Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática

Verificação Automática de Especificações OBLOG

Dissertação de Mestrado submetida ao Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa como parte dos requisitos para a obtenção do grau de Mestre em Informática.

por

Paulo Jorge Fernandes Carreira

Dezembro de 1999

Automatic Verification of OBLOG Specifications

This dissertation was prepared by Paulo Jorge Fernandes Carreira under the supervision of Prof. Dr. José Luiz Lopes Fiadeiro at the Department of Informatics of the Faculty of Science of the University of Lisbon in partial fulfillment of the requirements for the degree of Master in Computer Science. In the curricular part of his master's studies the author completed the following courses: *Concurrent Programming*, *Distributed Artificial Intelligence*, *Formal Specification*, *General Systems Theory*, *Neural Networks*, *Neural Dynamics*, *Machine Learning* and *Object Oriented Programming*.

December 1999

Verificação Automática de Especificações OBLOG

Esta dissertação foi preparada por Paulo Jorge Fernandes Carreira sob a supervisão do Prof. Doutor José Luiz Lopes Fiadeiro no Departamento de Informática da Faculdade de Ciências da Universiade de Lisboa como parte dos requisitos para a obtenção do grau de Mestre em Informática. Na parte curricular dos estudos o autor realizou as seguintes disciplinas: *Aprendizagem*, *Especificação Formal*, *Inteligência Artificial Distribuída*, *Neurodinâmica*, *Programação Centrada em Objectos*, *Programação Concorrente*, *Redes Neuronais* e *Teoria dos Sistemas Gerais*.

Dezembro de 1999

Abstract

Designing correct software has been a problem for as long as computers have existed. Although Software engineering methodologies have contributed to the increasing quality of software systems, they cannot be successful without appropriate tools. Recent advances in a branch of software engineering that borrows concepts from discrete mathematics named Formal Methods, has enabled the automatic verification of computer systems although with some limitations. The two main verification approaches that emerged from Formal Methods are called *Automatic Theorem Proving* and *Model-Checking*. These two approaches left academia and have achieved success in the verification of real world systems. An obstacle for the extended use of these tools is requiring the formalization of systems in mathematical based notations. This thesis intends to take a step further in the direction of automatic verification of software by proposing a strategy for the verification of a high-level object-oriented specification language. The approach presented here is based on a translation of a subset of OBLOG specifications into two different formalisms: The *Process Algebraic* language LOTOS and *Communicating Automata*. These concepts were tested in the verification of the Alternating Bit Protocol with acceptable results.

Keywords

Automata, Automatic Verification, Computer Aided Verification, Formal Methods, Formal Specification, Model-Checking, Object-Oriented Languages, Process Algebras, Specification Languages, Temporal Logic.

Resumo

O desenho de software que opere correctamente tem constituído um problema desde que os primeiros computadores foram construídos. Muito embora a adopção de métodos de Engenharia de Software tenha dado um contributo visível no aumento da qualidade dos sistemas de software, não se podem atingir os padrões de qualidade necessários sem ferramentas apropriadas. Avanços recentes numa área da Engenharia de Software que importa muitos dos seus conceitos do campo das matemáticas discretas, denominada Métodos Formais, tem vindo a tornar possível a verificação automática de sistemas computadorizados ainda que com algumas limitações. As duas aproximações principais têm por nome *Demonstração Automática de Teoremas* e *Verificação Baseada em Modelos*. Tanto uma como outra deixaram o meio académico alcançando sucesso na verificação de sistemas de tamanho real. Um dos obstáculos que se põe ao uso extensivo destas técnicas é o requisito de utilizar uma notação matemática para representar os sistemas que pretendemos verificar. Esta tese tem por objectivo dar mais um passo em frente na direcção da verificação automática de Software propondo uma estratégia para a verificação de especificações escritas numa linguagem de alto nível. As ideias apresentadas assentam na tradução de um sub-conjunto das especificações OBLOG para dois formalismos diferentes, a saber: A linguagem LOTOS, baseada na *Álgebra dos Processos* e os *Autómatos Comunicantes*. Os conceitos apresentados foram testados na verificação de uma versão do Protocolo do Bit Alternante com resultados aceitáveis.

Palavras Chave

Álgebras de Processos, Autómatos, Especificação Formal, Métodos Formais, Linguagens Object-Oriented, Lógica Temporal, Verificacão Automática, Verificação Assistida por Computador, Verificação Baseada em Modelos.

Preface

By the praxis of dissertations this is the section in which I should thank everyone involved in this work. The task is sometimes unfortunate for the writer (me) – do I have enough space to thank all my closest friends? And what about all those anonymous speakers and writers that sparked some thought on me and made me rethink my work? I am also indebted to them, am I not? So, just because your name is not here that doesn't mean that I don't remember your smile.

This thesis is the result of a one-year research project in Automatic Verification as a grantee sponsored by OBLOG Software. The working environment at OBLOG is great both in people and logistics. Many thanks go to all my friends there.

I won't only thank but single out my advisor, Prof. José Luiz Fiadeiro for starting me into this wonderful field of Computer Science. For his commitment in finding money for me, for giving me the opportunity to meet researchers from other countries, for his patience and many times, inspiring good humor – Thanks. I believe this can only be compensated with excellent work.

Some notes for some friends; Gonçalo: Thanks for taking me home. Rui Bastos: Thanks for suffering some of my chapters. José Cruz: Our discussions about Turing Machines were inspiring and I definitely understand your dilemma of dishes after dinner.

Thanks to Asia, Gosia, Justyna, Karolina and Magda, I can now 'concat' some words in Polish – Your commitment in teaching me is greatly acknowledged. I recognize it was a good anti-stress exercise reading a foreign language at night, in those days of theory. Especially Karolina, who helped me with this proverb: *Dzisiaj lepiej niż wczoraj. Jutro lepiej niż dzisiaj* – Karolina, *codziennie*! (I can e-mail the translation to the most curious minds).

Finally to my dearest Susana, whose love and support played a significant deal in the necessary energy to carry out this job. Thanks. Thanks for being so generously that 'parallel' part of me, so that I could work on thesis problems. You are great!

To all of you who were touched by my sometimes-rude temperament, please forgive me.

I end up saying that I have been blessed, with family and friends that made my life so much richer, so much fuller, and so much happier. In this work, I'm convinced, there is also a part of each of them.

Paulo Carreira

Index

Automatic Verification of OBLOG Specifications ii
Verificação Automática de Especificações OBLOG ii
Abstract iii
Keywords iii
Resumo iv
Palavras Chave iv
Preface v
Index vi
Introduction 1
Chapter 1 – Automatic Verification in Software Engineering 3
1.1 Problems of the classical software engineering processes 3
1.2 Formal methods for software engineering 4
1.3 Automatic verification 6
Chapter 2 – State Transition Systems and Temporal Logic 10
2.1 Transition systems 11
2.2 Modal Logic 12
2.3 Linear temporal logic 13
2.3 Branching time logics 15
2.4 Hennessy-Milner Logic 18
2.5 Mu-Calculus 19
2.6 Specification of properties using temporal logic 21
Chapter 3 – Model-Checking 26
3.1 Process overview 26
3.2 State space generation 27
3.3 Model-Checking approaches 28
3.4 Optimizing the Model-Checking process 30
3.5 Classes of systems 32
3.6. Some of the existing verifiers 34
Chapter 4 – The OBLOG specification language 37
4.1 Objects, Operations and Methods 37
4.2 Features left out of the OBLOG language 38
4.3 Behavior Components 39
4.4 Some aspects of coding behavior components 41
4.5 Specification of the Alternating Bit Protocol 44
Chapter 5 – Process algebraic verification approach to OBLOG specifications 47
5.1 Technical framework 47
5.2 Translating OBLOG specifications into LOTOS 53
5.3 Case study – verifying the Alternating Bit Protocol 63
Chapter 6 – Verifying OBLOG specifications using Communicating Automata 68
6.1 Technical framework 68
6.2 Translating OBLOG programs into IO-Automata 71
6.3 Case study – verifying the Alternating Bit Protocol 80
Conclusion 85

References 90
Appendix A – Syntax of the OBLOG specification language.. 98
Appendix B – Initial OBLOG specification of the ABP... 100
Appendix C – Syntactic sugar free specification of the ABP....................................... 103
Appendix D – LOTOS specifications of the ABP 107
Appendix E – Summary of properties verifyed with CADP 120
Appendix F – UPPAAL .ta specification of the ABP... 121
Appendix G – UPPAAL .ta specification of test automata and properties 129

Introduction

Whenever we look around, to parking meters, medical instrumentation, power plants, telephone switching equipment or aircraft navigation systems, we can perceive an increasing dependence of humans on *computerized systems*. These systems exhibit complex behavior because they are constituted of many interacting parts that operate simultaneously and interact with their surrounding environment in many different ways.

In general, the human being does not have an inborn ability to think of every possible interaction emerging from the kind of systems just mentioned. For that reason, when designing software for them, misconceptions are frequent leading to errors that are not manifested in the test phases but only during systems operation as a result of complex interactions with the environment. Consequences of failures of systems of this kind can be ranked in material and human loss or even in environmental damage. The development of better concurrent programming languages and techniques for quality measurement, test and verification of software has become imperative.

About 30 years ago, Floyd and Hoare [Floyd 67, Hoare 69] proposed the first techniques for verifying program correctness. However, these techniques were limited only to a subset of the class of *imperative programs* that are not suited, in general, to program the systems mentioned above.

The quest of program verification was greatly influenced in the late 1970s, when A. Pnueli proposed the use of Temporal Logic to reason about the behavior of *concurrent* and *reactive* programs [Manna & Pnueli 92, Manna & Pnueli 95].

The results of A. Pnueli together with efforts in protocol verification lead to the development of a technique known as *Model-Checking* [Burch & al. 90]. This verification technique, as opposed to some previous ones, allowed the verification process to be completely automated making it attractive for many applications [Clarke & al. 86].

Model-Checking is carried out in two phases. The first phase is the construction of a graph (the model) from the program through exhaustive enumeration of all possible states into which the system can run, connecting them by arcs representing state transitions and labeling them with actions taken by the system. The second phase consists in exploring this graph in order to verify if it meets some specific properties. For this technique to be feasible, the number of states in the system model must be finite. It happens in practice that, although finite, the number of states of even small real world systems is often so high that it falls out of the capacity of the machines that most of us have access to.The *state space explosion* problem is certainly the prominent obstacle of the Model-Checking approach. This problem has been motivating the upsurge of diverse theoretical results and techniques that applied in a conjoint way can make possible the verification of real world systems [Clarke & al. 94, Courcoubetis & al. 92].

Although hardware industry is already using Model-Checking techniques with success for almost one decade [Clarke & al. 92] only recently verification tools appeared for software systems and still with some limitations. This situation is manly due to the intrinsic complexity of software verification, which is higher than hardware's and to the reliability needs of hardware that prompted for this technology earlier.

Verification tools for software also offer very low level languages that are normally based on *automata* or *process calculi*. Nevertheless, we have reasons to believe that software verification will become a reality taking into account that the size of systems handled by verification tools has been growing exponentially. In 1983 the first model checkers only supported 10^4 states, whereas in 1998 it has been possible to verify a system with 10^{120} states.

In this context, it is reasonable to pursue the study of techniques for verification of OBLOG specifications [Andrade & Sernandas 96]. The OBLOG approach is constituted of an object-oriented specification language and a box of auxiliary tools that allow the transformation of specifications into of-the-shelf programming languages. Providing a *correct* development framework for the OBLOG toolbox can be done by verifying the correctness of the specifications and the correctness of the transformations. This work constitutes a first contribution towards the verification of specifications while the verification of the transformations is subject for future research work.

Our proposal for verifying OBLOG specifications relies in creating an interface to take advantage of existing Model-Checking tools instead of building a model checker specifically for OBLOG. The first part of this work consisted in an evaluation of fitness-for-purpose of different existing tools focusing on their performance and capacity to analyze software systems. Pragmatic aspects were taken into account like the degree of automation of the tools and their integration with other tools, so that the results here contained could be used in the development of a prototype for verification of OBLOG specifications.

Two verification suites were selected: 1) The French tool CADP [Garavel 98] developed at INRIA, mainly due to its interfacing capabilities and its capacity to handle very large specifications like the specification of the Airbus Flight Warning Computer. 2) The second tool is the Swedish/Danish tool UPPAAL [Bengtsson & al. 95] jointly developed by the University of Uppsala and by the University of Aalborg. This second choice was motivated by the need of a complementary approach that could give us a somewhat different view and by the performance of this tool that was about ten times the speed of similar tools.

Finally, a subset of the OBLOG specification language (Chapter 4) was selected and a set of rules to convert OBLOG specifications into the input languages of both verification tools was developed (Chapter 5 and 6). The results contained in these chapters were tested with an example that also allowed tuning some of the translation rules.

This dissertation is organized as follows: In Chapter 1, the motivation and use of formal methods and verification technologies are discussed. Chapter 2 presents the necessary mathematical background of this work. The presentation of Model-Checking technology is made in Chapter 3. In Chapter 4, the subset of the OBLOG language is presented. Chapter 5 describes the verification approach using a *process algebraic* coding and Chapter 6 presents the *automata theoretic* coding.

Chapter 1 – Automatic Verification in Software Engineering

This chapter aims to frame the Automatic Verification procedures in the context of Software Engineering and Formal Verification. We start by drawing a sketch of the Software Engineering process together with its weaknesses. Formal Methods are then introduced, proposing to minimize some of the problems found in the traditional Software Engineering approach such as ambiguities and lack of automation, among others. Still in this section, a further insight is carried out on the usefulness of Formal Methods and on their integration within the traditional approach. In the final section, some observations that motivated the use of Automated Verification are given, introducing two different approaches: Proof-Theoretic and Model-Theoretic verification.

1.1 Problems of the classical software engineering processes

Many organizations use a software development process that is some variant of the V life cycle diagram presented in Figure 1.1. Starting from requirements, the development process proceeds through several phases until a final product is delivered. Then, the maintenance phase begins.

This traditional approach raises some problems. A central one is that errors are discovered too late in the software development life cycle, when they are expensive to repair. As a result, the user usually gets lately delivered and misconceived software. In [Pressman 97] it is outlined that early stages of the software development life-cycle are error prone and moreover, errors made here can have lasting influence on reliability and cost of the system.

Nowadays, in the software industry, the various issues affecting the classical approaches to software engineering process are somewhat common sense. These issues can be stated generally, as the incapacity to understand the client expectations, followed by an imprecise requirements specification and ending with the inability to verify conformance of requirements. The result is obvious: We do not know whether we are building the right program, and even if we do know, we will not be able to tell if we are building it right.

Currently, the methodology used in earlier stages of the software development process (like Requirements Specification and System Analysis) are rather ad hoc. They are often based of informal or semi-formal diagrammatic descriptions, retaining ambiguities that will reduce the usefulness of future inspections made to the implementation, because the inspector and the implementer's interpretation of the specification might be different. Furthermore, such descriptions do not allow automation

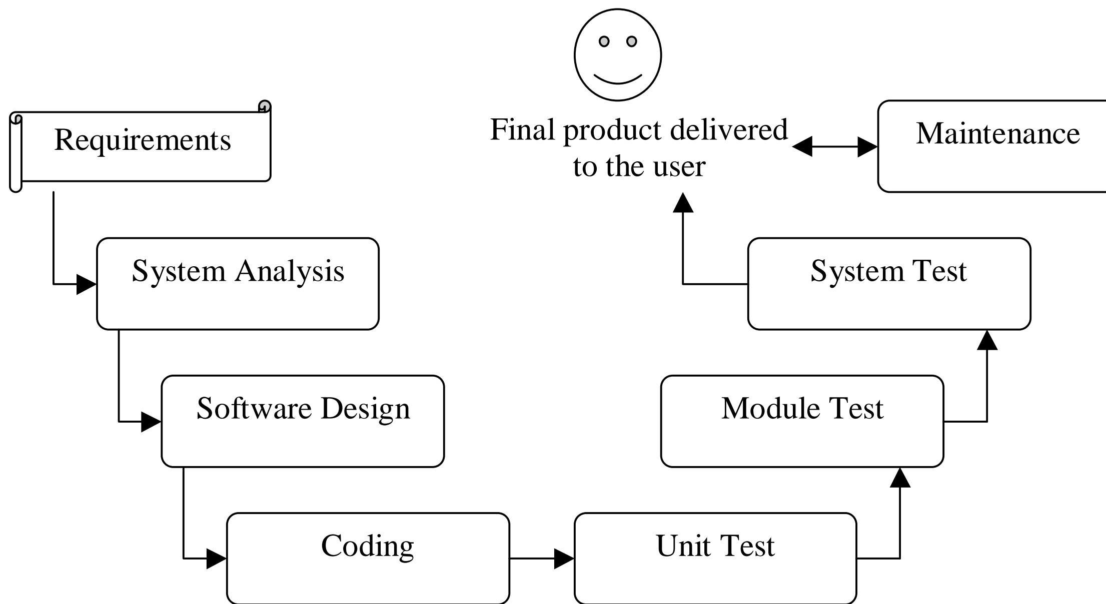


Figure 1.1 – The V software life cycle diagram

tools, forcing the development team to perform a manual and tedious tasks of testing and reviewing. We are interested in tools to perform these inspections automatically.

1.2 Formal methods for software engineering

The term *Formal* refers to the use of techniques from logic and discrete mathematics that aim at a rigorous approach to the development of computer systems and software in the tasks of specification, construction and validation.

The specification task is carried during the Systems Analysis and Software Design phases and is often referred to as *Formal specification of properties*. In this phase, essential properties of the system are identified and written in a language that can express the desired evolution of the system in time. Examples of properties are

1. Deadlock-freedom: e.g.: "A system constituted of two processes is always able to take an action, either through P_1 or through P_2.
2. System response: e.g.: "After performing a withdraw operation the balance is decreased".

Essentially, a formal specification is a set of formulas $\{\varphi_i\}$ where each φ_i expresses a property of the system.

In the construction task, a *specification of the system behavior* (or *implementation*) is carried out, usually during the Software Design and Coding phases. The behavioral description of the system expresses, in some way, a *transition system*. This is a set of *system states* and a relation between them called *transition relation*. Informally, the states of a program (or behavior description) are vectors where each component is a valuation for a program variable. The transition relation describes to which states we can go from a certain state by performing a certain action. Intuitively, the transition relation describes the effects of commands on states. Several languages exist for the purpose of behavior specification. Usually they are based on Process Algebras like

CSP, CCS, LOTOS or some automata theoretic framework like Communicating Automata.

The validation task consists of checking whether the specified properties follow from behavior specification. The meaning of "follow from" is explained in Section 1.3. In real-life software projects, sometimes due to the lack of appropriate tools, not every property and behavior of the system is specified.

We can look at Formal Methods in two perspectives and see two different roles of Formal Methods. As noted above, Formal Methods intend to be applied in determining the quality of specifications, for example if they are consistent (i.e. we cannot conclude contradictory facts), if certain properties are consequences of the specified requirements or whether one level of design implements another, among others. In such cases the focus of Formal Methods is *analytical.* From the other viewpoint, we say that the focus is *descriptive*. The descriptive focus of Formal Methods refers to the clarification of the systems requirements documentation through the use of a *formal specification language* that facilitates communication among development teams helps the inspections or reviews and allows quality certification of the system being developed. Formal methods can be usefully applied in both analytical and descriptive perspectives; in real-life projects however, this may not be so straightforward.

The encouragement of systematic enumeration and exploration of cases needed to cope with formalization in the System Analysis and Software Development phases leads to the detection of more design problems than it would otherwise be possible with traditional methodologies. However, drawbacks of Formal Methods also exist and in the absence of good tools and a skilled team we are certainly voted to failure.

The application of Formal Methods need not cover the complete software development cycle. Instead, they can be used to enhance individual phases. In companies where a software development method is already well established, its enhancement with the application of Formal Methods is often very rewarding. An explanation of this fact can be given if we take a look at each phase of the development cycle separately:

- In the System Analysis phase it as been noted above the advantage of early fault detection. Here, Formal Methods provide accuracy where it is lacking in conventional analysis approaches[1].
- In the Software Design phase, architectural aspects are considered. By focusing on critical modules of the system we are able to ascertain which modules should receive Formal Methods analysis. This task is also known as *criticality assessment*.
- In the Coding phase, a tool that guides us through the functionalities that need to be implemented next, that gives us some measure of how much functionality was implemented by comparing our implementation with the specification, would be certainly useful. This is feasible with the application of Formal Methods. Finally, the code could be verified in some automatic or semi-automatic fashion.

In this context, Formal Methods can certainly be usefully employed.

---

[1] A noteworthy work linking Formal Methods and Systems Analysis can be found in [Moreira 94]. Herein, Moreira proposes the *Rigorous Object-Oriented Analysis* (ROOA) – A method that integrates *formal specification languages* with *object oriented analysis methods*.

1.3 Automatic verification

Production-line control equipment, aircraft navigation systems, medical instruments or power plant monitoring systems are some examples of highly complex computerized systems whose failure can directly lead to personal injury, death or perhaps provoke high environmental damages. These systems must be highly reliable and verifying them becomes essential.

Safety-critical systems demand for the most sophisticated correctness assurance technology available. Meanwhile, the fact that "bugs cost money"[2] is progressively attracting the focus of industry to the practices of verification. For example the Pentium FDIV bug may have cost Intel 500 million USD. Another example was the explosion of the Ariane space launcher in June 1996 ultimately due to a software failure [ESA 96]. The loss was estimated at 610 million Euros, about 675 million USD.

Computer systems verification (software and hardware) can be done in a non-formal or formal way. What distinguishes formal verification from other approaches is the use of a mathematical based process. This process ensures that if a system is formally verified to have a certain property then it will never exhibit a behavior that violates this property (assuming that the verification procedure is correct). The non-formal (and traditional) ways of attaining reliability standards are based on performing exhaustive testing, but testing real-world software systems to be 100% reliable is infeasible. Because of this, a variety of non-formal techniques exist that try to enhance the quality of the system in the production phase where testing efforts are cleverly balanced to achieve greater coverage on rather critical parts of the system. However, it is commonly recognized by the industry that even these efforts are insufficient for many applications. In general, testing can only demonstrate the presence of errors and not their absence.

Today, there is a consensus among verification practitioners that verification is too complex to be carried out by hand, at least from a pragmatic point of view. The main advantage of formal verification is enabling the automation of the verification process, this means that unto some point, a machine can do the job. Often referred as the "*push-button*" principle of verification or *automatic verification*, this kind of formal verification consists in giving the machine a specification of the system together with the specification of a property, pushing a button and waiting until the machine stops and yields either a yes or no. A 'yes' means that the system has the required property and a 'no', that the system does not have that property.

The recent upsurge of interest in formal methods was greatly influenced by the appearance of techniques to perform automatic verification. Although far from being an El Dorado, automatic verification as proven to be successful in many domain-specific applications like communication protocols and hardware systems [Clarke & al. 92].

We may identify two classes of programs (often referred as systems)[3]. One class, is the class of those programs that are ordinarily described as *sequential programs*. The second class is the class of *continuously operating programs* (or *reactive systems*), like operating systems, communication protocols or traffic control systems.

---

[2] Quotation from [Hu 95]

[3] Some authors prefer to distinguish the terms "program" and "system", for instance the program can be considered a part of the system. Herein however, a "program" is a description of a "system" at some level of abstraction. In the current context the two terms can be used interchangeably without confusion.

Classical approaches for reasoning about sequential programs are based on formalisms like Floyd's *strongest consequents*, Dijkstra's *weakest pre-conditions* or Hoare's Logic. Floyd's work introduced the first formalism for verifying the correctness of computer programs. He introduced the idea of *strongest verifiable consequent,* this is the strongest assertion which holds after the execution of a statement, given an assertion that holds before that statement. By giving a set of rules to automatically compute the strongest verifiable consequent for every language construct, Floyd presented a formalism to verify the correctness of programs (without loops). Dijkstra and Hoare expanded Floyd's work. Hoare starts from the post-condition and computes the pre-condition.

These classical verification formalisms were developed based on the observation that sequential programs initially accept some input, perform some computation and, finally, output a result. We can see them as a *transformation* from *initial states* to *final states*, or, from *pre-conditions* to *post-conditions*. However, in the class of reactive programs, we can observe an arbitrarily long, possibly non-terminating behavior that maintains ongoing interaction with its *environment*, where outputs can influence future inputs. We must remark that, environment in this setting, possibly contains users, other programs, or hardware capable of producing events and data. Thus, the class of reactive systems subsumes many programs commonly labeled as concurrent, distributed or parallel. Since reactive programs are continuously operating and yield infinite computations, there is no final state. So, in this framework, initial-state/final-state formalisms like Hoare Logic are of little use.

In the late 1970s A. Pnueli showed that Temporal Logic could constitute an effective formalism to reason about reactive and concurrent systems. Details can be found in [Manna & Pnueli 92, Manna & Pnueli 95].

Proposed techniques to achieve the goal of systems verification can be cast into two groups: *Proof-Based Methods* (also called *Deductive Methods*) and *Model-Theoretical methods* (also called *State-Exploration Methods*). Both have been intensively studied in the literature and numerous algorithms have been proposed. Deductive methods comprehend two techniques: *Proof Checking* and *Theorem Proving*, whereas Model-Theoretical methods refer to a technique known as *Model-Checking*. Throughout this dissertation the Model-Checking technique will receive extensive treatment. More detailed analysis of Proof-Checking and Theorem Proving can be found in, e.g. [NASA 97]. A reference to the distinction drawn between Proof-Theoretical Methods and Model-Theoretical Methods can be found in [Emerson 91].

Deductive Methods

Deductive methods refer to the use of *deductive reasoning*. In order to give an informal definition of deductive reasoning we must recall two of the constituting parts of a logic : *Axioms* and *Inference Rules*. These two together denote the infinite set of all theorems of a logic L, by successive instantiation of Inference Rules on Axioms and previously obtained theorems. Starting from a set of L-formulas A (or *Assumptions*), the *set of consequences* of A in L may be described as the successive instantiation of Inference Rules of L on Assumptions of A, Axioms of L and previously obtained consequences of A. Deductive reasoning is the task of verifying whether a L-formula C is a consequence of a set of Assumptions A. If this task succeeds, we can say that C *follows from* A, C can be *proved* from A or symbolically, $A \vdash_L C$.

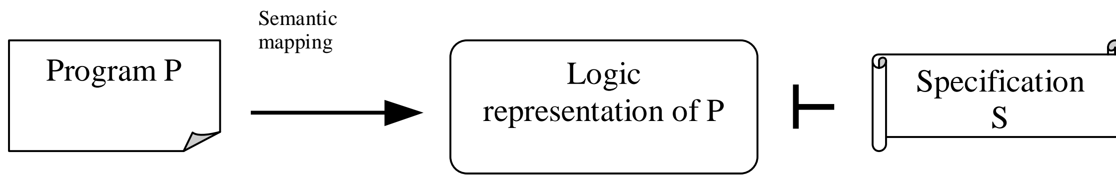


Figure 1.2 – Deductive approach to software verification

In the context of Automatic Verification, in order to make use of deductive methods we are interested in having a logic representation of the program P. This is achieved through a *semantic mapping* that produces a set of formulas from P to be used as assumptions. Using these assumptions, we are able to verify the conformance of program P with specification S. A program P is conformant with a specification S if every formula of S is a consequence of the assumptions drawn from P.

An important technique intimately related to theorem proving is *Proof Checking*. The idea behind it is the following: the user builds the proof and the Proof Checker double-checks it. This kind of work can be quite tedious, but nowadays proof-checking tools provide a great degree of automation, automatically proving intermediate steps. Larch Prover and Z/EVES are two examples of Proof Checkers that have been used with success for verifying circuits and algorithms.

A *Theorem Prover* can be seen as a Proof Checker with higher degree of automation. Ideally these tools should automatically exhibit a demonstration in any given logic. Unfortunately due to the complexity of this kind of procedure this is not the case. Theorem Provers provide less control over the proof, normally they request for user guidance although they are supposed to do most of the work, whereas in the Proof Checker case, the user should do the work and request for automation.

Some of the existing tools can be seen as hybrids that act both as Proof Checkers and Theorem Provers. Good examples of this are the tools ACL2 and PVS that are being used with success for instance in circuit verification.

Model-Theoretic Methods

Model-Theoretical techniques appeared out of several developments in the late 1970s and early 1980s. Two main factors seemed to have attracted the focus of researchers. Firstly, Pnueli proved that deciding the truth of a linear temporal formula over a finite structure was decidable [Pnueli 77]. Secondly, the realization that many concurrent programs could be seen as communicating finite-state machines. Together, these two factors lead to a new approach to software verification that, until then, was carried out by deductive techniques when possible, or with pencil and paper. This new approach is based on the construction of the program's state space followed by its exhaustive exploration checking for the validity of a formula that represents a property.

The program's state space can be roughly seen as the enumeration of all possible configurations of the set of variables of the program. Each configuration of program variables is called a *state vector*. A common representation of the system's behavior is taking each state vector as a node of a graph were each edge represents the possible changes of state form vectors to vectors. When the program's state space is finite we can

use a technique called Model-Checking [Clarke & al. 94]. Model-Checking is the task of checking for the validity of a formula φ of the specification S in a specific model.

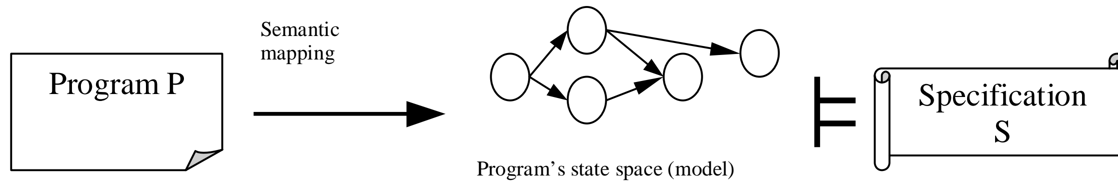


Figure 1.3 – Finite-State approach to software verification

In Figure 1.3, we can see a sketch of the Model-Checking procedure: Firstly, the program is translated through an appropriate semantic mapping to a graph that represents the behavior. Then, this graph is used as a model where formulas of specification S are to be interpreted.

Early applications of this technique (or variants of it) can be dated between 1978-1980 in the context of protocol validation [Brand & Joyner 78, Razouk & Estrin 80]. Shortly after in the early 1980s, Sifakis and his students started their work on the French validation system CESAR. Somewhat simultaneously, Clarke and Emerson introduced Temporal Logic Model-Checking algorithms as seen in [Clarke & Emerson 81] and [Clarke & Emerson 82] that lead to the development of the Extended Model Checker (or EMC) [Clarke & al. 86]. Holzmann in 1980 built the first general-purpose protocol analyser [Holzmann 81]. The work of Sifakis, Clarke and Emerson and Holzmann lead to three of the most popular verification tools today, respectively, CADP [Garavel 98], SMV [McMillan 92] and SPIN [Holzmann 91].

