified. The complete specification is given in Appendix A.

specification abp_service[User1,User2]: **noexit**

behavior

hide send1, recv1, send2, recv2, LOST **in**

abp[User1, User2, send1, recv1, send2, recv2, LOST](0 **of** Bit)

where

process abp[User1, User2, send1, recv1, send2, recv2, LOST]

(s_seq:Bit):**noexit**:=

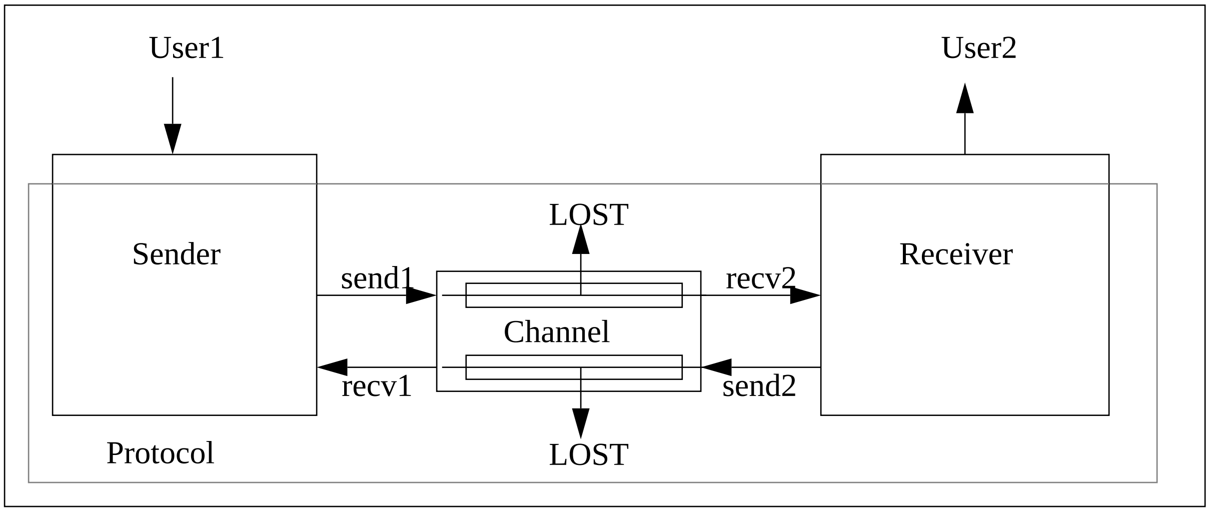


Figure 6-2 Alternating Bit Protocol Structure

The service *abp_service* has two interaction points through gates *User1* the source and *User2* the sink. They both allow one event of sort *Data*. A value of sort *Data* is simply any natural number.

The protocol, specified in process *abp*, is a composition of three entities, the sender, the receiver and the unreliable channel. The sender entity sends PDUs and receives acknowledgments by communicating with the channel through gates *send1* and *recv1* respectively. The receiver entity receives PDUs and sends acknowledgments by communicating with the channel through gates *send2* and *recv2* respectively. The PDUs and the acknowledgments are both of sort *Mess*. Gate *LOST* is added to the channel process to identify the loss of a message.

Verification

We give now the definition of the protocol behaviour properties and of the semantic properties. The protocol behaviour properties are applied on *B1(X) = abp[User1, User2, send1, recv1, send2, recv2, LOST](X)*, where X is the current sequence number, and the semantics properties are applied on *B2* = abp_service*[User1, User2]*. For better understanding of the message exchange between the protocol entities, the resulting traces are also shown using message sequence diagrams (MSDs). Each vertical line in an MSD identifies the entity from which a message is sent or received. The gates used in the communications are identified at the top of the vertical lines. Some gates are associated with an arrow to indicate the direction of the messages which will occur on this gate.

Protocol Behaviour Properties

P1 “Provide examples of traces that lead to the loss of a message”. This can be described as:

(LOST, B1(0))/{} =t*$\Rightarrow^{+}$*B’*

This will be satisfied with many execution traces, among others (see Figure 6-3 and Figure 6-4 respectively):

t = 〈 *User1 !D:Data,*
send1 !makepdu(D, 0):Mess,
LOST !makepdu(D, 0):Mess 〉

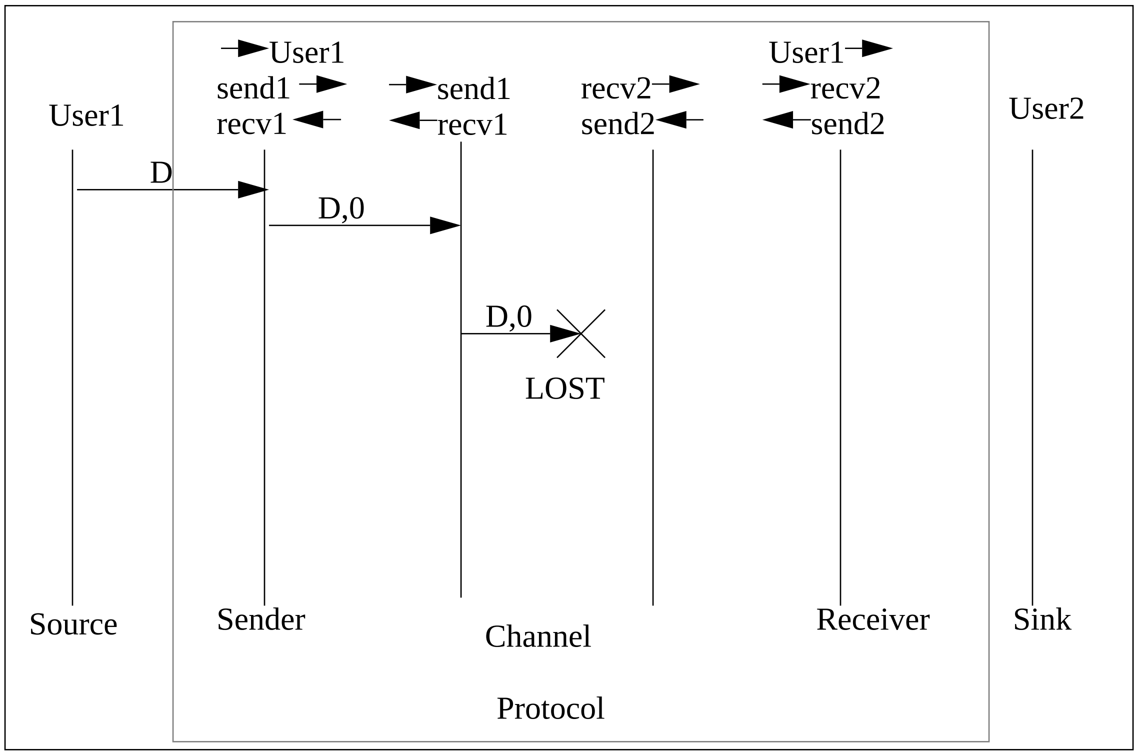


Figure 6-3 MSD 1 for Protocol Property P1

and

t = 〈 *User1 !D:Data,*
send1 !makepdu(D,0):Mess,
recv2 !makepdu(D,0):Mess,
send2 !makeack(0):Mess,

LOST !makeack(0):Mess 〉

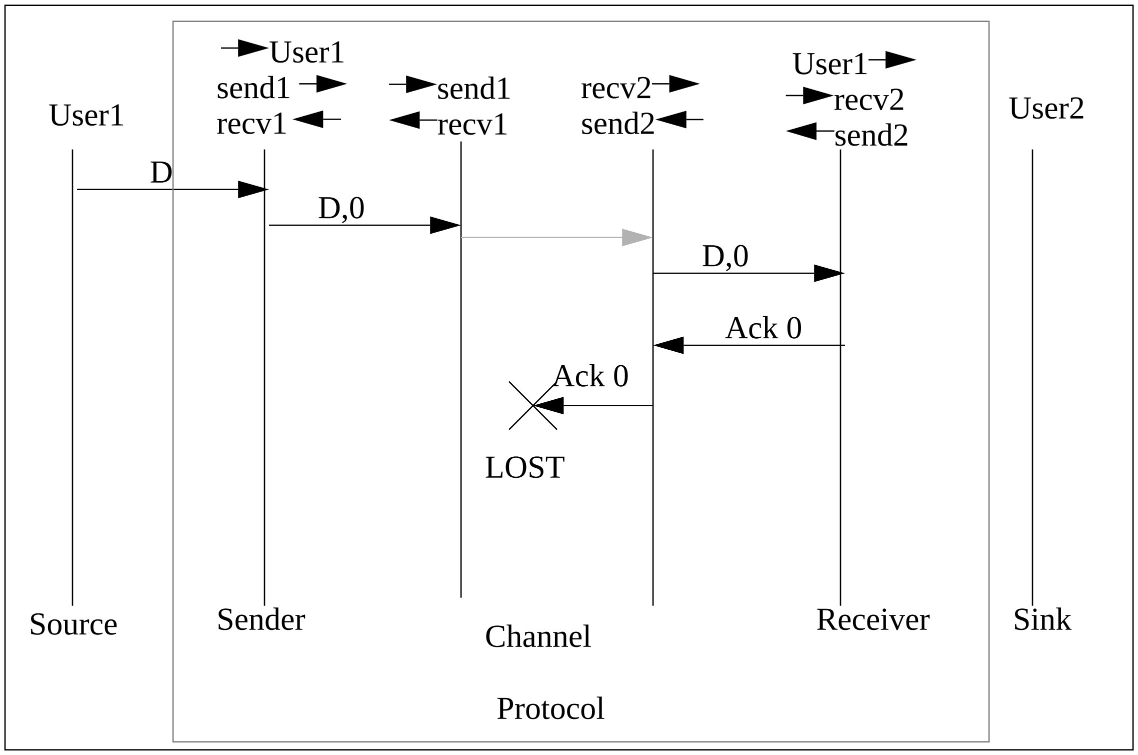


Figure 6-4 MSD 2 Protocol Property P1

From the above two traces, we see that *M:Mess* can have a value of *makepdu(D,0)* or a value of *makeack(0)* indicating that the types of messages that can be lost are PDUs and Acknowledgments. Note that in reality, the above query has an infinite number of solutions, but due to the fact that heuristics are added to static derivation paths and the guided-inference rules definition, only a finite number of solutions is provided. See sections 4.1.3 and 4.2.3. Note also that the variable *D:Data* in the above traces is left free since its value does not affect the execution trace in any way, i.e. it is not involved in any condition encountered during derivation.