Limitations of using Formal Verification techniques are twofold. Firstly, the complexity involving the verification process can be exponential in the size of the program, making it difficult or even impossible to verify many real-world systems. Secondly, the use of verification tools still requires both familiarity with the problem domain and acquaintance with the verification procedures. Understanding how the user should interact with verification tools is currently a research topic, see e.g. [Valmari & Setälä 95] or [Stevens & Stirling 98].

Chapter 2 – State Transition Systems and Temporal Logic

The basis of Model-Theoretic verification is the creation of a *finite model* of the system to be verified. The whole idea is to have a model that captures the evolution of the system at some abstraction level and yet, simple and homogeneous enough to allow efficient reasoning.

Transitions Systems (or TS) have been proposed to serve as models of systems specifications that are constituted by states (represented as points) and transitions (represented as arcs connecting the points). Two major classes of transition systems can be found in literature: Those that attach information to states and those that attach information to actions, respectively named*, Kripke structures* and *Labeled Transition Systems* (or LTS).

These two classes of models reflect a distinction in specification languages. If the language is based on data transformations, like, for example the imperative language Pascal, then the models of specifications of that language are Kripke structures. One the other hand, we have languages that do not care much about data. Instead these languages are action (or event) oriented. The models of these languages are Labeled Transition Systems. For a more extensive treatment of TS refer to [Arnold 94].

Having our model in hand, we want to know whether it has certain properties. We are interested in analyzing the evolution of the system and verifying that it does not perform incorrect computations and it does not take defective sequences of actions.

The use of a specialized language called Temporal Logic (or TL) for specifying properties of computer programs, especially those that are concurrent and non-terminating, was proposed by A. Pnueli in the 1970s [Pnueli 77]. His ideas have been studied and extended by many researchers and today TL is an active area of research. This effort has lead to proposals and to effective use of TL on almost every aspect of concurrent programming, including design, specification, verification, composition and automatic synthesis. To support these developments a great deal of theoretical machinery was developed, leading among other results, to a variety of TLs.

To specify properties of reactive systems we need a logic that describes the relative ordering of events in time. An obvious way of doing this would be introducing special variables into a predicate logic to represent the instants at which the events occur. Instead, we present a propositional family of TLs that can be used to specify properties about state transition systems.

A multitude of TLs has been defined to specify and study the properties of concurrent and reactive systems. Attempting to give a comprehensive presentation of the existing temporal logic systems if far from the scope of this dissertation. Instead, we shall concentrate on those that are meaningful for subsequent chapters. Occasionally however,

an extension or some other modification to a TL system may be presented. For a more extensive treatment of the subject see [Emerson 91, Fagin & al. 95].

Using criteria similar to that of Transition Systems, TLs will be grouped in logics that express properties on states and on actions. The two groups of temporal logics are presented with a further distinction about linear-time and branching-time, preceded by a presentation of the underlying Modal Logic. Linear-time logics allow us to express properties concerning one individual execution path whereas Branching-time logics allow us to express properties concerning tree executions. Tree executions are found in transition systems where a state can possibly have several successor states

2.1 Transition systems

The transition systems that attach information to states are called *Kripke structures* and are commonly used to interpret temporal logics like LTL or CTL.

Definition 2.1 A Kripke structure (or KS) is a 4-tuple M = (S, AP, L, Σ) where,

- S is a set of states.
- AP is a finite, nonempty set of atomic propositions.
- The mapping $L: S \rightarrow 2^{AP}$ is called the *proposition labeling*.
- $\Sigma \subseteq S \times S$ is a transition relation. An element $(s_1, s_2) \in \Sigma$ is written as $s_1 \rightarrow s_2$.

In the literature, when referring to KSs, states are often designated as *worlds* and the transition relation Σ is often designated as *accessibility relation*. The set AP is omitted in where it can be taken from the context.

A second class of transition systems, called *Labeled Transition Systems* attach information to transitions instead of attaching information to states. Labeled Transition Systems are frequently used as interpretation models for action logics like Henessy-Milner logic [Hennessy & Milner 85] or ACTL [De Nicola & Vaandrager 91] and also serve as models for process algebras like CSP [Hoare 85] or CCS [Milner 89], and LOTOS [ISO 88].

Definition 2.2 A Labeled Transition System (or LTS) is a structure M = (S, A, Σ) where,

- S is a set of states.
- A is finite, non-empty set of actions.
- $\Sigma \subseteq S \times A \times S$ is the *transition relation.* An element $(s_1, a, s_2) \in \Sigma$ is called a *transition* and usually written as $s_1 \rightarrow^a s_2$. The transition $s_1 \rightarrow^a s_2$ means that, from state s_1, we can reach state s_2 by performing action a.

Definition 2.3 A transition relation Σ is said to be *nondeterministic,* if for a given action a, there exist states s_1, s_2 and s_3 such that $(s_1, a, s_2) \in \Sigma$ and $(s_1, a, s_3) \in \Sigma$ and $s_1 \neq s_2$. Otherwise, if for all states we have, $(s_1, a, s_2) \in \Sigma$ and $(s_1, a, s_3) \in \Sigma$ implies $s_2 = s_3$, the transition relation is said to be *deterministic*. Informally, a nondeterministic relation allows more than one labeled arrow $\rightarrow^a$ starting at a given state, while a deterministic relation allows only one.

Some more powerful temporal logics like the Mu-Calculus [Kozen 83] allow us to express properties both about states and actions. These logics are naturally interpreted over models that include information both about states and actions. A new version of transition system is presented that can be regarded as a combination of the two previously presented transition systems, KSs and LTSs.

Definition 2.4 A *Mixed Labeled Transition System* (or MLTS) is a 5-tuple M = (S, A, AP, L, Σ) where,

- S is set of states.
- A is a set of actions.
- AP is a finite, nonempty set of atomic propositions.
- The mapping $L: S \rightarrow 2^{AP}$ is called the *proposition labelling*.
- $\Sigma \subseteq S \times A \times S$ is a transition relation.

2.2 Modal Logic

The class of Modal Logics was originally developed by philosophers to study different "modes" of truth. We can give meaning to assertions of the form *necessary p* or *possibly p* (necessary and possibly are called *modalities*). Further, an assertion p may be true in the present moment and false in the next moment. The underlying framework used to capture these notions is the one of worlds connected by an accessibility relation. Together these can be represented by a Kripke structure. In each world w, a set of assertions holds, and given a certain accessibility relation R, a different set of assertions hold in each $w_i \in R(w)$. If, given a world w, an assertion p holds in every world $w_i \in R(w)$, then necessary p holds in world w. Dually, if given a world w, an assertion p holds for some world $w_i \in R(w)$, then possibly p holds in world w. Usually the notation used is □p (box p) for 'necessary p' and ◇p (diamond p) for 'possibly p'. These concepts are illustrated below for an accessibility relation $R = \{(w_0, w_1), (w_0, w_2)\}$.

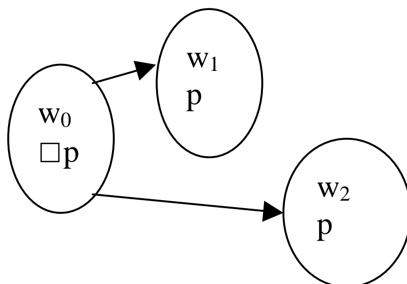


Figure 2.1 – A scenario where "box p" is satisfied.

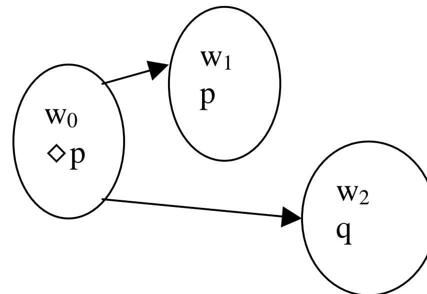


Figure 2.2 – A scenario where "diamond p" is satisfied.

In order to present a logic from a more formal point of view, one needs to present a language, rules to construct formulas using the language and some way to tell what is the meaning of a given formula (its semantics). Herein, we present the language of modal logic (as a set of symbols), the grammar that produces formulas (as a set of rules) and the mapping ⊨ that gives meaning to formulas within Kripke structures. A deeper insight on modal logic can be found in [Hughes & Cresswell 96].

Modal logic formulas are identified as follows:

Definition 2.5 The *set of formulas of modal logic* is recursively defined by the following grammar, where p range over propositional variables like p, q, r … and φ over modal formulas:

- $\varphi ::= p \mid \neg\varphi \mid \varphi_1 \Rightarrow \varphi_2 \mid \Box\varphi$.

The absence of $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\Diamond\varphi$ in the grammar definition is justified by the following definitions:

- $\varphi_1 \wedge \varphi_2 \equiv \neg(\varphi_1 \Rightarrow \neg\varphi_2)$
- $(\varphi_1 \vee \varphi_2) \equiv \neg\varphi_1 \Rightarrow \varphi_2$
- $\Diamond\varphi \equiv \neg\Box\neg\varphi$

In subsequent sections these simplifications to grammar presentations will be introduced and explained when necessary.

Definition 2.6 We define the *semantics of a modal logic formula* φ with respect to a Kripke structure (a model) M=(S, AP, L, Σ) inductively in the structure of φ. For a given state s, $\models$ is defined as a relation satisfying the following conditions:

- $s_0 \models p$ iff $p \in L(s_0)$, for a propositional symbol φ;
- $s \models \neg\varphi$ iff $s \not\models \varphi$;
- $s \models \varphi_1 \Rightarrow \varphi_2$ iff $s \models \varphi_1$ implies $s \models \varphi_2$;
- $s \models \Box\varphi$ iff for every $s' \in S$ such that $(s,s') \in \Sigma$, $s' \models \varphi$;

2.3 Linear temporal logic

Temporal logic (or TL) can be seen as a specialization of modal logic in which the accessibility relation is established between time points. The operator *necessary* is called *always* and the operator *possibly* is called *sometimes*. In *Linear Temporal Logic* (or LTL), each time point has only one successor.

Before we formally present the syntax of LTL we should consider the availability of diverse notations concerning temporal operators. The equivalence between two of the most common notations is shown below in Table 2.1.

Definition 2.7 The set of *LTL formulas* is defined through the following grammar, where p is a propositional variable and φ is path formula:

- $\varphi ::= p \mid \neg\varphi \mid \varphi \Rightarrow \varphi \mid X\varphi \mid \varphi U \varphi$

		Possible readings
$\Box\varphi$	$G\varphi$	Always φ; Globally φ
$\Diamond\varphi$	$F\varphi$	Eventually φ; Sometime in the future φ
$\circ\varphi$	$X\varphi$	Nexttime φ
$\varphi U\psi$	$\varphi U\psi$	φ Until ψ

Table 2.1

The operators G and F are defined through the following definitions:

- $F\varphi \equiv trueU\varphi$
- $G\varphi \equiv \neg F \neg\varphi$

Example 2.8 The formula $GFp \vee FGp$ means that either we have p infinitely often or from some point on we have always p.

The reader can draw some intuition on the semantics of linear temporal operators introduced above by looking at Figure 2.3.

Definition 2.9 Given a Kripke structure M=(S, AP, L, Σ), we define *path* as an infinite sequence $\pi=(s_0, s_1, \ldots)$ of states such that $\forall i, (s_i,s_{i+1})\in\Sigma$.

Definition 2.10 We define *suffix of a path* $\pi=(s_0, s_1, \ldots, s_j, \ldots)$, as being the path $\pi' = (s_j, s_{j+1}, \ldots)$, symbolically : π_j.

Definition 2.11 The i-th state of a path $\pi=(s_0, s_1, \ldots)$ is denoted by $\pi(i)$.

Definition 2.12 We define the *semantics of a LTL formula* φ with respect to a Kripke structure M=(S, AP, L, Σ) inductively in the structure of φ. We write $M, \pi \vDash\varphi$ to mean that: "in structure M, the formula φ holds in path π". When the structure M is understood, we write $\pi\vDash\varphi$.

- $\pi\vDash p$ iff $p\in L(\pi(0))$, for a propositional symbol p;
- $\pi\vDash\neg\varphi$ iff $\pi\nvDash\varphi$;
- $\pi\vDash\varphi_1\Rightarrow\varphi_2$ iff $\pi\vDash\varphi_1$ implies $\pi\vDash\varphi_2$;
- $\pi\vDash G\varphi$ iff for every $i\in\mathbb{N}$, $\pi_i\vDash\varphi$;
- $\pi\vDash F\varphi$ iff exits $i\in\mathbb{N}$ such that $\pi_i\vDash\varphi$;
- $\pi\vDash X\varphi$ iff $\pi_1\vDash\varphi$;
- $\pi\vDash\varphi_1 U\varphi_2$ iff exists j such that ($\pi_j\vDash\varphi_2$ and for every $k<j$ $\pi_k\vDash\varphi_1$).

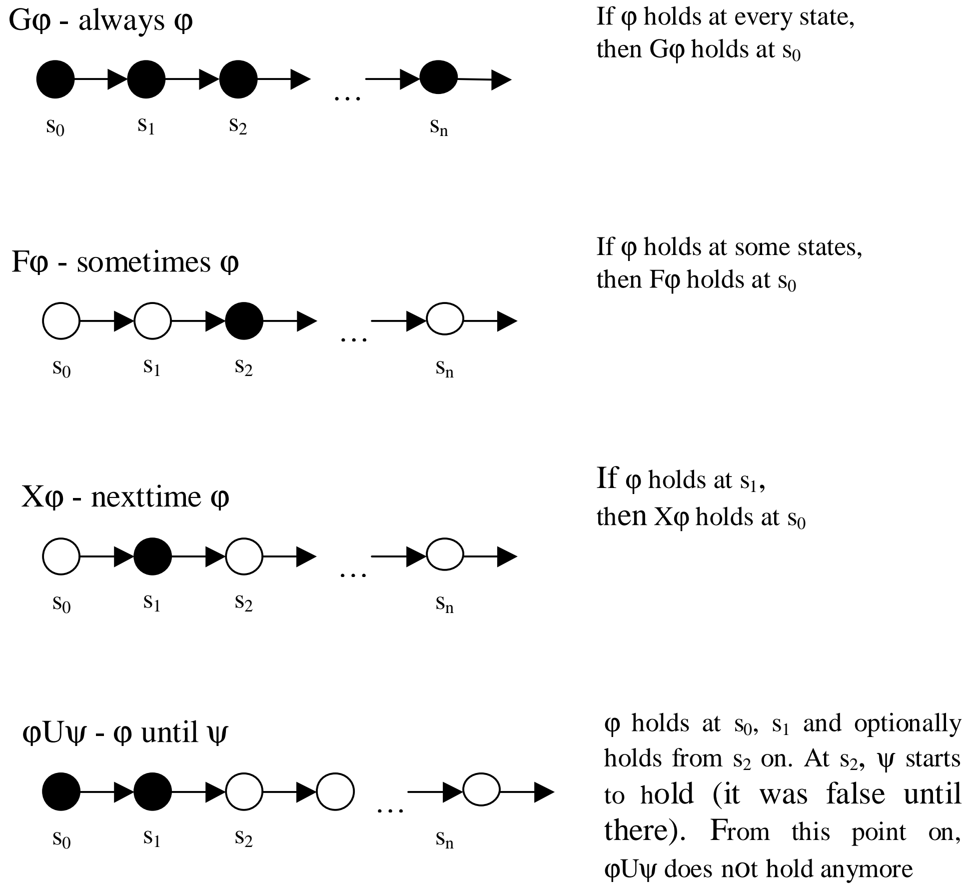


Figure 2.3 – An intuition of semantics of the LTL operators.

The temporal operators 'F' and 'G' are obtainable from the operator 'U' as presented in Definition 2.7.

2.3 Branching time logics

In *branching-time temporal logics* (or BTL) the underlying structure of time is a tree instead of a linear sequence as in linear time logic. This means that we now allow more than one possible future. In these trees we allow a node to have infinitely many successors while requiring each node to have at least one successor. Branching over this tree produces *paths* isomorphic to $\mathbb{N}$.

Computational tree logic (or CTL) defined by Clarke and Emerson [Clarke & Emerson 81] constitutes one of the more representative systems within BTL systems. Syntactically, it can be seen as an addition of two extra operators to LTL, A ("for all futures") and E ("for some future") that are used to quantify over branches of a tree. We present two logics CTL and CTL*. The latter is more expressive than the former as we will see. Informally, CTL*, which is sometimes referred as *full branching-time logic*, extends CTL by allowing Boolean combinations and nestings of linear-time operators.

The syntax of CTL may be obtained by combining the path quantifier operators A and E to the existing X and U operators.

Definition 2.13 The *syntax of CTL formulas* is defined by the following grammar, where φ, φ_1 and φ_2 are state formulas, and ψ is a path formula:

- $\varphi ::= p \mid \neg\varphi \mid \varphi_1 \Rightarrow \varphi_2 \mid E\psi \mid A\psi$;
- $\psi ::= X\varphi \mid \varphi_1 U \varphi_2$.

CTL* extends CTL by allowing Boolean combinations and nesting of the linear operators X and U.

Definition 2.14 The *syntax of CTL* formulas* is defined by the following grammar where also ψ_1 and ψ_2 are path formulas:

- $\varphi ::= p \mid \neg\varphi \mid \varphi_1 \Rightarrow \varphi_2 \mid E\psi$;
- $\psi ::= \varphi \mid \neg\psi \mid \psi_1 \Rightarrow \psi_2 \mid X\psi \mid \psi_1 U \psi_2$.

The following definition is introduced

- $A\psi \equiv \neg E \neg \psi$.

Also, the following abbreviations are common in literature:

- $F^{\infty}\ \varphi \equiv GF\varphi$.
- $G^{\infty}\ \varphi \equiv FG\varphi$.

The semantics of CTL and CTL* formulas is given as follows:

Definition 2.15 We define the *semantics of a CTL formula* φ with respect to a Kripke structure $M=(S, AP, L, \Sigma)$ inductively in the structure of φ. We write $M,s \models \varphi$ to mean that: "in structure M, the formula φ holds in state s" and $M,\pi \models \varphi$ to mean that: "in structure M, the formula φ holds in path π". Since M is obvious in this context it will be omitted. Let s_0 be the initial state in the computation tree:

- $s_0 \models p$ iff $p \in L(s_0)$, for a propositional symbol p;
- $s_0 \models \neg\varphi$ iff $s_0 \not\models \varphi$;
- $s_0 \models \varphi_1 \Rightarrow \varphi_2$ iff $s \models \varphi_1$ implies $s \models \varphi_2$;
- $s_0 \models E\psi$ iff exists path $\pi=(s_0, s_1, \ldots)$ such that $\pi \models \psi$ in M;
- $s_0 \models A\psi$ iff for every path $\pi=(s_0, s_1, \ldots)$ such that $\pi \models \psi$ in M;
- $\pi \models X\varphi$ iff $\pi_1 \models \varphi$;
- $\pi \models \varphi_1 U \varphi_2$ iff exists j such that ($\pi_j \models \varphi_2$ and for every $k<j$, $\pi_k \models \varphi_1$).

The semantics of operators F and G is given below:

- $\pi \models G\varphi$ iff for every $i \in \mathbb{N}$, $\pi_i \models \psi$;
- $\pi \models F\varphi$ iff exists $i \in \mathbb{N}$ such that $\pi_i \models \varphi$;

Definition 2.16 We define the *semantics of a CTL* formula* φ with respect to a Kripke structure $M= (S, AP, L, \Sigma)$ inductively in the structure of φ. We write $M,s \models \varphi$ to mean that: "in structure M, the formula φ holds in state s" and $M, \pi \models \varphi$ to mean that: "in

structure M, the formula φ holds in path π". Since M is obvious in this context it will be omitted. Let s_0 be the initial state in the computation tree:

- $s_0 \vDash \varphi$ iff $\varphi \in L(s_0)$, for a propositional symbol φ;
- $s_0 \vDash \neg\varphi$ iff $s_0 \nvDash \varphi$;
- $s_0 \vDash \varphi_1 \Rightarrow \varphi_2$ iff $\pi \vDash \varphi_1$ implies $\pi \vDash \varphi_2$;
- $s_0 \vDash E\varphi$ iff exists a path $\pi=(s_0, s_1, \ldots)$ such that $\pi \vDash \varphi$ in M;
- $s_0 \vDash A\varphi$ iff for every path $\pi=(s_0, s_1, \ldots)$, $\pi \vDash \varphi$ in M;
- $\pi \vDash \varphi$ iff $\pi=(s_0, s_1, \ldots)$ and $s_0 \vDash \varphi$;
- $\pi \vDash \neg\psi$ iff $\pi \nvDash \psi$;
- $\pi \vDash \psi_1 \Rightarrow \psi_2$ iff $\pi \vDash \psi_1$ implies $\pi \vDash \psi_2$
- $\pi \vDash X\psi$ iff $\pi_1 \vDash \psi$;
- $\pi \vDash \psi_1 U \psi_2$ iff exists j such that ($\pi_j \vDash \psi_2$ and for every $k<j$, $\pi_k \vDash \psi_1$).

Sometimes we may wish to restrict the logics CTL and CTL* so that they cannot express the existence of paths in a Kripke structure. Technically this can be achieved by restricting the use the existential path quantifier 'E'. This leads to logics where one can only quantify universally over paths. One last remark must be made to possibility of nesting universal quantifiers. We require formulas to be expressed in *positive normal form* to avoid the appearance of existential quantifiers (through negation of universal quantifiers).

Definition 2.17 A formula φ is said to be in the *positive normal form* if no universal quantifier falls in the scope of a negation operator.

The logics obtained by this process are called *Universal CTL* (or ∀CTL) and *Universal CTL** (or ∀CTL*).

A comparative analysis of branching time versus linear time logics can be given now, considering three different aspects:

- Expressiveness;
- Complexity;
- Suitability for behavior specification.

Expressiveness. Among the criteria for choosing some specification language, expressiveness is central. By expressiveness we should understand the capacity to describe certain classes of properties. The more classes of properties a language is able to describe, the more expressive it is. From this viewpoint CTL and LTL are not directly comparable. That is, each of them can define properties that the other can not. There are classes of properties expressible in either language. In CTL we have quantification over paths, this feature is absent in LTL and because of this we are unable to express the notions of *potentiality* and *inevitability* in LTL. The notion of potentiality is obtained through $EF\varphi$ formulas, meaning that, there is a path where φ holds at some point. The notion of inevitability is obtained through $AF\varphi$ formulas, meaning that, in every path, φ

holds at some point. Conversely, and due to the syntactic restrictions imposed by CTL, i.e., we cannot nest linear temporal operators, we are unable to express the class of *fairness* properties. The Fairness properties include the notions of *almost everywhere* and *almost always*. Almost everywhere can be expressed in LTL by a formula of the form FGφ, which means that, from some point on, we always verify φ. Whereas, almost always, can be expressed by a formula of the form GFφ, that means that, in all instants we know that φ will hold. The logic CTL* subsumes both LTL and CTL allowing to express all the above mentioned classes of properties.

Complexity. The CTL logic is less complex than LTL with respect to the time required for evaluating a formula φ over a model M. Classical algorithms based on Büchi automata are linear both on formula and model sizes for CTL. Whereas for LTL, automata algorithms are exponential in the formula's size and linear in the model's size. In the case of CTL*, the complexity is the same than that of LTL.

Suitability for behavior specification. This point has much to do with Expressiveness. In LTL we can only express properties about individual computation paths. In CTL, however, we can quantify over paths. For instance in LTL we can not distinguish the behavior of a.(b[]c) and a.b[]a.c. This is, we can not write a property in LTL that will hold in one of a.(b[]c) and a.b[]a.c and not hold in the other.

2.4 Hennessy-Milner Logic

Action based logics have been introduced as formalisms to characterise properties of programs written in process algebras. *Hennessy-Milner* logic (or HML) [Hennessy & Milner 85] is considered a standard representative of this class of logics. Its syntax and semantics are given below.

Definition 2.18 The *syntax of HML formulas* is defined by the following grammar, where $a \in A$ represents an action, α, α_1 and α_2 are action formulas and φ, φ1 and φ2 are state formulas.

- $\alpha ::= tt \mid a \mid \neg\alpha \mid \alpha_1 \Rightarrow \alpha_2$;
- $\varphi ::= tt \mid \neg\varphi \mid \varphi_1 \Rightarrow \varphi_2 \mid [\alpha]\varphi$.

The symbols tt and ff are abbreviations for true and false respectively. For succinctness the following definitions were used:

- $ff \equiv \neg tt$
- $\langle\alpha\rangle \equiv \neg [\alpha]\neg\varphi$.

Definition 2.19 We define the *semantics of a HML formula* φ with respect to LTS M= (S, A, Σ) inductively in the structure of φ where $F \subseteq S$ is a set of sates and $E \subseteq A$ is a set of actions in the following way:

- $E \vDash tt$ iff for every $a \in A$, $a \in E$;
- $E \vDash a$ iff $a \in E$;
- $E \vDash \neg\alpha$ iff $A \backslash E \vDash \alpha$;

- $E \models \alpha_1 \Rightarrow \alpha_2$ iff $E \models \alpha_1$ implies $E \models \alpha_2$;
- $F \models tt$ iff for every $s \in S$, $s \in F$;
- $F \models \neg\varphi$ iff $S \backslash F \models \varphi$;
- $F \models \varphi_1 \Rightarrow \varphi_2$ iff $F \models \varphi_1$ implies $F \models \varphi_2$;
- $F \models [\alpha]\varphi$ iff $F = \{s_1 : s_1 \rightarrow^a s_2$ for every $a \in E'$, for every $s_2 \in F'$ where $E' \models \alpha$ and $F' \models \varphi\}$;

Example 2.20 In HML we can express deadlock freedom with the following property

- $\neg$ ([tt]ff).

This means that there are no states without successor states.

2.5 Mu-Calculus

The *Mu-calculus* [Kozen 83] appears as an extension of PDL and HML. Hennessy-Milner logic only allows expressing global properties and is too weak from a practical point of view. Extending its assertions with recursive definitions constitutes a simple way of greatly increasing its expressiveness.

Definition 2.21 The *syntax of Mu-Calculus formulas* is given by the following grammar where $a \in A$ represents an action, $p \in AP$ is an atomic proposition, Y is a propositional variable, α, α_1, α_2 are action formulas and φ, φ_1 and φ_2 are state formulas.

- $\alpha ::= tt \mid a \mid \neg\alpha \mid \alpha_1 \Rightarrow \alpha_2$;
- $\varphi ::= tt \mid p \mid Y \mid \neg\varphi \mid \varphi_1 \Rightarrow \varphi_2 \mid [\alpha]\varphi \mid \mu Y.\varphi$.

The greatest fixed point operator ν is defined as:

- $\nu Y.\varphi \equiv \neg\mu Y.\ \varphi[\neg Y/Y]$.

The definition of the Mu-Calculus semantics needs some further results. We can define its semantics over MLTS, but if we restrict ourselves to a subset of the Mu-Calculus a simpler structure is sufficient. If we skip actions, we can present the Mu-Calculus semantics using a Kripke structure, whereas if we skip atomic propositions (i.e. information on states), it is possible to define the Mu-Calculus semantics over a LTS.

Definition 2.22 A variable Y is said to be *bound* in formula φ if there is a sub-formula of φ of the form $\mu Y.\varphi'$ and Y occurs in φ'. A variable Y that is not bound in φ is said to be *free*.

Definition 2.23 A formula φ is said to *syntactically monotonic* iff every *bound* variable of φ is placed under an even number of negations.

Definition 2.24 An *environment* is a partial function ρ: Var $\rightarrow 2^S$, that associates a set of states to each propositional variable Y.

Definition 2.25 We define the *Mu-Calculus semantics of a syntactically monotonic formula* φ with respect to a MLTS, M = (S, A, L, Σ) and environment ρ, inductively in the structure of φ, where E is a set of actions and F a set of states, in the following way:

- $E \vDash_\rho tt$ iff for every $a \in A$, $a \in E$;
- $E \vDash_\rho a$ iff $a \in E$;
- $E \vDash_\rho \neg\alpha$ iff $A \setminus E \vDash_\rho \alpha$;
- $E \vDash_\rho \alpha_1 \Rightarrow \alpha_2$ iff $E \vDash_\rho \alpha_1$ implies $E \vDash_\rho \alpha_2$;
- $F \vDash_\rho tt$ iff for every $s \in S$, $s \in F$;
- $F \vDash_\rho p$ iff for every $s \in F$, $s \in L(p)$;
- $F \vDash_\rho Y$ iif $F = \rho(Y)$;
- $F \vDash_\rho \neg\varphi$ iff $S \setminus F \vDash_\rho \varphi$;
- $F \vDash_\rho \varphi_1 \Rightarrow \varphi_2$ iff $F \vDash_\rho \varphi_1$ implies $F \vDash_\rho \varphi_2$;
- $F \vDash_\rho [\alpha]\varphi$ iff $F = \{s_1 : s_1 \rightarrow^a s_2$ for for every $a \in A'$, for every $s_2 \in S'$ where $A' \vDash_\rho \alpha$ and $S' \vDash_\rho \varphi\}$;
- $F \vDash_\rho \mu Y.\varphi$ iff $\cap\{S' \subseteq S : S'' \vDash_\rho \varphi$ and $S'' \subseteq S'\}$.

The Mu-calculus is an expressive formalism, allowing the translation of other TLs. Historically, the Mu-Calculus as been used as a kind of "Assembly Language" where CTL formulas were to be translated. Some verification tools like SMV take advantage of these translations as explained in [McMillan 93]. A translation for CTL operators is given in Table 2.2.

$EX\varphi$	$\langle tt\rangle\varphi$
$AX\varphi$	$\langle tt\rangle tt \wedge [tt]\varphi$
$E[\varphi_1 U \varphi_2]$	$\mu Y.(\varphi_2 \vee (\varphi_1 \wedge \langle tt\rangle Y))$
$A[\varphi_1 U \varphi_2]$	$\mu Y.(\varphi_2 \vee (\varphi_1 \wedge \langle tt\rangle tt \wedge [tt]Y))$

Table 2.2 – Translation of CTL operators into the Mu-Calculus.

2.6 Specification of properties using temporal logic

Experiments with program verification point to the use of about 70% of formulas of the form $AG\neg\varphi$. These temporal logic formulas are called *safety formulas* for they assert about the system safety, this is, that the system never enters into a bad condition specified by φ.

As with safety properties, we can also classify other kinds of properties, which assert about the program good operation like *response properties.*

The hierarchy of temporal formulas can be displayed in a diagram like that of Figure 4.1. This diagram can be found in [Manna & Pnueli 95]. Each node displays a canonical formula of a certain kind (*reactivity*, *response*, *persistence*, *obligation*, *safety* and *guarantee*). Edges define an inclusion relation between classes of properties specifiable by formulas of a certain kind. For instance, the class of safety formulas is included in the class of obligation formulas. Also a persistence formula is a reactivity formula. Any formula φ is said to be a k-formula, where k ranges over the set of kinds defined above, if it is equivalent to a canonical k-formula. Correspondingly, a k-property is a property specifiable through a k-formula.

Since we are working with propositional temporal logic, we can attach useful propositions to certain distinguished states like, for example, *init* or *terminate*. The following three types of propositions are often used in a concurrent system framework:

- at_*l*
- *enabled_p*$_i$
- *terminate*

The *at_l* proposition holds in sates where the program is at *control location* l. Locations can be simply understood as labels in the program about which we want to write our properties. Technically, one can add to the program specification a control variable whose domains are the location labels. The *enabled_p*$_i$ proposition is used to specify the fact that the process p_i is ready to make a transition. The *terminate* proposition will hold in a location where the program the program terminates.

Definition 2.26 A program P is said to have (or enjoy) a property φ if all computations of models of P are models of φ.

Safety properties

A safety property asserts that "nothing bad happens". We illustrate this intuition by pointing two of the most important safety properties of a concurrent system: *mutual exclusion* and *deadlock freedom.*

Definition 2.27 Given a program P with n processes a *mutual exclusion formula* is formula of the form $G\neg(at_cs_1 \wedge \ldots \wedge at_cs_n)$ where for $i\in\{1...n\}$, at_cs_i are predicates meaning that the process p_i is in the critical section.

A mutual exclusion formula states that, in a program P, its constituent processes can never be at their critical sections simultaneously. Note that, in this rule the "nothing bad" concept is "never at critical section simultaneously".

Definition 2.28 Given a program P with n processes a *deadlock freedom formula* is formula of the form G($enabled_p_1 \vee \ldots \vee enabled_p_n$) where for $i \in \{1...n\}$, $enabled_p_i$ are predicates meaning that the process p_i is enabled.

The deadlock freedom formula encodes the fact the program P can always run, is this context some process p_i is always enabled. In other words, it is never the case that all processes of P are disabled (meaning that we can there are no further transitions). For a deadlock freedom formula, the "nothing bad" fact is "never disabled".

The safety formulas presented above, mutual exclusion and deadlock freedom, represent special cases of *invariance* properties. Safety properties can be divided in *local invariance* and *global invariance* properties.

Definition 2.29 Given a program P, a *local invariance formula* is a formula of the form $G(at_l \Rightarrow \varphi)$.

Definition 2.30 Given a program P, a *global invariance formula* is a formula of the form $G\varphi$.

Local invariance formulas mean that a property must hold when the program is at some location. Whereas, global invariance formulas express properties that should hold no matter where the program's control is.

Guarantee properties

A guarantee property formula is specified as $F\varphi$. This formula states that at some point the program will meet a state where φ holds.

Definition 2.31 Given a program P, a *guarantee formula* is a formula of the form $F\varphi$.

The intuition behind guarantee formulas is that, a program that enjoys a formula $F\varphi$, guarantees that φ holds at least once. If, for instance, φ is related with an event, $F\varphi$ sates that the event must occur but guarantees no repetitions of it. Therefore, guarantee formulas may be used to specify events that occur only once in the program's lifetime, such as termination.

Definition 2.32 Given a program P, and *terminate* a predicate identifying the program termination states. A formula F*terminate*, is a *termination formula.*

Obligation properties

Some properties can not be expressed by safety or guarantee formulas alone. Obligation formulas take the form of conjunctions of pieces like: $G\varphi_1 \vee F\varphi_2$, because of the combination of safety and guarantee formulas. Noting that $G\varphi_1 \vee F\varphi_2 \equiv \neg F\varphi_1 \vee F\varphi_2$, we can write $F\varphi_1 \Rightarrow F\varphi_2$. This states that, if at some point during the program's execution φ_1 has become true, then at some point, φ_2 has become true or will certainly become true.

Definition 2.33 Given a program P, an *obligation formula* is a formula of the form $\bigwedge_{i \in \{1..n\}} [G\varphi_i \vee F\psi_i]$, where φ_i and ψ_i are also formulas.

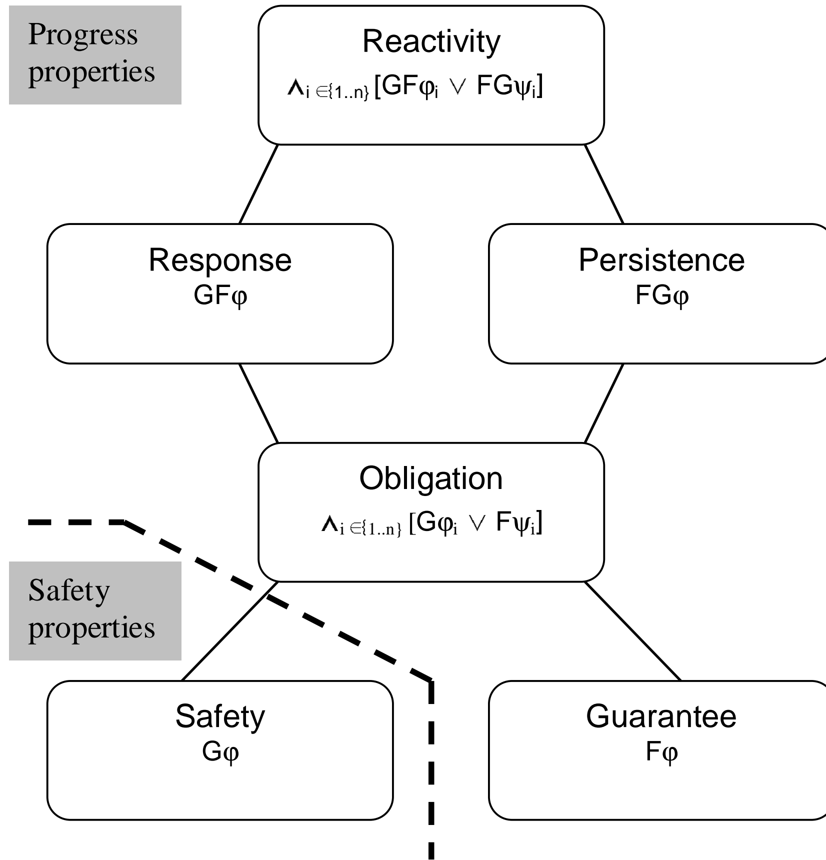


Figure 2.4 – Hierarchy of temporal formulas proposed in [Manna & Pnueli 95].

Example 2.34 Lets suppose that, in the context of some program we want to ensure the following relation between two variables x and y: $F(x=1) \Rightarrow F(y=1)$. This means that if x=1 sometime during the execution of the program, then, also y=1 holds during the program's execution. One may wrongly get the feeling that some ordering exists between the assignments of x and y, in the sense that firstly x=1 and only after y=1. In fact this is not the case, we are not stating with the above formula that: y=1 is a response to x=1, this fact is captured by a different formula like $F((x=1) \Rightarrow F(y=1)$.

Response properties

A response property asserts about the system responsiveness. System responsiveness is the ability of a system (represented as a computer program) to give a response to a stimulus (event).

Definition 2.35 A *response formula* is a formula reducible to a canonical formula $GF\varphi$.

In the formula $GF\varphi$ we are stating that φ holds infinitely many times. A rather intuitive way of thinking about response formulas is to consider the equivalent of the canonical response formula: $G(\varphi_1 \Rightarrow F\varphi_2)$ also known as the *temporal implication formula*.

Example 2.36 Lets consider the case where we have a program P that receives a reset signal and resets its internal state by setting the variables x=0 and y=0. The formula that captures the fact that "whenever P sees a reset signal, P resets its internal state" is $G((reset{=}true) \Rightarrow F(x{=}0 \wedge y{=}0))$.

Persistence properties

Persistence formulas are used to sate properties about systems stabilization. In other words, beyond some point the system enters in some defined situation and stays like that forever.

Definition 2.37 A *persistence formula* is a formula reducible to a canonical formula $FG\varphi$.

We can look at a response formula in an equivalent way using the formula $\varphi_1 \Rightarrow FG\varphi_2$.

Reactivity properties

A reactivity formula can be possibly seen as a more elaborate response formula. In a response formula, we were guaranteeing response to only one stimulus. However, with reactivity formulas, we want to ensure infinitely many responses but only when we have infinitely many stimuli.

Definition 2.38 A *reactivity formula* is a formula reducible to a canonical formula $FG\varphi_1 \vee GF\varphi_2$.

We have presented a safety/progress classification of system properties. However, another classification is commonly found in literature, the *safety* (or *invariance*) properties versus the *liveness* (*eventuality* or *progress*) properties. Liveness properties are informally associated to the sentence "something good may happen". A liveness property asserts about the eventual behavior of the system. Formally we can define safety and liveness properties as follows:

Definition 2.39 We say that φ is a *safety formula* iff any infinite sequence σ that violates φ, i.e. $\sigma \vDash \neg\varphi$, contains a finite prefix $\sigma' = \sigma[0,k]$ where all extensions of σ' violate φ.

Definition 2.40 We say that φ is a *liveness formula* iff any finite sequence σ can be extended to an infinite sequence σ' satisfying φ.

Although $G\varphi$ captures the set of safety formulas, $F\varphi$ does not capture all liveness formulas. In fact, $F\varphi$ is a *live-guarantee* formula, there are also *live-obligation*, *live-response*, *live-persistence* and *live-reactivity* formulas [Manna & Pnueli 92]. A live-k formula is a liveness formula that is also a k-formula. This observation motivates the following claim:

Claim 2.41 The class of liveness formulas is contained in the class of progress formulas.

Reachability properties

Many interesting properties of a system can be stated as the *reachability* of a given set of states. Examples of reachability properties are the safety properties: 'A system is safe if it never reaches undesired states'. This property is formalised in the form $init \Rightarrow \neg \mathrm{EF} unsafe$ or equivalently $init \Rightarrow \mathrm{AG} safe$, where *init*, *safe* and *unsafe* are propositions denoting sets of states.

Chapter 3 – Model-Checking

The Model-Checking process (or MC) is carried out in two phases. The first one consists in obtaining the model of the system, given its specification. The second one explores the model, checking for conformance with some previously specified property. Efficient MC algorithms (linear in the size of the state-space) for finite-state systems have been obtained for a number of logics. The problem of these algorithms appears when applying MC even to moderate-size systems: We often witness a combinatorial explosion of the state-space.

This chapter analyzes the limitations and virtues of MC tools. We start by detailing the main phases of MC pointing out the possible sources for the state-explosion problem. In order to cope with this problem, a number of optimization techniques and algorithms are discussed. A classification of different kinds of systems is also presented in an effort to understand what systems are less likely to be verified via MC. The chapter ends with a survey on the features of some of the most popular MC tools.

3.1 Process overview

Along its life, a system receives stimuli, computes and acts (produces stimuli). We are interested in verifying if a description of a system performs correct computations and takes the right actions when stimulated in a specific manner. A way to verify the correctness of a system's specification is enumerating all possible behaviors of the system and check their correctness. For example, the fragment *if c then B_1 else if d then B_2 else B_3* has tree different behaviors B_1, B_2 and B_3 depending on the values of *c* and *d*. The evolution of the system (the next state from the current state) depends on the values of the variables at that state[4]: in the example above, the state is consists of variables 'c' and 'd' that determine whether B_1, B_2 or B_3 are executed, thus leading to a different state.

The question is, given a state, how do we compute the next state[5]? A primitive solution could be providing an exhaustive enumeration of current-state/next-state relation pairs. In practice however, this relation is induced from a set of rules that usually constitute an *operational semantics* of the system specification language also describing in a accurate way what happens to a state when a command is executed on it.

Starting from an initial state and using rules as described above, we can compute all the reachable states of a system specification. Looking at states as vectors of variables, the reachable state-space of a system is a subset of the vector space generated by all the possible valuations of the variables in their respective domains.

---

[4] It could be claimed this sentence is imprecise. What happens is that, in some languages, next-states do not depend on values of variables but on stimuli from the environment. Process algebras are good examples.

[5] Possibly a state may lead to more than one next-state.

Let us suppose that we are interested in verifying if in system S, is always the case that $x \geq 0$. This is the same as checking if in every reachable state s of S, $s \models x \geq 0$. Properties about the system behavior can be written as properties of its reachable state space. In the previous chapter we introduced mathematical frameworks both for representing state spaces (Transition Systems) and for writing properties about them (Temporal Logics). Temporal Logics can be used to write properties that concern the relative ordering of events in time.

Below, in Figure 3.1, we present a sketch of the MC process. In it, the system specification is translated into a graph representing its reachable state-space, which is checked against a formula representing a desired property of the system. The *Model-Checking Engine* (Checker) can be regarded as a form of search algorithm for graphs, it stops yielding a *yes* or a *no*, meaning that the model satisfies the formula or not. Many of these Checkers also provide a trace that can be used for debugging purposes if the model does not satisfy the property formula. Technically this trace corresponds to a dump of the stack of the graph search algorithm.

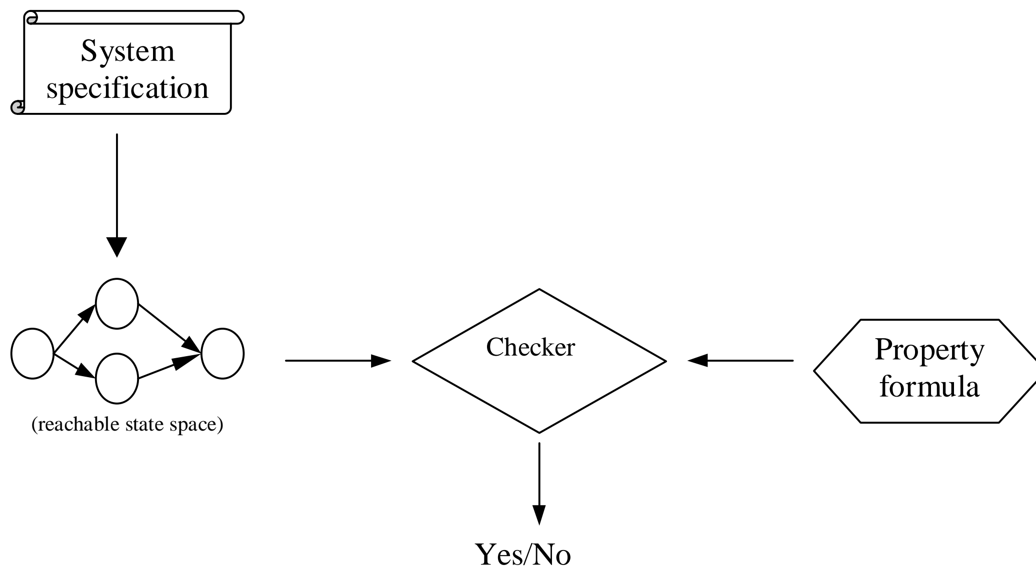


Figure 3.1 – Sketch of the Model Checking verification procedure.

3.2 State space generation

One of the underlying principles of the MC process is the generation of reachable states of a system. This can be done using a set of rules. These rules determine the flow operation of the program under certain valuations of its variables or reactions to external events. Starting from an initial state, and a program, applying the rules will produce new states and subprograms. Applying again rules to subprograms and new states we obtain more states and subprograms and so on. In the example of Figure 3.2, a very simple CSP process that either performs infinitely many ‘a’ or performs finitely many ‘a’ and a ‘b’, a ‘c’ and exits is presented with the corresponding state space.

(1) *test* $=_{def}$ *(a. test) [] (b. c.* ***exit****)*

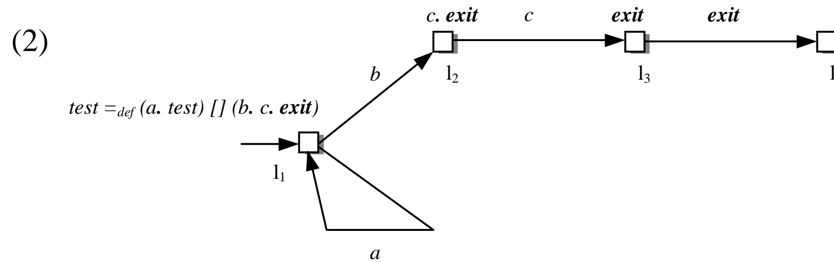


Figure 3.2 – A very simple CSP process and the corresponding state-space. (1) Process definition. (2) The process state space.

The reachable state-space relation of a program can be formalized as a Kripke structure or a LTS on which properties can be evaluated. For the above example we have a LTS like M=(S, A, Σ) where, S={l_1, l_2, l_3, l_4}, A={a, b, c, exit} and Σ={(l_1,a,l_1), (l_1,b,l_2), (l_2,c,l_3), (l_3,exit,l_4)}.

Generating state-spaces exhaustively in a naïve way as presented above would be impractical. For this reason, several techniques are employed. These techniques rely on the use of intermediate representations like *behavior expression graphs* [Savola 95], *Petri-Nets* [Garavel 89] or BDDs [Hu 95]. The use of these intermediate representations allows the optimization of the state-space through the employment of graph manipulation operations. More information about *state space generation*, can be found in [Valmari 88, Krimm & Mounier 94, Kokkarinen 98].

The state explosion problem arises during the process of state-space generation. It may have several origins. For example the number of parallel processes greatly influences the size of the resulting state space. In a concurrent system the state-space is known to be exponential in the number of parallel components. Two more sources of the state explosion problem are the increase of external non-determinism through the use variables representing inputs from the environment (which greatly increases the number of successor states for each state) and the use of *unbounded recursion*, where a process calls itself without performing any stop test.

3.3 Model-Checking approaches

Model-Checking techniques can be classified into two approaches: *Temporal Logic Model-Checking* and *Language Containment Checking*. Temporal Logic Model-Checking was pioneered by Clarke and Emerson [Clarke & Emerson 82]. Their algorithm was polynomial in the size of the model (determined by the size of the program) and in the size of the temporal logic formula. The algorithm consisted of a strategy to perform the interpretation of the Temporal Logic formula by unfolding it over the underlying model. Formally the strategy was based in *semantic tableau* which is a tree-like top-down

proof constructed by successive instantiations of *semantic proof rules*. The proof rules are presented with conclusions written below premisses. We call *goal* to a conclusion and *sub-goals* to the premisses. References to this divide-and-conquer approach also called *truth simplification* can be found in the work of Cleaveland and Stirling about MC of the modal Mu-calculus [Cleaveland 90, Stirling 91]. In their book, Manna and Pnueli present an algorithm for checking the satisfiability of Temporal Logic formulas [Manna and Pnueli 95] also based in semantic tableau.

We can characterize the behavior of a program by the sequences of actions it performs. When a program is running, actions are performed in some ordered way producing a language that may be recognized by an automaton. In fact many specification languages are based on automata theory and programs are automata descriptions at some level of abstraction. Moreover, many languages can be given semantics in terms of automata.

In this setting, verifying that a specification behaves correctly is the same as ensuring that the language recognized by the corresponding automaton is correct. For example, in order to verify that a program P never performs action 'a' one could check that $a \notin L(A_p)$ where A_p is the automaton representing the behavior of P.

Realistic systems are often accompanied by elaborated specifications that request for *language containment checking* instead of simple word containment checking. We are usually interested in ensuring properties that are modeled as cyclic patterns of behavior like safety properties. A straightforward verification technique was proposed based on the idea of language containment checking: Given a program P and a property φ, checking that P enjoys property φ is the same as checking that $L(A_p) \subseteq L(A_\varphi)$, where A_p and A_φ are the automaton representations of P and φ respectively.

For a more extensive treatment of the automata theoretical approach to verification refer to [Kurshan 94].

The Algorithm

The algorithm for system verification using automata is based on language containment checking. In practice the algorithm performs a language emptiness check based on the result of Proposition 3.1.

Proposition 3.1 Given two automata A and B with $L(A) \neq \emptyset$, then $L(A) \subseteq L(B) \Rightarrow L(A \times B) \neq \emptyset$. See [Kurshan 94] for more details.

The algorithm works as follows: Given a program P and a property φ, the first step is obtaining the automaton A_p for the program P and the automaton $A_{\neg\varphi}$ for $\neg\varphi$. After, an emptiness check is performed on the product automaton $A_p \times A_{\neg\varphi}$. In order to verify whether $L(A_p \times A_{\neg\varphi}) \neq \emptyset$, efficient algorithms exist that are based on searching techniques[6] as proposed in [Courcoubetis & al. 92]. The advantage of performing the product is that $A_p \times A_{\neg\varphi}$ instead of $A_p \times A_\varphi$, is that the first automaton will only accept rejecting sequences and there will be few of them or none if the program is correct.

[6] Adapting distributed searching algorithms for language emptiness checking could constitute a straightforward approach for a *distributed verification architecture*.

Infinite behavior

If a program stops, its behavior may be captured by a simple *-automaton, but if it is non-stopping then we are in the presence of infinite behavior captured by infinite languages that are generated by ω-automata like Büchi automata, Muller automata or Rabin automata.

Expressiveness enhancements

Arbitrary long but finite behaviors may be modeled by *-automata. Infinite behavior must be modeled using ω-automata, but they are as expressive as LTL logics because they can only characterize linear behavior. Expressiveness enhancements towards CTL branching-like properties require the use of yet another class of automata, the *tree-automata*. For instance the formula AGEFp, which can be informally stated as "no matter where you are it is possible to eventually get to an occurrence of p", has no ω-automaton counterpart. The usual results for *-automata and ω-automata may generalized to tree-automata, in particular, language containment checking may be performed in much the same way with the same complexity. A comprehensive treatment of the topic of ω-automata and tree-automata if given in [Thomas 94].

Complexity of the algorithm

The complexity of the MC algorithm depends on the state space size and also on the specification language. For example, if the specification language is LTL or CTL, the algorithm is linear in the size of the model but exponential in the size of formula. As highlighted above, the emptiness problem for ω-automata is decidable in linear time. Relating the automata based approach to the Temporal Logic approach can be done translating the temporal formula to an automaton, which is exponential in space. A recent work [Daniele et al. 99] presents an improved algorithm for translating LTL formulas into automata. However, we still run into automata with $2^{O(n)}$ states in worst case conditions, where *n* is the number of sub-formulas of the specification.

3.4 Optimizing the Model-Checking process

Implementing the Temporal Logic MC approach or the Automata based approach naïvely can lead to some disappointment. The algorithms will reveal to be extremely ineffective and of limited practical usefulness. Optimization techniques must be employed.

The first class of techniques try to optimize the exploration of the model, avoiding its complete generation (*on-the-fly* techniques), avoiding its complete exploration (*partial-order* techniques) or working with an abstraction of it (*abstraction* techniques). A second class of techniques tries to improve the representation of state spaces (*symbolic* techniques).

In practice, model-checking algorithms do not include a separate stage where the state space is first generated before being analyzed. A technique denominated *on-the-fly state space exploration*, evaluates the formula while simultaneously constructing the state space. When a counterexample is found the procedure stops since there is no need to keep on constructing the state-space. This technique has the potential of considerably reducing the memory and time required in verification. Details can be found in [Courcoubetis & al. 92, Kurshan 94]. An implementation of the algorithm is proposed in [Gerth & al. 95].

Intuitively, the parallel composition of processes may result in a large number of "equivalent" paths joining two states that only differ by some irrelevant ordering of actions. This happens due to the many possible interleavings of concurrent behavior. The notion of *equivalent paths* is different among specification languages. Some of these do not distinguish between some sequences of actions, but others do. If executing transition 'a' after transition 'b' produces the same result as executing transition 'b' after transition 'a' then we say that 'a' and 'b' are *independent*. Based on this observation (that some transition systems are commutative) we can define a relation between independent transitions called *independence relation*.

Partial-order techniques aim to relieve the state explosion problem using the concept of transition independence. Instead of expanding all transitions leaving from a specific state, only a small subset is expanded according to some specific criteria. In the end a reduced version of the state-space is generated. Different methods to achieve partial order reduction are based, for example, in *ample-sets* [Peled 93] or *stubborn sets* [Valmari 88]. A very good survey on the subject is presented in [Peled 98].

Abstraction techniques can greatly reduce the state-space by eliminating information irrelevant to the property being verified, abstracting away unnecessary detail. For example, if we want to prove a property like AG(x>0), then we can build an abstract model based on two equivalence classes of states (those where x>0 and those where ¬(x>0)). The use of abstraction was advanced in [Clarke & al. 94b] and can provide a major source of reductions on state space sizes. However, this technique is not gaining full acceptance because automatic ways of generating abstractions have not yet emerged. Abstraction has even been regarded as impractical in software verification [Jackson & al. 94]. An application of abstraction techniques is reported in [Heitmeyer & Bharadwaj 97]

Other techniques to alleviate the state space size rely on efficient representations. The representation strategy can adapt to the system being considered. There are two main representation strategies: *Symbolic* or *implicit* state space representation and *reachable state space* or *explicit* representation.

Symbolic state space representations have been used with great success in a technique known as *symbolic model checking* by providing very compact representations of state spaces. The *Binary Decision Diagram* (or BDD) [Bryant 86] is certainly the most well known symbolic representation of a state space.

The usage of BDDs has been successful in industry [Clarke & al. 92] by allowing the verification of a very complex hardware communication protocol. The work of [Burch & al. 90] describes comprehensively this technique that has been generating dozens of new results each year. Basically, a BDD is a representation of a Boolean function. The information contained in each state is converted into a Boolean vector, and the transition relation R(x,y) is viewed as function from Boolean vectors to Boolean vectors. See [Kurshan 94] for more theoretic material on BDDs.

We can construct a BDD for a given Boolean function in the following way:

1. First, build a decision tree for the desired function, that obeys the following constraints: 1.1) Along any path from the root to the leaf, no variable appears more than once. 1.2) Along every path form the root to the leaf, the variables should always appear in the same order.
2. The second step is to repeatedly applying the following two reduction rules: 2.1) Merge duplicate nodes (nodes with same label and same children). 2.2) If

both child pointers of a node point to the same child, one of the nodes (child or parent) should be deleted, because one of them is redundant.

This procedure converts a binary decision tree into a *binary directed acyclic graph*, which is a BDD as shown in Figure 3.5.

In fact, practical applications of BDDs do not construct binary decision trees but generate them directly from higher level representations and manipulate them always on their fully reduced form. In [Hu 95] it is explained how to build Boolean function representations from high level descriptions.

The resulting BDD of any binary operation between two Boolean functions can be computed in time proportional to the product of their underlying BDDs sizes. Evaluating a BDD-represented function can be achieved in time linear to the number of variables.

Empirical evidence has shown that for the same classes of problems, the introduction of BDDs has enabled a much larger set of systems to be verified. Their advantages are that they are reasonably small[7] and they can be directly manipulated to effectively compute every Boolean operation. Despite their advantages some problems persist: In some situations BDDs run slower and are more sensitive to implementation patterns. They perform poorly in programs that, for example, make hard use of counters, because counters yield very irregular state spaces. Another difficulty is the variable ordering. As referred in [Hu & Dill 93] the size of the resulting BDD is very sensitive to the way in which the variables are selected to build the diagram. Variable ordering can make the difference between a linear and an exponential BDD and finding the optimal ordering is an $O(n^2 3^n)$ complex problem. Because of this, they have not yet replaced explicit enumeration algorithms.

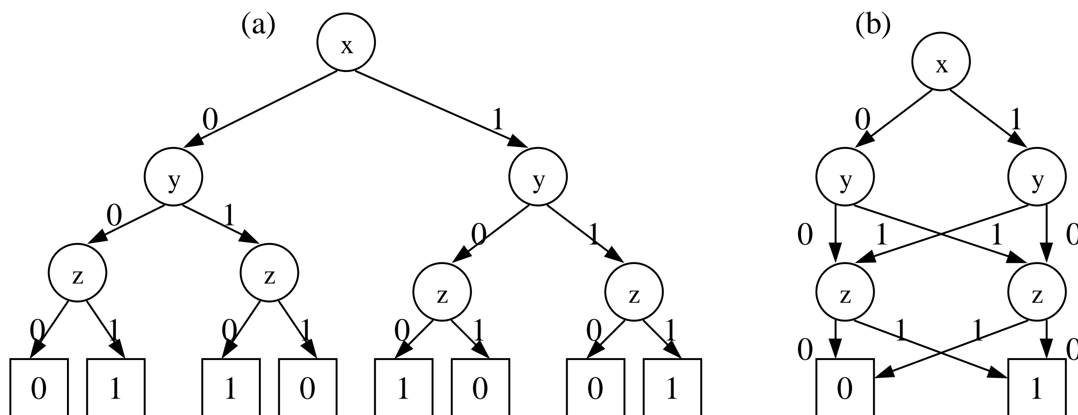


Figure 3.5 – Presentation of a Binary Decision Diagram for the function f(x,y,z)= x *xor* y *xor* z. Part (a) shows the decision tree for the function f(x,y,z) is shown. This tree can be reduced to the BDD as shown in part (b). Nodes represent decision point and squares (leafs of the tree) represent resulting values.

3.5 Classes of systems

Systems state-spaces can be quite different from one another. The reason for this to happen is not only the presence of a particular specification but also the nature of the system being specified. Some systems generate state-spaces with similar patterns and

[7] Truth tables, for instance, have exponential size in the number of variables.

according to these patterns we can classify them in classes. However, comprehensive references to systems classes have not been proposed yet, nor have *state space patterns* of systems been studied in a systematic way. Below, the author advances a classification of systems. The impact of the different classes of systems in their corresponding state spaces is also appropriately outlined.

Systems can be classified as *open systems* and *closed systems.* An open system is one that interacts with the environment. When constructing the state-space of an open system, we often witness an explosion of the state space because of the nondeterminism of external actions. For instance, a process with the following action ***c?x:****int*, that receives a value on channel 'c' and stores it in variable 'x', can generate as many states as the cardinality of the domain *int*. There are two ways of diminishing the uncertainty about the data that can be stored in variable 'x'. The first one is establishing lower and upper bounds to *receiving actions* from the environment by enforcing a specification discipline that promotes the use of guards, like for instance: ***c?x:****int* ***where*** *(x>0 and x<10).* The second way is specifying the environment, but this is not always possible because sometimes we do not know its working rules. By specifying the environment we can have a better idea of what values are produced on a channel by using a type-inference mechanism. This information can then be used to reduce the state space. For many problems it is possible to specify the environment, leading to the so-called closed system. Frameworks exist where specification of open systems is not possible. The specification must always take the form Env || Sys[8], where Env is the environment specification, Sys is the system specification and '||' is some composition operator.

In a *loosely coupled* system, there are not many actions being performed or resources shared by more than one component (also called process) of a system. Because of this, each component of the system can proceed somewhat independently from the others causing the number of interleavings to grow very quickly in the number of components. On the other hand, a system can be *tightly coupled*, meaning that almost every action synchronizes with other parts of the system. This fact causes the number of interleavings to be greatly reduced.

The majority of the existing verification frameworks only allow for the use of very *simple data types* like Booleans, Integers and Enumerations. It is still not obvious how to formally verify systems with *complex data types* like linked lists or trees. Systems with complex data types produce much larger and irregular state spaces than those systems using only simple data types.