P2 “Is it possible for the protocol sink entity to receive the expected PDU after the medium loses the PDU once?”. This can be described using relation $\Rightarrow^{\times}$ by first reaching the loss of a PDU *M*, indicated by the associated selection predicate *is_pdu(M)*, and then receiving at gate *recv2* the expected message, expressed by the associated selection predicate *seq(M) eq 0*, where 0 is the expected sequence number:

(⟨LOST?M:Mess[is_pdu(M)], recv2!M[seq(M) eq 0]⟩, B1(0))/{} =t⇒$^{\times}$B'

This succeeds with the following trace, where the actions identified in the query match the third and the fifth action (see Figure 6-5):

t = ⟨ *User1 !D:Data,*
send1 !makepdu(D, 0):Mess,
LOST !makepdu(D, 0):Mess,
send1 !makepdu(D, 0):Mess,
recv2 !makepdu(D, 0):Mess⟩

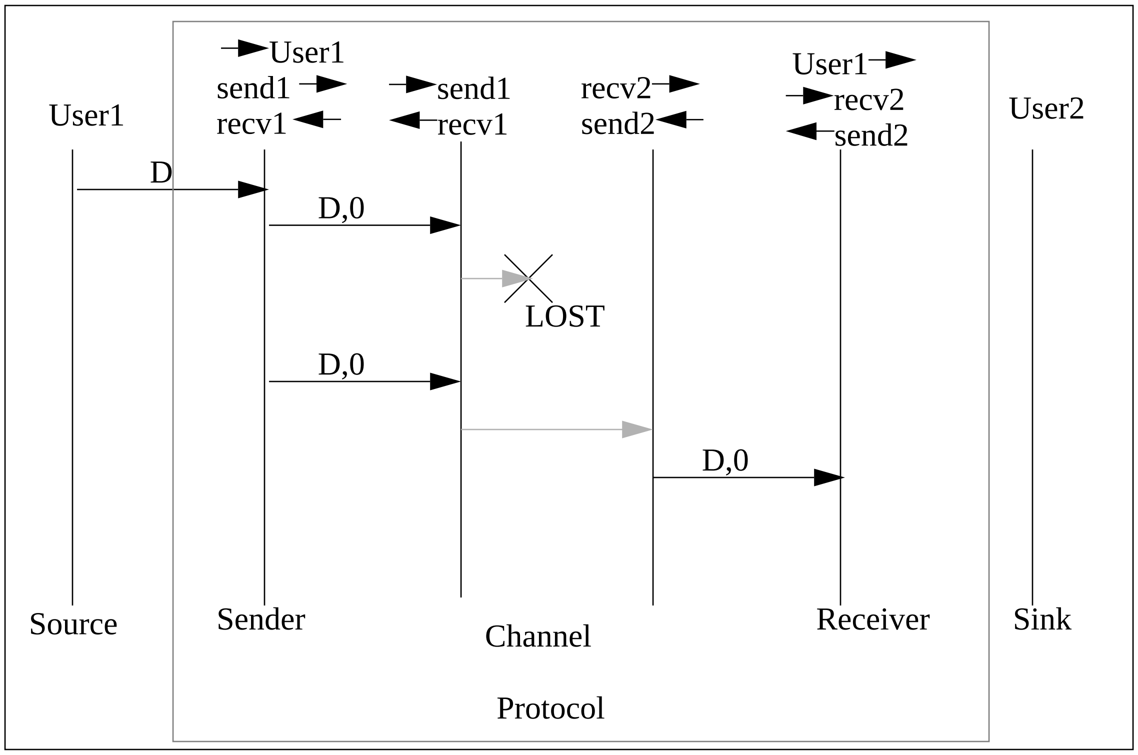


Figure 6-5 MSD 1 Protocol Property P2

P3 "Is it possible that when an acknowledgment is lost by the medium, the protocol source entity will re-send the same PDU with the same sequence?". This can also be described by associating selection predicates to the desired ordered actions as:

(⟨ send1?M1:Mess,
LOST?M2:Mess[is_ack(M2) and (seq(M1) eq seq(M2))],
send1!M1 ⟩, B1(0))/{} =t⇒$^{\times}$B'

This succeeds with (see Figure 6-6):

t = 〈 *User1 !D:Data,*
send1 !makepdu(D, 0):Mess,
recv2 !makepdu(D, 0):Mess,
send2 !makeack(0):Mess,
LOST !makeack(0):Mess,
send1 !makepdu(D, 0):Mess〉

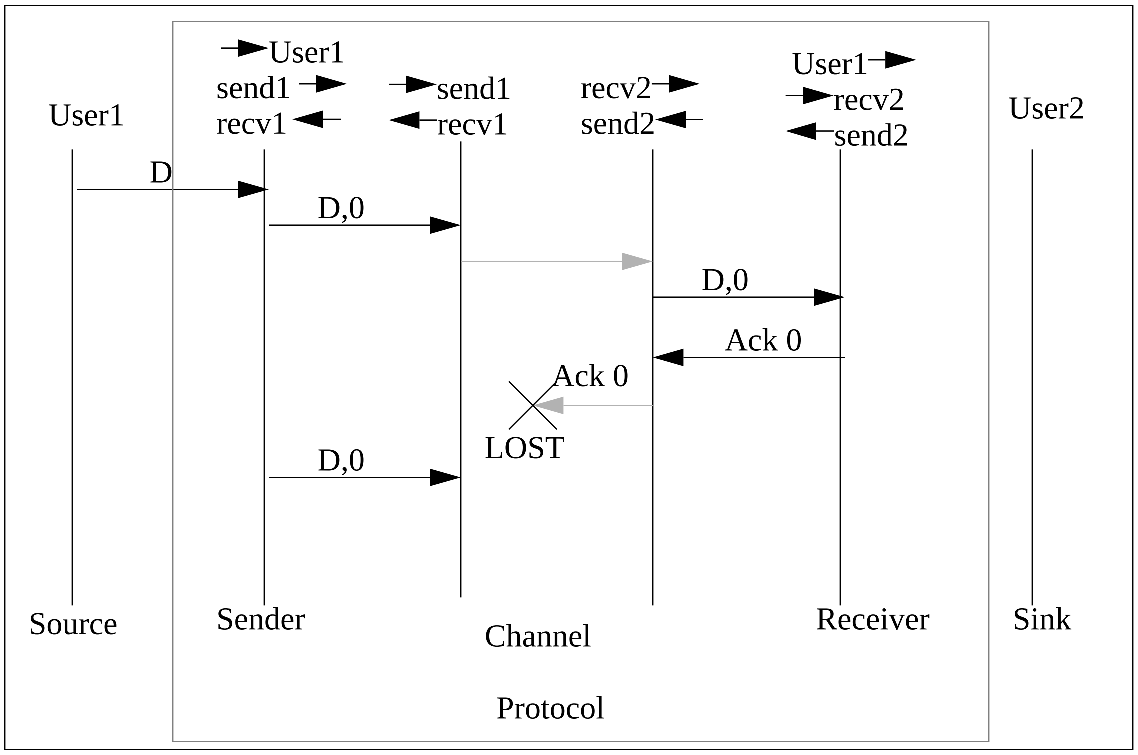


Figure 6-6 MSD 1 Protocol Property P3

P4 “Provide examples of traces where the source entity sends a PDU with sequence number *X*, after receiving an acknowledgment with sequence number *complement(X)*”. This can described as:

(〈 *recv1?M1:Mess[is_ack(M1)],*
send1?M2:Mess[is_pdu(M2) and (seq(M2) eq compl(seq(M1)))]〉, *B1(0))/{} =t*$\Rightarrow^{\times}$*B’*

The following is one possible trace (see Figure 6-7):

t = 〈 *User1 !D1:Data,*
send1 !makepdu(D1, 0):Mess,
recv2 !makepdu(D1, 0):Mess,
send2 !makeack(0):Mess,
User2 !D1:Data,
recv1!makeack(0):Mess,
User1 !D2:Data,
send1 !makepdu(D2, Succ(0)):Mess〉

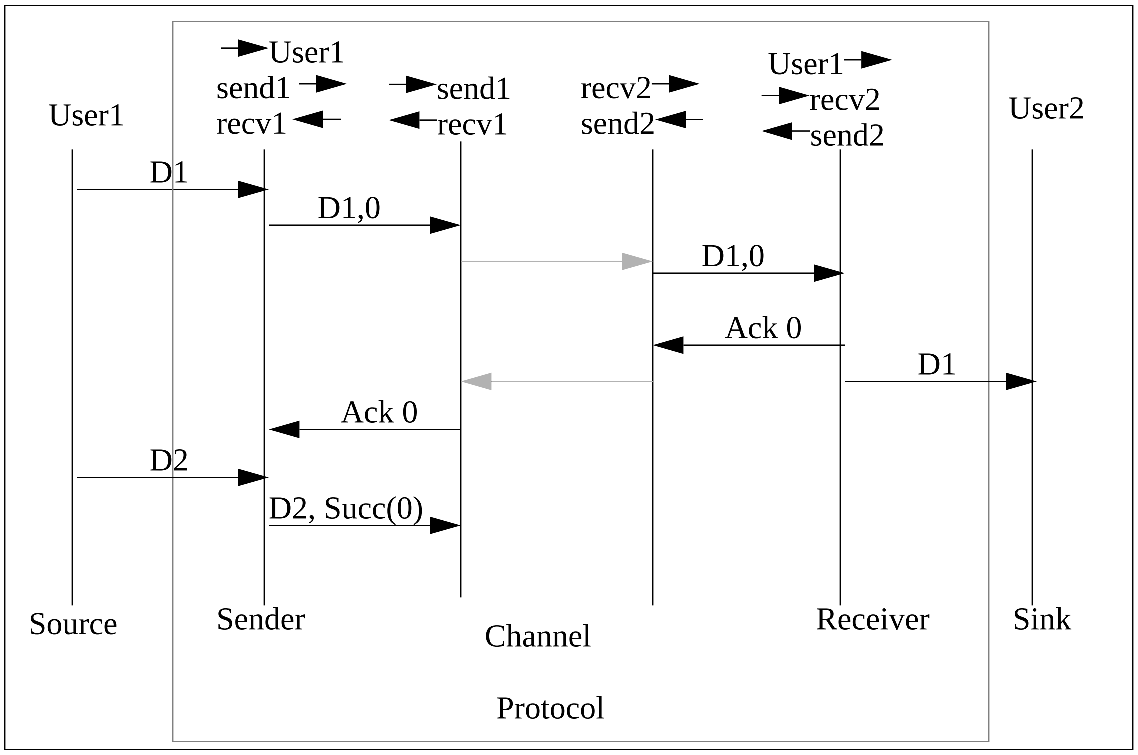


Figure 6-7 MSD 1 Protocol Property P4

P5 "Verify that it is not possible for the source entity to send a PDU with sequence number *X*, immediately after receiving an acknowledgment with the same sequence number *X*". This can described as:

(〈 *recv1?M1:Mess[is_ack(M1)],*
send1?M2:Mess[is_pdu(M2) and (seq(M2) eq seq(M1))]〉*,* *B1(0))/{} =t*$\Rightarrow^{\times}$*B'*

This goal is evaluated to false, so the query is satisfied.

Note that satisfying queries P4 and P5 implies that the following property is satisfied:

"When the source entity sends a PDU after receiving an acknowledgment with sequence number *X*, the sequence number of that PDU is *complement(X)*"

Semantic Properties and Queries

S1 "Provide an example of a trace that leads User2 to receive a message?". This can be described as:

$(User2?D{:}Data,\ B2)/\{\} =t\Rightarrow^{+} B'$

This will be satisfied with

$t = \langle User1!D{:}Data,\ User2!D{:}Data\rangle$

for any data message *D* supplied by *User1*. Note that here the underlying protocol that guarantees the delivery of a message from *User1* to *User2* is hidden from the environment and, therefore, its actions, i.e. sequences of **i**'s, are not part of the desired traces.

S2 "Is it possible for *User2* to receive a message if *User1* does not send a message?". This query can be described by reaching *User2* without passing through *User1* as follows:

$(User2?D{:}Data,\ B2)/\{User1\} =t\Rightarrow^{+} B'$

This relation will not hold, indicating that it is not possible for *User2* to receive a message without *User1* sending a message. Note that the failure of this relation is caused by the fact that no static derivation paths that lead to *User2* without passing through *User1* were found, i.e. $\Sigma(User2?D{:}Data,\ B2,\ \{User1\}) = \emptyset$. See lemma 4-3.

S3 "Check that if *User2* receives a message then this message is first sent by *User1*". This property is stronger than the previous one in the sense that *User2* will receive a message only if it is sent by *User1*. This can be accomplished by first defining the property as a logical expression on traces where $\Rightarrow$, $\vee$, $\wedge$, and $\neg$ are the *implication*, *or*, *and*, and *negation* symbols respectively. The operators defined in section 3.2.3, are also used in the expressions.