Globality refers to the global shared use of resources between system components. For example if there are many components accessing resources in a shared-memory fashion we can have big state spaces. If we promote *locality* we reduce the number of possible interleavings in resource accesses. Locality means that resources are only visible to a limited number of sub-components of the system that are said to be in their scope. This information can be used to greatly optimize the state-space generation by allowing to abstract away from transitions that do not interfere with the property being verified. Some references can be found in [Kokkarinen 98], in [Valmari 96, Krimm &

[8] In some frameworks it is often referred that, from the point of view of one system component everything else is the environment. If spec=P||Q||R, from R's point o view spec=R||Env where Env=P||Q. This view however, is not incompatible with the one presented above.

Mounier 97] techniques are presented for taking advantage of locality expressed using the LOTOS *hiding* operator when generating LTSs.

Currently, the MC verification approach effort in industry is applied for the verification of hardware systems and for the verification of software systems. Although not dissociated, this reflects a distinction often referred in literature as hardware/software systems dichotomy. This hardware/software MC dichotomy is caused due to differences in the nature of the underlying systems architectures, such as:

- Hardware is mostly synchronous whereas software is mostly asynchronous.
- Very simple data types are used in hardware systems opposed to very complex data types present in software.
- System variables are mostly global in hardware whereas in software the use of locality is encouraged.

These peculiarities have great influence in the representation strategy and latterly, in the algorithm. Different kinds of MC algorithms have been proposed that rely only on scarce empirical observations about the systems nature. However, by pursuing the study of classes of systems it is hoped to design MC algorithms better adapted to different kinds of applications

3.6. Some of the existing verifiers

An overview on some of the existing verifiers is presented below. This is by no means a comprehensive list but an attempt to describe some features of tools that the author analyzed. A comparative analysis on some MC tools is given in [Dong & al. 99], where substantial benchmarking is presented focusing in expressiveness aspects of their corresponding specification languages.

CADP (CAESAR/ALDÉBARAN Development Package)

This toolset is suited for the verification of small to medium-sized systems like communication protocols or distributed systems written in process languages [Garavel 98]. CADP is an open architecture, currently constituted of a toolbox and an Application Programming Interface (API) that allows for further development of tools to interface with the existing ones. The toolbox is constituted of compilers and verifiers. The compilers translate systems specifications written in process languages into transition systems represented by graphs. Compilers exist for ESTEREL, LOTOS and LUSTRE. The underlying technology is comparison of labeled transition systems, i.e., equivalence checking. The verifiers include Model-Checkers and equivalence checkers that compare graphs up to some equivalence relation such as, for example, *weak bisimulation*.

KRONOS

The KRONOS tool [Yovine 97] aims at the verification of real-time hybrid systems represented as timed-automata. Automata are described using an automata description language and properties are specified in TCTL. Timed automata and timed temporal logics underlay the principles of this tool. The MC algorithm [Yovine 98] performs reachability analysis of timed-transition systems using on-the-fly techniques, partial order techniques and BDDs.

Murφ

The Murφ verifier [Dill 96] has been used for hardware verification although it has been reported to be also very effective in software verification [Dong & al. 99]. The Murφ description language is based on Chandy and Misra's Unity language [Chandy & Misra 88]. A system description consists of constant and variable declarations, procedure declarations, rule definitions, a description of the start state and a collection of invariants. Each rule is constituted of a guard and an action. Correctness requirements are written as invariants, boolean expressions that must be true at given state. Whenever one of these is violated an error message is displayed and an error trace generated. Properties in Murφ are written in a subset of LTL. The underlying technology is depth-first or breath-first to systematically generate reachable states together with Symmetry reduction, Partial-Order methods, replicated component abstraction, probabilistic algorithms and reversible rule application.

SMV (Symbolic Model Verifier)

The SMV system [McMillan 92] has been used mainly in hardware verification although it is claimed that this system can be used for software verification. The language used by SMV can be seen as a system of equations to describe the next state and hence, the transition relation as a finite Kripke structure. The underlying technology is Symbolic Model-Checking [McMillan 93], the transition relation is represented implicitly by Boolean formulas implemented by BDDs. The system properties are specified using CTL, which are translated into Mu-Calculus. Computing the satisfaction of formulas over the state space represented as BDDs is accomplished through fix-point computations as presented in [Clarke & al. 86, Burch & al.90, Clarke & al. 92].

STeP (Stanford Temporal Prover)

The main goal of STeP [Bjørner & al. 96] is the specification and verification of concurrent and reactive systems. Specifications are written in SPL (Simple Programming Language) [Manna & Pnueli 92], a very expressive Pascal-like language with message-passing primitives and parallel composition among other features. STeP is able to verify both finite and infinite state systems by using Model-Checking and deductive methods respectively. The properties are written using LTL.

SPIN

The SPIN verifier [Holzmann & Peled 96] is a Model-Checker for asynchronous systems specified in the language PROMELA. PROMELA is a non-deterministic guarded command language designed to specify protocols by modeling process interaction and coordination. The language provides variables and general control flow structures in the tradition of Dijkstra's guarded command language and Hoare's CSP. The underlying technology of this tool is based automata on providing LTL model-checking capability. Properties to be checked are represented as Büchi automata. Reachability analysis is performed through a depth-first search and a single-pass, on-the-fly verification algorithm coupled with partial order techniques.

UPPAAL

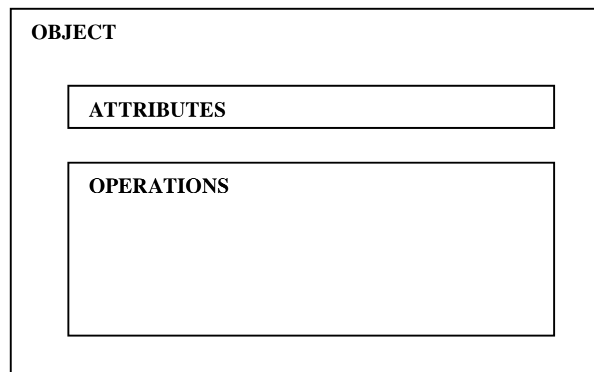
UPPAAL is a tool for the modeling, simulation and verification of real-time systems [Larsen & al. 97]. Its theoretical foundations were presented in [Yi & al. 94] where a restriction imposed of the property description language (a simpler version of TCTL) seemed a shortcoming when compared to other tools that supported timed Mu-Calculus. However, the option of simpler property language enabled for the verification of much larger systems, which constitutes an ingredient of success for this tool. Timed automata are described in a description language referred as the *.ta* format. The underlying technology is on-the-fly symbolic model-checking of timed systems by constraint checking.

Chapter 4 – The OBLOG specification language

In this chapter we propose a subset of an object-oriented language that has been be used to develop high-level specifications. The OBLOG language was designed for the development of large-scale information systems and evolved from studies in object logic specification languages [Sernadas & Ehrich 91, Jungclaus & al. 91]. Since the full OBLOG language used in industry contains far too many features for a first approach to verification, we restrict to a simpler version of the language. Its semantics will be given informally while presenting the syntax. Some notions needed for extracting syntactic information about identifiers of our specifications are also presented. The chapter ends with a presentation of the Alternating Bit Protocol.

4.1 Objects, Operations and Methods

In OBLOG, a system specification is a community of *objects* running in parallel. The central constituting part of an object-oriented system is the *object*, which can be seen as an abstraction of an entity with a public interface and an internal body. OBLOG objects encapsulate a set of *attributes* and a set of *operations* as shown in Figure 4.1. The only way one can change the object attributes is by requesting the object to perform operations.



object ::= '**`object`**' *id* '**`is`**' *attributes operations* '**`end`**'

Figure 4.1 – Structure of an object and syntax of an object declaration

The state of an object is given by the set of values of the object's *attributes*. Changing the state of an object means changing the value of some attribute. Attribute value changes are made through *set* and *get* operations defined in the object's interface.

For example, the object ‘Point’ should provide the operation `getX(out x:integer)` in order to allow other objects to access read the ‘Point.X’ attribute in read mode.

An operation may be carried out through several *methods*. Each method is constituted of an *enabling condition* and a *behavior component*. Depending on the valuations of the input parameters of an operation, different methods may be selected to execute from the set of *enabled methods*. Every method has an *enabling condition* that defines for which values of the input values of the operation the method is *enabled*. If a method fails to execute then an alternative method is selected until one of them executes successfully. If no method executes, we say that the operation failed to execute.

In Figure 4.3, a graphical representation of the structure of a method is given. A method works as follows: If the enabling condition holds, then the behavior component is executed under a context augmented with local variables.

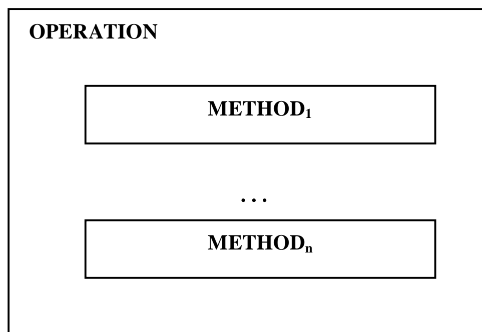


Figure 4.2 – Structure of an operation

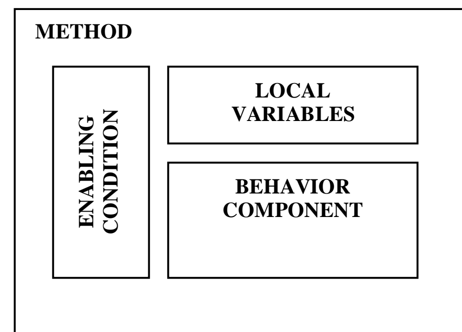


Figure 4.3 – Structure of a method

operation ::= ‘**`operation`**’ *id* ‘**`(`**’ *inparms* ’**`,`**’ *outparms* ‘**`)`**’ *methods*
method ::= ‘**`method`**’ *id* ‘**`enabling`**’ *boolexpr* ‘**`local`**’ *variabledecs* ‘**`do`**’ *bhexpression*

Figure 4.4 – Syntax of operation and method declarations

A method executes successfully if its behavior component executes successfully. If, on the other hand, the behavior component of a method fails then we say that the method also fails.

4.2 Features left out of the OBLOG language

The simple version of OBLOG treated herein leaves out some features that can be found in common programming languages. These features are *Inheritance*, *Dynamic Creation of Objects* and *Exception Handling*.

The use of *Inheritance* in an object-oriented language encourages the reuse of code; it supports development of software in a style called *differential programming*, in which the developer builds a new object by stating the difference to a previous one. The semantics of the OBLOG inheritance mechanism was not clear at the time of developing this dissertation. The author investigated the semantics of inheritance mechanisms of common object-oriented programming languages, which are based in Hailpern and Nguyen object-oriented model [Hailper & Nguyen 87]. In this model, let us suppose that

an object ‘B’ inherits from an object ‘A’; the invocation of an operation of ‘B’ inherited from ‘A’ will be forwarded from ‘B’ to ‘A’ which is the real implementer of the operation. In OBLOG, the inheritance mechanism is somewhat more complex because it involves the enrichment of behavior from the parent object in ways that are not easily captured by the mechanisms just mentioned. The inheritance mechanism present in OBLOG is studied in greater detail in [Andrade 99].

Features involving creation of new entities at run-time like *dynamic allocation of memory* and dynamic creation of objects were also left out. Verification tools based on MC technology do not allow the verification of systems with dynamic creation of processes. Since we are coding objects as processes,[9] we cannot verify systems with dynamic creation of objects. This means that the number of objects is fixed in the lifetime of the system.

Some object-oriented languages also provide a mechanism for handling extraordinary events or conditions. This mechanism known as Exception Handling is not treated in this work. The main reason is that the semantics of Exception Handling involves dynamic creation of objects of type *exception* and the manipulation of object references. Coding these features for MC verification tools is a subject further research.

4.3 Behavior Components

The behavior of an object is modeled by specifying *behavior components*. We shall use P and Q to designate behavior components and C to designate a Boolean condition. Behavior components can be of two kinds, *atomic components* and *composite components*. An atomic component is an elementary action and a composite component is an assembly of behavior components. Every behavior component is a *producer* and *consumer* of information; it consumes a set *I* of input values and produces a set *O* of output values.

Skip

This component is also referred as inaction or null behavior. It does not consume or produce any information.

skip ::= ‘`skip`’

Fail

The *fail* component causes immediate interruption of the current behavior component and starts the closest alternative behavior component branch (Definition 3.1). This component does not produce or consume any information.

fail ::= ‘`fail`’

Assert

The *assert* component causes a condition to be evaluated. If the condition evaluates to *false* the assertion component behaves as a *fail* component; if the condition

[9] As shown in chapters 5 and 6.

evaluates to *true* then the component behaves as a *skip* component. The information consumed by the assert component is the set of variables of the condition.

assert ::= '**`assert`**' *boolexpr*

Set

The *set* behavior component evaluates an expression and stores the resulting value in a variable. It consumes the variables present in the expression *expr* and produces the variable *v*.

set ::= '**`set`**' *id* '<<' *expr*

Send

The send behavior component is used to send values over an operation channel. Send is synchronous, i.e., it blocks on the channel corresponding to an operation until values are taken out from it. It consumes the variables present in the list *exprs* and produces no information.

send ::= '**`send`**' *exprs* '**`in`**' *id*

Receive

The *receive* behavior component is used to receive values from an operation channel. As its *send* counterpart it is also synchronous in the sense that it blocks until information is sent on the operation channel. This component can also receive a failure over an operation identifier behaving as *fail.* It consumes no information and produces the information a variable list *ids*.

receive ::= '**`receive`**' *ids* '**`in`**' *id*

Composite components 'aggregate' atomic components to produce elaborate behavior expressions. The consumption and production of information is the union of the variables used by the aggregates.

Iteration

The *iteration component* behaves as common procedural languages, executing P while C evaluates to true.

iteration ::= '**`while`**' *boolexpr* '**`do`**' *bhcomponent*

Sequential composition

The *sequential behavior component* provides for the specification of sequential behavior. Writing `P;Q` where P and Q are behavior components, means that Q will only execute after P.

sequence ::= *bhcomponent* '**`;`**' *bhcomponent*

Alternative composition

We are often interested in specifying what happens when a command fails. Using *alternative behavior components* we may do it an elegant way. The fact that Q should execute if P fails is written as '`P` **`alt`** `Q`' .

alternative ::= *bhcomponent* '**`alt`**' *bhcomponent*

Definition 3.1 Given a sequence of alternatively composed behavior components in the form P_1 **`alt`** … **`alt`** P_n we define *Closest alternative behavior component branch* as being P_{n-1} for P_n. P_1 propagates the failure to its environment.

Local variables

Local variables allow for encapsulation of information exchange between behavior expressions. Local variables have a scope that is constituted of a behavior expression, it can be any behavior expression but to take advantage of it, it should in fact be a composite behavior expression. Writing **`local`** $v_1:s_1=i_1$, …, $v_n:s_n=i_n$ **`in`** `P` means that the variables $v_1,\ldots,v_n$ of sorts $s_1, \ldots, s_n$ are only visible inside P with initial values $i_1, \ldots, i_n$.

local ::= '**`local`**' *variabledecs* **'`in`'** *bhcomponent*

Choice (OR) composition

The choice behavior component provides for the choice among a list of behavior components in a CSP-like fashion. Writing **`or`**`(`bh_1, …, bh_n`)` can be expressed CSP as *bh_1 [] ... [] bh_n*.

choice ::= '**`or(`**' *bhcomponents* '**`)`**'

Alternative or (XOR) composition

The alternative or behavior component can be regarded as some kind of multiway alternative behavior component. When one of the operands fail, say bh_k, in **`xor(`**bh_1, …, bh_{k-1}, bh_k, bh_{k+1},…, bh_n**`)`**, the state is rolled back and the 'xor' component behaves like **`xor(`**bh_1, … bh_{k-1}, bh_{k+1}, …, bh_n**`)`**. Where bh_k is left out. When no behavior component is left, the 'xor' component fails.

xor ::= '**`xor(`**' *bhcomponents* '**`)`**'

4.4 Some aspects of coding behavior components

In the task of isolating the subset of the language that is to be analyzed automatically, many versions were investigated. The first reason is that some features are very difficult to code and the scope of this work had to be reformulated. The second reason is that some language constructs turned out to be expressible in terms of simpler ones. Due to this, some core behavior components were isolated that also provided a deeper understanding of the language features. For example the *preconditioning* behavior component is

primitive in the industrial version of the language, however in this simpler version it is presented as syntactic sugar over the *assert* behavior component.

Another aspect is related to information about behavior components. In the earlier versions of the translation rules presented in chapters 5 and 6, a type system for OBLOG terms was presented in order to ease the task of obtaining information about terms. A closer look revealed that the information needed could be easily given by a set of auxiliary definitions. These definitions are presented below because they will be used in forthcoming chapters.

Failure dynamics

The notion of *failure* is primitive in OBLOG. Although some languages also offer the notion of failure (sometimes called *exception*), few of them provide an automatic roll-back mechanism upon failure. The notion of failure in OBLOG comprehends two levels: the level of behavior components and the level of methods. Instead of developing two different mechanisms, an effort was put in finding a unifying mechanism onto which the notion of failure of behavior components and methods could be coded.

When a behavior component fails, the failure is propagated until it is captured by an alternative behavior component. The notion of *failure propagation* has to do with the failure of a behavior component upon the failure of one of its sub-components – we say that the sub-component propagated the failure to the 'father'. A failure can be propagated either *directly* or *remotely*. A *direct failure propagation* is the one that propagates through composition operators (not through *alt*). For example, in a sequential composition where R≡(P;Q), if P fails then R fails. By *remote failure propagation*, we mean the transmission of failure on operation calls. When calling an operation, if the operation fails, this information should be transmitted to the behavior component performing the operation call. In order to model OBLOG synchronous operation calls we extended the language with the syntactic sugar construct **`failon`**.

Syntactic sugar

The specifications are more readable if we introduce some syntactic sugar. We start with pre and post conditioning and conditional blocks as follows:

```
pre C in P ≡ assert C; P

pos C in P ≡ P; assert C

if C then P else Q ≡ (pre C in P) alt Q
```

Another natural coding can be given to the synchronous call of a method as:

```
call op(inparms, outparms) ≡ send inparms in op; receive outparms in op
```

where receive is also potentially failing. It receives an extra parameter with failure information. This is used to implement the remote failure mechanism described above. Sending failure over an operation channel can be done using **`failon`** which is simply the *send* behavior component augmented with one parameter that transmits the failure.

The list *iparms* contains expressions and the list *oparms* contains variables. Operations can also be seen as a composition of simpler behavior components as:

```
operation op(iparms, oparms) method1 … methodn ≡ receive iparms in op;
local oparms in (method1 alt … alt methodn) alt failon op
```

Where each method is expanded as:

```
method mi enabling Ci local Varsi do Bi ≡ assert Ci local Varsi in (Bi;
send oparms in op)
```

The specification of the ABP is presented in Appendix B without syntactic sugar and without syntactic sugar in Appendix C.

Obtaining information about OBLOG behavior components

The subsequent chapters present rules that rely on syntactic information about behavior components that we define below. We start by defining the sets of identifiers that are used for consumption of information and for storing the newly produced information.

Definition 3.2 We define the *set of identifiers consumed by a behavior component* B inductively in the structure of B, written *I*(B):

- *I*(**skip**) = *I*(**fail**) = ∅.
- *I*(**assert** C) = *I*(C).
- *I*(**set** `X` **<<** `Expr`) = *I*(`Expr`).
- *I*(**send** `values` **in** `op`) = *I*(`values`).
- *I*(**receive** `values` **in** `op`) = ∅.
- *I*(`P ; Q`) = *I*(P)∪*I*(Q).
- *I*(P **or** Q) = *I*(P **xor** Q) =*I*(P)∪*I*(Q).
- *I*(P **alt** Q) = *I*(P)∪*I*(Q).
- *I*(**while** C **do** P) = *I*(C)∪*I*(P).
- *I*(**local** $v_1:s_1=i_1,\ldots,v_n:s_n=i_n$ **in** B) = *I*(B) – $\{v_1,\ldots,v_n\}$

Definition 3.3 We define the *set of identifiers produced by a behavior component* B inductively in the structure of B, written O(B):

- *O*(**skip**) = I(**fail**) = O(**assert** `C`) = ∅.
- *O*(**set** `X << Expr`) = {`X`}.
- *O*(**send** `values` **in** `op`) = O(`values`).
- *O*(**receive** `values` **in** `op`) = ∅.
- *O*(P ; Q) = *O*(P or Q) = *O*(P xor Q) = *O*(P **alt** Q) = *O*(P)∪*O*(Q).
- *O*(**while** C **do** P) = O(C)∪O(P).
- *O*(**local** $v_1:s_1=i_1,\ldots,v_n:s_n=i_n$ **in** B) = *O*(B) – $\{v_1,\ldots,v_n\}$.

Definition 3.4 The set of variables used by a behavior component, written *Vars(BhExp)*, is defined as the union of the set of identifiers consumed and produced by the behavior component.

Definition 3.5 We define the *set of operation identifiers used by a behavior component* B inductively in the structure of B, written *OP*(B):

- *OP*(**skip**) = *OP*(**fail**) = *OP*(**assert** C) = *OP*(**set** X **<<** Expr) = ∅.
- *OP*(**send** values **in** op) = {op}.
- *OP*(**receive** values **in** op) = {op}.
- *OP*(P; Q) = *OP*(P)∪*OP*(Q).
- *OP*(P **or** Q) = *OP*(P **xor** Q) = *OP*(P **alt** Q) = *OP*(P)∪*OP*(Q).
- *OP*(**while** C **do** P) = *OP*(C)∪*OP*(P).
- *OP*(**local** v_1:s_1=i_1,…,v_n:s_n=i_n **in** B) = *OP*(B)

Every identifier must have some predefined sort. OBLOG expressions induce a sort assignment mapping that assigns sorts to variable identifiers.

Definition 3.6 We define the *sort assignment mapping of identifiers in a behavior component* B inductively in the structure of B, written *SORT*(B):

- *SORT*(**skip**) = *SORT*(**fail**) = *SORT*(**assert** C) = *SORT*(**send** values **in** op) = ∅.
- *SORT*(**set** X **<<** Expr) = {Expr→s}, where s=*SORT*(X).
- *SORT*(**receive** v_1**:**s_1,…,v_n**:**s_n **in** op) = {$v_1 \rightarrow s_1, \ldots, v_n \rightarrow s_n$}.
- *SORT*(P; Q) = *SORT*(P **or** Q) = *SORT*(P)∪*SORT*(Q).
- *SORT*(**while** C **do** P) = *SORT*(C)∪*SORT*(P).
- *SORT*(**local** v_1:s_1=i_1,…,v_n:s_n=i_n **in** B) = {$v_1 \rightarrow s_1, \ldots, v_n \rightarrow s_n$} ∪ *SORT*(B).

4.5 Specification of the Alternating Bit Protocol

The Alternating Bit Protocol [Bartlett & al. 69] is a simple link layer communication protocol used in many verification case studies. In this protocol setup there are two communicating objects named *sender* and *receiver* which offer the communicating parties message sending and receiving services respectively (we do not intend to model communicating parties here). Furthermore there exist two objects that represent *transmission* and *acknowledge* lines.

The protocol can be summed up textually as follows: The sender transmits the message and augments it with a bit. When the acknowledge signal arrives the sender checks if the acknowledge bit is the same of the message that was sent, if it does not, it ignores the acknowledgement. After some amount of time, if an acknowledgement does not arrive, the sending party (using the protocol) re-transmits the message. If necessary the message is retransmitted a third time and so on until an acknowledgement arrives. The receiver recognizes a new message because the appended bit is different from the one that it has got. If the bit is the same then it is a re-transmission, which the receiver ignores. Otherwise, it sends an acknowledge and inverts its internal bit. Both the transmission and acknowledge lines are unreliable in the sense that they may fail to deliver their messages.

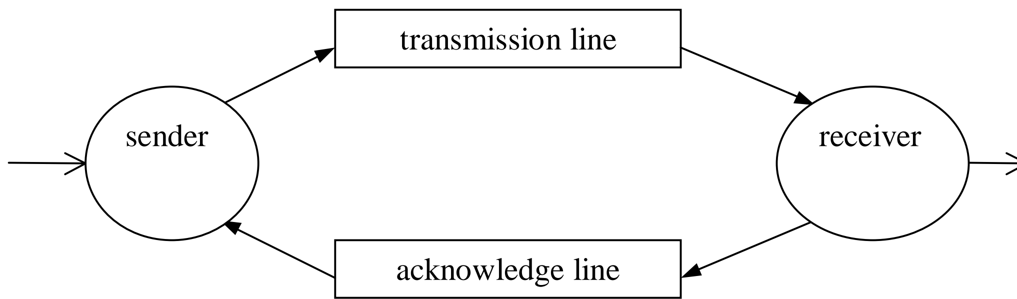


Figure 6.16 – Alternating bit protocol setup

In the OBLOG implementation, the 'sender', the 'transmission line', the 'receiver' and the 'acknowledge line' are objects. The communication is started when the user tries to send a message. For this purpose, it should call the *sender.accept* operation of the 'sender object'. The message is received on the other side when a user tries to receive a message from the receiver object by calling the *receiver.deliver* operation. The transmission process is triggered in the *sender.accept* operation, which causes the 'sender' object to request a further operation on the 'transmission line' calling *transline.sendMessage* operation. The transmission line can fail or deliver the message, and if the message is new to the 'receiver' then the corresponding acknowledge bit is sent throughout the 'acknowledge line'.

sender	**Transmission line**	**Acknowledge line**	**receiver**
Accept	SendMessage	SendAcknowledge	deliver
GetStatus	ReceiveMessage	ReceiveAcknowledge	getStatus

Table 4.5 – Summary of operations of the ABP objects

Modeling communication lines failure

As outlined above, the 'transmission line' and the 'acknowledge line' are unreliable communication lines. In our specification, this notion is explicitly coded in state of the objects that represent communication lines. When a send operation is issued, the communication line either fails, by setting the internal state to a special failure constant, or succeeds by copying the message and the control bit into an internal memory area composed of object state attributes. This non-deterministic effect is obtained by an alternative composition of two assignment blocks using the OBLOG '**or**' operator as illustrated in Figure 4.6.

Handling message re-transmission

Several different versions of the ABP exist that use retransmission. The 'sender' waits for some amount of time for the sent message to be acknowledged, after which it retransmits the message. In practical applications, the retransmission process does not run forever in situations where a message is always lost, but the sender object usually returns an error to

```
operation sendMessage( in m : message, in b : bit )
     method send
     enabling true
     do
        or
        ( (
              set lStatus << STATGOOD;
              set buffMsg << m;
              set buffBVal << b
           )
           set lStatus << STATERROR;
        )
```

Figure 4.6 – OBLOG specification of the SendMessage operation illustrating the non-deterministic behavior of a communication line captured by the assignment to the variable lStatus.

the user after some pre-fixed number of retransmissions. In the specification we developed, we adopted a simple strategy of providing the 'sender' object and the 'receiver' object *getStatus* operations. These operations enable the user of these objects to periodically query them for message acknowledge in the case of the 'sender' object or new message was delivery in the case of the 'receiver' object. Given this setup, message retransmission must be handled outside our specification.

Chapter 5 – Process algebraic verification approach to OBLOG specifications

In this chapter we present a verification procedure for the OBLOG object-oriented formal specification language based on the translation to another specification language. The translation will be performed to the ISO 8807 norm LOTOS process description language [ISO 88].

The LOTOS language is based on the process algebras CSP and CCS with some extensions. Several proposals of translation of object-oriented languages into process algebras exist in the literature; an example is the semantics of the POOL object-oriented language family given in terms of Milner's π-calculus that can be found in [Walker 92] and [Walker 95]. Although inspiring, the work of Walker can not be directly applied in our setting because it makes use of the capability of sending names in channels present in the π-calculus, a feature not present in LOTOS.

5.1 Technical framework

The LOTOS language is a message passing process algebra that combines and extends features of both CSP [Hoare 85] and CCS [Milner 89]. Success stories of industrial applications of LOTOS exist, namely, the specification of a portion of the Airbus Flight Warning Computer that constitutes, at the time of writing, one of the largest existing formal specifications. This specification was developed using the CADP toolset [Garavel 98], which is an environment for specification and verification using the LOTOS language.

A LOTOS[10] specification includes two parts: A *data-type specification* part and a *behavior specification* part. The specification of data-types is done using the language ACT-ONE [Ehrig & Mahr 85] whereas the behavior of the system is specified in LOTOS in terms of processes, possibly consisting of several sub-processes (that are processes themselves, see Figure 5.1). This enables the creation of specifications in a modular way. The identifier `process-id` stands for the name of the process and `gate-list` contains the names of the channels where externally observable actions are performed. A process can also accept parameters in `parameter-list` and the `functionality` part describes the finishing behavior of the process. If the process being described is non-stopping then the functionality is specified as **`noexit`**. If, on the other hand, the system

[10] LOTOS refers to the term Full LOTOS often found in literature. Full LOTOS is Basic LOTOS plus data-types.

```
process process-id[gate-list](parameter-list) : functionality :=

      behavior-expression

where

      process-declarations

endproc
```

Figure 5.1 – Template of LOTOS process specification.

stops, yielding a list of values the functionality is specified as **exit**($S_1, \ldots, S_n$). The heart of a process is the behavior expression.

Aspects of LOTOS syntax and semantics

The semantics of executing LOTOS behavior expressions is obtained by performing transitions. Thus, the execution of a process yields a LTS. Let us recall that a transition takes the form of Figure 5.2 where Bh_1 and Bh_2 are behavior expressions and 'a' is an action. The intuitive notion behind transitions is that Bh_1 performs action 'a' and transforms itself in Bh_2. Actions are events performed on gates.

$$Bh_1 \xrightarrow{a} Bh_2$$

Figure 5.2 – A transition executed by performing action 'a'.

An action can be considered *visible* meaning that the environment of the process that produced it can synchronize with it, or *invisible* (denoted by the symbol '**i**') meaning that the action is invisible to the process environment. By using the *hiding* expression (Table 5.1) we can turn actions invisible and control the level of abstraction of the system being analyzed. If for instance we are interested in analyzing the communication occurring among two concurrent processes we can concentrate on actions performed on selected gates and abstract away from actions performed inside the processes behavior expressions that do not directly interfere with the processes communication.

The communication mechanism for LOTOS processes is somewhat similar to ADA's *rendezvous* mechanism – when reaching the rendezvous point, the processes must wait for every process in a selected group to also reach that point and only then they will be unblocked to perform more actions. This mechanism often referred as *multiway synchronization* is absent both is CSP and CCS.

In LOTOS, the *parallel composition operator* (also called *synchronization operator*) contains a list of gates in which the processes must synchronize (where processes perform *synchronous actions*). When two processes running in parallel engage in performing an action in a gate selected in the parallel composition operator they must do it simultaneously. Any other action is performed independently by each process.

Processes in LOTOS also have a reserved gate 'δ' used to signal successful terminations. Taking process P and Q, 'a' an action, gates $g_1, \dots, g_n$ and $h_1, \dots, h_n$.We can summarize the behavior expressions as in Table 5.1.