$\forall t\ ((t\wedge = User2?D2{:}Data) \Rightarrow (\ (User1?D1{:}Data \in\ t) \wedge (D1=D2)\ )\ )$

which is equivalent to

$\forall t\ (\neg(t\wedge = User2?D2{:}Data) \vee (\ (User1?D1{:}Data \in\ t) \wedge (D1=D2)\ )\ )$

and transform $\forall$ to $\exists$.

$\neg\exists t\ (\neg(\neg(t\hat{} = User2?D2{:}Data) \vee (\ (User1?D1{:}Data \in\ t) \wedge (D1{=}D2)\ )\ ))$

which is equivalent to

$\neg\exists t\ ((t\hat{} = User2?D2{:}Data) \wedge (\ \neg(User1?D1{:}Data \in\ t) \vee \neg(D1{=}D2)\ )$

which can be divided into:

$\neg\exists t\ ((t\hat{} = User2?D2{:}Data) \wedge \neg(User1?D1{:}Data \in\ t)\ ),$ and

$\neg\exists t\ ((t\hat{} = User2?D2{:}Data) \wedge (\ (User1?D1{:}Data \in\ t) \wedge \neg(D1{=}D2)\ )\ )$

These can be described using the following queries respectively:

$(User2?D1{:}Data,\ B2)/\{User1\} =t\Rightarrow^{+} B'$ should not hold, and

$(\langle User1?D1{:}Data,\ User2!D2{:}Data[D1\ ne\ D2]\rangle,\ B)/\{\} =t\Rightarrow^{\times} B'$ should not hold, and

we add another query to determine the existence of a trace that leads *User2* to receive the same message sent by *User1* as:

$(\langle User1?D{:}Data,\ User2!D{:}Data\rangle,\ B)/\{\} =t\Rightarrow^{\times} B'$ should hold.

The first two queries are evaluated to false while the third query yields to

$t = \langle User1!D{:}Data,\ User2!D{:}Data\rangle$

S4 “If a message is sent by *User1* then *User2* will definitely receive the same message”. This property is not expressible by relations $\Rightarrow^{+}$ and $\Rightarrow^{\times}$, since we have to guarantee that all possible traces after *User1* sends a message lead to *User2* receiving the same message, i.e. absence of livelocks or deadlocks between *User1* and *User2*. For example, the fact that relation $(\langle User1?D{:}Data,\ User2!D{:}Data\rangle,\ B)/\{\} =t\Rightarrow^{\times} B'$ holds simply implies that a trace satisfying the property can be found and does not imply that *User2* will definitely receive the message sent by User1. Modification of relations $\Rightarrow^{+}$ and $\Rightarrow^{\times}$ to express deadlocks and livelocks is an item for future work, see section 7.2.

6.3 Scope of Application

Other than being a relief strategy for state space explosion, we foresee several applications for the techniques discussed in this thesis. These applications are described in the following sections.

6.3.1 Step-by-step execution

Narrowing Technique

In step-by-step Execution, the narrower can help in: (a) eliminating actions associated with unsatisfiable predicates, (b) supplying the user with possible values for variables in an action satisfying its corresponding predicate. For example, consider the following LOTOS process:

```
process dummy[a,b,c] (X:nat) :=
        a?Y:Nat[Y<X]; c!Y!X; stop
        []
        b?Y[Y>=X]; dummy[a, b, c](succ(X))
        []
        a; b; p[a,b]
endproc.
```

The possible initial actions for the process call *dummy[g1,g2,g3](0)* are: *g1?Y:Nat[Y<0]*, *g2?Y[Y>=0]* and *g1*.

The predicate *[Y<0]* of the first action cannot be satisfied for any value of *Y*, and therefore the action can be eliminated. On the contrary in the second action, the narrower can provide a value *succ(0)* for *Y*.

Static Analyser

During step-by-step execution, the user can determine if the execution of an offered action may lead to his/her intended targeted action. This is possible by applying the static analyser on the resulting behaviour expression of the action. If no static derivation paths were found by the analysis, then the action is not suitable for execution, see lemma 4-3. For example, using the process definition above, the possible initial actions for the process call *dummy[g1,g2,g3](0)* are: *g1?Y:Nat[Y<0]*, *g2?Y[Y>=0]* and *g1*. If the user's intent is to reach an action with gate *g3*, the static analyser can determine that selecting action *g1* is undesirable.

The user may also choose one of the generated static derivation paths to be followed by step-by-step execution. In this case, the possible next actions that do not comply with the chosen SDP will be eliminated.

Goal-Oriented Execution

The user may request reaching the targeted action directly by applying goal-oriented execution, or reaching a desired intermediate action then continue the normal step-by-step execution.

6.3.2 Symbolic execution

Narrowing Technique

The narrowing technique can help in pruning branches from the symbolic graph with unsatisfiable guards or predicates. This will reduce the graph considerably. Also, the narrower can help in transforming paths from the symbolic graph into execution traces by providing values for the free variables, which will satisfy all guards and predicates.

Guided-Inference System

A symbolic graph is constructed by first generating all possible next actions and their resulting behaviour expressions using an unguided inference system. The same process is then repeated on all the resulting behaviours that have not been already encountered. Using the guided-inference system, the construction of a symbolic graph can be restricted by a given property. This is very useful in the sense that the graph will only include the possible symbolic paths that satisfy the property.

6.3.3 Random Walk

Narrowing Technique

Narrowing can also assist in exercising randomly the dynamic behaviour of a given process, by providing valid values for variables encountered during the walk. For example, random values can be generated by applying a random function on a set of values generated by the narrower.

Static Analyser

Similar to step-by-step execution, by applying the static analyser on the resulting behaviour expressions of all offered actions, random walk can select (randomly) one of the next actions that may lead to an intended targeted action. Actions with no static derivation paths found by the analysis on their resulting behaviours are excluded from the selection set.

Goal-Oriented Execution

Random walk can also be applied by the goal-oriented execution algorithm where static derivation paths and values offered by the narrower can be selected randomly, see step 2 and step 6 of the algorithm in section 3.8.

6.3.4 Data-Flow Analysis

Static Analyser/Guided-Inference System

van der Schoot and Ural [136] have demonstrated the use of our techniques to perform data flow analysis and to generate data flow oriented test sequences. First, a flow graph is constructed modeling both control and data flow aspects expressed in the specification. In this flow graph, definitions and uses of each variable occurrence employed in the specification are identified. Static derivation paths for a specific node in the graph are then obtained and fed to a guided-inference system to obtain test sequences satisfying a specific data flow criterion. Unfortunately, further explanation of their technique would require the introduction of many definitions.

6.3.5 Temporal Logic Properties

Goal-Oriented Execution

Some temporal logic properties can be checked using the relations $\Rightarrow^{+}$ and $\Rightarrow^{\times}$ described in chapter 3. The following are some examples.

“from the current behaviour B, if action a!true can be reached then action b!false!0 can be reached after”

can be expressed as:

if $(a!true,B)/\{\} =t_1\Rightarrow^{+} B'$ holds then $(b!false!0,B')/\{\} =t_2\Rightarrow^{+} B''$ also holds, and the property

“*from the current behaviour B, it is not possible to reach action a!true without reaching an action with gate name b*”

can be expressed as:

$(a!true\ ,B)/\{b\} =t\Rightarrow^{+} B'$ should not hold,

and the property

“*from the current behaviour B, it is not possible to reach action a!true after reaching action b!false!0*”

can be expressed as:

$(\langle b!false!0, a!true\rangle,B)/\{\} =t\Rightarrow^{\times} B'$ should not hold,

6.3.6 Test Cases generation

Goal-Oriented Execution

A large part of testing theory relates to the problem of selecting test sequences satisfying certain requirements, called test intents. Our technique can help finding such sequences by representing the test intents using relations $\Rightarrow^{+}$ and $\Rightarrow^{\times}$. For example, the results of the queries applied in section 6.2 on the specification of the Alternating Bit Protocol can be mapped into test cases for the implementation of such a design.

Chapter 7 Conclusion

7.1 Contributions

In chapter 2, we have summarized existing work in the area of protocol validation and verification, and we have presented some existing intrinsic problems, mainly the state space explosion problem. We have surveyed many attempts to overcome these problems in various ways, especially those using Communicating Finite State Machines models and the formal description technique LOTOS. In the following chapters, we have presented our approach to these problems for protocols specified in LOTOS using a search technique called *goal-oriented execution*. In this technique, traces satisfying a given property are derived. These traces are modeled using relations $\Rightarrow^{+}$ and $\Rightarrow^{\times}$. $(a,B)/G =t\Rightarrow^{+} B'$ defines the derivation of behaviour B on a trace t leading to a matching targeted action a' without passing through any other action with gate name in G. $(\langle a_1,\ldots,a_n\rangle, B)/G =t\Rightarrow^{\times} B'$, on the other hand, defines the derivation of behaviour B on a trace t, such that t contains a predetermined series of matching actions $\{a_1',\ldots,a_n'\}$, not necessarily contiguously, without passing by any other action with gate name in $G \cup \alpha(\langle a_1,\ldots,a_n\rangle)$. The goal-oriented execution technique results from the combination of:

1- a *static analyser* - establishes the scope of the search by determining where, in the LOTOS specification, a given property can possibly hold,

2- *guided-inference system* - a new type of inference system which uses the scope information generated by the static analyser to generate variable traces satisfying the temporal ordering restriction specified in the given property, and

3- *narrower engine* - is an automatically generated tool, called ERNAL, which is capable of evaluating and solving LOTOS ADTs expressions.

In chapter 4, we have presented the formal definition of the static analyser and of the guided-inference system and we have illustrated a sketch for the correctness of such techniques. Also in that chapter, we have exposed the limitations of the applicability of the static analyser and of the guided-inference system techniques, along the added heuristics. The method of automatically generating the narrower engine from a given Abstract Data Type definition was explained in detail in chapter 5. Finally, in chapter 6, we have demonstrated how the goal-oriented execution technique can be applied for verification, and how it can be used, along with other techniques discussed in this thesis, to enhance existing verification methods.

The goal-oriented execution technique was first to be reported for basic LOTOS in [68]. In parallel with this thesis' work, a similar technique applying full LOTOS was developed at University of Twente [38]. Their approach differs from ours in the following ways:

- The goal used in their technique is a selected sub-behaviour expression in the overall specification. They claim that it is more logical to have a state of the specification as a goal than an action denotation. However, since there may exist sub-behaviour expressions that do not correspond to reachable states. For example, the sub-behaviour expression *a;**stop*** in the expression *b;a;c;**stop** |[a,b]| b;<u>a;**stop**</u>* is not a reachable state, although *a;c;**stop** |[a,b]| a;**stop*** is a state.
- Their algorithm is not recursive in nature. For example, to reach a sub-behaviour that occurs on the right hand side of an enable operator, an execution trace that leads to an action with gate δ is needed from the left hand side of the enable operator. To do so, they are obliged to use a method that reaches an action denotation, like ours, rather than reaching a behaviour expression.
- To select a sub-behaviour expression as a goal, the verifier must be familiar with the code of the specification, i.e. the specification must be seen as a white box. Our technique is suitable for a black box verification strategy, where the verifier needs only the knowledge of the interaction structures. See the introduction of chapter 6.
- The initial derivation of their static information is re-visited to remove all unsatisfied paths. This is not needed in our approach, since unsatisfied paths are not even constructed.
- Their inference system is less powerful than ours in handling synchronizations caused by parallel operators. For example, when their system is directed into one side of a parallel operator towards an action, say *a*, that must synchronize, a *synch-failure* exception is raised if no transition is found on the other side of the parallel operator that can synchronize with action *a*. The exception must then be handled by the user. This situation is completely handled by our inference system. See section section 4.2.1.

- Their narrower technique is borrowed from [152], while the technique used by our narrower is new. We believe that our approach can lead to a more efficient execution.