Behavior Expression	Description	Meaning
stop	stop the process	No more transitions occur in the process.
exit	exit from the current process	A transition is carried out performing action on gate δ.
i; P	invisible action	An invisible action is performed and then P.
a**;** P	visible action	The visible action 'a' is performed and then P.
P **[]** Q	choice	Either an action of P is performed and then P', or an action of Q is performed and then Q.
P**|[**g_1**, …,**g_n**]|**Q	synchronization	An action 'a' performed on one of the gates $g_1, \dots, g_n$ causes the process P (respectively Q) to block and wait for process Q (respectively P) to perform an action on that same gate.
P **>>** Q	sequence	P until an action is performed on δ, then Q.
P **[>** Q	disabling	P until Q is able to take action, then Q. If P performs an action on δ, Q will not be able to perform any action.
hide g_1**,…,**g_n **in** P	hiding	Any actions occurring in one of the gates $g_1, \dots, g_n$ is invisible.
Pr**[**(h_1,…,h_n)/(g_1, …,g_n)**]**	process instantiation	Process P defined as Pr will run with gates $g_1, \dots, g_n$ are substituted by gates $h_1, \dots, h_n$.
Pr**[**g_1,…,g_n**]** **:=** P	process definition	Process P is defined referred as Pr**[**$g_1, \dots, g_n$**]**.

Table 5.1 – Summary of LOTOS behavior expressions.

Given a process defined as $Pr[g_1, ..., g_n] =_{def} P$, we say that $Pr[h_1, ..., h_n]$ is an *instantiation* of Pr, denoting a behavior expression P where each h_i replaces a g_i. Actions take the form *gate**!**atom* where *atom* is a closed expression without variables. The notation *gate**?**X* is also used in LOTOS, but with a different meaning from the CCS *receive* action[11]. In LOTOS the expression *gate**?**variable* is syntactic sugar for a choice on actions like *gate**!**atom*$_1$ *[] … [] gate**!**atom*$_n$, where $atom_1, \dots, atom_n$ range over the sort

[11] Note that in CCS, the *receive* action (denoted by '**?**') is symmetric to the *send* action (denoted by '**!**'), i.e., CCS processes synchronize through send/receive action combinations. LOTOS process synchronize by performing the same action simultaneously.

of variable X. In the synchronization operator P|[g_1, …, g_n]|Q, if the set of gates of either P or Q is included in {g_1, …, g_n} then P|[g_1, …, g_n]|Q can be written as P||Q. The expression P|[]|Q can be written as P|||Q. We will also consider the existence of formal variables and a behavior expression of the form **[**BoolExp**]**→BhExp where if *eval(BoolExp, Vars)=true*, then BhExp. We will not present formal variables here due to the lack of space, please refer to [Garavel 89].

The semantics of behavior expressions is given over labeled transition systems as presented below in table 5.2. Let L be a LTS where L=(S, A, Σ) and gates g_1, …, g_n, h_1, …, $h_n \in \Sigma-\{\delta\}$. Also, let 'a' be an action such that $a \in \Sigma \cup \{i\}$.

Behavior Expression	Assumptions	Transitions
`stop`		(no transitions)
`exit`		**exit** $\rightarrow^{\delta}$ **stop**
`i; P`		**i;** P $\rightarrow^{i}$ P
`a; P`		a**;** P $\rightarrow^{a}$ P
`P [] Q`	$(P \rightarrow^{a} R) \vee (Q \rightarrow^{a} R)$	P [] Q $\rightarrow^{a}$ R
`P|[g1, …, gn]|Q`	$(P \rightarrow^{a} P') \wedge a \notin \{g_1, ..., g_n, \delta\}$	P […] Q $\rightarrow^{a}$ P' […] Q
	$(Q \rightarrow^{a} Q') \wedge a \notin \{g_1, ..., g_n, \delta\}$	P […] Q $\rightarrow^{a}$ P […] Q'
	$(P \rightarrow^{a} P') \wedge (Q \rightarrow^{a} Q') \wedge a \in \{g_1, ..., g_n, \delta\}$	P […] Q $\rightarrow^{a}$ P' […] Q'
`P >> Q`	$(P \rightarrow^{a} P') \wedge a \neq \delta$	P>>Q $\rightarrow^{a}$ P'>>Q
	$(P \rightarrow^{\delta} P')$	P>>Q $\rightarrow^{i}$ Q
`P [> Q`	$(P \rightarrow^{a} P') \wedge a \neq \delta$	P[>Q $\rightarrow^{a}$ P'[>Q
	$(P \rightarrow^{\delta} P')$	P[>Q $\rightarrow^{\delta}$ P'
	$(Q \rightarrow^{a} Q')$	P[>Q $\rightarrow^{a}$ Q'
hide `g1,…,gn` **in** `P`	$P \rightarrow^{a} Q \wedge a \notin \{g_1, ..., g_n\}$	**hide** … **in** P $\rightarrow^{a}$ Q
	$P \rightarrow^{a} Q \wedge a \in \{g_1, ..., g_n\}$	**hide** … **in** P $\rightarrow^{i}$ Q
`P[(h1,…,hn)/(g1, …,gk)]`	$P \rightarrow^{a} Q \wedge a \notin \{g_1, ..., g_n\}$	P[(h_1,…,h_n)/(g_1, …,g_k)] $\rightarrow^{a}$ Q[(h_1,…,h_n)/(g_1, …,g_k)]
	$P \rightarrow^{a} Q \wedge a = g_i \in \{g_1, ..., g_n\}$	P[(h_1,…,h_n)/(g_1, …,g_k)] $\rightarrow^{h_i}$ Q[(h_1,…,h_n)/(g_1, …,g_k)]
`Pr[g1,…,gk] := P`	P[(h_1,…,h_n)/(g_1, …,g_k)] $\rightarrow^{a}$ Q	Pr[h_1,…,h_n] $\rightarrow^{a}$ Q

Table 5.2 – Summary of the semantics of LOTOS behavior expressions.

Properties about the behavior of LOTOS processes in CADP can be specified in two different ways. The first is trough *logic specifications* using a version of the Mu-Calculus that does not allow expressing properties on states. The second is through *behavioral specifications*.

The semantics of this version of the Mu-Calculus is the same as that of Chapter 2 but without propositions on actions.

Definition 5.1 We define the *Mu-Calculus without propositions* by the following grammar where 'a' is an action, Y is a propositional variable, φ, φ_1 and φ_2 are state formulas. The semantics of $[-]\varphi$ is the same as $[tt]\varphi$.

- $\varphi ::= tt \mid Y \mid \neg\varphi \mid \varphi_1 \Rightarrow \varphi_2 \mid [a]\varphi \mid [-]\varphi \mid \mu Y.\varphi$.

Additionally to the usual Boolean operators that can be obtained from $\neg\varphi$ and $\varphi_1 \Rightarrow \varphi_2$, there is also a number of other useful modal operators that can be defined such as the *greatest fixed-point operator* $\nu Y.\varphi \equiv \neg\mu Y.\neg\varphi$, the *always operator* (in all paths, at every moment, φ holds) defined as $AG\varphi \equiv \nu Y.(P \wedge [-]Y)$, the *potentially operator* (in some path, at some moment, φ holds) defined as $EF\varphi \equiv \mu Y.(P \vee \langle - \rangle Y)$ and the *inevitably operator* (in all paths, at some moment φ holds) defined as $AF\varphi \equiv \mu Y.(P \vee ([-]Y \wedge \langle - \rangle tt)$.

Behavioral specifications describe the expected behavior of the program viewed as a black-box (i.e. observed from a certain abstraction level, for example, only considering visible a subset of its actions). The properties expressed by behavior specifications can be directly coded in LTSs or in any other language that can be compiled to an LTS. After having both the program and the specification translated into LTSs, the verification process consists in comparing the two LTSs to see if the transitions taken by the program are related to the transitions taken by the specification[12].

The logical specification approach and the behavior specification approach are complementary. Some properties are easily specified in one of the approaches and not so much with the other[13]. Logic specifications are often checked by tools using the Temporal Logic Model-Checking techniques defined in Chapter 3, whereas behavioral specifications are checked by tools using the Automata Theoretic techniques.

In the approach of behavioral specifications LTSs are compared according to a class of relations based on the notion *simulation relation.* From this relation, different notions of equality can be offered differing in the treatment of invisible actions (processes internal actions) of the two LTSs. Before discussing the class of simulation relations some notions need to be recalled.

An *equivalence* relation is a reflexive, transitive and symmetric relation. A *preorder* is a transitive relation but does not need to be symmetric. An equivalence relation (noted $\approx$) is said to be a *congruence* with respect to some function f, iff $L_1 \approx L_1' \wedge \ldots \wedge L_n \approx L_n'$ imply $f(L_1,\ldots,L_n) \approx f(L_1',\ldots,L_n')$. Correspondingly, a preorder (noted $\lesssim$) is said to be a *precongruence* with respect to f, iff $L_1 \lesssim L_1' \wedge \ldots \wedge L_n \lesssim L_n'$ imply $f(L_1,\ldots,L_n) \lesssim f(L_1',\ldots,L_n')$. The operators '||' and '**hide**' can be regarded as functions between LTSs as presented in [Valmari 96] and [Krimm & Mounier 97]. Some of the relations between LTSs have interesting properties, they are often congruencies or precongruencies with respect to '||' and '**hide**' enabling a form of compositional verification that we will analyse later. Equivalencies can be ranked according to their

---

[12] This approach is intimately related to the automata theoretic verification procedure presented in Chapter 3, where the notion of transition taken by a program coincides with the notion of language accepted by a LTS viewed as an automaton.

[13] Experiments run by the author suggested that logical specifications are better suited for liveness properties whereas behavioral specifications are better suited for safety properties.

capability to make distinctions between LTSs. We say that an equivalence '$\approx_2$' *weaker* or *coarser* than '$\approx_1$', iff $L_1 \approx_1 L_2$ imply $L_1 \approx_2 L_2$. Furthermore, a relation between two LTSs is said to preserve a property iff a property holding in L_1 also holds for every LTS L_i related to L_1 by some equivalence or preorder. Below we present a survey on three equivalence relations that will be employed for verification later in this chapter. A good presentation of various equivalence relations can be found in [Fernandez 88].

Strong Bisimilarity

Strong bisimilarity (also named *bisimulation equivalence*) is the *strongest* equivalence relation (the one that distinguishes more LTSs) found in literature. For two systems to be strongly bisimilar they must simulate each other as in Definition 5.2. We can simply state that initial states must simulate each other, because the notion of simulation is recursive. A state s_1 of L_1 simulates s_2 of L_2 if every outgoing transition of s_2 arrives at a state that can also be simulated by the ending state of an outgoing transition from s_1 with the same label. See [Milner 89].

Definition 5.2 Let $L=(S, A, \Sigma)$ be an LTS with initial state $s_0 \in S$. A binary relation $\approx_{sb} \subseteq S \times S$ is a strong bisimulation, iff for every s, s_1, $s_2 \in S$ such that $s_1 \approx_{sb} s_2$ and every action $a \in \Sigma \cup \{i\}$ the following holds:

- If $s_1 \rightarrow^a s_2$, then there is an $s' \in S$ such that $s \approx_{sb} s' \wedge s_2 \rightarrow^a s'$.
- If $s_1 \rightarrow^a s_2$, then there is an $s' \in S$ such that $s' \approx_{sb} s \wedge s_1 \rightarrow^a s'$.

Branching bisimulation

This relation (noted $\approx_{bb}$) is weaker than strong bisimilarity. Moreover, whenever the two LTS under comparison are livelock-free (they do not contain any circuit of internal actions) this relation preserves liveness properties [Valmari 96]. Note that a specification should be considered as a liveness property whenever the behavior it defines must be eventually executed by the program. We write $L_1 \lesssim_{bb} L_2$ when each transition of the form $s_1 \rightarrow^a s_2$ (where $a \neq i$) is simulated by a sequence of transitions of the form $s_2 \rightarrow^i \ldots \rightarrow^i s_2'' \rightarrow^a s_2'$ where $s_1 \lesssim_{bb} s_2''$ and $s_1' \lesssim_{bb} s_2'$. Transitions of the form $s_1 \rightarrow^i s_2$ are simulated by a sequence $s_2 \rightarrow^i \ldots \rightarrow^i s_2'$ or by staying in s_2. Moreover, $L_1 \approx_{bb} L_2$ iff $L_1 \lesssim_{bb} L_2$ and $L_2 \lesssim_{bb} L_1$.

Safety equivalence

Safety properties state that visible actions do not appear in an illegal order. This relation (noted $\approx_{se}$) relates two LTS that verify the same set of safety properties and is weaker than branching bisimulation. Note that a specification can be considered as a safety specification when it defines a super set of the expected behavior of the program. In other words, the program can be considered *safe* whenever its behavior remains included in the one defined in the specification. Safety equivalence does not preserve liveness properties.

The CADP toolkit [Garavel 98] is composed of a LOTOS compiler that interfaces with a number of verification tools for LTSs including a Model-Checker. The compiler, named CAESAR, translates a restricted version of Full-LOTOS specifications into LTSs.

The specification is firstly pre-processed and translated into a simpler process algebra called Sub-LOTOS which replaces LOTOS constructs with other semantically equivalent ones. In a second stage, the Sub-LOTOS specification is represented by a Petri net and the LTS is then generated by performing reachability analysis on an optimized version of this Petri net representation. This tool does not support some ISO LOTOS features involving dynamic creation of processes and gates, and recursive process instantiation under parallel composition operators. CAESAR automatically detects these violations after a detailed syntax and semantic analysis of the LOTOS specification. Another tool named Aldébaran [Fernandez 88] can be connected with CAESAR and also accepts LTSs in several formats generated by tools other than CAESAR. This tool can be used to compare two LTSs according to several equivalence relations like *strong bisimulation*, *branching bisimulation* and *safety equivalence*.

Other tools exist for the specification and animation of LOTOS specifications, but few of them offer Model-Checking capabilities, an open format for LTS files and an API as does CADP. It is worth mentioning the Finnish toolset named ARA Toolset, that includes the ARA State Space Generator [Savola 95], a tool for generating LTSs from LOTOS specifications and the ARA Comparator Tool that work closely like CAESAR and Aldébaran respectively.

5.2 Translating OBLOG specifications into LOTOS

The translation of OBLOG specification into LOTOS is achieved by assigning a process to each object and behavior component. An object will be modeled as a process that waits for messages in a list of channels corresponding to the object operations (see Figure 5.3). Behavior components are also modeled as processes, and their composition as composition of processes. For example the sequential behavior component is modeled using the '>>' sequential composition of processes operator.

```
process stack[empty, push, pop, top] : noexit :=

...

where

...

endproc
```

Figure 5.3 – Skeleton of a Stack object coded in LOTOS.

Object attributes are coded by a process that offers and accepts values for attributes in special gates. This process runs in parallel with the object body process that is constituted of the object operations. This scenario can be better understood by looking at the following process algebraic-like equations, where *attributes* designates a *local memory* process:

$$object =_{def} attributes \parallel body$$
$$body =_{def} (op_1[\,]\ldots[\,]op_n);\ body$$

Operations are also coded as processes. Methods and behavior expressions are coded into processes that send a Boolean atom through the δ gate. This atom named *recover value* is necessary to keep track of the *success* of the execution of methods and behavior expressions.

Identifiers

It will be necessary to build new identifiers from old ones. In order to do this we introduce the concept of renaming operator on identifiers and on sets of identifiers. A further operation is defined that allows making the substitution of an identifier in a set of identifiers.

Definition 5.5 Let I be a set of identifiers. The notation $I|_v$ is used to denote the set of identifiers $\{id_v : id \in I\}$ where each id_v is obtained concatenating the string v to id. Given a set of identifiers $I=\{i_1, \ldots, i_{k-1}, i_k, i_{k+1}, \ldots, i_n\}$, we define substitution of an identifier $i_k \in I$, by some symbol X, written $I[X/i_k]$ in the following way : if $1 \le k \le n$ then $I[X/i_k] \equiv \{i_1, \ldots, i_{k-1}, X, i_{k+1}, \ldots, i_n\}$.

Gate sets

We model behavior components as processes. Behavior components need to communicate with their environments, for example asking for attribute values or calling operations. For this reason each process corresponding to a behavior component must be defined with a set of gates to access the object attributes and to invoke other objects operations (by sending messages on this gates).

Definition 5.6 Given a behavior component bh, by *gate set* of bh (written *GSet*(bh)) we mean the union of set of the gates obtained form the names of the operations and attributes that bh accesses. More formally: $GSet(bh)=OP(bh)\cup Vars(bh)$.

Behavior component identifiers

In our version of OBLOG, behavior components are unnamed (anonymous) entities. In LOTOS every process declaration must have a name. For this reason, we will use a mapping that assigns unique names to every behavior component. The mapping is defined as $Id: bhcomponent \to ProcName$. Where *ProcName* is the set of process names.

Backup and Restore templates

These templates are use to make the presentation of the semantics simpler and will be used later in the *rollback* process in the semantics of alternative composition.

Definition 5.7 Given a set of attribute gates $at_1, \ldots, at_n$ and a gate 'c' we define backup and restore templates in the following way:

- *Backup*($at_1,\ldots,at_n$, c) ≡ `at`$_1$`!READ?v`$_1$`; …;at`$_n$`!READ?v`$_n$`; c!v`$_1$`!…!v`$_n$
- *Restore*($at_1,\ldots,at_n$, c) ≡ `c?v`$_1$`?…?v`$_n$`; at`$_1$`!WRITE!v`$_1$`;…;at`$_n$`!WRITE!v`$_n$

where each v_i is a variable used to hold the a value received or sent through a gate.

We are now in position the give the rule for translating OBLOG specification into LOTOS specifications. The rules are given in a denotational way, having a set of semantic preconditions on the left side.

Skip

The rule for *skip* is constituted of a process that always terminates successfully by delivering *false* as recover value.

Semantic pre-conditions	*Denotation*
pid = *Id*(**skip**)	⟦ **skip** ⟧ ≡

```
process pid : exit(Bool) :=
  exit(false)
endproc
```

Fail

The rule for *fail* produces a process that provokes a recovery by signalling a *true* on the δ gate.

Semantic pre-conditions	*Denotation*
pid = *Id*(**fail**)	⟦ **fail** ⟧ ≡

```
process pid : exit(Bool) :=
  exit(true)
endproc
```

Expression Evaluation

The *expression evaluation* process is intended to encapsulate the evaluation of an expression. The values of the attributes $A_1, \ldots, A_n$ are read and put into a set of variables $v_1, \ldots, v_n$. The resulting value is built from the evaluation of *Expr*, an expression using only object attributes and local variables. Evaluating an expression never fails.

Semantic pre-conditions:

pid = *Id*(Expr)
gset = *GSet*(Expr)
$\{A_1, \ldots, A_n\}$ = *I*(Expr)
S_i = *SORT(Expr)*(A_i)
ExprType = *SORT*(*Expr*)

Denotation

⟦ Expr ⟧ ≡

```
process pid[gset]: exit(ExprType) :=
  A1!READ?v1:s1;…;An!READ?vn:sn;
    exit(Expr[vn/An])
endproc
```

Assert

The *assert* component is modeled as a process that evaluates the condition 'C' (a Boolean expression) and then it behaves as *skip* (by sending *false* as recover value) or as *fail* (by sending *true* as recover value). Note that the translation of the process performing the evaluation of condition 'C', denoted by ⟦Expr⟧, is nested the in the **where** section below, as a sub-process declarations. In subsequent rules we will also follow this pattern.

Semantic pre-conditions

pid = *Id*(**assert** C)
pexpr = *Id*(Expr)
gset = *GSet*(**assert** C)
exprgset = *GSet*(C)

Denotation

```
⟦ assert C ⟧ ≡
process pid[gset] : exit(Bool) :=
  pexpr[exprgset] >> accept bval in
  (
   [bval eq true] -> exit(true)
   []
   [bval eq false] -> exit(false)
  )
where
 ⟦ Expr ⟧
endproc
```

Set

The *set* component is modeled by a process that evaluates *Expr* and writes the result in the state of the object. The gate 'X' is included in 'gset'.

Semantic pre-conditions

pid = *Id*(**set** *[…]*)
pexpr = *Id*(Expr)
gset = GSet(**set** *[…]*)
exprgset = GSet(Expr)

Denotation

```
⟦ set X << Expr ⟧ ≡
process pid[gset] : exit(Bool) :=
  pexpr[exprgset] >> accept
  rval in X!WRITE!result; exit(false)
where
 ⟦ Expr ⟧
endproc
```

Send

The *send* component results in a process that evaluates all expressions used as parameters and sends all values through the 'opId' gate.

Semantic pre-conditions

pid = *Id*(**send** *[…]*)
$pexpr_1$ = *Id*($Expr_1$)
…
$pexpr_n$ = *Id*($Expr_n$)
gset = *GSet*(**set** *[…]*)
$exprgset_1$ = *GSet*($Expr_1$)
…
$exprgset_n$ = *GSet*($Expr_n$)

Denotation

```
⟦ send Expr1, …, Exprn in opId ⟧ ≡
process pid[gset] : exit(Bool) :=
  pexpr1[exprgset1] >> accept res1 in
  …
  pexprn[exprgsetn] >> accept resn in
      opId!res1!…!resn; exit(false)
where
 ⟦ Expr1 ⟧
 …
 ⟦ Exprn ⟧
endproc
```

Receive

The *receive* component receives a set of formal variables from 'opId'. This component also receives an extra parameter 'v_r' allowing to receive a remote recover status. This feature is used to code synchronous calls with failure, this component fails if it receives a recover value of *true*. If the recover value is *false*, the remaining values are committed to a set of object attributes and/or local variables. An optimization could be worked out: It is possible to discover channels that will never be used to transmit failure information by type analysis on operation channels. We could then supply a different coding for receiving in the channels, skipping the recover value test.

Semantic pre-conditions

pid = *Id*(**receive** *[…]*)
gset = *GSet*(**receive** *[…]*)
$\{A_1, \ldots, A_n\} = O(\text{Expr})$
$S_i = SORT(\textbf{receive} [\ldots])(A_i)$

Denotation

```
⟦ receive v1, …, vn in opId ⟧ ≡
process pid[gset] : exit(Bool) :=
  opId?v1:s1?…?vn:sn?vr:Srecover;
  (
    [vr eq true] -> exit(true)
    []
    [vr eq false]-> A1!WRITE!v1:s1;…;
An!WRITE!vn:sn; exit(false)
  )
endproc
```

Iteration

The coding of the *iteration behavior* component consists in a process behaving as a while loop operational semantics rule. This process evaluates 'Expr', and while it is *true* behaves like 'BhComp'. The denotation includes the nesting of ⟦Expr⟧ and ⟦BhComp⟧. If 'BhComp' fails, the *Iteration* component also fails as illustrated by the code fragment that evaluates the recover value of the process representing 'BhComp'. Note also that, like in other components, *exprgset*⊆*gset* and *bhcompgset*⊆*gset*.

Semantic pre-conditions

pid = *Id*(**while** *[…]*)
pexpr = *Id*(Expr)
bhcompid = *Id*(BhComp)
gset = *GSet*(**while** *[…]*)
exprgset = *GSet*(Expr)
bhcompgset = *GSet*(BhComp)

Denotation

```
⟦ while Expr do BhComp ⟧ ≡
process pid[gset] : exit(Bool) :=
  pexpr[exprgset] >> accept bval in
  (
   [bval eq false] -> exit(false)
   []
   [bval eq true] -> bhcompid[bhcompgset]
>> accept rval1 in
      [rval1 eq true] -> exit(true)
      []
      [rval1 eq false] -> pid[gset] >>
accept rval2 in exit(rval2)
  )
where
 ⟦ Expr ⟧
 ⟦ BhComp ⟧
endproc
```

Sequential Composition

The *sequential composition* component is coded by a process containing two sub-processes. After exhibiting the behavior of the first sub-process corresponding to ⟦$\mathtt{BhComp}_1$⟧, the main process checks whether it is needed to recover or not, if yes it sets recover value to *true*, otherwise it executes ⟦$\mathtt{BhComp}_2$⟧ and returns its recover value.

Semantic pre-conditions	*Denotation*
pid = *Id*($BhComp_1$ **;** $BhComp_2$) pid_1 = *Id*($BhComp_1$) pid_2 = *Id*($BhComp_2$) gset = *GSet*($BhComp_1$ **;** $BhComp_2$) $gset_1$ = *GSet*($BhComp_1$) $gset_2$ = *GSet*($BhComp_2$)	⟦ $BhComp_1$; $BhComp_2$ ⟧ ≡

```
process pid[gset] : exit(Bool) :=
  pid1[gset1] >> accept rval in
      [rval1 eq true] -> exit(true)
      []
      [rval1 eq false] -> pid2[gset2]
>> accept rval2 in exit(rval2)
where
 ⟦ BhComp1 ⟧
 ⟦ BhComp2 ⟧
endproc
```

Alternative Composition

The *alternative composition operator* is coded by analyzing the recover value of ⟦$\mathtt{BhComp}_1$⟧ and using a backup/restore discipline. Firstly, the values of those variables altered inside '$BhComp_1$' are backed up in the process denoted *Backup*(*O*($BhComp_1$), c) that will run in parallel with the body of the alternative composition. After executing ⟦$\mathtt{BhComp}_1$⟧ its recover value is analyzed: if *true* this means that this component failed and ⟦$\mathtt{BhComp}_2$⟧ is executed after a restore process denoted *Restore*(*O*($BhComp_1$), c). As a first attempt to code the alternative composition one might feel tempted to use the LOTOS disabling operator '[⟩'[14], where 'P [⟩ Q' means that P executes as long as Q is not enabled to take any transitions. The strategy to use the disabling operator consists in sharing some

Semantic pre-conditions	*Denotation*
pid = *Id*($BhComp_1$ **alt** $BhComp_2$) pid_1 = *Id*($BhComp_1$) pid_2 = *Id*($BhComp_2$) gset = *GSet*($BhComp_1$ **alt** $BhComp_2$) $gset_1$ = *GSet*($BhComp_1$) $gset_2$ = *GSet*($BhComp_2$) $ogset_1$ = *O*($BhComp_1$)	⟦ $BhComp_1$ **alt** $BhComp_2$ ⟧ ≡

```
process pid[gset] : exit(Bool) :=
  hide c in
    Backup(ogset1, c)|||
    (
      pid1[gset1] >> accept rval1 in
       [rval1 eq true] -> Restore(ogset1,
c); pid2[gset2] >> accept rval2 in
exit(rval2)
        []
        [rval1 eq false] -> exit(false)
     )
where
 ⟦ BhComp1 ⟧
 ⟦ BhComp2 ⟧
endproc
```

[14] CSP interruption operator

distinguished recovery channel ‘c’ among P and Q. When P signals its failure through ‘c’, it automatically enables Q (and disables P by the semantics of ‘[〉’). However, because of the semantics of this operator, many ‘undesirable’ transitions are produced. The LOTOS compiler produces a transition of the form $p_i \rightarrow^a q_{init}$ for each transition $p_i \rightarrow^a p_j$ where q_{init} is the starting state of the LTS representing Q and p_i and p_j are states of P.

Choice composition

The *choice composition* is coded in a straightforward way making use of the ‘[]’ operator. One of the processes is selected to run and the recover value is the one of the process selected.

Semantic pre-conditions	*Denotation*

pid = *Id*(**or** *[…]*)
pid_1 = *Id*($BhComp_1$)
…
pid_n = *Id*($BhComp_2$)
gset = *GSet*(**or** *[…]*)
$gset_1$ = *GSet*($BhComp_1$)
…
$gset_n$ = *GSet*($BhComp_n$)
	⟦ **or** ($BhComp_1$ … $BhComp_n$) ⟧ ≡
	process pid**[**gset**] : exit(**Bool**) :=**
	   pid_1**[**$gset_1$**]**
	  []
	   …
	  []
	   pid_n**[**$gset_n$**]**
	where
	 ⟦ $BhComp_1$ ⟧
	 …
	 ⟦ $BhComp_n$ ⟧
	endproc

Local Variables

Local variables are coded as a process that synchronizes in gates that correspond to variable names. The process is recursive and it is constituted of choices between actions to read variables of the form *VarName!READ!atom* and actions to write new values in variables of the form *VarName!WRITE!newValVariable*

Semantic pre-conditions

pid = *Id* (**local** *[…]*)
gset = *GSet*(**local** *[…]*)
bhcompid = *Id* (BhComp)
gsetbhcomp = *GSet*(BhComp)
S_i = *SORT*(**local** *[…]*) (A_i)

Denotation

```
⟦ local A1 : S1 = I1, …, An : Sn = In in BhComp ⟧ ≡
 process pid[gset] : exit(Bool) :=
  hide A1,…,An, Break in
  (
   lvars[A1,…,An, Break](I1,…,In)|[A1,…,An]|
    (
      bhcompid[bhcompgset] >> accept
rval:Bool in
      (
         Break!go;
         exit(rval)
      )
where
  process lvars[{A1,…,An}|p]({A1:I1,…,An:In}|v) :
exit :=
      A1!WRITE?XA1:S1;
lvars[{A1,…,An}|p]({A1,…,An}|v[XA1/A1])
     []
      …
     []
      An!WRITE?XAn:Sn;
lvars[{A1,…,An}|p]({A1,…,An}|v[XAn/An])
     []
      A1!READ!V1; lvars[{A1,…,An}|p ]({A1,…,An}|v)
     []
      …
     []
      An!READ!Vn; lvars[{A1,…,An}|p]({A1,…,An}|v)
     []
      Break; exit
  endproc
  ⟦ BhComp ⟧
endproc
```

Alternative or (XOR) composition

The *alternative or* operator combines *alt* and *or* operators as described in Chapter 4. In order to code the successive elimination of those behavior components that failed, we use a recursive process with a set of flags used as Boolean guards in a choice. When a behavior component fails, the modified attributes are restored and the process starts again with the flag ‘$comp_i$’ corresponding to the behavior component ‘$BhComp_i$’ set to *false*. This strategy leaves ‘$BhComp_i$’ out from the next round. If in the end, all behavior components fail, the alternative composition of all of them also fails.