7.2 Future Work

We believe that our technique is a powerful verification tool for LOTOS specifications. It provides an attractive and simple representation for the user objectives and is supported by an efficient implementation. In order to make the technique even more valuable, the following items are considered for future work:

1- Applying the technique to verify real-size protocols. Additional heuristics will probably be needed to cut search time.

2- Maximizing the functionalities of the static analyser. Doing so, the execution complexity can be reduced. One functionality that can be added is the static construction of complete derivation paths. For example,

   $\Sigma(d, \underline{a;d\ |[a]|\ b;a;\textbf{stop}}, \{\}) = \{\ [parallel([prefix, prefix], [prefix, prefix])]\ \}$

   where the left behaviour and the right behaviour of the parallel operator need to be evaluated, while

   $\Sigma(d, \underline{a;d\ |[a]|\ b;a;\textbf{stop}}, \{b\}) = \emptyset$.

3- Modifying the relation $\Rightarrow^{\times}$ to represent contiguous actions. For example, one could write "a_1,^a_2" to express the fact that action a_2 must immediately follow action a_1 in the desired trace.

4- Other than the restricted gate set that applies globally on all actions in relations $\Rightarrow^{+}$ and $\Rightarrow^{\times}$, restricted gate sets for specific actions can also be applied. For example "a_1, a_2/G" can identify a restricted gate set G specifically for action a_2, i.e. all actions between action a_1 and action a_2 should not have a gate name in G.

With these new notations, we claim the following equivalences:

$$(\langle a_1, \hat{}a_2\rangle, B)/G =t\Rightarrow^{\times} B' = (\langle a_1, a_2/\{*\}\rangle, B)/G =t\Rightarrow^{\times} B'$$

In other words, saying that action a_2 must follow immediately action a_1 (expressed by "a_1,^a_2"), is the same as saying that the set of all actions between action a_1 and action a_2 are restricted (expressed by "a_1, $a_2/\{*\}$").

$$(\langle a_1,\ldots,a_n\rangle, B)/G =t\Rightarrow^{\times} B' = (\langle a_1/G,\ldots,a_n/G\rangle, B)/\{\} =t\Rightarrow^{\times} B'$$

This identifies the fact that the application of the global restricted set *G* is equivalent to its application on each identified action.

5- Modifying the restricted gate set to represent the absence of livelocks and deadlocks identified by ∞ and $\perp$ respectively. This is obviously undecidable especially when dealing with an infinite execution space, but we believe that tackling these problems with appropriate limitations is definitely important. For example, the following property of the alternating bit protocol, stated in section 6.2,

“If a message is sent by *User1*, then *User2* will definitely receive the same message”,

can be expressed as:

$$(\langle User1?D{:}Data,\ User2!D{:}Data/\{\perp, \infty\}\rangle,\ B)/\{\}\ {=}t{\Rightarrow}^{\times}\ B'$$

assuming that fairness is applied. This implies that any execution path taken after action *User1?D:Data* is executed will eventually lead to action *User2!D*:*Data*. In our technique, fairness is handled by limiting recursion. See section 4.1.3 and section 4.2.3.

References

[1] Ashkar, P. Symbolic Execution of LOTOS Specification, *Master's Thesis*, University of Ottawa, 1992.

[2] Afek, Y., and Gafni, E. Election and traversal in unidirectional networks, *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, Vancouver, Aug. 1984, 190-198.

[3] Ansart, J.P. Issues and Tools for Protocol Specification. In *The Advanced Course on Distributed Systems - Methods and Tools for Specification*.

[4] Azema, P., Drira, K., and Vernadat, F. A Bus Instrumentation Protocol Specified in LOTOS, in Juan Quemada., José Mañas, and Enrique Vázquez, editors, *Proceedings of 3rd International Conference on Formal Description Techniques FORTE '90*, Nov. 1990, Madrid, Spain.

[5] Bainbridge, S. and Mounier, L. Specification and Verification of a Reliable Multicast Protocol, Technical Report HPL-91-163, Hewlett-Packard Laboratories, Bristol, U.K. Oct. 1991.

[6] Barbeau, M. and Bochmann, G.V. Extension of the Karp and Miller Procedure to Lotos Specifications, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1991, vol. 3,103-119.

[7] Belina, F., and Hogrefe, D., The CCITT Specification and Description Language SDL, *Computer Networks and ISDN Systems*, 1989, vol.16, 311-341.

[8] Belnes, D., Moller-Pedersen, B., and Dahle, H.P. Rational and Tutorial on OSDL: an Object Oriented Extension of SDL, SDL '87: State of the Art and Future Trends, *Proceedings of the Third SDL Forum*, in R. Saracco and Tilanus, eds., Leidschendam, The Netherlands, Apr. 1987, 413-426.

[9] Berlinguette, P. and Gueraichi, D. The Alternating Bit Protocol in LOTOS:"Textual" and "Graphical" Representation, Technical Report TR-88-25, Department of Computer Science, University of Ottawa, 1988.

[10] Berthelot G., and Terrat, R. Modelisation et validation des protocoles de transport pour reseaux de Petri a predicats, *Congres de Conception des Systemes Telematiques*, Nice, France, June 1981.

[11] Berthelot G., and Terrat, R. Petri nets theory for the correctness of protocols, *IEEE Transactions on Communications*, Dec. 1982, 30(12):2497-2505.

[12] Blumer, T.P., and Sidhu, D.P. Mechanical Verification and Automatic Implementation of Communication Protocols. *IEEE Transactions on Software Engineering*, Aug. 1986, SE-12(8):827-843.

[13] Bochmann, G.V. Finite State Description of Communication Protocols. In *Computer Networks Symposium*, University of Liege, Belgium, Feb 1978.

[14] Bochmann, G.V., and Sunshine, C. Use of formal methods in communication protocol design. *IEEE Transactions on Communications*, 1980, 28(4):624-631.

[15] Bochmann, G.V. Usage of protocol development tools: The results of a survey, in H. Rudin and C.H. West, editors, *Protocol Specification, Testing, and Verification VII*. North-Holland, 1987, 139-161.

[16] Bochmann, G.V. Specification of a simplified transport protocol using different formal description techniques, *Computer Networks and ISDN Systems*, June 1990, 18(5):335-377.

[17] Bogaards, K. LOTOS-Supported System Development, in K.J. Turner, editor, *Proceedings of 1st International Conference on Formal Description Techniques FORTE '88*, North-Holland, 1988, 279-294.

[18] Bolognesi, T., and Brinksma, E. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 1987, vol.14, 25-59.

[19] Bolognesi, T., and Caneve, M. Squiggles: A tool for the analysis of LOTOS specifications, in K.J. Turner, editor, *Proceedings of 1st International Conference on Formal Description Techniques FORTE '88*, North-Holland, 1988, 201-216.

[20] Bolognesi, T., and Caneve, M. Equivalence Verification: Theory, Algorithms, and a Tool, in P.H.J. van Eijk et al., editors, The Formal Description Technique LOTOS-Results of the ESPRIT/SEDOS Project, North-Holland, 1989, 303-326.

[21] Boudol, G., Simon, R., and Vergamini, D. Experiment with AUTO and AutoGraph on a single case of sliding window protocol. *Rapport RR870*, INRIA, 1988.

[22] Brand, D., and Joyner jr., W.H. Verification of Protocols Using Symbolc Execution. *Computer Networks*, Oct 1978, 2:351-360.

[23] Brand, D., and Zafiropulo, P. On Communicating Finite-State Machines. *J. ACM*, Apr. 1983, 30(2):323-342.

[24] Brinksma, E. The Specification Language LOTOS, in Proceedings NGI-SION symposium 3, NGI, Amsterdam, 1985.

[25] Brinksma, E., Scollo, G., and Steenbergen, C. LOTOS specifications, Their Implementation and Thier Tests, in *6th Internation Workshop on Protocol Specification, Testing, and Verification*, Montreal, June 1986.

[26] Cavalli, A.R. A method for automatic proofs for the specification and validation of protocols, ACM SIGCOMM, S*ymp. on Communications Architectures and Protocols*, Montreal, 1986, 9-1 to 9-18.

[27] CCITT, Recommendation Z.100, Specificaion and Description Language SDL, Study Group X, 1992.

[28] Chang, E., and Roberts, R. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configuration of Processes. *Comm. ACM 22*, 5 (May 79), 281-283.

[29] Cheng, K.E., and Jackson, L.N. A Definition and Description Techinique for Translating SDL Specifications to Implementation, SDL '89, The Language at Work, *Proceedings of the Fourth SDL Forum*, in O. Faergemand and M. M. Marques, eds., Lisbon, Portugal, Oct. 1989, 293-302.

[30] Choi, T.Y., and Miller, R.E. A Decomposition Method for the Analysis and Design of Finite State Protocols. In *Data Communication Symposium, ACM SIGCOMM*, 1983, 167-176.

[31] Chow, C., Gouda, M.G., and Lam, S.S. A Discipline for Constructing Multiphase Communication Protocols. *ACM Transactions on Computer Systems*, Nov. 1985, Vol. 3(4):315-342.

[32] Clarke, E.M., and Emerson, E.A. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic, in *Proc. Workshop on Logics of Programs*, Lecture Notes in Computer Science, (Springer, Berlin) 131:52-71.

[33] Day, J.D., and Zimmermann, H. The OSI Reference Model. *IEEE Transactions on Communications*, 71, Dec 1983, 1334-1340.

[34] Dershowitz, N., and Jouannaud, J.P. Rewrite Systems, in J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 66, North Holland, Amsterdam, 1990.

[35] Dershowitz, N., and Plaisted, D.A. Equational Programming, in J. E. Hayes, D. Michie, and J. Richards, editors, *Machine Intellengence 11: The logic and acquisition of knowledge*, Oxford Press, Oxford, 1988, chapter 2, 21-56.

[36] Dershowitz, N., and Sivakumar, G. Goal-directed equation solving, in *proceedings of the Seventh National Conference on Artificial Intelligence*, St. Paul, MN, Aug. 1988, 166-170.

[37] Diaz, M., and Vissers, C. SEDOS: Designing open distributed systems. IEEE Software, Nov. 1989, 6(6):24-33.

[38] Eertink, H., Simulation Techniques for the Validation of LOTOS Specification, *Ph.d. Thesis*, The Netherlands, 1994.

[39] Eertink, H., and Wolz, D., Symbolic Execution of LOTOS Specifications, *Fifth International Conference on Formal Description Techniques FORTE '92*, France, Oct. 1992.

[40] Ehrig, H., Fey, W., and Hansen, H. ACT ONE: An Algebraic Specification Language Specification with Two Levels Semantics, *Bericht-Nr.*, Mar. 1983.

[41] Ehrig, H., Mahr, B., Fundamentals of Algebraic Specification 1, *SpringerVerlag*, Berlin,1985.

[42] Ernberg, P., Fredlung, L., and jonsson, B. Specification and Validation of a Simple Overtaking Protocol using LOTOS. T 90006, Swedish Institute of Computer Science, Kista, Sweden, Oct. 1990.

[43] Faci, M., Logrippo, L., and Stepien, B. Formal Specification of Telephone Systems in LOTOS: The Constraint-Oriented Approach. To appear in Computer Networks and ISDN Systems.

[44] Fehri, M.C. A System for Validating and Executing LOTOS Data Abstractions (SVELDA), *Master Thesis*, University of Ottawa, 1987.

[45] Fernandez, J.C. ALDÉBARAN: un système de vérfication par réduction de processus communicants, *Thèse de Doctorat*, Université Joseph Fourier (Grenoble), May 1988.

[46] Fernandez, J.C., An implementation of an efficient Algorithm for Bisimulation Equivalence, *Science of Computer Programming*, May 1990, 13(2-3):219-236.

[47] Fernandaz, J.C., and Mounier, L. Verifying Bisimulations "On the Fly", in Juan Quemada., José Mañas, and Enrique Vázquez, editors, *Proceedings of 3rd International Conference on Formal Description Techniques FORTE '90*, Nov. 1990, Madrid, Spain.