Semantic pre-conditions

pid = *Id*(**xor** *[…]*)
pid_1 = *Id*($BhComp_1$)
…
pid_n = *Id*($BhComp_n$)
gset = *GSet*(**xor** *[…]*)
$gset_1$ = *GSet*($BhComp_1$)
…
$gset_n$ = *GSet*($BhComp_n$)
$ogset_1$ = *O*($BhComp_1$)
…
$ogset_n$ = *O*($BhComp_n$)

Denotation

```
⟦ xor (BhComp1 … BhCompn) ⟧ ≡
process pid[gset] : exit(Bool) :=
  xorbody[gset](true,…,true)
where
  process xorbody[gset](comp1:Bool, …,
compn:Bool) : exit(Bool)
     [comp1 eq true] ->
       hide c in
         Backup(ogset1, c)|[c]|
         (
          pid1[gset1] >> accept rval1 in
           [rval1 eq true] ->
            Restore(ogset1, c);
            xorbody[gset2](false, …, compn)
             >> accept rval2 in exit(rval2)
           []
           [rval1 eq false] -> exit(false)
         )
     []
      …
     []
     [compn eq true] ->
        hide c, r1 in
         Backup(ogsetn, c)|[c]|
         (
          pidn[gsetn] >> accept rval1 in
           [rval1 eq true] ->
            Restore(ogset1, c);
            xorbody[gset2](comp1, …, false)
             >> accept rval2 in exit(rval2)
           []
           [rval1 eq false] -> exit(false)
         )
     [(comp1 eq true) and … and (compn eq
true)] -> exit(true)
where
 ⟦ BhComp1 ⟧
 …
 ⟦ BhCompn ⟧
endproc
```

Objects

We will code objects as processes representing the local memory constituted by the attributes and running in parallel with another process that is a choice on operations. After an operation is called and executed we return to the choice of operations again.

Semantic pre-conditions

pid = *Id*(**object** *[…]*)
gset = *GSet*(**object** *[…]*)
$gsetop_1$ = *GSet*(**operation** op_1 *[…]*)
…
$gsetop_n$ = *GSet*(**operation** op_n *[…]*)

Denotation

```
⟦ object
   attribute A1 : S1 = I1
   ...
   attribute An : Sn = In
   operation OpId1 is […]
   ...
   operation OpIdn is […]
⟧ ≡
process pid[gset] : noexit :=
   lvars[A1,…,An](I1,…,In)|||
   (
      opid1[gsetop1]
     []
      …
     []
      opidn[gsetopn]
   ) >> accept rsilence:Bool in
       pid[gset]
where
   process  lvars[{A1,…,An}|p]({A1:I1,…,An:In}|v)
: noexit :=
      A1!WRITE?XA1:S1;
lvars[{A1,…,An}|p]({A1,…,An}|v[XA1/A1])
     []
      …
     []
      An!WRITE?XAn:Sn;
lvars[{A1,…,An}|p]({A1,…,An}|v[XAn/An])
     []
      A1!READ!V1; lvars[{A1,…,An}|p
]({A1,…,An}|v)
     []
      …
     []
      An!READ!Vn;
lvars[{A1,…,An}|p]({A1,…,An}|v)
   endproc
   ⟦ operation OpId1 is […] ⟧
   ...
   ⟦ operation OpIdn is […] ⟧
endproc
```

System

The system process is obtained as the parallel composition of processes corresponding to objects. The gates (opg_i) included in each parallel composition operator are obtained by a simple procedure of intersecting the union of gates of process $P_1, \ldots, P_i$ with the gates of the processes $P_{i+1}, \ldots, P_n$, where n is the number of objects in the system.

Semantic pre-conditions	*Denotation*

pid = *Id*(**object** *[…]…[…]*)
gset = *GSet*(**object** *[…]*)
$gset_1$ = *GSet*(**operation** op_1 *[…]*)
…
$gset_n$ = *GSet*(**operation** op_n *[…]*)
$opg_1 = gset_1 \cap (gset_2 \cup \ldots \cup gset_n)$
…
$opg_{n-1} = (gset_1 \cup gset_{n-1}) \cap gset_n$
	⟦ **object** o_1 *[...]* ... **object** o_n *[...]* ⟧ ≡
	process pid[gset] **: noexit :=**
	    pid_1[$gset_1$]
	   |[opg_1]|
	   …
	   |[opg_{n-1}]|
	    pid_n[$gset_n$]
	where
	 ⟦ **object** o_1 *[...]* ⟧
	  …
	 ⟦ **object** o_n *[...]* ⟧
	endproc

5.3 Case study – verifying the Alternating Bit Protocol

After translating the OBLOG program into a LOTOS specification (Apendix D) and feeding the CADP toolset, we will check properties from the LOTOS specification using a Temporal Logic Model-Checker.

The limitation of this method resides in the size of the LOTOS specifications accepted by the CADP tools. Our experiments have shown that the translation of the ABP protocol specification of Appendix C with the rules presented in Section 5.2, results in more than 1000 lines of LOTOS code resulting in more than 3 million states. For this reason, the LOTOS specification had to be divided and checked using a technique known as *compositional verification*.

Verification framework

Since the LOTOS specification (a process P) obtained by translation from an OBLOG specification results in a very large LTS, analysis via 'brute force' approach is infeasible. Instead of verifying a property φ over the LTS produced from a process $P \equiv P_1 \,||\, \ldots \,||\, P_n$, we can verify φ over a smaller LTS resulting from $Q \equiv Q_1 \,||\, \ldots \,||\, Q_n$. This process Q is such that $Q \approx P$, where '≈' is an equivalence which preserves the property φ and $Q_i \approx P_i$

Concerning the specifications of properties in CADP, they are usually specified in the Mu-Calculus although Temporal Logic macros are defined in order to alleviate the notation. A limitation arising from the use of the hide operator, (not present in the tool UPPAAL presented in Chapter 6) is that we can not examine the state of the object directly. This introduces some obstacles in formalizing some properties related to the object internal state. To work around this shortcoming some properties were restated.

In order to verify the system $Sys \equiv$ P*sender* || P*ackline* || P*transline* || P*receiver*, we need to avoid the state-space explosion resulting from the parallel composition operators. The idea is to avoid expanding the original LTS for *Sys* and to engage in producing a reduced LTS compositionally. First, we independently produce and reduce modulo safety equivalence, a separate LTS for each process. Second, we perform the parallel composition of the reduced LTSs producing an LTS, which is equivalent to the original for *Sys* according to safety equivalence.

The tool Aldébaran was used for two purposes. To minimize the LTSs obtained from the LOTOS specifications using the CAESAR compiler and to perform the parallel composition of the reduced LTSs. The properties were model-checked using the EVALUATOR tool over the reduced LTS. Aldébaran also helped in earlier phases when the LOTOS specification was translated by hand from the OBLOG ABP specification (Appendix B and C). Behavioral specifications of the processes were developed and Aldébaran was used as a Model-Checker providing counter examples that were used to debug the translation framework and, in some cases, the ABP specification.

CADP was installed in a 450Mhz Pentium computer with 128MB of RAM running the Linux operating system. Verification using compositional verification took only about 2 minutes. Previous experiments without compositional verification, after running an hour did not yield any answer.

Operation of a single component – the acknowledge line

Our 'acknowledge line' is a very simple component. Checking its correct operation amounts to see if it exhibits two expected behaviors:

1. "The line may loose an acknowledge signal"
2. "The line correctly delivers the acknowledge signal in the absence of errors"

The first property states that an 'ackline.getAcknowledge' operation call can *fail* after having sent an *acknowledge signal* by calling then 'ackline.sendAcknowledge' operation. For this property we must check that:

$$Ackline_{spec} \vDash \varphi_{\text{ack}} \text{ where },$$
$$\varphi_{\text{ack1}} \equiv \text{EF}(ackline.sendacknowledge(a_i) \Rightarrow \text{EF}(ackline.getAcknowledge.fail{=}\text{true}))$$

Where a_i is an *acknowledge signal*, either '0' or '1'.

The second property cannot be verified directly because we cannot access the internal state of the process in order to analyze the occurrence of an error in the acknowledge line (if no error occurred then *Ackline.lStatus*=STATGOOD). The property must be restated as the following reachability property: "From those states where the 'acknowledge line' reported to have no errors the *acknowledge signal* is inevitably delivered correctly". To verify this property we checked the following formula:

$$\varphi_{\text{ack2}} \equiv \text{EF}((ackline.sendacknowledge(a_i) \wedge \text{AF}(ackline.getAcknowledge.fail{=}\text{false})) \Rightarrow \text{AF}(ackline.getAcknowledge(a_i))$$

Global Operation of the system

Another class of properties concerns the joint operation of the components. We will now concentrate on the send-receive properties and abstract away from details of individual components. We can verify the following properties.

- "A sent message can be lost and never received"
- "A sent message is always received if there is no error in the transmission line"
- "A received message is acknowledged to the sender if there is no error in the acknowledge line".

The first property states about the unreliability of the protocol. It states that a message sent via 'sender.accept' may be lost and not received when we call 'receiver.deliver'. This property holds if:

$$Sys_{spec} \vDash \varphi \text{ where,}$$
$$\varphi_{global1} \equiv \mathrm{EF}(\mathit{sender.accept}(m_i) \Rightarrow \mathrm{AF}(\mathit{receiver.deliver}(m_j)) \wedge \mathrm{AF}(\mathit{receiver.getStatus}(\mathrm{SAMEMESSAGE})))$$

For the second property, we need to find a way of characterizing the states where there was no error in the transmission line. The transmission of a message from the 'sender' to the 'receiver' objects involves a sequence of calls to the 'transmission line' which we can observe and reason about. We will also restate property 2 as: "A message is always received from states where: (2.1) the message is sent and (2.2) we inevitably get to states where we observe a correct reception form the transmission line."

$$\varphi_{global2} \equiv \mathrm{AF}((\mathit{sender.accept}(m_i) \wedge \mathrm{AF}(\mathit{transline.receiveMessage.fail}=\mathrm{false})) \Rightarrow \mathrm{AF}(\mathit{receiver.deliver}(m_i)))$$

The third property in conjunction with the second will close the send-receive-acknowledge cycle of our protocol. As with the second property, we must also restate this last one in order to capture the states where no error occurred in the acknowledge line as: "A message is acknowledged to the sender from those states where calling *ackline.getAcknowledge* does not fail"

$$\varphi_{global3} \equiv \mathrm{AF}(\mathit{ackline.getAcknowledge.fail}=\mathrm{false}) \Rightarrow \mathrm{AF}(\mathit{sender.getStatus}(\mathit{SENTANDACK}))$$

The properties written in CADP concert syntax can be found in Appendix E.

Other verification strategies in CADP

CADP is a very flexible tool allowing various forms of verification not completely explored in this work. During the verification of the ABP, the task of compositional verification outlined the utility of comparing specifications at different levels of abstraction for verifying safety properties. This was due to the notable incidence of violations of safety properties because implementations frequently exhibit behavior that falls out of that one allowed by the more abstract specification. An error in the

translation of the *alternative behavior* component previously defined with the LOTOS *disabling operator* was detected using behavioral specifications. Yet another related to the *local memory* process was found and corrected. Some comments about different forms of verification possible in CADP are mentioned below.

Behavior specifications. The use of behavior specifications is sometimes regarded as an escape provided by those verification tools for process algebras like LOTOS that do not offer Temporal Logic Model-Checkers. The strategy consists in using an abstract specification to describe a superset of the behavior of the implementation and check if the behavior of the implementation is contained in that previously defined superset (safety checking). On the other hand one can specify a subset of the implementation behavior and check that this behavior is in fact carried out by the implementation (liveness checking). Both the specification and the implementation are specified using the same language. The advantages are that some properties are very naturally specified in the language of the implementation and that modeling of environments (see below) is made easier. Disadvantages are usually related to checking liveness properties, which are often more difficult to specify. A major strength of comparing behavior specifications appears to be detecting errors in message sequencing. Experiments run in this project uncovered some errors both in implementations and specifications.

Environment modeling. From a theoretical point of view, when we are verifying a process P with a set of gates G we are admitting that the process P can receive any possible sequence of messages through gates G. In fact, when verifying the process P with a tool that enumerates the possible states of P like CADP, a huge number of states is produced as result of this interleaving of messages sent by the environment to the process via G. In this situation, we say that P is an *open process*. This approach is only feasible in practical applications when the domains of the messages accepted in the gates are very small, let us say 3 or 4 elements. For this reason we have to abstract the context in which the process operates by modeling the environment with a process E and compose it with P. Some strategies for building environment processes have been advanced, like using the concept of *message generators* (processes that generate messages on gates). These processes have a constraint section that allows incremental addition of constraints has the analysis of the system progresses providing for the incremental specification approach of the environment. The main limitation of this method is ensuring that the model of the environment is general enough and that its is not hiding sequences that could lead our system into undesirable states.

Test process. The use of the *test process* technique is advanced in Chapter 6 as necessary for performing verification of some classes of properties in UPPAAL. Literature related to CADP does not refer this approach although the author believes that its use is straightforward. Safety properties can be coded inside the system specification as *special message detectors*, i.e., processes that make progress when they detect unsafe message sequences and take a special action, say *sigfail*. The verification task consists in detecting whether the system takes *sigfail* actions. Liveness properties can be coded in a similar way. A somewhat close approach is taken in the ARA Toolset [Savola 95] where special actions exist to signal some types of failure of the system. The failure states are

directly coded in the system specification that is compared against a failure model via a special notion of equivalence called *failure equivalence* by Antti Valmari [Valmari & Tienari 95]. The Murφ verifier [Dill 96] takes a similar approach, failure states are also explicitly specified.

Chapter 6 – Verifying OBLOG specifications using Communicating Automata

Model-Checking of real time systems has been intensively studied in the last few years leading to the emergence of tools like HyTech [Henzinger & al. 95], Kronos [Olivero & Yovine 93] and UPPAAL [Bengtsson & al. 95].

In this chapter, we propose a framework for verifying simple safety and liveness properties of OBLOG programs using available verification tools for IO-Automata like UPPAAL. The procedure consists in translating the specification of an object community into a network of communicating timed-automata [Yi & al. 94] and checking properties from it.

The OBLOG language is not a real-time specification language although there are plans to extend it with time primitives. At the current state of affairs we are mainly interested in taking advantage of the very efficient model checking algorithms present in UPPAAL and not so much in verifying timing properties.

6.1 Technical framework

Concurrent communicating state machines are straightforward models of many real world systems. It is common in industry to model concurrent systems as communities of communicating processes whose internal structure is implemented with state machines. This formalism forms the base of some specifications languages like SDL [ITU-T 94].

The tool UPPAAL, is a verification tool for closed real-time systems that has proved to be successful in real world problems specified as networks of communicating timed automata with variables [Larsen & al. 97, Kristoffersen & Petterson 95, Bengtsson & al. 98]. The UPPAAL kernel consists of a collection of very efficient algorithms that are used to perform reachability analysis over a model of the system being verified [Larsen & al. 95, Larsen & al. 95b]. As a result, we are restricted to the verification of *reachability* properties. The escape for this limitation resides in the technique of *test automata*. A property φ is specified as a *test automaton* $T\varphi$, the system enjoys property φ if the test automaton $T\varphi$ never reaches any distinguished reject node when combined with the system specification as $T\varphi \| S$.

Verification tools like UPPAAL accept specifications of nets of Timed IO-Automata as input. The framework of timed IO-Automata can be obtained in two ways, 1) a timed generalisation of classical IO automata like Mealy machines [Springintveld & al. 97] or 2) as an IO version of Timed Automata like those of [Alur & Dill 94]. Alur and Dill extended the standard finite-state automata with a finite collection of real-valued clocks that proceed at the same rate and measure the elapsed time since they were reset.

Definition 6.1 Let K be a set real-valued clock identifiers. We define the set of *clock constraints* over K, also written $\Phi(K)$, as being the set of formulas generated by the grammar $\varphi ::= k \leq c \mid c \leq k \mid k < c \mid c < k \mid \varphi_1 \wedge \varphi_2$, where c is a constant and k a clock variable.

Definition 6.2 A *timed automaton* is a tuple $A = \langle L, L_0, \Sigma, K, I, E \rangle$ where

- L is a finite set of locations (also called states or nodes)
- $L_0 \subseteq L$ is the set of initial locations
- Σ is a finite set of labels (also called actions)
- K is a finite set of clocks
- I is a mapping that assigns a clock constraint in $\Phi(K)$ (also called invariant condition) to every location of L.
- $E \subseteq L \times \Sigma \times 2^K \times \Phi(K) \times L$ is called the set of switches. A switch $\langle l_1, a, r, \lambda, l_2 \rangle$ represents a transition from location l_1 to location l_2 by performing action a. r is a clock constraint over K that specifies when the switch is enabled and the set $\lambda \subseteq K$ specifies the set of clocks to be reset.

Below, in Figure 6.1 we present an example of a timed automaton. The example intends to model a very simple device – a *timed ringer*. The ringer should ring regularly for 60 time units in specified intervals. When ringing, the user may push the *stop-ringer* button or ear the ringer for 60 time units. The variable m_{clk} is the main clock and r_{clk} is the ringer clock. The purpose of the first one is to start the ringer and the second measures the elapsed time since the ringer started in order to stop it.

In order to have a more expressive language and ease the modelling task, an extension of timed automata with more data types (like Integers and Booleans) is introduced in the UPPAAL verification tool [Larsen & al. 97] extending Alur and Dill initial model that only provided clock variables. One could argue that introducing variables could cause the undecidability of the Model-Checking procedure if we allow their domains to be infinite. However, upper and lower bounds on the variable domains are established in order to guarantee the termination of the verification procedure.

When verifying OBLOG programs, we are not particularly interested in time but, instead, in variables of sorts other than clocks. Because of this, a model of communicating automata will be used, where time variables are abstracted away and other sorts of variables are introduced as in Definition 6.3.

Program properties in UPPAAL must be specified using a restricted version of TCTL based on [Larsen & al. 95] whose syntax is presented in Definition 6.4. Currently, the UPPAAL model checker only allows specification of properties with nested of modal operators for specifying *reachability* properties.

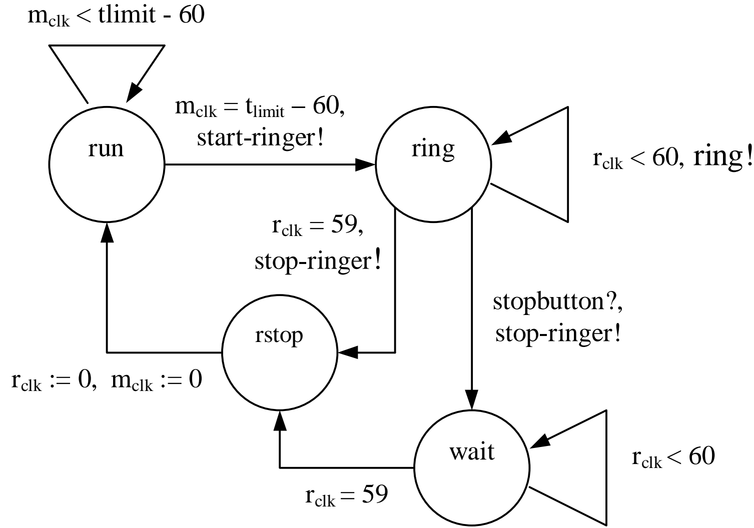


Figure 6.1 – A timed automaton example of ringer

Definition 6.3 An *extended time-abstracted automaton* (or ETA) is a tuple $A=\langle L, l_i, l_f, \Sigma, V, E\rangle$ where

- L is a finite set of locations (also called states or nodes)
- $l_i \in L$ is the initial location
- $l_f \in L$ is the final location
- Σ is a finite set of labels (also called actions)
- V is a finite set of variables of different sorts.
- $E \subseteq L\times\Sigma\times\Gamma(V)\times 2^{\Lambda(V)}\times L$ is called the set of switches. A switch $\langle l_1, a, \gamma, \lambda, l_2\rangle$ represents a transition from location l_1 to location l_2 by performing action a, γ is a constraint over V that specifies when the switch is enabled and the set λ is a variable assignment in V specifies the new values for variables.

Definition 6.4 Assume K is finite set of clocks. The set of formulas over K is defined by the following abstract syntax where c is a clock constraint over K, A_i is a component of the system (an automaton) and $l\in L$ a location.

$\varphi ::= \forall\Box\beta \mid \exists\Diamond\beta$
$\beta ::= a \mid \beta_1 \wedge \beta_2 \mid \neg\beta$
$a ::= c \mid A_i$ **at** l

Intuitively $\forall\approx\beta$ means that β must be satisfied in every reachable state, $\exists\Diamond\beta$ means that β should be satisfied in some reachable state. The atomic formulas are either propositions over clock variables of the form $x \sim y$ where $\sim \in \{<, \leq, \geq, >, =\}$ or of the form A_i **at** l meaning that the Automaton A_i reached location l.

As explained in [Bengtsson & al. 95], the above logic is a restriction of the one presented in [Larsen & al. 95] and it is not possible to express some desirable properties with it, for example bounded liveness properties. In UPPAAL the problem is overcome unto so extent by using a technique known as *test automata*. Given a property φ to Model-Check, the user should provide a test automaton for it. The test automaton $T\varphi$

should be such that the system specification S enjoys the property φ if S||Tφ (S interacting with Tφ) never reaches any bad states of Tφ. Constructing test automata from temporal properties is a tedious and error-prone task. In the work of [Aceto & al. 98] a process for automatic compilation of test automata form real-time logic formulas is presented. Using this logic is possible to specify progress properties in a more elegant way, skipping the need for the construction of test automata.

Experimental comparisons of UPPAAL with other tools have been carried out, as shown in e.g. [Larsen & al. 95b]. These experiments showed that UPPAAL is not only faster but is also able to verify systems with a higher number of processes. From the technical point of view, both HyTech and Kronos first compute the state-space resulting from the automata network before carrying out the verification process. Instead of doing this, UPPAAL produces the state space on-the-fly. The difference is a time versus space trade-off from which UPPAAL is earning the best share. First computing the state space provides for the reuse of states, thus avoiding to compute the same state several times. This approach has a high cost in terms of space. On the other hand, adopting on-the-fly approaches enables the verification of properties computing only a small fraction of the state-space.

Another issue is related with the expressiveness of the underlying logic: Kronos documentation points for full TCTL support whereas UPPAAL documentation explicitly refers the support of restricted versions of TCTL or timed Mu-calculus. Expressiveness is sacrificed yielding a faster verification procedure. In a recent work, Aceto [Aceto & al. 98b] presented an expressive logic that goes beyond the *reachability* properties expressible with the logic of Definition 6.4. A presentation on the full TCTL and timed Mu-calculus can be found in [Henzinger & al. 92].

6.2 Translating OBLOG programs into IO-Automata

In the work of Yi [Yi & al. 94] a CCS-like algebra of processes with clocks is presented. This algebra may serve as a formal description language for real-time communicating systems. A procedure for translating this timed process algebra into communicating timed automata is also presented deriving in this way a parallel composition operator for timed-automata which can be used to construct more complex system descriptions from components.

We adapt Yi's ideas of structural description of real-time systems in a CCS-like language to the OBLOG specification language. In our framework, objects and behavior components describe automata. Composition of behavior components is modeled as composition of automata.

It could be argued that translating OBLOG specification into CCS could constitute an easier task than performing a translation to timed automata. A closer look at OBLOG will reveal that a translation to CCS would raise more questions than it would solve. It would be difficult to model the OBLOG notions of *failure* and *alternative behavior composition*. Instead, the solution adopted was handcoding the notion of failure and introducing a new composition operator between automata, the *recovery composition operator*.

We will now introduce some preliminary notions that will be used to specify the rules allowing the translation of OBLOG specifications into ETAs.

Distinguished no-action symbol

When labelling edges, it is be possible to specify that no action is taken. This is accomplished through the distinguished action 0. Please note that this action is not the same as the invisible action τ. By definition all nodes have an edge with label (tt, 0, $\emptyset$) onto themselves.

Sequential composition operator '$\rightarrow$'

Sequential composition of two automata means wiring the end state of the first to the initial state of the second one. The behavior produced in this manner corresponds to the sequential composition between OBLOG commands, "execute the first command and then execute the second".

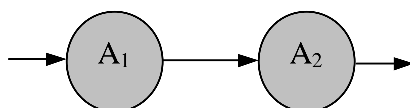


Figure 6.3 – Automaton template for the sequential behavior component

Definition 6.5 Given two automata A_1 and A_2 we define *sequential composition of A_1 and A_2*, symbolically $A_1 \rightarrow A_2$, in the following way: Let $A_1 = \langle L_1, l_{1i}, l_{1f}, \Sigma_1, V_1, E_1\rangle\rho_1$ and $A_2 = \langle L_2, l_{2i}, l_{2f}, \Sigma_2, V_2, E_2\rangle\rho_2$ then $A_1 \rightarrow A_2 = \langle L_1 \cup L_2, l_{1i}, l_{1f}, \Sigma_1 \cup \Sigma_2, V_1 \cup V_2, E\rangle\rho$, where $E = E_1 \cup E_2 \cup \{(l_{1f}, 0, tt, \emptyset, l_{2i})\}$ and $\rho = \rho_1 \cup \rho_2$.

Recovery composition operator $|_\rho$

When giving semantics to OBLOG alternative command constructs of the form `P` **`alt`** `Q`, the idea is to produce an automaton that wires possible failure nodes (also named *recovery locations*) of P to the start of Q. In Figure 6.4, a simplified sketch of this procedure is shown. Grey nodes are used to represent the recovery locations like those of **`assert`** commands.

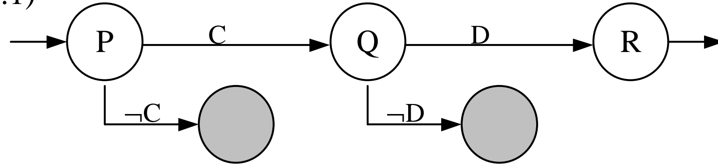
Definition 6.6 Given two automata A_1 and A_2 and a set ρ of recovery locations, we define *recovery composition of A_1 and A_2*, symbolically $A_1/_\rho A_2$, in the following way: Let $A_1 = \langle L_1, l_{1i}, l_{1f}, \Sigma_1, V_1, E_1\rangle\rho_1$ and $A_2 = \langle L_2, l_{2i}, l_{2f}, \Sigma_2, V_2, E_2\rangle\rho_2$ then $A_1/_\rho A_2 = \langle L_1 \cup L_2, l_{1i}, l_{1f}, \Sigma_1 \cup \Sigma_2, V_1 \cup V_2, E\rangle\rho_2$, where $E = E_1 \cup E_2 \cup \{(l_r, 0, tt, \emptyset, l_{2i}), (l_{2f}, 0, tt, \emptyset, l_{1f})$ such that $l_r \in \rho_1\}$.

When two automata A_1 and A_2 are composed with the recovery composition operator, the recovery locations specified in ρ_1 are *instantiated* with the automaton A_2, in the sense that the edges reaching recovery locations are wired to the initial location of A_2. Both final nodes of A_1 and A_2 are wired to a new final node (see Figure 6.4).

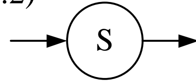
For simplicity we will omit the ρ symbol in $\langle L, l_i, l_f, \Sigma, V, E\rangle\rho$ when $\rho=\emptyset$.

(1) `(P; assert C; Q; assert D; R) alt S`

(2.1)



(2.2)



(3)

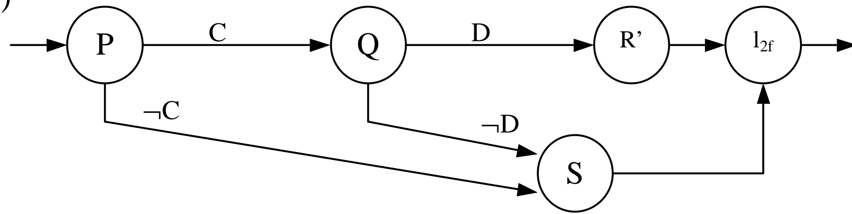


Figure 6.4 – Recovery composition operator procedure. (1) OBLOG specification fragment. (2.1) representation of the first branch of the **alt** command with two failure points. (2.2) representation of the second branch. (3) automata representation of (1) with S being used to recover from `(P; assert C;Q; assert D; R)`

Operations associated memory

We further extend our model by associating a memory region with each operation. This memory region is a set of variables that will be used to allow parameter passing. These variables are associated with operations through a mapping that returns a set of variables for each operation, defined as $AssocMem\text{:}\mathrm{OP} \rightarrow 2^{\mathrm{VAR}}$.

Backup automaton

In order to give semantics to alternative composition, constructs are needed to backup data in order to code the notion of *rollback*. Rollback is coded using a backup-restore discipline and will be detailed later in this document. We present the notion of *Backup automaton* below:

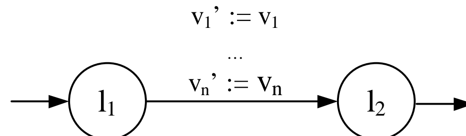


Figure 6.5 – Sketch of the Backup automaton

Definition 6.7 Given a vector of variables V, we define *Backup automaton* over V, written *Back(V)*, as being the automaton $B = \langle \{ l_1, l_2 \}, l_1, l_2, \varnothing, V \cup V', E \rangle$ where V' is a set of fresh primed versions of the variables in V and $E = \{(l_1, 0, tt, \{ v' := v \}, l_2)\}$ for every $v \in V$ and $v' \in V'$, the corresponding primed version of v.

Restore automaton

The dual notion of Backup automaton is the *Restore automaton*. This automaton will be used later to restore the values of the variables to perform the rollback process.

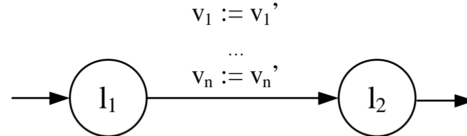


Figure 6.6 – Sketch of the Restore automaton

Definition 6.8 Given a vector of variables V, we define the *Restore automaton* over V, written *Rest(V)*, as being the automaton $R = \langle \{ l_1, l_2 \}, l_1, l_2, \varnothing, V \cup V', E \rangle$ where V' is a set of fresh primed versions of the variables in V and $E = \{(l_1, 0, tt, \{ v := v' \}, l_2)\}$ for every $v \in V$ and $v' \in V'$, the corresponding primed version of v.

With the above notions we are now able to present the translation rules more succinctly. We will start with the atomic components, *skip*, *fail*, *assert*, *set*, *send*, *receive* and, the more elaborate components like *iteration*, *sequence composition*, *alternative composition*, *xor composition* and *or composition*.

Skip

The *skip* component is composed of only one node. Optionally we could think of coding *skip* without any nodes. This could cause some problems, for example, in the alternative composition operator we would have to treat *skip* as a special case.

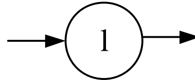


Figure 6.7 – Automaton for the skip behavior component

$[\![\texttt{skip}]\!] = \langle \{l\}, l, l, \varnothing, \varnothing \rangle$

Fail

The presence of two extra nodes in the *fail* component is necessary in the context of alternative composition. The coding could be done using only one node to introduce the failure location but, as with *skip*, we would cause extra trouble in the rule for alternative composition operator.

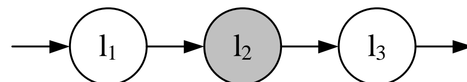


Figure 6.8 – Automaton for the fail behavior component

⟦**fail**⟧ = $\langle\{l_1, l_2, l_3\}, l_1, l_3, \varnothing, E\rangle\rho$ where $E = \{(l_1, 0, tt, \varnothing, l_3), (l_1, 0, tt, \varnothing, l_2)\}$and $\rho=\{l_2\}$.