[48] Fernandez, J, Garavel, H., Mourier, J., Rasse, A., Rodriguez, C., Sifakis, J. Une boite a outils pour la verification de programmes LOTOS, in O. Rafiq (ed.), CFTP'91, Ingenierie des protocoles, Renne, 1991, 479-500.

[49] Floyed, R.W. Assigning Meanings to Programs, In *Symposia in Applied Mathematics*, XIX, 1967, 19-32.

[50] Gabbay, D., Pnuelli, A., Shelah, S., and Stavi, Y. On thr temporal analysis of fairness, in *Proc. 7th annu. ACM Symp. Principles of Programming Languages*. Las Vegas, NV, Jan. 1980, 163-173.

[51] Gallager, R.G., Humblet, P.A. and Spira, P.M. A distributed algorithm for minimum-weight spanning trees, *ACM Transactions on Programming Languages and Systems 5*, 1, Jan. 1983, 66-77.

[52] Gallouzi, S. Trace Analysis of LOTOS Behaviours, *Master's Thesis*, University of Ottawa, 1989.

[53] Gallouzi, S., Logrippo, L., and Obaid, A. An Expressive Trace Theory for LOTOS, in B. Jonson, J. Parrow, and B. Pehrson. (eds.) *Protocol Specification, Testing, and Verification, XI,* North-Holland, 1991.

[54] Gallouzi, S., Logrippo, L., and Obaid, A. A Hoare-Style Proof System for LOTOS, in Juan Quemada., José Mañas, and Enrique Vázquez, editors, *Proceedings of 3rd International Conference on Formal Description Techniques FORTE '90,* Nov. 1990, Madrid, Spain.

[55] Garavel, H. Compilation et vérification de programmes LOTOS, *Thèse de Doctorat*, Université Joseph Fourier , Grenoble, Nov. 1989.

[56] Garavel, H. Compiation of LOTOS Abstract Data Types, in S.T. Vuong, editor, *2nd International Conference on Formal Description Techniques FORTE '89*, Vancouver, B.C., Canada, Dec. 1989, 147-162.

[57] Garavel, H. and Sifakis, J. Compilation and Verification of LOTOS Specifications, in L. Logrippo, R.L. Probrt, and H. Ural, (eds.), Proceedings of the 10th International Symposium on Protocol Specfication, Testing and Verification, Ottawa, June 1990, 359-376.

[58] Garcia-Molina, H. Elections in distributed computing systems, *IEEE Transactions on Computers C31*, 1, Jan. 1982, 48-59.

[59] Gouda, M.G. and Han J.Y. Protocol Validation by Fair Progress State Exploration. *Computer Networks and ISDN Systems,* 1985, (9):353-361.

[60] Gribi, B. A Model Cheker for LOTOS, Master's Thesis, University of Ottawa, 1992.

[61] Gribi, B., and Logrippo, L. A Validation Environment for LOTOS, in *13th IFIP Symposium on Protocol Specification, Testing and Verification*, May 1993.

[62] Gouda, M.G., and Yu, Y.T. Protocol Validation by Maximal Progress State Exploration. *IEEE Transactions on Communications*, Jan. 1984, COM-32:94-97.

[63] Guillemot, R., Haj-Hussein, M., and Logrippo, L. Executing Large LOTOS Specifications. In: Aggarwal, S., and Sabnani, K. (eds.) *Protocol Specification, Testing, and Verification VIII*. North-Holland, 1988, 399-410.

[64] Hailpern, B.T. Verifying Concurrent Processes using Temporal Logic, *Ph.D. Thesis*, Computer Science Laboratory, Dept. of Electrical Engineering, Stanford University, Aug. 1980.

[65] Hailpern, B.T., and Owicki, S.S. Modular Verification of Computer Communication Protocols. *IEEE Transactions on Communications*, Jan.1983, vol. COM-31(1):56-68.

[66] Haj-Hussein, M. ISLA: An Interactive System for LOTOS Applications, *Master's Thesis*, Unicersity of Ottawa, 1989.

[67] Haj-Hussein, M., and Logrippo, L. Specifying Distributed Algorithms in LOTOS, to appear in *Procedings of Computer Networks Conference*, Wroclaw, 1991.

[68] Haj-Hussein, M. A LOTOS Data Type Definition for Integers, *Technical Report*, University of Ottawa, 1991.

[69] Haj-Hussein, M., Logrippo, L., and Sincennes, J. Goal-Oriented Execution for LOTOS, *Fifth International Conference on Formal Description Techniques FORTE '92*, France, Oct. 1992.

[70] Hallsteinsen, S.O., Venstad, A., Nyeng, A., and Martinsen, H. Transformational Program Development- An Approach for Translating SDL to CHILL, SDL '89, The Language at Work, *Proceedings of the Fourth SDL Forum*, in O. Faergemand and M. M. Marques, eds., Lisbon, Portugal, Oct. 1989, 283-292.

[71] Hoare, C.A.R. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[72] Holzmann G.J. Automatic Protocol Validation in Argos: Assertion Proving and Scatter Searching. *IEEE Transactions on Software Engineering*, Jun. 1983, SE13(6):683-696.

[73] Holzmann, G.J. Tracing Protocols. *AT&T Technical Journal*, Dec 1985, 64(10).

[74] Holzmann, G.J. Algorithms for automated Protocol Verification.*AT&T Technical Journal*, Jan.-Feb. 1990, 69(1):32-44.

[75] Holzmann, G.J. Design and Validation of Computer Protocols, Prentice Hall, Englewood Cliffs, NJ, 1991, ISBN 013-539925-4.

[76] Huet, G., and Lankford, D.S. On the Uniform Halting Problem for Term Rewriting Syatems, Laboratory Report 283, INRIA, Le Chesnay, France, Mar. 1978.

[77] Huet, G., and Oppen, D.C. Equations and Rewrite Rules: A Survey, in R. Book, editor, Formal Language Theory: Perspectives and Open Problems, Academic Press, New York, 1980, 349-405.

[78] Humblet, P.A. A distributed algorithm for minimum weight directed spanning trees, *IEEE Trans. on Communication Comm-31*, 6, June 1983, 756-762.

[79] IBM Europe, Technical improvements to CCITT recommendation X.25, *submission to study group VII*, Oct. 1978.

[80] International Organization for Standardization. Information Processing Systems. Open Systems Interconnection. Basic Reference Model for Open Systems Interconnection (ISO International Standard 7498), 1984.

[81] International Organization for Standardization. Information Processing Systems. Open Systems Interconnection. Formal specification of IS 8072 (Transport Service) in LOTOS, *ISO/TC97/SC6/WG4/N13*, Aug. 1986.

[82] International Organization for Standardization. Information Processing Systems. Open Systems Interconnection. Formal specification of IS 8073 (Transport Protocol) in LOTOS, *ISO/TC97/SC6/WG6/N233*, Mar. 1987.

[83] International Organization for Standardization. Information Processing Systems. Open Systems Interconnection. ESTELLE - A Formal Description Technique Based on an Extended State Transition Model, DIS 9074, 1987.

[84] International Organization for Standardization. Information Processing Systems. Open Systems Interconnection. LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour (ISO International Standard 8807), 1988.

[85] Itoh, M., and Ichikawa, H. Protocol Verification Algorithm using Rediced Reachability Analysis. In *The Transactions on the IECE of Japan*, Feb. 1983, vol. E66, 88-93.

[86] Kakuda, Y., Wakahara, Y., and Norigoe, M. A New Algorithm for Fast Protocol Validation. In *COMPASAC*, *IEEE*, 1986, 228-236.

[87] Knuth, D.E., and Bendix, P.B. Simple Word Problems in Abstract Algebra, in J. Leech (ed.), Computational Problems in Abstract Algebra, Pergamon Press, 1969, 263-297.

[88] Korach, E., Rotem, D., and Santoro, N. Distributed election in a circle without a global sense of orientation, *Int. J. of Computer Mathematics 16*, 1984, 115-124.

[89] Kuiper, R., and de Roever, W. Fairness assumptions for CSP in a temporal logic framework, in D. Bjoner, editor, Proceedings of TC.2 Working Conference on Formal Description of Programming Concepts, Garmisch Partenkirchen, North Holland, 1983.

[90] Kutten, S. A unified approach to the efficienr construction of distributed leader-finding algorithms, *Proc. IEEE COnf. on Communication and Energy*, Montreal, Oct. 1984.

[91] Lai, W.S. Protocol Traps in Computer Networks. A Catalog. *IEEE Transactions on Communications*, vol. COMM-30, No. 16, 1434-1449.

[92] Lam, S.S., and Shanker, A.U. Protocol Verification via Projections. *IEEE Transactions on Software Engineering*, Jul. 1984, SE-10(4):325-342.

[93] Lamport, L. 'Sometime' is sometime 'not never': On the temporal logic of programs, in *Proc. 7th annu. ACM Symp. Principles of Programming Languages*. Las Vegas, NV, Jan. 1980, 174-185.

[94] Lamport, L. Specifying Concurrent Program Modules, *ACM Transactions on Programming Languages and Systems*, Apr. 1983, 5(2):190-222.

[95] Le Moli, G. A theory of colloquies, in *First European Workshop on Computer Networks*, Arles, France, 1973, 153-173.

[96] Lin, F.J., Chu, P.M., and Liu, M.T. Protocol Verification Using Reachability Analysis: The State Space Explosion Problems and Relief Strategies. In *ACM SIGCOMM*, Aug 1987.

[97] Logrippo, L., Faci, M., and Haj-Hussein, M. An Introduction to LOTOS: Learning by Examples. To appear in Computer Networks and ISDN Systems.

[98] Logrippo, L., Obaid, A., Briand, J.P., and Fehri, M.C. An Interpreter for LOTOS, a Specification Language for Distributed Systems. *Software-Practice and Experience*, *18*, 1988, 365-385.

[99] Loureiro, A.A.F., Chanson, S.T. and Vuong, S.T. FDT Tools for Protocol Development, *Fifth International Conference on Formal Description Techniques FORTE '92*, Tutorial Paper, France, Oct. 1992, 38-78.

[100] Madelaine, E., and Vergamini, D. AUTO: A verfication tool for ditributed systems using reduction of finite automata networks, in S.T. Vuong, editor, *2nd International Conference on Formal Description Techniques FORTE '89*, Vancouver, B.C., Canada, Dec. 1989, 77-84.

[101] Merlin, P.M. A study of the recoverability of computing systems, *Ph.D. Thesis*, Dept. of Information and Computing Sciences, Univ. of California, Irvine, 1974.

[102] Merlin, P.M. A methodology for the design and implementation of communications protocols, *IEEE Transactions on Communications*, June 1976, 24(6).

[103] Milner, R. A Calculus of Communicating Systems. vol. 92 of *Lecture Notes in Computer Science*, Sringer-Verlag, 1980.

[104] Manas, J.A., and de Miguel-More, T. From LOTOS to C. In: K.J.Turner (ed.) *Formal Description Techniques.* North-Holland, 1989, 79-84.

[105] Musser, D.R. Abstract Data Type Specification in the AFFIRM System*, IEEE Trans. Software Engineering*, vol. SE-6(1).

[106] Nguyen, H.T., Jackson, L.N., and Parker, K.R. Reachability Graph Generator for SDL, SDL '89, The Language at Work, *Proceedings of the Fourth SDL Forum*, in O. Faergemand and M. M. Marques, eds., Lisbon, Portugal, Oct. 1989, 219-230.

[107] Nicola, R., and Hennessey, M. Testing equivalences for processes, *Theoretical Computer Science*, Nov. 1984, 34(1,2):83-133.

[108] Ohmaki, K et al. Design and implementation of an Application Interface for LOTOS Irocessors, in K. Parker and G. Rose, editors, *4th International Conference on Formal Description Techniques FORTE '91*, 1991.

[109] Park, D. Concurrency and Automata on Infinite Sequences, Proc. 5th GI Conference, *Lecture Notes in Computer Science 104*, 1981, 167-183.

[110] Pavón, S., and Llamas, M. The testing functionalities of LOLA, in Juan Quemada., José Mañas, and Enrique Vázquez, editors, *Proceedings of 3rd International Conference on Formal Description Techniques FORTE '90*, Madrid, Spain, Nov. 1990.

[111] Petri, C.A.Communication with automata, *Ph.D. Thesis*, Darmstat Institute of Technology, Bonn, Germany, 1962.

[112] Pnuelli, A. The Temporal Logic of Programs, In *Proc. of the 18th Symposium on Foundations of Computer Science*, Nov. 1977, 46-57.

[113] Pnuelli, A. A temporal semantics for concurrent programs, in Semantics of Concurrent Computation. New York: Springer-Verlag, 1979, 1-20.

[114] Queille, J.P., and Sifakis, J. Fairness and Related Properties in Transition Systems - A temporal Logic to Deal with Fairness, Acta Informatice, 1983, 19:195-220.

[115] Quemada, J, Pavón, S., and Fernández, A. Transforming LOTOS Specifications with LOLA. The Parameterised Expansion. In: K.J.Turner (ed.) *Formal Description Techniques.* North-Holland, 1989, 45-54.

[116] Quemada, J. Compressed State Space Representation in LOTOS with the Interleave Expansion. In B. Jonsson, J. Parrow, and B. Pehrson, editors, *Proceedings of Eleventh International Conference on Protocol Specification, Testing, and Verification,* North-Holland, 1991.

[117] Rasse, A. CLEO: Diagnostic des Erreurs en XESAR, *Thèse de Doctorat,* Institut National Polytechnique de Grenoble, June 1990.

[118] Rety, P., Kirchner, C., Kirchner, H., and Lescanne, P. NARROWER: a new algorithm for unificaton and its application to logic programming, in Dijon, editor, T*he First International Conference on Rewriting Techniques and Applications*.

[119] Richard, J., and Linn, Jr. The Features and Facilities of Estelle. In *Fifth IFIP Workshop on Protocol Specification, Testing, and Verification,* North-Holland, 1986, 271-296.

[120] Rodriquez, C. Spécification et validation de systèmes en XESAR. *Thèse de Doctorat*, Institut National Polytechnique de Grenoble, May 1988.

[121] Rubin, J. Testing Communication Protocols Using Random Legal Inputs. In *COMNET ‘85,* pages 645-662, Budapest, Hungary, North-Holland, Amesterdam, Oct 1985.

[122] Rudin, H. Automated Protocols Validation: One Chain of Development. *Computer Networks,* 1978, 2(4/5):373-380.

[123] Rudin, H., and West, C.H. Validation of Protocols Using State Enumeration: A Summary of some experience. In *INWG/NPL Workshop on Protocol Testing*, Oct 1981, vol. 1, 371-375.

[124] Rudin, H., and West, C.H. An Improved Protocol Validation Technique. *Computer Networks,* 1982, 6:65-73.

[125] Rudin, H., West, C.H., and Zafiropulo, P. Automated Protocol Validation-One Chain of Development. In *Computer Networks Symposium,* University of Liege, Belgium, Feb 1978.

[126] Saracco, R., and Tilanus, P.A.J. *CCITT SDL: Overview of the LAnguage and its Applications,* Computer Networks and ISDN Systems, 1987, vol. 13, 65-74.

[127] Schultz, G.D, Rose, D.B., West, C.H, and Gray, J.P. Executable, description and validation of SNA, *IEEE Transactions on Communications*, 1980, 28(4):661-677.

[128] Scollo, G. OSI TRASPORT SERVICE. A Constraint Oriented Specification in LOTOS, Reference Number: SEDOS/119.6. PROYECT ST 410 SEDOS. *PUBLIC REPORT, TASK C1*. Oct 1987.

[129] Scollo, G, and van Sinderen, M. Architecture Design of the Formal Specification of the Session Standards in LOTOS, in *Proceedings of the 6th international workshop on protocol specification, testing and verification,* T. Kalin, editor, North Holland, Amsterdam, 1987, 475-488.

[130] Shrivastava, S.K., and Ezhilchelvan, P.D. rel/REL: A Family of Reliable Multicast Protocol for High-Speed Networks. Technical Report, University of Newcastle, Dept. of Computer Science, U.K., 1990.

[131] The SPECS Consortuim and Bruijning, J. Evaluation and Integration of Specification Languages, *Computer Networks and ISDN Systems*, 1987, vol. 13, 75-89.

[132] Turner K.J. A LOTOS-based develpment strategy, in S.T. Vuong, editor, *2nd International Conference on Formal Description Techniques FORTE '89*, Vancouver, B.C., Canada, 157-174.

[133] Sunshine, C. Interprocess communication protocols for computer networks, *Ph.D. thesis*, Stanford University, Technical Report #105, Dec. 1975.

[134] Symons, F.J.W. Introduction to numerical Petri nets, a general graphical model of concurrent processing systems, *Australian Telecommunication Research*, 1980, vol. 14, no. 1, 28-32.

[135] Turner, K., editor, Using Formal Description Techniques, Wiley, 1993.

[136] van der Schoot, H. Validation Activities for LOTOS based on Static Data Flow Analysis, Master Thesis, University of Twente, 1993.

[137] van der Schoot, H., and Ural, H. Data Flow Oriented Test Selection for LOTOS, to appear in Computer Networks and ISDN Systems.

[138] van Eijk, P. The Design of a simulator tool, in P. van Eijk et al. (eds), *The Formal Description Technique LOTOS*, 1989, North Holand, Amsterdam, 351-390.

[139] van Eijk, P., Vissers, C.A., and Diaz, M.*The Formal Description Technique LOTOS*. North-Holland, 1989.

[140] van Eijk, P., and Eertink, H., Design of the LOTOSPHERE symbolic LOTOS simulator, in Juan Quemada., José Mañas, and Enrique Vázquez, editors, *Proceedings of 3rd International Conference on Formal Description Techniques FORTE '90,* Madrid, Spain,Nov. 1990, 709-712.

[141] van Eijk, P. Tool demonstraction: The Lotosphere integrated tool environment *lite*, in K. Parker and G. Rose, editors, *4th International Conference on Formal Description Techniques FORTE '91*, 1991.

[142] Vissers, C.A., Scollo, G., and Van Sinderen, M. Architecture and Specification Style in Formal Descriptions of Distributed Systems. In Aggarwal, S., and Sabnani, K., (eds.) *Protocol Specification, Testing, and Verification, VIII,* North-Holland, 1988, 189-204.

[143] Vissers, C.A. FDTs for open distributed systems: A restrospective and a prospective view, in L. Logrippo, R.L. Probrt, and H. Ural, (eds.), *The 10th International Symposium on Protocol Specfication, Testing and Verification*, Ottawa, June 1990, Invited Paper.

[144] Vuong, S.T. and Cowan, D.D. Reachability Analysis of Protocols with FIFO Channels. In *Communication Architectures and Protocols, ACM SIGCOMM*, University of Texas at Austin, Mar. 1983.

[145] Vuong, S.T. Hui, D.D., and Cowan, D.D. A Tool for Protocol Validation Via Reachability Analysis, in *Protocol Specification, Testing, and Verification VI,* North-Holland, 1987, 35-41.

[146] West, C.H. General Technique for Communication Protocol Validation, *IBM J. Res. Develop.*, 1978, 22(4):393-404.

[147] Wang, Y. and Larsen, K.G. Testing Probabilistic and Nondeterministic Processes, in R.J. LINN, Jr. and M.U. Uyar, editors,.*Protocol Specification, Testing, and Verification XII*, Florida, U.S.A, June 1992.

[148] West, C.H. Applications and Limitations of Automated Protocol Validation. In *Second IFIP Workshop on Protocol Specification, Testing, and Verification,* North-Holland, 1982, 361-371.

[149] West, C. Protocol Validation by Random State Exploration. *Protocol Specification, Testing, and Verification, VI.* North-Holland, Amsterdam, 1986, 233-242.

[150] West, C. Protocol Validation in Complex Systems. *SIGCOMM'89 Computer Communications Review,* Sept. 89, vol. 19, no. 4, 303-312.

[151] West, C.H. Protocol Validation - Principles and Applications, In *10th IFIP Symposium on Protocol Specification, Testing and Verification*, June 1991.

[152] West, C.H. and Zafiropulo, P. Automated Validation of a Communication Protocol: the CCITT X.21 Recommendation, *IBM J. Res. Develp.*, 1978, 22(1):60-71.

[153] Woltz, D. Design of a Compiler for Lazy Pattern Driven Narrowing, *Proceedings of the 7th international workshop on the specification of abstract data types*, Springer LNCS 534, 1991.

[154] Wu, C., and Bochmann, G. Fairness in LOTOS, in K. Parker and G. Rose, editors, *4th International Conference on Formal Description Techniques FORTE '91*, 1991.

[155] Zafiropulo, P. Protocol Validation by Duologue Matrix Analysis. *IEEE Transactions on Communications*, 1978, COM-26:1187-1194.

[156] Zafiropulo, P., West, C.H., Rudin, H., Cowan, DD., and Brand, D. Towards Analysing and Synthesizing Protocols. *IEEE Transactions on Communications*, 1980, 28(4):651-661.

[157] Zhao, J.R., and Bochmann, G.V. Reduced Reachability Analysis of Communication Protocols. In *Protocol Specification, Testing, and Verification, VI*, North-Holland, 1987 , 243-254.

Appendix A - Alternating Bit Protocol

This protocol provides a reliable, uni-directional data transfer service between two users, User1 the source and User2 the sink.

The protocol uses an unreliable full duplex one place channel to transfer protocol data units (PDUs) and acknowledgments. To ensure that the messages sent by User1 are received in the correct order by User2, the protocol associates a sequence number, alternating between 0 and 1, with the delivered (PDUs) and acknowledgments. Figure A-1 illustrates the overall composition of the Sender and the Receiver entities, associated with User1 and User2 respectively, and the unreliable channel. The gates used by the protocol to communicate with the channel are hidden from the environment, i.e. the users. The overall structure can be seen as follows:

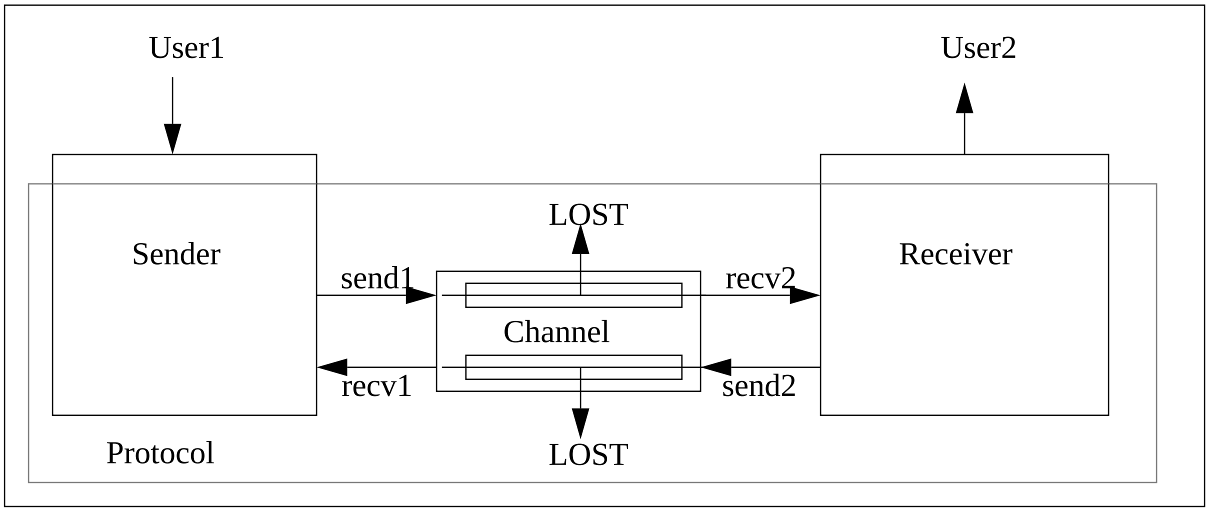


Figure A-1 Overall structure of alternating bit protocol and service

The service *abp_service* has two interaction points through gates *User1* the source and *User2* the sink. They both allow one event of sort *Data*. A value of sort *Data* is simply any natural number.