Assert

The *assert* behavior component tests a condition C and then behaves as *skip* or as *fail* depending on the *truth value* of C.

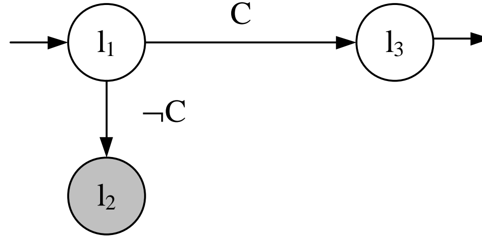


Figure 6.9 – Automaton for the assert behavior component

⟦**assert** C⟧ = $\langle\{l_1, l_2, l_3\}, l_1, l_3, \varnothing, E\rangle\rho$ where $E = \{(l_1, 0, C, \varnothing, l_3), (l_1, 0, \neg C, \varnothing, l_2)\}$, V=*Vars(*C*)* and $\rho=\{l_2\}$.

Set

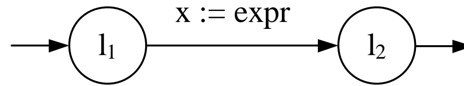


Figure 6.10 – Automaton for the set behavior component

⟦**set** x **<<** expr⟧ = $\langle\{l_1,l_2\}, l_1, l_2, V, \{(l_1, 0, tt, \{x:=expr\}, l_2)\}\rangle$ and V=*Vars(*set x << expr*)*.

Send

The automaton for the *send* behavior component works as follows: It calls the operation by synchronising on channel op_{call}, it then stores the values of the expressions in the associated memory region of the operation and waits for the parameters to be read, synchronising on the channel op_{parm}.

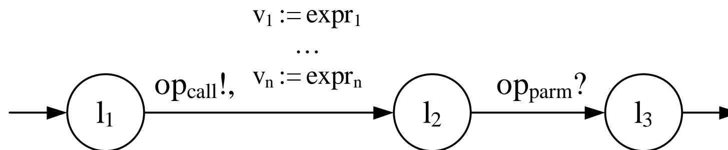


Figure 6.11 – Automaton for the send behavior component

⟦**send** *exprlist* **in** op⟧ = $\langle\{l_1,l_2, l_3\}, l_1, l_3, \{op_{call}!, op_{parm}?\}, V, E\rangle$, where $v_i \in$ *AssocMem(*op*)*, $expr_i \in$ *exprlist* and

- $V = \cup(Vars(expr_i) \cup \{v_i\})$;

- $E = \cup(\{(l_1, op_{call}!, tt, \{v_i := expr_i\}, l_2)\}) \cup \{(l_2, op_{parm}?, tt, \emptyset, l_3)\}$.

Receive

The receive behavior component is the dual of the send component. The receive component copies the values from the operation's associated memory (a_x) to the *varlist* (v_x) as below (where $x \in \{1..n, recover\}$).

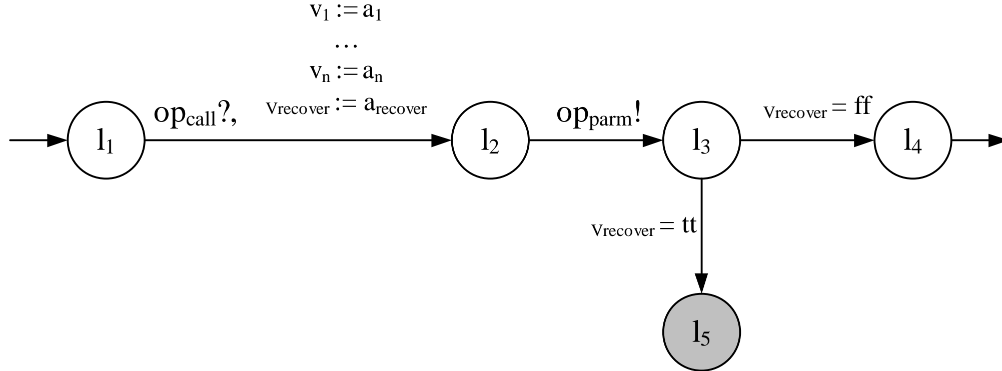


Figure 6.12 – Automaton for the receive behavior component

⟦**receive** *varlist* **in** op⟧ $= \langle\{l_1,l_2,l_3,l_4,l_5\}, l_1, l_4, \{op_{call}?, op_{parm}!\}, V, E\rangle\rho$, where $v_i \in$ *varlist*, $a_i \in$ *AssocMem(*op*)*, $\rho=\{l_5\}$and:

- $V = \cup(\{v_i\}\cup\{a_i\}) \cup\{v_{recover}, a_{recover}\}$;
- $E = \cup(\{(l_1, op_{call}?, tt, \{v_i := a_i\}, l_2)\}) \cup \{(l_2, op_{parm}!, tt, \emptyset, l_3), (l_3, 0, v_{recover}=ff, \emptyset, l_4), (l_3, 0, v_{recover}=ff, \emptyset, l_5)\}$.

Coding the *receive* component involves a further peculiarity. A potential failure may arise in case the component receives a failure notification from the operation. The failure is reported in the extra parameter $v_{recover}$ (recover value).

Iteration

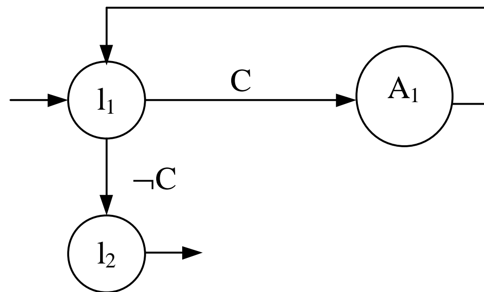


Figure 6.13 – Automaton template for the Iteration behavior component

⟦**while** C **do** P⟧ $= \langle L, l_1, l_2, \Sigma, V, E \rangle$ where ⟦P⟧ $= A_1 = \langle La, la_i, la_f, , \Sigma a, Va, Ea\rangle$ and

$L = \{l_1,l_2\} \cup La$, $\Sigma = \Sigma a$, $V=Vars(C) \cup Va$, $E = \{(l_1, 0, C, \emptyset, la_i), (l_1, 0, \neg C, \emptyset, l_2), (la_f, 0, tt, \emptyset, l_1)\} \cup Ea$.

Sequential composition

The *sequential composition* behavior component may be obtained in straightforward way as the sequential composition of two extended timed automata. If $A_1 = [\![P]\!]$ and $A_2 = [\![Q]\!]$ then the template is the one of Figure 6.9.

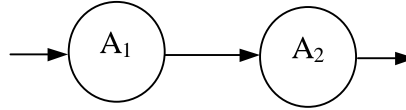


Figure 6.14 – Automaton template for the sequential behavior component

$[\![\texttt{P;Q}]\!] = [\![\texttt{P}]\!] \rightarrow [\![\texttt{Q}]\!]$

Alternative composition

The alternative behavior component translation is somewhat elaborated. Let us recall that the intended semantics of `P alt Q` is that "if P fails then Q should execute as if P never executed". In order to achieve this we must provide means for backing up and restoring the data that will be used by Q.

$[\![\texttt{P alt Q}]\!] = (Back(\ O(\texttt{P})\ ) \rightarrow [\![\texttt{P}]\!])\ \ /_\rho\ \ (Rest(\ O(\texttt{P})\ ) \rightarrow [\![\texttt{Q}]\!])$ where $O(\texttt{P})$ represents the variables where the information produced P is stored as in Definition 3.3.

This semantic equation states that the operation of the alternative composition operator is obtained as the recovery composition of the resulting automata of P and Q, backing up the variables used by Q before executing P and restoring them in case of failure before starting to execute Q.

Alternative or (XOR) composition

In the absence of failure, the *xor* behavior component behaves like a kind of *multiway CSP choice operator*[15]. When one of the operands fails, say bh_k, in **xor**(bh_1, …, bh_{k-1}, bh_k, bh_{k+1}, …, bh_n), the state is rolled back and the *xor* component behaves like **xor**(bh_1, …, bh_{k-1}, bh_{k+1}, …, bh_n). When no behavior is left, the *xor* component fails.

A schema like that of Figure 6.15 can emulate this. In the beginning, the set of flags $component_1$, …, $component_n$ is reset. When a branch corresponding to a behavior component is chosen, the flag $component_k$ is set to true asserting that the behavior component bh_k has been selected. If bh_k executes (represented by an automaton A_k) then the control reaches location l_3, otherwise is sent back to l_2, but this time bh_k is no longer

[15]In CSP the choice operator is binary. By a *multiway choice operator* we mean an n-ary (for $n \geq 2$) operator, written in prefix notation as $[](P_1,\ldots,P_n)$. The semantics of this operator is a generalization of the semantics of the binary choice operator.

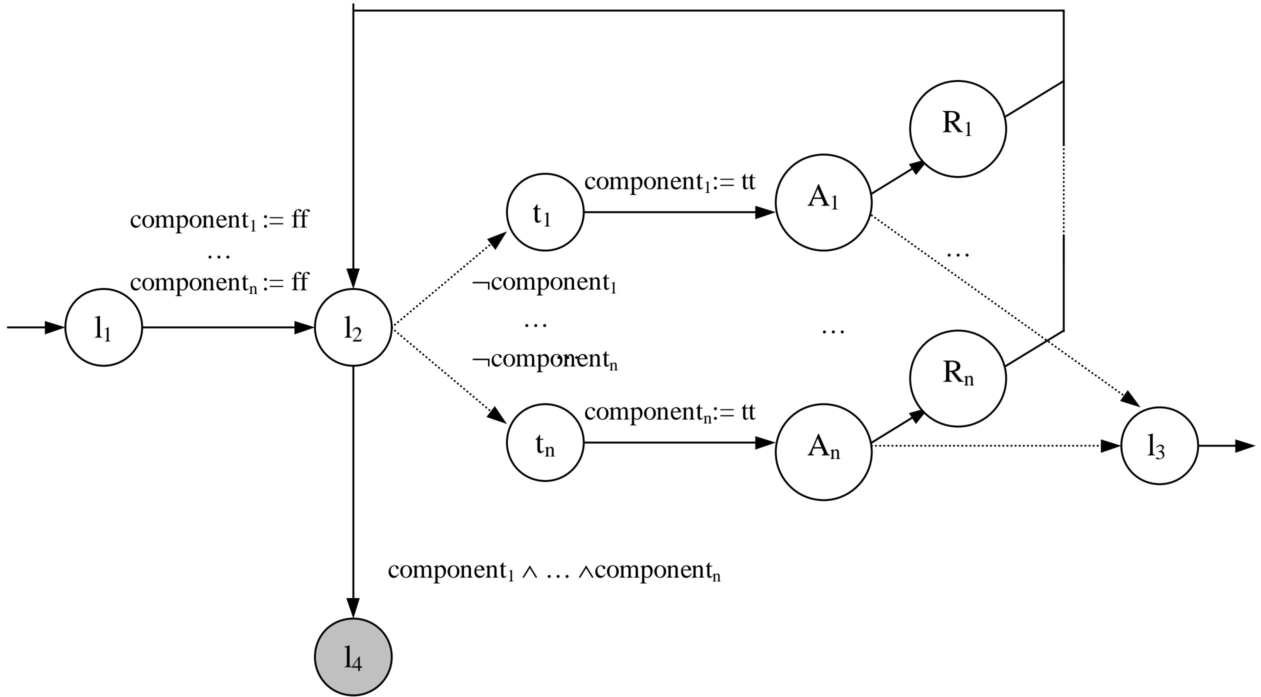


Figure 6.15 – Automaton for the receive behavior component

available because the guard $\neg$*component$_k$* is *fals*e. After having failed to execute every behavior component the *xor* component fails and reaches l_4.

Each automaton $A_k = \langle L_k, l_{1k}, l_{2k}, \Sigma_k, V_k, E_k\rangle$ is obtained by appending the back up of data modified by bh_k with the automaton corresponding to bh_k. Failure points of A_k should be directed to another automaton R_k responsible for restoring the values modified within A_k. Thus, $A_k = Back(\ O(Bh_k)\ ) \rightarrow [\![Bh_k]\!]$ and $R_k = Rest(\ O(Bh_k)\ )$.

$[\![$**xor** (op$_1$, …, op$_n$)$]\!]$ = $\langle L, l_1, l_3, \Sigma, V, E\rangle$ ρ, where $k\in\{1..n\}$ and

- $A_k = \langle La_k, lai_k, laf_k, \Sigma a_k, Va_k, Ea_k\rangle$ and $R_k = \langle Lr_k, lri_k, lrf_k, \Sigma r_k, Vr_k, Er_k\rangle$.
- $L = \{l_1, l_2, l_3, l_4\} \cup \{t_1, \ldots, t_n\} \cup La_1 \cup \ldots \cup La_n \cup Lr_1 \cup \ldots \cup Lr_n$.
- $\Sigma = \Sigma a_1 \cup \ldots \cup \Sigma a_n$.
- $V = \{component_1, \ldots, component_n\} \cup Va_1 \cup \ldots \cup Va_n \cup Vr_1 \cup \ldots \cup Vr_n$.
- $E = E_1 \cup E_2$ where,
  - $E_1 = \{(l_1, 0, tt, \{component_k := ff\}, l_2), (l_2, 0, \neg component_k, \varnothing, t_k), (t_k, 0, tt, \{component_k := tt\}, lai_k), (laf_k, 0, \varnothing, \varnothing, l_3), (lrf_k, 0, \varnothing, \varnothing, l_2)$« } for $k\in\{1..n\}$.
  - $E_2 = \{(l_r, 0, tt, \varnothing, lri_k)\}$, for each location $l_r \in \rho a_k$.

Note that $\Sigma r_k = \varnothing$.

Choice (OR) composition

In practice, the *xor* operator is often used as CSP choice operator. While translating *xor* introduces a lot of unnecessary detail, the translation of a CSP-like *or* operator is straightforward as shown below:

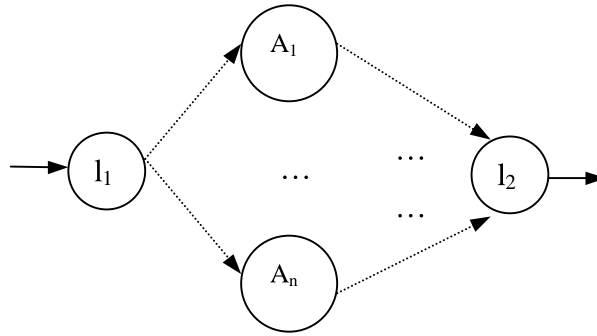


Figure 6.16 – Automaton template for the OR behavior component

$[\![$**`or`** `(op1, …, opn)`$]\!] = \langle L, l_1, l_2, \Sigma, V, E\rangle$ ρ, where $k \in \{1..n\}$ and

- $A_k = \langle L_k, li_k, lf_k, \Sigma_k, V_k, E_k\rangle$;
- $L = \{l_1, l_2\} \cup L_1 \cup \ldots \cup L_n$;
- $\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_n$;
- $V = Va_1 \cup \ldots \cup Va_n$;
- $E = E_1 \cup \ldots \cup E_n \cup \{(l_1, 0, tt, \emptyset, li_k), (lf_k, 0, tt, \emptyset, l_2)\}$.

The Object level

Operations are composed to form objects. After initializing its attribute values, the object waits for a message that will select an operation to run. When the operation finishes, the object returns to a state where it is again ready to accept another message. The resulting automaton can be seen as a choice composition of operations as illustrated by the CSP process: *offerOperations* $=_{def}$ *(op_1 [] … [] op_n); offerOperations* where *object* $=_{def}$ *init; offerOperations*.

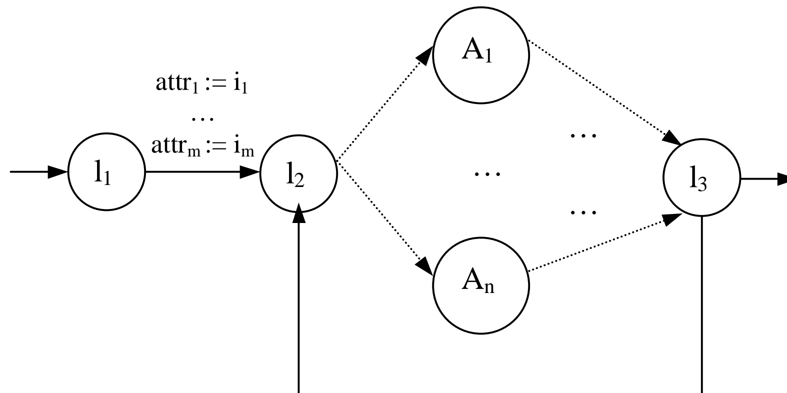


Figure 6.17 – Automaton template for the object composition operator

$[\![$**`object`** `oid` **`is`** `adec1, …, adecm op1, …, opn`$]\!] = \langle L, l_1, l_3, \Sigma, V, E\rangle$, where for $i \in \{1..n\}$:

- $[\![op_i]\!] = A_i = \langle L_i, li_i, lf_i, , \Sigma_i, V_i, E_i\rangle$;
- $L = \cup(L_i) \cup \{l_1, l_2, l_3\}$;

- $\Sigma = \cup(\Sigma_i)$;
- $V = \cup(V_i)$;
- $E = \cup(E_i) \cup \{(l_1, 0, tt, \{attr_1:=i_1, \ldots, attr_m:=i_m\}, l_2)\} \cup \cup(\{(l_2, 0, tt, \emptyset, li_i), (lf_i, 0, tt, \emptyset, l_3)\})$.

The system level

A community of objects forms a system. In OBLOG, objects work in parallel thus easing the modeling task, i.e., an OBLOG specification can be seen as a network of ETAs where each ETA corresponds to an object. In our setting, the network is constructed by composing ETAs using the parallel composition operator between ETAs. A system $S = O_1 \ldots O_n$ will be coded as $[\![O_1 \ldots O_n]\!] = [\![O_1]\!] \parallel \ldots \parallel [\![O_n]\!]$.

6.3 Case study – verifying the Alternating Bit Protocol

In this section we explain the verification of properties in order that ensure the proper operation of our specification of the ABP (Appendix F and Appendix G).

The specification of a protocol is essentially a specification of passive objects that offer communication services. Because of this, and since we want to make assertions about the correct progress of our system, we must cause the system progress by interacting with it. In a more general framework, we must compose our system with an environment and check that it exhibits a correct behavior upon environment stimuli. Since we want to check progress properties, which are not specifiable in the UPPAAL property language, we must cause the system evolution by calling objects operations.

Verification framework

Given a system specification *Spec* in UPPAAL we can only verify properties of the form $Spec \models \forall\Box\varphi$ or $Spec \models \exists\Diamond\varphi$ (see Definition 6.4). Taking into account that the specification implicitly codes the initial state (formally *init*) by giving initial values to every variable, every property specified with the UPPAAL temporal logic takes the form $Spec \models init \Rightarrow \forall\Box\varphi$ or $Spec \models init \Rightarrow \exists\Diamond\varphi$, which apparently coincide with the characterization of reachability properties.

The subtlety here is that, although UPPAAL allows for the verification of reachability properties from the initial state, what we are really interested in, is specifying reachability from states other than the system initial state. For example, we would like to verify the property $transmit_I \Rightarrow \forall\Box\exists\Diamond receive_i$, where $transmit_i$ and $receive_i$ are formulas characterizing states of successful transmission and reception, respectively and, 'i' ranges over the domain of messages. We are not able to write such a formula in UPPAAL logic. We can however, produce a special test automaton $T\varphi$ to interact with the system as introduced in Section 6.1. This automaton will test for a special condition like $\neg receive_i$ that will isolate the 'bad states' of the system. Then, we check that the test automaton never reaches the 'bad' states, those where $\neg receive_i$ (from $transmit_i$) holds. This is not sufficient for we must also ensure that the transmission and the reception are in fact carried out. A straightforward way to do it is adding a special state named *performed* such that $T\varphi$ will reach this state only after having performed a transmit and a receive operation, as shown if Figure 6.19.

What we will be doing is translating reachability properties into a combination of UPPAAL reachability properties plus a test automaton. Constructing test automata

involves a certain degree of ingenuity. For that reason, we propose a systematic way of doing this by using the template *ElaborateTA*(A,C) of Definition 6.9 where A is an automaton corresponding to a program that will interact with our system specification and C is a condition that defines a 'bad' situation of the system. When the test automaton reaches the *bad* state this means that the system is also at some 'bad' state as depicted in Figure 6.19. The safety property verification is attained through the verification of the property $\forall\Box\neg$(Tφ **at** *bad*) together with the assurance of system progress by checking $\exists\Diamond$(Tφ **at** *performed*).

Definition 6.9 Given an automaton A and a Boolean condition C we define the Test Automaton for A with condition C as *ElaborateTA*(A, C) $\equiv \langle L, l_1, l_2, \Sigma, V, E\rangle$ where

- $A = \langle La, la_i, la_f, \Sigma a, Va, Ea\rangle$;
- $L = \{l_p, l_b, l_f\} \cup La$;
- $\Sigma = \Sigma a$;
- $V = Vars(C) \cup Va$;
- $E = \{(la_f, 0, tt, \varnothing, l_p), (l_p, 0, C, \varnothing, l_b), (l_p, 0, tt, \varnothing, l_f)\} \cup Ea$.

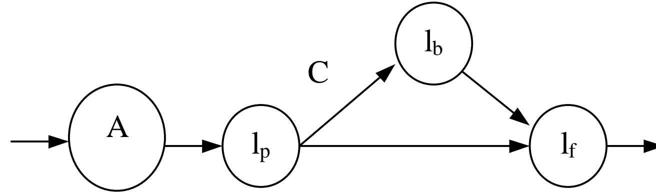


Figure 6.18 – Automaton template for *ElaborateTA*. The edge l_p stands for *performed*, l_b stands for *bad* and l_f for *final*.

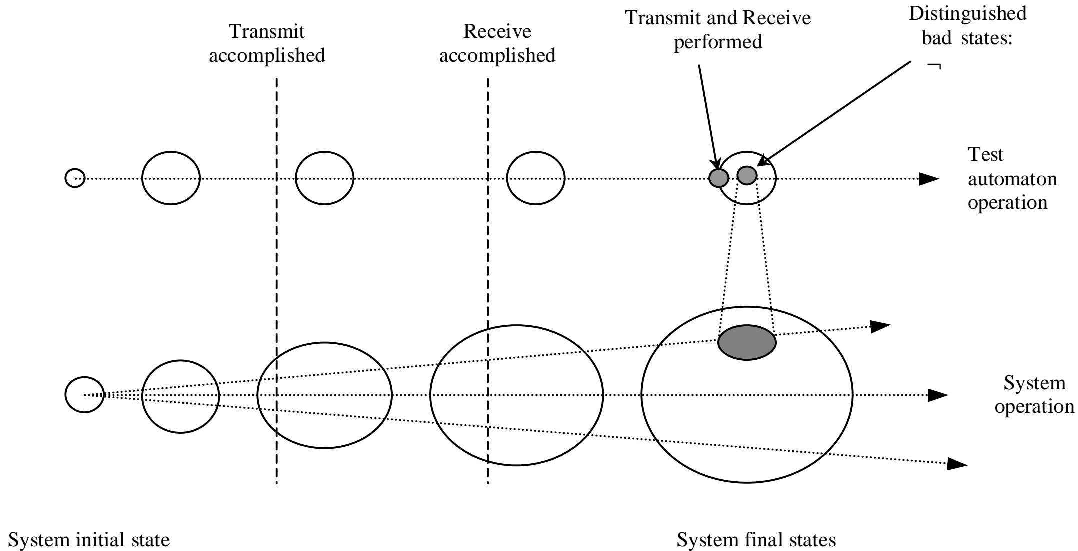


Figure 6.19 – Interaction of a test automaton with a system specification. The circles represent the state space reachable by the system and by the test automaton.

After calling object operations we will need to test the output variables and reason about operation failure. Output variables will be referenced in the usual fashion and the failure of an operation call will make a special variable *object.operation.fail* to be set to *true* or to *false* if the operation call succeeded. Object attributes are referenced as *object.attribute*.

Operation of a single component – the acknowledge line

Our 'acknowledge line' is a very simple component. We propose checking the following properties:

1. "The line may loose an acknowledge signal"
2. "The line correctly delivers the acknowledge signal in the absence of errors"

For the first property we can check whether a 'ackline.getAcknowledge' operation call fails after having sent an acknowledge signal with 'ackline.sendAcknowledge'. Using the UPPAAL tool we check that in the program below:

$$T\varphi \equiv \mathit{ElaborateTA}([\![\textbf{call } \text{ackline.sendAcknowledge(0)}; \textbf{ call } \text{ackline.getAcknowledge(ack)}]\!], \text{false})$$

This property holds:

$$T\varphi // \mathit{ABP}_{Spec} \vDash \exists\Diamond((T\varphi \textbf{ at } \mathit{performed}) \wedge (\mathit{ackline.getAcknowledge.fail}=\text{true}))$$

There is no need to test for bad states because we are interested in knowing if the program denoted by $T\varphi$ executes with success. The second property asserts something about the good operation of the line. For that, we can call 'ackline.getAcknowledge' after 'ackline.sendAcknowledge' and check that no failure occurred and that the right acknowledge signal was returned. We want to show that in our system:

$$\mathit{ABP}_{Spec} \vDash (\mathit{sendAcknowledge}_i \wedge \neg(\mathit{ackLine.lStatus}=\mathit{Error})) \Rightarrow \forall\Box\exists\Diamond \mathit{getAcknowledge}_i$$

In UPPAAL we must do it as:

1)	$T\varphi \equiv \mathit{ElaborateTA}([\![\textbf{call } \text{ackline.sendAcknowledge(ack)}; \textbf{ call } \text{ackline.getAcknowledge}(\text{ack}_2)]\!], \neg(\text{ack}=\text{ack}_2) \wedge \neg(\mathit{ackline.lStatus}=\mathit{Error}))$
2)	$T\varphi // \mathit{ABP}_{Spec} \vDash \exists\Diamond(T\varphi \textbf{ at } \mathit{performed})$
3)	$T\varphi // \mathit{ABP}_{Spec} \vDash \forall\Box\neg(T\varphi \textbf{ at } \mathit{bad})$

In UPPAAL syntax we have:

```
E<>(call_ackline_send_get.call_performed))
A[](not (call_ackline_send_get.call_bad))
```

If these properties hold then we know that: The test automaton never reaches a bad state, i.e., a state where there no error occurred in the transmission and $\neg(\text{ack}=\text{ack}_2)$. To prevent this property to hold in cases where there is no progress, the liveness property 2)

above must also hold ensuring that the test automaton reaches the states precisely before the 'bad' states (this is, the automaton gets to *performed*) and therefore also tests the condition.

A small subtlety persists however. The program was written in some first order fashion, using the 'ack' variable to mean 'for any acknowledge value'. In practice the variable is be instantiated to '0' or to '1' before calling the 'ackline.sendAcknowledge' operation. The variable ack_2, is an output variable and will be instantiated by the program ('ack' is an input variable).

Global operation of the system

Another class of properties refers to the joint operation of the components. We will now concentrate on the send-receive properties and abstract away from the details of individual components.

Let us first look at two properties on the unreliability of our protocol:

1. "A sent message can be lost and never received"
2. "A sent message is always received if there is no error in the transmission line"

Both properties can be verified using the same program that sends a message through the 'sender.accept' operation and then tries to receive it through the 'receiver.deliver' operation. The 'receiver.getStatus' operation will allow the failure report of the receiver. The symbols written in capital letters like SOMETHING and SAMEMESSAGE are enumerations that can be found in Appendix B.

$$T\varphi \equiv \mathit{ElaborateTA}(\llbracket \textbf{call } \mathit{sender.accept}(\text{SOMETHING}) \textbf{; call } \mathit{receiver.deliver}(\text{m}) \textbf{; call } \mathit{receiver.getstatus}(\text{s}) \rrbracket, \text{false})$$

For the first property we must verify that:

$$T\varphi // ABP_{Spec} \vDash \exists\Diamond((T\varphi \textbf{ at } \mathit{performed}) \wedge (s\text{=SAMEMESSAGE}))$$

In order to make our specification simpler we abstracted away the message contents. For this reason, the message data type can only have two values: 'NIL' and 'SOMETHING'. After calling 'receiver.deliver' we must get 'SOMETHING' if the transmission line worked correctly. Furthemore, the 'receiver.getStatus' operation will report that this message is new. The property can by written as:

$$(\mathit{send}_i \wedge \neg(\mathit{transline.lStatus}\text{=}\mathit{Error})) \Rightarrow \forall\Box\exists\Diamond \mathit{deliver}_i$$

In UPPAAL we must work around as:

1)	$T\varphi \equiv \mathit{ElaborateTA}(\llbracket \textbf{call } \mathit{sender.accept}(\text{SOMETHING}) \textbf{; call } \mathit{receiver.deliver}(\text{m}) \textbf{; call } \mathit{receiver.getstatus}(\text{s}) \rrbracket, \neg(\text{s=NEWMESSAGE}) \wedge \neg(\mathit{transline.lStatus}\text{=}\mathit{Error}))$
2)	$T\varphi // ABP_{Spec} \vDash \exists\Diamond(T\varphi \textbf{ at } \mathit{performed})$
3)	$T\varphi // ABP_{Spec} \vDash \forall\Box\neg(T\varphi \textbf{ at } \mathit{bad})$

To complete the sequence send-receive-acknowledge we are only missing the following property: "A received message is acknowledged to the sender if there is no error in the acknowledge line". For that, we must add to our program a 'send.getStatus' operation call and check if the sent message was acknowledged. The operation call 'receiver.getStatus' is irrelevant for this property because the call to 'receiver.deliver' causes the acknowledge signal to be sent, so we will take it away. The property is verified as before by constructing the test automaton and verifying two properties as follows:

1)	$T\varphi \equiv \mathit{ElaborateTA}(\llbracket$**call** *sender.accept*(SOMETHING)**; call** *receiver.deliver*(m); *sender.getstatus*(s)$\rrbracket$, $\neg$(s=NOACKNOWLEDGE) $\wedge$ $\neg$(*ackline.lStatus*=*Error*))
2)	$T\varphi // ABP_{Spec} \vDash \exists\Diamond(T\varphi$ **at** *performed*)
3)	$T\varphi // ABP_{Spec} \vDash \forall\Box\neg(T\varphi$ **at** *bad*)

Putting together the second and the third properties we have ensured that our protocol can correctly deliver messages in the absence of errors.

Conclusion

With this work, the author hopes to have contributed towards the verification of software systems, in particular for the verification of OBLOG specifications. Despite all the limitations of currently available tools it is hopped that, taking into account the efforts being put in this area, in the next few years we will witness a systematic employment of these tools in a more pragmatic way. Only producing software will not suffice. As customers get educated they will request for quality certified software. Thus, the evolution of software engineering methodologies and production tools in the direction of correctness appears as a natural route.