The protocol, specified in process *abp*, is a composition of three entities, the sender, the receiver and the unreliable channel. The sender entity sends PDUs and receives acknowledgments by communicating with the channel through gates *send1* and *recv1* respectively. The receiver entity receives PDUs and sends acknowledgments by communicating with the channel through gates *send2* and *recv2* respectively. The PDUs and the acknowledgments are both of sort *Mess*. Gate *LOST* is added to the channel process to identify the loss of a message.

```
specification abp_service[User1,User2] : noexit
library
 Boolean, NaturalNumber, Bit
endlib
        (* messages sent by User1 or received by User2 are of type Data, which are simply
         natural numbers *)
type DataType is NaturalNumber renamedby
   sortnames Data for Nat
endtype
        (* Acknowledgments and PDUs are both of type Mess.An acknowledgment is constructed
         by the operator makeack(Bit) that carries the sequence bit of the message to be
         acknowledged.A PDU is constructed by the operator makepdu(Data, Bit) that carries the
         message to be sent and the associated sequence bit. is_ack(Mess) and is_pdu(Mess) are
         boolean operators that determine if the given Mess is an acknowledgment or a PDU
         respectively *)
type message is DataType, Boolean, Bit
        sorts Mess
        opns
                makeack    : Bit -> Mess
                makepdu   : Data,Bit-> Mess
                compl: Bit -> Bit
                is_ack: Mess -> Bool
                is_pdu: Mess -> Bool
                data  : Mess -> Data
                seq  : Mess -> Bit
        eqns forall D:Data, S1,S2:Bit
        ofsort Bool
                is_ack(makeack(S1)) = true;
                is_ack(makepdu(D, S1)) = false;
                is_pdu(makeack(S1)) = false;
                is_pdu(makepdu(D, S1)) = true;
        ofsort Data
                data(makepdu(D,S1)) = D;
        ofsort Bit
                seq(makeack(S1)) = S1;
                seq(makepdu(D, S1)) = S1;
                compl(1) = 0;
                compl(0) = 1;
endtype
behavior
    hide send1, recv1, send2, recv2, LOST in
        abp[User1, User2, send1, recv1, send2, recv2, LOST](0 of Bit)
    where
```

```
process abp[User1, User2, send1, recv1, send2, recv2, LOST]
                                             (s_seq:Bit):noexit:=
   (sender [User1, send1, recv1, LOST] (s_seq)
    |||
    receiver[User2, send2, recv2] (s_seq) )
    |[send1, recv1, send2, recv2, LOST]|
    channel [send1, recv1,  send2, recv2, LOST]

   where
   (* The protocol entity 'sender' receives a message from User1 and delivers it, then
   repeats the same process *)
   process sender[User1, send1, recv1, LOST](s_seq:Bit) : noexit :=
         User1?D:Data;
         (deliver[send1, recv1, LOST](D, s_seq) >>
               sender[User1,send1, recv1, LOST](compl(s_seq)))
   where
   (* The delivery process sends a PDU containing the message sent by User1 along with
    the associated sequence bit. If it receive the proper acknowledgment for the PDU then
    the message is delivered. If not, it re-sends the same PDU once again until the proper
    acknowledgment is received*)
         process deliver[send1, recv1, LOST](D:Data; s_seq:Bit) : exit :=
          send1!makepdu(D, s_seq) ;
         (wait_ack[recv1, LOST](s_seq) >> accept ok : Bool in
          ([ok] -> exit
          []
          [not(ok)] ->deliver[send1,recv1,LOST](D, s_seq) (*re-deliver message *)
          )
         )
         where
               process wait_ack[recv1,LOST] (s_seq:Bit) : exit(Bool) :=
                     (recv1?M:Mess[is_ack(M)] ; exit(seq(M) eq s_seq))
                     []
                     LOST?M:Mess; exit(false)(* LOST *)
               endproc
         endproc
  endproc
   (* The protocol entity 'receiver' waits until a PDU is ready to be received from the
    channel. When it receives a PDU it acknowledges it and then it delivers the associated
    message to User2 only if the associated sequence bit is the one expected. *)
   process receiver[User2, send2, recv2](r_seq:Bit) : noexit :=
         recv2?M:Mess [is_pdu(M)];
         ([seq(M) eq r_seq] ->          send2!makeack(r_seq) ;
                                        User2!data(M);
                                        receiver[User2, send2, recv2](compl(r_seq))
         []
         [seq(M) ne r_seq] ->           send2!makeack(compl(r_seq));
```

```
                                        receiver[User2, send2, recv2](r_seq)
          )
     endproc
     (* unreliable channel: when a message is sent through the channel on gate send1 or send2,
      at can be delivered on gate recv1 or recv2 respectively or the channel may lose it if
      the internal action i is executed *)
     process channel [send1, recv1, send2, recv2, LOST] : noexit :=
          uni_channel[send1, recv2, LOST] ||| uni_channel[send2, recv1, LOST]
          where
          process uni_channel[send,recv,LOST]: noexit :=
               send?Msg:Mess;
               (recv!Msg; uni_channel[send,recv,LOST]
               []
               i; LOST!Msg; uni_channel[send,recv,LOST])(* message lost *)
          endproc
     endproc
  endproc
endspec
```

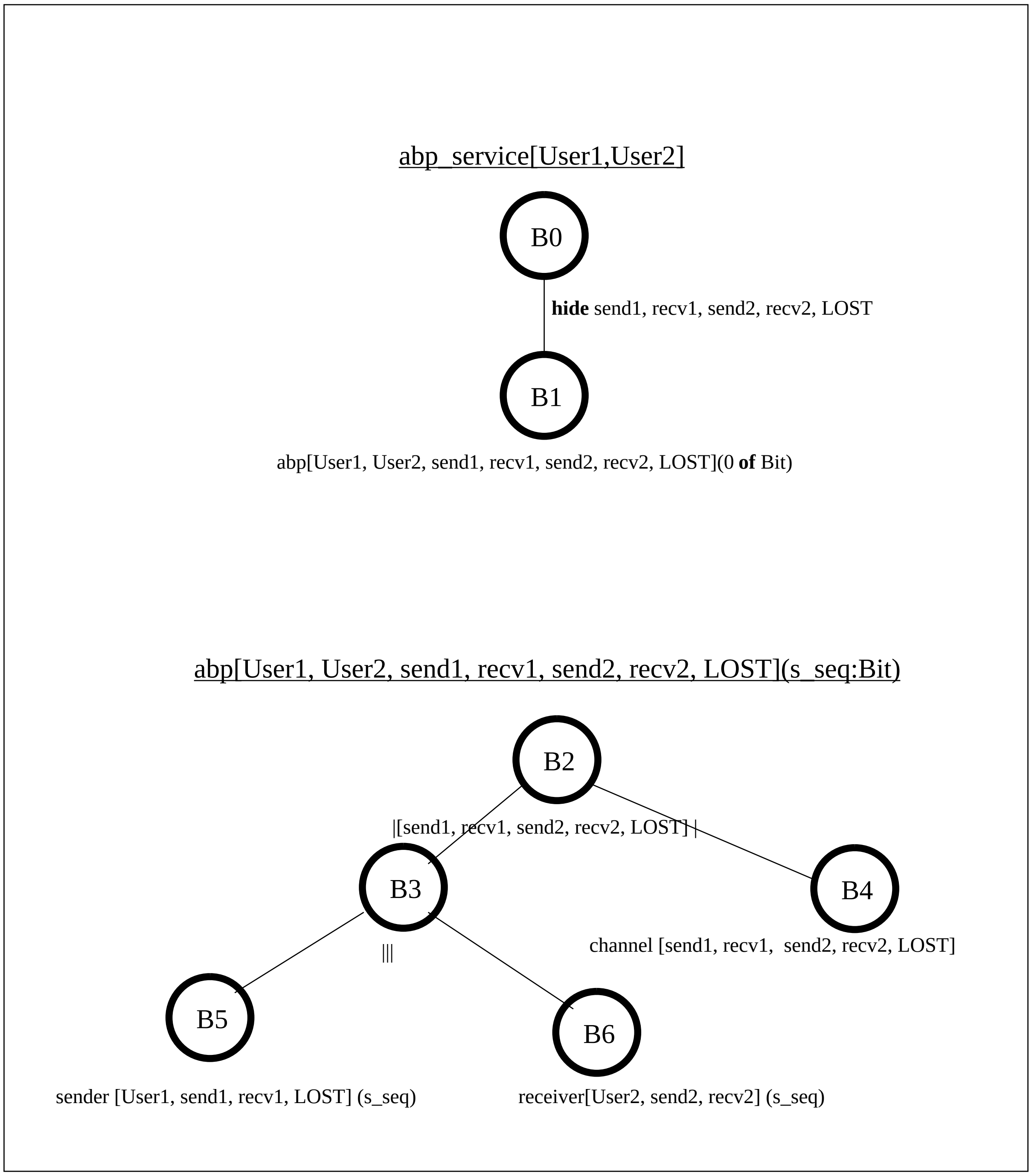

abp_service[User1,User2]
B0
hide send1, recv1, send2, recv2, LOST
B1
abp[User1, User2, send1, recv1, send2, recv2, LOST](0 **of** Bit)
abp[User1, User2, send1, recv1, send2, recv2, LOST](s_seq:Bit)
B2
|[send1, recv1, send2, recv2, LOST]|
B3
B4
channel [send1, recv1, send2, recv2, LOST]
|||
B5
B6
sender [User1, send1, recv1, LOST] (s_seq)
receiver[User2, send2, recv2] (s_seq)

Figure A-2 Top level view of alternating bit protocol and service

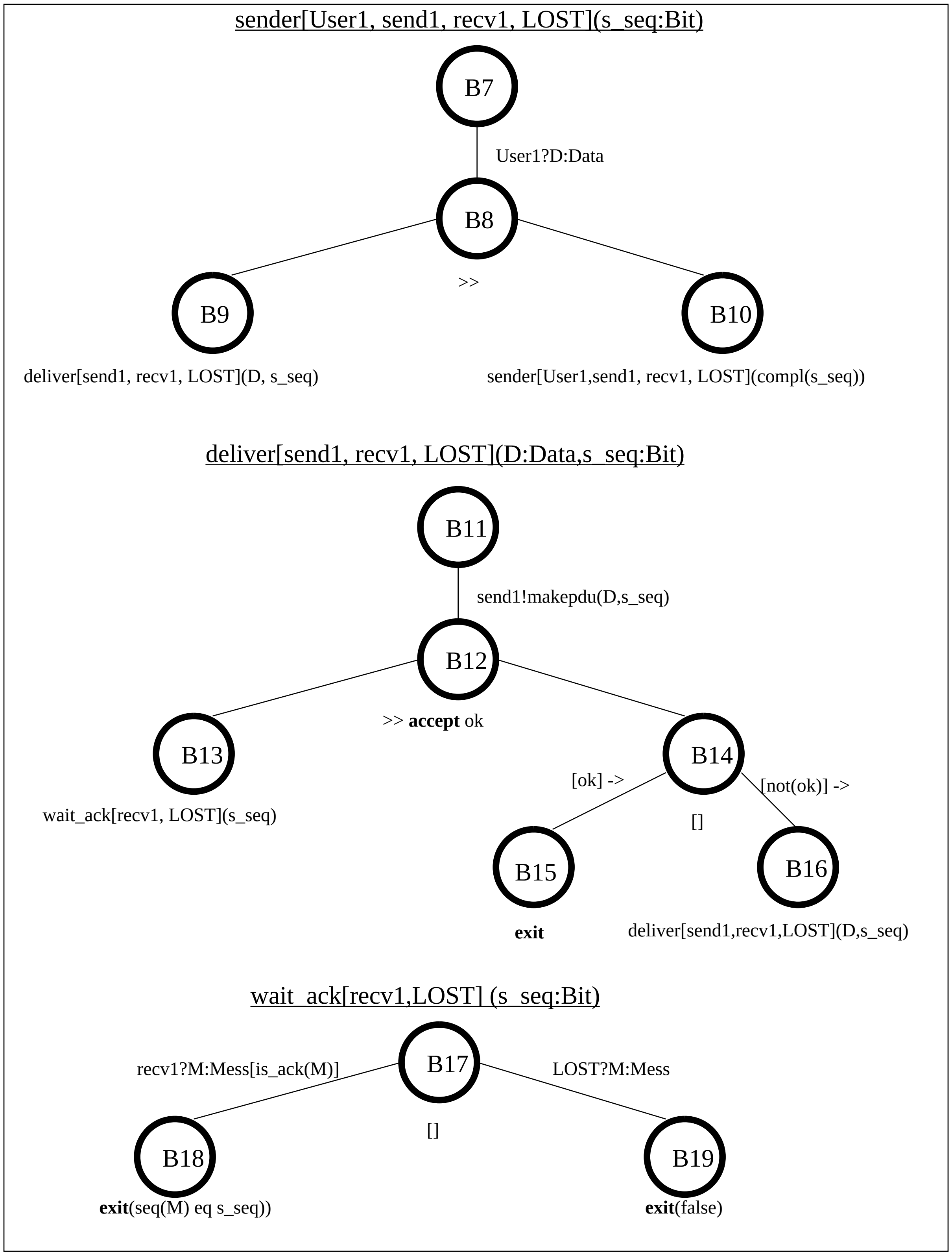

sender[User1, send1, recv1, LOST](s_seq:Bit)
B7
User1?D:Data
B8
>>
B9
B10
deliver[send1, recv1, LOST](D, s_seq)
sender[User1,send1, recv1, LOST](compl(s_seq))
deliver[send1, recv1, LOST](D:Data,s_seq:Bit)
B11
send1!makepdu(D,s_seq)
B12
>> **accept** ok
B13
B14
wait_ack[recv1, LOST](s_seq)
[ok] ->
[not(ok)] ->
[]
B15
B16
exit
deliver[send1,recv1,LOST](D,s_seq)
wait_ack[recv1,LOST] (s_seq:Bit)
recv1?M:Mess[is_ack(M)]
B17
LOST?M:Mess
[]
B18
B19
exit(seq(M) eq s_seq))
exit(false)

Figure A-3 Alternating Bit Protocol - Sender

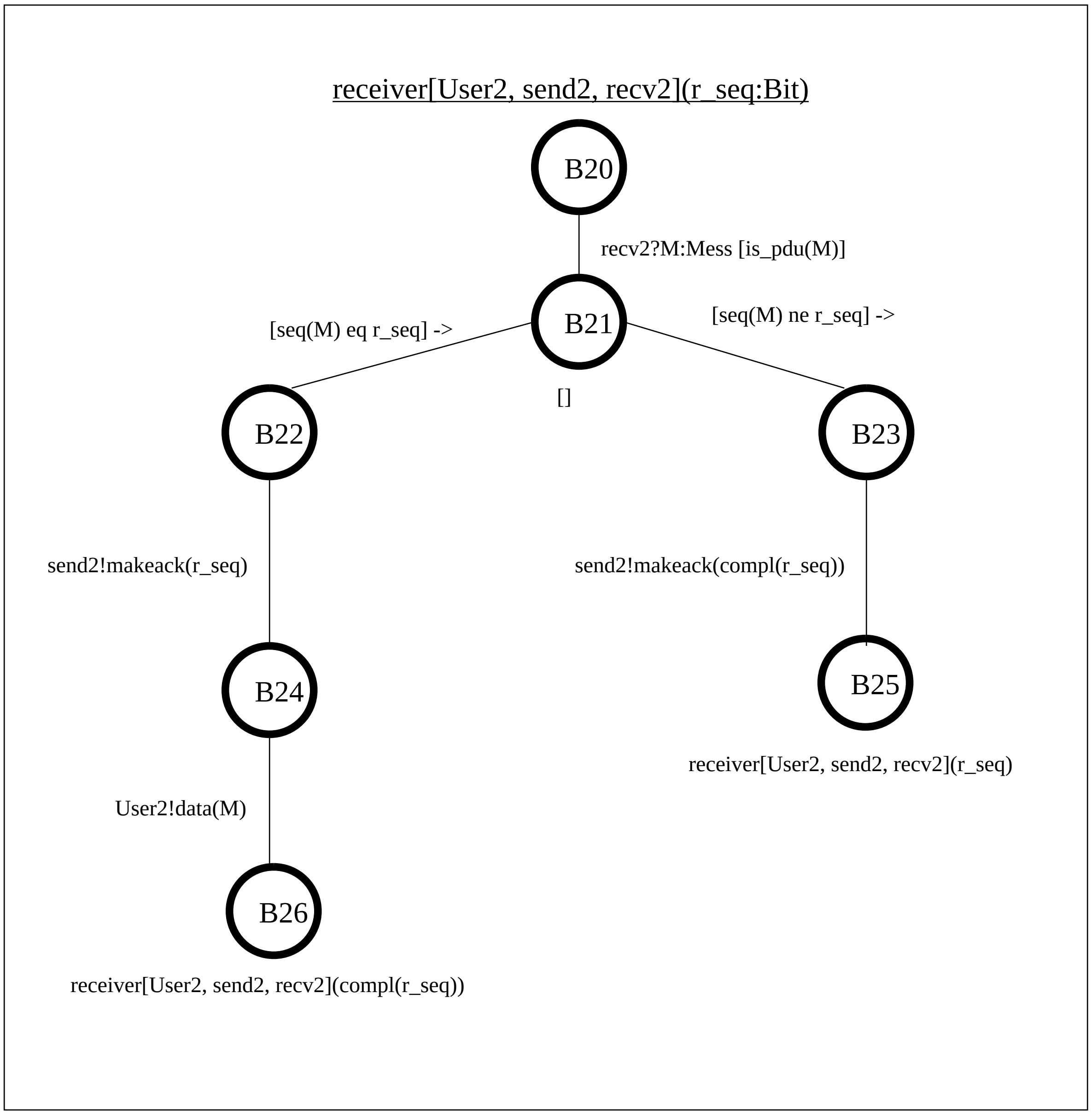

receiver[User2, send2, recv2](r_seq:Bit)
B20
recv2?M:Mess [is_pdu(M)]
B21
[seq(M) eq r_seq] ->
[seq(M) ne r_seq] ->
[]
B22
B23
send2!makeack(r_seq)
send2!makeack(compl(r_seq))
B24
B25
receiver[User2, send2, recv2](r_seq)
User2!data(M)
B26
receiver[User2, send2, recv2](compl(r_seq))

Figure A-4 Alternating Bit Protocol - Receiver

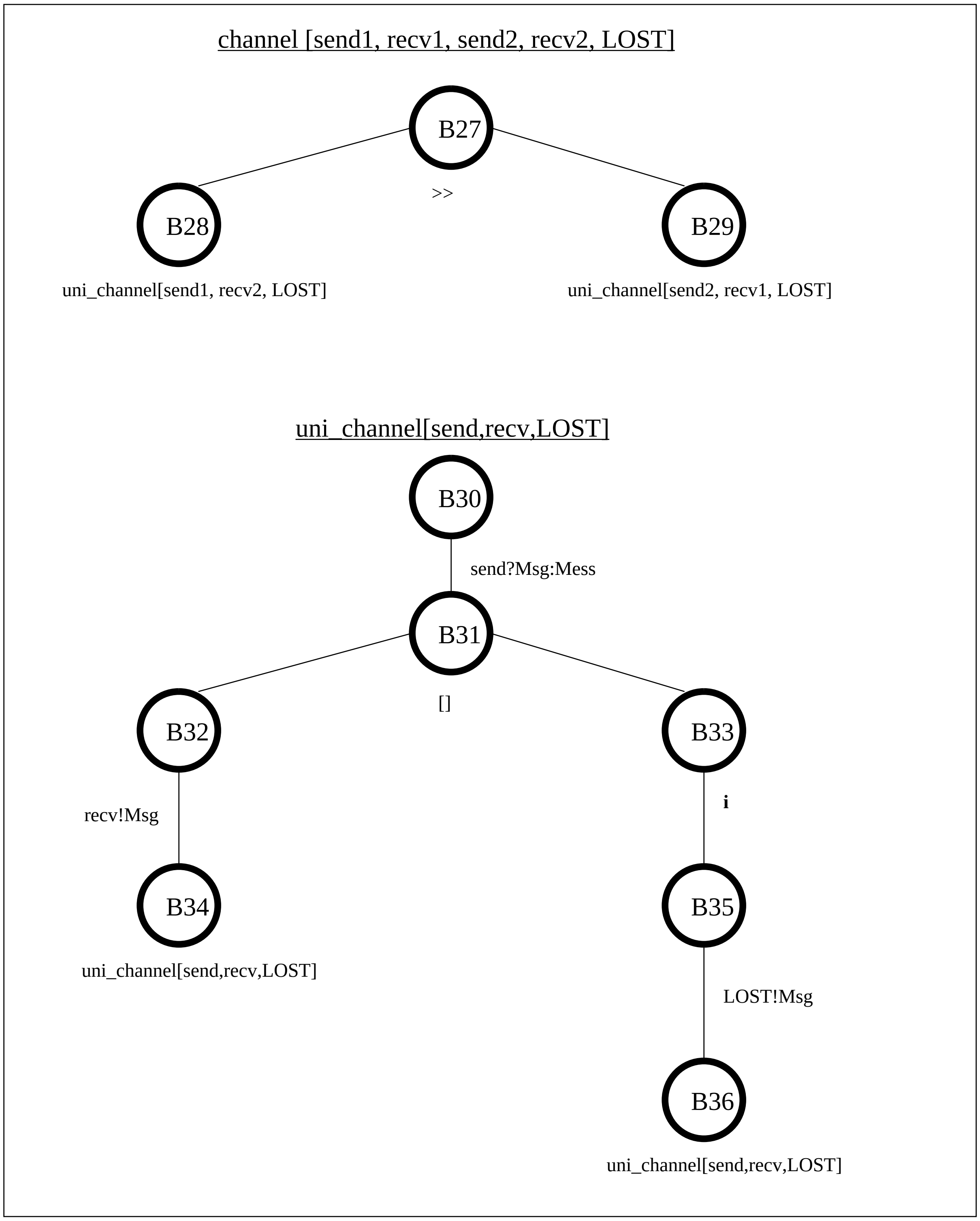

channel [send1, recv1, send2, recv2, LOST]
B27
>>
B28
B29
uni_channel[send1, recv2, LOST]
uni_channel[send2, recv1, LOST]
uni_channel[send,recv,LOST]
B30
send?Msg:Mess
B31
[]
B32
B33
recv!Msg
i
B34
B35
uni_channel[send,recv,LOST]
LOST!Msg
B36
uni_channel[send,recv,LOST]

Figure A-5 Alternating Bit Protocol - Unreliable Channel