Use of tools and techniques

The tools used in this work are Model-Checkers, which theoretically operate under the push-button principle. In practice however, it does not work so simply: On one hand, the complexity of real systems is too high for current Model-Checkers, asking for the employment of additional techniques like compositional verification. On the other hand, the gap between the problem domain and the specification languages is too big, still requiring the user to be an expert in formal specification and having a deep understanding of the problem domain. This situation can be greatly minimized by embedding verification tools into software development environments like OBLOG and providing an intuitive interface to manage compositional verification. The translation frameworks presented in chapters 5 and 6 from domain specific specification languages into low level specification languages can be used to fill the above-mentioned gap between domain languages and specification languages.

Specification of properties

The specification of properties is still done using Temporal Logic and Behavioral Specifications. While Temporal Logic is a natural way of specifying properties for the computer scientist, it is also a technicality that most users would like to see adapted to their problem domain. This is possible through the predefinition of macros over temporal logic operators for the most common properties. In turn, Behavioral Specifications are a very powerful means of specifying safety properties. However, it suffers from the drawback that the specification must be written in the same vocabulary as that of the implementation, which is limiting when the specification is still too abstract. It is a fact that specifications are also evolving themselves, being a subject of the spiral development process. In this context, specifications could be given initially in a form of Temporal Logic that could be refined until we have an acceptable specification of the system. The refinement can consist in nesting Temporal Logic formulas by Behavioral Specifications.

Model-Checking technology

Many theoretical developments concerned with the optimizations of state-space generation, representation and exploration have been pushing Model-Checking technologies from academia to industry. One of the major sources of reduction of state space-size is a technique called *abstraction* that relies in the synthesis of a smaller abstract system from the initial one. It has been mentioned that the main obstacle for

application of this technique is that a way of automatically computing abstractions has not yet emerged. The author believes that the solution resides in the development of a specialized type analysis algorithm that outputs an abstraction of a program given a program and a formula. An attempt to treat abstraction in this way is given in [Jackson & al. 94]. Abstraction was informally used in the specification of the ABP to abstract away from the contents of messages.

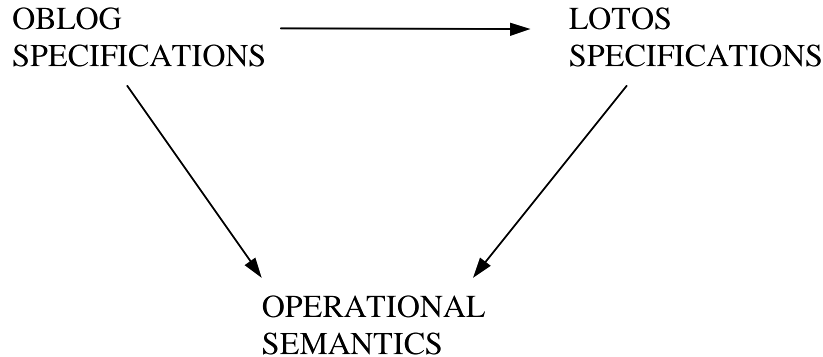
Yet another possible enhancement that can influence the translation of object-oriented specifications is the development of more flexible parallel composition operators. In our proposals for translation of OBLOG specifications, the community of objects was coded using parallel composition of processes. A deeper insight into the nature of concurrency in an object-oriented system will reveal differences to other concurrent systems. For example, in a hardware system, all components run in parallel and sometimes synchronize. In a somewhat different way, the flow of object-oriented can be seen as a sequential chain of delegations (modeled by synchronous calls) where concurrency appears as new threads of delegations. In most object-oriented systems there is a very small number of *active objects*, being the majority *passive objects*. Putting apart intra-method concurrency we can say that the system exhibits a behavior that can be emulated by a system with as many processes as the number of active objects. We can optimize the state-space generation by skipping a large number of meaningless transition interleavings by using composition operators that regulate the degree of interference or parallelism of the object in contact with the system. In [Krimm & Mounier 97] a parallel composition operator called *semi-composition* is presented and used to allow on-the-fly generation on state-spaces from LOTOS specifications. In a paper by Karisto [Karisto 97] the idea of a more flexible parallel composition operator is explored as way to handle data variables in LTSs.

The OBLOG language

It can be pointed out that the subset of the language is too narrow and is not representative of the full OBLOG specification language. The approach taken in the development of this work was not to advance a general theory to convert all OBLOG specifications into another specification language. There are two main obstacles: The first, is the absence of a formal semantics of the language. The second is that many concepts of OBLOG can not be coded into the languages offered by the verification tools. For example, the dynamic creation of objects has no apparent coding in the subset of LOTOS supported by CADP. Some important object-oriented concepts like *inheritance* have been left out. This dissertation intends to evaluate the capabilities of automatic verification tools and to advance a first procedure for the verification of OBLOG specifications.

Future work will consist in broadening the subset of specifications analyzable automatically by systematically introducing codings for new concepts like, for example, *Exception Handling* and *Inheritance*. Alternative composition should also be reviewed, its semantics involves the rollback of the whole state of system and in our approach, rollbacks were carried out only at the object level because coding the roll-back of the whole system seemed too complex. It is expected to reach the decidability barrier with the introduction of dynamic creation of objects. At this point, the integration of Model-Checkers with Automated Theorem Provers must be considered.

As a last commentary, if we had a formal definition of the operational semantics of the language we could have proved that our translation is correct by proving that the following diagram commutes.



CADP and the LOTOS translation

The CADP toolset is very flexible, allowing the specifications to be given directly in LTSs. One the other hand, LOTOS is a functional language such that, some concepts can be coded in a straightforward way (e.g. method invocation) and others reveal to be very difficult (e.g. roll-back). Although there is room for many optimizations, there will always be unnecessary transition overhead as a result of the use of LOTOS as an intermediate language. Given this, it appears that the development of an operational semantics for OBLOG specifications allowing a translation directly to LTSs is a good direction to pursue.

Another problem is the impossibility of stating properties about the internal state of objects. This can be very useful for debug purposes and for the specification of some reachability properties

In a case-study related to the LOTOS specification of the Airbus Flight Warning Computer the problem of the '[⟩' disabling operator which has an *interrupt-and-terminate* semantics versus *interrupt-and-resume* was also outlined. The solution followed was an ad-hoc treatment expressing this behavior with a different language. Please note that we also had a problem in using this operator as pointed in Chapter 5.

UPPAAL and the Communicating Automata translation

Although UPPAAL is a tool for analyzing timed systems, it also allows the modeling of systems without clock variables. This tool is equipped with a high-performance Model-Checking procedure based on reachability analysis. The performance comes at a price; the specification of some classes of properties can be quite tricky – Since we only have a subset of TCTL logic available. The specification of complex properties must be given has a combination of a simple formula plus a test automaton. How this can be done automatically is not obvious. In a recent paper [Aceto & al. 98] proposed a procedure for generation of test automata from a version of the timed Mu-Calculus. The impact of his contribution in this work was not analyzed in detail at the time of writing.

The language offered by UPPAAL is based on communicating automata with shared memory. This allows the specification of reachability properties over the attributes of objects but increases the complexity of message passing because the parameters of operations can only be passed through synchronization on a shared memory area.

Comparing the two approaches

We have presented a translation strategy for two different languages. One of them is a functional process algebraic language whereas the other is an automata-based shared memory like language with synchronization. In trying to establish a comparative analysis of the suitability of them to code OBLOG specifications it must be remarked that these languages are different in nature. Some aspects of object-oriented technology are better coded using a functional style while other could benefit from a shared memory style. UPPAAL provides a lower level language than LOTOS. Both languages offer synchronization but in different ways: LOTOS synchronization uses a *multiway rendezvous* mechanism and while UPPAAL follows the thread of process algebraic languages with symmetric synchronization. The net effect of this is that, for example, coding *broadcast* is straightforward in LOTOS and elaborate in UPPAAL. Yet another aspect of this difference has to do with the verification of components when detached from the system. The difference is motivated by the semantics of the two languages: A LOTOS specification is always able to perform actions on channels unless we force its synchronization with another process. In UPPAAL an action on a channel can only be taken if there is a process offering the symmetric action. If we are interested in verifying the progress of a LOTOS process, the semantics of the language automatically induces an environment process which always agrees on every observable action that the first one is trying to take. In turn, an UPPAAL process with actions only progresses if we provide the specification of an environment with symmetric actions.

Concerning data-types, CADP presented a very powerful setting for the specification of abstract data-types with the language ACT-ONE. UPPAAL offered only integers and enumerated types

The verification of properties using CADP sometimes took times ranging form 30 seconds to 15 minutes approximately, while UPPAAL took times ranging from 10 seconds to 3 or 4 minutes at most. The explanation for this difference is related both with the technology of the tools and with the coding of OBLOG specifications into each one. CADP is more flexible and offers more powerful specification logic than UPPAAL, so its natural to think that CADP is not as performing as UPPAAL. Please note that UPPAAL documentation explicitly refers that expressiveness was sacrificed for speed. On the other hand there are aspects related to the target language: The LOTOS language is a higher level language than the language of Communicating Automata. Thus, when translating to Communicating Automata we are in some sense 'nearer' to LTSs than when translating to LOTOS. Furthermore, because of the functional qualities of LOTOS we had to code the state of objects as a process running in parallel with the object body, which seems to have introduced a lot of overhead. Yet another aspect is the coding of *failure* and *rollback*, this feature is absent is both languages and had to be emulated by means of *backup* and *restores*. This emulation is introducing an overhead that was not quantified in this work. A deeper insight should be carried out into this subject since it can confirm the necessity of producing LTSs directly form OBLOG. Although Communicating model of

flat shared memory model is not the best way to code object states, it seems to have worked better in our setting.

References

[Aceto & al. 98]
Luca Aceto, Augusto Bergueno and Kim G. Larsen, "Model Checking via Reachability Testing for Timed Automata.", In Proceedings of the 4th International Workshop on Tools and algorithms for the construction and Analysis of Systems. Gulbenkian Foundation, Lisbon , Portugal, 31, March, 2nd April, 1996. LNCS 1384, pages 263-280, Bernhard Steffen (Ed.).

[Aceto & al. 98b]
Luca Aceto, Patricia Bouyer, Augusto Burgueño and Kim G. Larsen,"The Power of Reachability Testing for Timed Automata", In Proceedings of Foundations of Software Technology and Theoretical Computer Science, December 17--19, 1998, Chennai, India.

[Andrade & Sernadas 96]
Luis Filipe A. Andrade and A. Sernadas, "Banking and Management Information System Automation", In proceedings of the 13th world congress of the International Federation of Automatic Control, San Francisco, USA, 1996, Volume L., pp. 113-136. Elsevier Sience.

[Andrade 99]
Luis Filipe A. Andrade, "An Object-Oriented and Transformational Approach for Software Development", Ph.D. Thesis, in preparation.

[Arnold 94]
André Arnold, "Finite Transition Systems", Prentice-Hall International Series in Computer Science, 1994.

[Bartlett & al. 69]
K. A. Bartlett, R. A. Scantlebury and P. T. Wilkinson, "A Note of Reliable Full-Duplex Transmission over Half-Duplex Links", Communications of ACM 12(5) pp. 260-261, 1969.

[Bengtsson & al. 95]
Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson and Wang Yi. "UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems.", In Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid systems, New Brunswick, New Jersey, 22-24 October, 1995.

[Bengtsson & al. 98]
Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, Wang Yi and Carsten Weise, "New Generation of UPPAAL.", In proceedings of the International Workshop on Software Tools for Technology Transfer. Aalborg, Denmark, 12 - 13 July, 1996.

[Bjørner & al. 96]
Nikolaj Bjørner, Anca Browne, Eddi Chang, Michael Colón, Arjun Kapur,, Zhoar Manna, Henny B. Sipma and Tomás E. Uribe, "STeP – The Stanford Temporal Prover", Computer Science Department, Stanford University, Stanford California 94305 USA, 1996.

[Brand & Joyner 78]
D. Brand and W. H. Joyner Jr., "Verification of Protocols Using Symbolic Execution", Computer Networks, 2, pg. 351-360, 1978.

[Bryant 86]
R. E. Bryant, "Graph Based Algorithms for Boolean Function Manipulation", IEEE Transactions on Computers, C-5, pp. 677-691, 1986.

[Burch & al. 90]
J. R. Burch, E . M. Clarke, K. L. McMillan, D. L. Dill and L. J. Hwang, "Symbolic Model Checking: 10^{20} Sates and Beyond.", Logic in Computer Science, 1990.

[Chandy & Misra 88]
K. Mani Chandy and Jayadev Misra, "Parallel Program Design — A foundation", Addison-Wesley 1988.

[Clarke & Emerson 81]
E. M. Clarke and E. A Emerson, "Characterizing properties of Parallel programs as Fixed-Points", 7th Colloquium on Automata Languages and Programming. Vol. 85 of LNCS, Springer-Verlag, 1981.

[Clarke & Emerson 82]
E. M. Clarke and E. A Emerson, "Design and Synthesis of Synchronization Skeletons Using branching Time Temporal Logic", Logics of Programs Workshop Proceedings, Yorktown Heights, NY, USA. pg. 52-71, Vol. 131 of LNCS, Springer-Verlag, 1982.

[Clarke & al. 86]
E. M. Clarke, E. A. Emerson, A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications", ACM Transactions on Programming Languages and Systems, pg. 244-263, Vol. 8, No. 2, April 1986.

[Clarke & al. 92]
E. M. Clarke, J. R. Burch, O. Grumberg, D. E. Long and K. L. McMillan, "Automatic Verification of Sequential Circuit Designs", Phil. Trans. R. Soc. London, 339, 105-120, 1992.

[Clarke & al. 94]
E. M. Clarke, O. Grumberg and D. Long, "Verification Tools for Finite-State Concurrent Systems", In "A decade of Concurrency: Reflections and Perspectives", volume 803 of LNCS, Springer-Verlag 1994.

[Clarke & al. 94b]
E. M. Clarke, O. Grumberg and D. Long, "Model Checking and Abstraction", In Proceedings of Programming Languages (POPL), 1994.

[Cleaveland 90]
Rance Cleaveland, "Tableau-Based Model Checking in the Propositional Mu-Calculus", Acta Informatica 27, pp. 725-747, Springer-Verlag, 1990

[Courcoubetis & al. 92]
C. Courcoubetis, M. Vardi, P. Wolper and M. Yannakakis, "Memory-Efficient algorithms for the verification of Temporal Properties", Formal Methods in System Design 1-275-288, 1992.

[Daniele & al. 99]
Marco Daniele, Fausto Giunchiglia and Moshe Y. Vardi, "Improved Automata Generation for Linear Temporal Logic", In CAV'99 Conference on Automated Verification, vol. 1633 of LNCS, Springer-Verlag 1999.

[De Nicola & Vaandrager 90]
Rocco De Niccola and Fritz W. Vaandrager, "Action versus State Based Logics for Transition Systems", In proceedings Ecole de Printemps on Semantics of Concurrency, vol. 469 of LNCS, pp. 407-419. Springer Verlag, 1990.

[Dill 96]
David. L. Dill, "The Murφ verification system", In Computer Aided Verification. 8th International Conference, pp. 390-3, 1996.

[Dong & al. 99]
Yifei Dong, Xioqun Du, Y. S. Ramakrishna, C. R. Ramakrisnan, I. V. Ramakrisnan, Scott A. Smolka, Oleg Sokolsky, Eugene W. Stark and David S. Warren, "Fighting Livelock in the i-Protocol: A Comparative Study of Verification Tools", In Proceeding of the 5th International conference on Tools and algorithms for the Construction and Analysis of Systems, Amsterdam, Holland 1999, LNCS 1579, Springer-Verlag.

[Ehrig & Mahr 85]
H. Ehrig and B. Mahr, "Fundamentals of algebraic Specifications, Volume I", Springer-Verlag, 1985.

[Emerson 94]
E. allen Emerson, "Temporal and Modal Logic", Handbook of Theoretical Computer Science, Vol. B, Formal Methods and Semantics, Chapter 16, Elsevier Science publishers, MIT Press, 1994.

[ESA 96]
European Space Agency, "ARIANE 5 Flight 501 Failure", Report by the Inquiry Board, http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html, Paris, 1996 July 19.

[Fagin & al. 95]
Ronald Fagin, Joseph Y. Halpern, Yoram Moses, Moshe Y. Vardi, "Reasoning About Knowledge", MIT Press, 1995.

[Fernandez 88]
Jean-Claude Fernandez, "Aldébaran: Users Manual", Technical Report Spectre C14, LGI-IMAG Grenoble, 1988.

[Fernandez 89]
Jean-Claude Fernandez, "Aldébaran: A tool for verification of communicating processes", Technical Report Spectre C14, LGI-IMAG Grenoble, 1989.

[Fernandez 90]
Jean-Claude Fernandez and Laurent Mounier, "Verifying Bisimulations 'On the Fly' ", Proceedings of the 3rd International Conference on Formal Description Techniques, FORTE'90 Madrid, Spain, pp. 91-105, NH, Nov. 1990.

[Fisher & Ladner 77]
M. J. Fisher and R.E. Ladner, "Propositional Modal Logic of Programs", Proceedings of 9th ACM annual symp. on Theory of Computing. pp. 286-297, 1977.

[Fisher & Ladner 79]
M. J. Fisher and R.E. Ladner, "Propositional Modal Logic of regular Programs", J. Comput. System Science, 18 (2), pp. 194-211, 1979.

[Garavel 89]
Hubert Garavel, "Compilation et Vérification des programmes LOTOS", Thèse de Dotorat, Université Joseph Fourier (Grenoble), November 1989.

[Garavel 98]
Hubert Gravel, “OPEN/CAESAR: An Open Software Architecture for Verification Simulation and Testing”, In Proceeding of the 4th International conference on Tools and algorithms for the Construction and Analysis of Systems, Lisbon, Portugal 1998, LNCS 1384, Springer-Verlag.

[Gerth & al. 94]
R. Gerth, D. Peled, M. Y. Vardi, P. Wolper, “Simple On-the-fly Automatic Verification of Linear Temporal Logic”, In Proceedings of PSTV’95, Protocol specification testing and verification, Chapmann & Hall, Warsaw Poland, 1995.

[Hailpern & Nguyen 87]
Brent Hailpern and V. Nguyen, “A model for object-based inheritance”, In Bruce Shriver and Peter Wegner, editors, Research Directions in Object-Oriented Programming, pages 147-164. Computer Systems Series. MIT Press, 1987.

[Havelund & al. 99]
Klaus Havelund, Kim G. Larsen and Arne Skou, “Formal Verification of a Power Controller Using the Real-Time Model Checker UPPAAL”, Accepted for presentation at 5th International AMAST Workshop on Real-Time and Probabilistic Systems.

[Heitmeyer & Bharadwaj 97]
Constance Heitmeyer and Ramesh Bharadwaj, “Model Checking Complete Requirements Specifications Using Abstraction”, Naval Research Laboratory, Washington , DC 20375-5320, USA, November 10, 1997.

[Henziger & al. 92]
T. A. Henzinger, Z. Nicollin, J. Sifakis and S. Yovine, “Symbolic Model Checking for Real-Time Systems”, In Logic in Computer Science, 1992.

[Henzinger & al. 95]
T. A. Henzinger, P. H. Ho and H. Wong Toi, “HyTech: The next generation”, In proc. of the 16th Real-Time Systems Symposium, RTSS’95. IEEE Computer Society press, 1995.

[Hoare 85]
Charles A. R Hoare, “Communicating Sequential Processes.”, Prentice-Hall 1985.

[Holzmann 82]
Gerard J. Holzmann, “A Theory for Protocol Validation”, IEEE Transactions on Computers, C31(8), pp. 730-738, August 1982.

[Holzmann 91]
Gerard J. Holzmann, “Design and Validation of Computer Protocols”, Prentice-Hall, 1991.

[Holzmann & Peled 96]
Gerard J. Holzmann and D. Peled, “The state of SPIN”, In Computer Aided Verification (CAV’96), Vol. 1102 of LNCS, Springer-Verlag, July 1996.

[Hu & Dill 93]
Alan J. Hu and David L. Dill, “Reducing BDD Size by Exploiting Functional Dependencies," 30th ACM/IEEE Design Automation Conference, 1993, pp. 266-271.

[Hu & al. 93]
Alan J. Hu, David L. Dill, Andreas J. Drexler, and C. Han Yang, “Higher-Level Specification and Verification with BDDs,", Computer Aided Verification: Fourth International Workshop, 1992, reprinted in Lecture Notes in Computer Science Vol. 663, Springer-Verlag, 1993.

[Hu 95]
Alan J. Hu, “Techniques for Efficient Formal Verification Using Binary Decision Diagrams”, Ph.D. Thesis, Stanford University, December 1995.

[Hughes & Cresswell 96]
G. E. Hughes and M. J. Cresswell, “A new introduction to Modal Logic”, Routledge and T.J. Press, 1996.

[ISO 88]
International Organization for Standardizatio, “ISO/IEC. LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behavior. International Standard 8807”, – Information Processing Systems – Open Systems Interconnection, Genève, September 1998.

[ITU-T 94]
International Telecommunication Union, “Specification and Description Language (SDL)”, Z.100 norm, ITU-T, June 1994.

[Jackson & al. 94]
Daniel Jackson, Somesh Jha and Craig A. Damon, “Faster Model Checking of Software Specifications by Eliminating Isomorphs”, In Proceedings of Programming Languages (POPL), 1994.

[Jungclaus & al 91]
R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas, “Object-Oriented Specification of Information Systems: The TROLL Language”, Informatik-Bericht 94-03, TU Braunschweig, 1994.

[Karisto 97]
Konsta Karsisto, “A More Flexible Parallel Composition Operator”, Proceedings of the Fifth Symposium on Programming Languages and Software Tools, Jyväskylä, Finland, June 1997, University of Helsinki, Department of Computer Science, Series of Publications C, Report C-1997-37, pp. 13-23.

[Kozen 83]
D. Kozen, “Results on the propositional Mu-Calculus”, Theoretical Computer Science 27, pg. 333-354, North-Holland, 1983.

[Kokkarinen 98]
Ilka Kokkarinen, “A Verification-Oriented Theory of Data in Labeled Transition Systems.”, Dr.Tech. Thesis, Tampere University of Technology, Publications 234, Tampere 1998.

[Krimm & Mounier 97]
Jean-Pierre Krimm and Laurent Mounier, “Compositional State Space Generation from Lotos Programs”, Technical Report RR97-01, VERIMAG, January 1997.

[Kristoffersen & Petterson 95]
Kåre J. Kristoffersen and Paul Petterson, “Modelling and Analysis of a Steam Generator using UPPAAL”, In Proceedings of the 7th Nordic Workshop on Programming Theory, 1-3, November 1995.

[Kurshan 94]
Robert P. Kurshan, “Computer-Aided Verification of Coordinating Processes – The Automata Theoretic Approach”, Princeton Series in Computer Science, 1994.

[Larsen & al. 95]
Kim G. Larsen, Paul Pettersson and Wang Yi, “Diagnostic Model-Checking for Real-Time Systems.” , In Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems, New Brunswick, New Jersey, 22-24 October, 1995.

[Larsen & al. 95b]
Kim G. Larsen, Paul Pettersson and Wang Yi, “Compositional and Symbolic Model-Checking of Real-Time Systems.”, In Proceedings of the 16th IEEE Real-Time Systems Symposium, Pisa, Italy, 5-7 December, 1995.

[Larsen & al. 97]
Kim G. Larsen, Paul Pettersson and Wang Yi, “UPPAAL in a Nutshell”, In Springer International Journal of Software Tools for Technology Transfer, 1(1-2), December 1997, pages 134-152.

[Manna & Pnueli 92]
Zohar Manna and Amir Pnueli, “The Temporal Logic of Reactive and Concurrent Systems (specification)”, Springer-Verlag, 1992.

[Manna & Pnueli 95]
Zohar Manna and Amir Pnueli, “Temporal Verification of Reactive (safety)”, Springer-Verlag, 1995.

[McMillan 92]
Keneth L. McMillan, “The SMV system”, Carnegie-Mellon University, February 2, 1992.

[McMillan 93]
Keneth L. McMillan, “Symbolic Model Checking”, Kluwer Academic Publishers, 1993.

[Milner 89]
Robin Milner, “Communication and Concurrency”, Prentice-Hall International Series in Computer Science, 1989.

[Mateescu 98]
Radu Mateescu, “Vérification des Propriétes temporelles des programmes parallèles”, Ph.D. Thesis, Institut National Polytechnique de Grenoble, April 10, 1998.

[Moreira 94]
Ana M. D. Moreira, “Rigorous Object-Oriented Analysis”, Ph.D. Thesis. Department of Computing Science and Mathematics, University of Stirling, November 1994.

[NASA 97]
National Aeronautics and Space Administration, Office of Safety and Mission Assurance, “Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems Vol. II”, NASA-GB-001-97, May 1997.

[Olivero & Yovine 93]
Antonio Olivero and Sergio Yovine, “Kronos: A toolset for verifying real-time systems. User’s guide and reference manual”, VERIMAG, Grenoble, France, 1993.

[Peled 93]
Doron Peled, “all from one and one for all: On model checking using representatives”, In 5th International Conference on Computer-Aided Verification, Elounda, Greece, Vol. 697 of LNCS, pp. 164-177, 1993.

[Peled 98]
Doron Peled, "Ten Years of Partial Order Reduction", In Proceeding of the 4th International conference on Tools and algorithms for the Construction and Analysis of Systems, Lisbon, Portugal 1998, Vol. 1384 of LNCS, Springer-Verlag.

[Pnueli 77]
Amir Pnueli, "The Temporal Logic of Programs", Proc. 18th IEEE Annual Symposium on Foundations of Computer Science, pg. 46-57, 1977.

[Pressman 97]
R. S. Pressman, "Software Engineering – A practitioner's approach, 4th edition", McGraw-Hill International Editions, 1997.

[Razouk & Estrin 80]
B. Razouk and G. Estrin, "Modeling and Verification of Communication Protocols in SARA : the X21 interface", IEEE transactions on Computers, C-29 (12), pg. 1038-1052, 1980.

[Savola 95]
Reijo Savola, "A State Space Generation Tool for LOTOS Specifications.", VTT Publications 241, Technical Research Centre of Finland (VTT), Espoo, Finland 1995, 107 p.

[Springintveld & al. 97]
Jan Springintveld, Frits Vaandrager and Pedro R. D'Argenio, "Testing Timed Automata", Technical Report CSI-R9712, Computing Science Institute, University of Nijmegen, August 1997. also available as CTIT Technical Report 97-17, Centre for Telematics and Information Technology, University of Twente, September 1997.

[Sernadas & Ehrich 91]
Amilcar Sernandas and H.-D. Ehrich,, "What is an Object, After All?", Object Oriented Databases: Analysis, Design and Construction. R. Meersman, W. Kent, S. Koshla, eds. North-Holland, Amsterdam 1991.

[Stevens & Stirling 98]
Pedrita Stevens and Colin Stirling, "Practical Model Checking Using Games", In Proceeding of the 4th International conference on Tools and algorithms for the Construction and Analysis of Systems, Lisbon, Portugal 1998, Vol. 1384 of LNCS, Springer-Verlag.

[Stirling & Walker 91]
Colin Stirling and David Walker, "Local Model Checking in the modal mu-calculus", Theoretical Computer Science 89, pp. 161-177, Elsevier 1991.

[Thomas 94]
Wolfgang Thomas, "Automata on Infinite Objects", Handbook of Theoretical Computer Science, Vol. B, Formal Methods and Semantics, Chapter 4, Elsevier Science Publishers, MIT Press, 1994.

[Valmari 88]
Antti Valmari, "State Space Generation: Efficiency and Practicality.", PhD Thesis, Tampere University of Technology Publications 55, 1988, 169p.

[Valmari & Tienari 95]
Antti Valmari & Martti Tienari, "Compositional Failure-Based Semantic Models for Basic LOTOS.", Formal Aspects of Computing (1995) 7: 440-468. also Tampere University of Technology, Software Systems Laboratory, Report 16, Tampere, Finland, July 1993, 25 p.

[Valmari 96]
Antti Valmari, “Compositionality in State Space Verification Methods.”, Invited talk, Application and Theory of Petri Nets 1996, 17th International Conference, Osaka, Japan, June 1996, Proceedings, Vol. 1091 of LNCS, Springer-Verlag 1996, pp. 29-56.

[Walker 92]
D. Walker, “π-Calculus Semantics of Object-Oriented Programming Languages”, Theoretical Aspects of Computer Software, LNCS 526, pg. 532-547, Springer-Verlag, 1992.

[Walker 95]
D. Walker, “Objects in the π-Calculus”, Information and Computation 116, pg. 253-271, Academic Press Inc., 1995.

[Winskel 91]
Glyn Winskel, “A note on Model Checking the modal ν-calculus”, Theoretical Computer Science 83, pp. 157-167, Elsevier 1991.

[Wolczko 88]
Mario I. Woczko, “Semantics of Object-Oriented Languages”, Ph.D. Thesis, University of Manchester, March 1988.

[Yi & al. 94]
Wang Yi, Paul Pettersson and Mats Daniels, “Automatic Verification of Real-Time Communicating Systems by Constraint Solving.”, In Proceedings of the 7th International Conference on Formal Description Techniques, Berne, Switzerland 4-7 October, 1994.

[Yovine 97]
Sergio Yovine, “A Verification Tool for Real-Time Systems”, Springer International Journal of Software Tools for Technology Transfer, Vol. 1, N. ½, October 1997.

[Yovine 98]
Sergio Yovine, “Model Checking Timed Automata”, Lectures on Embedded Systems, G Rozenberg and F. W. Vaanndrager Ed., LNCS Vol. 1494, Springer Verlag, October 1998.

Appendix A – Syntax of the OBLOG specification language

Table of non-terminals

Non-terminal	**Description**
alternative	An alternative composition of behavior components
atomicbhcomponent	An atomic behavior component (one of skip, fail, assert, set, ...)
attribute	Attribute of an object
attributes	List of attributes of an object
assert	Assert condition
bhcomponent	Can be either an atomic behavior component or a composite one
bhcomponentlist	A sequence of behavior components
choice	A choice of two or more behavior components
compositebhcomponent	A composite behavior component
fail	Fail component
id	An identifier, a sequence of characters
ids	A list of identifiers
initval	The initial value given in a variable declaration or attribute declaration
inparms	Input parameters of an operation
iterate	Iterate component
local	Local variable component
method	A method
methods	List of methods
operation	A operation with header and methods
operations	A list of operations
outparms	Output parameters in an operation
receive	Receive component
send	Send component
sequence	Sequential composition component
set	Set component
skip	Skip component
sort	The sort of a variable, e.g., Boolean, Integer, ...
variabledec	A variable declaration
variabledecs	List of declarations of variables
xor	Xor behavior component

Grammar

object ::= ‘**`object`**’ *id* ‘**`is`**’ *attributes operations* ‘**`end`**’
attributes ::= *{attribute}**
attribute ::= *id* ‘**`:`**’ *sort* ‘**`=`**’ *initval*

operation ::= ‘**`operation`**’ *id* ‘**`(`**‘ *inparms* ’**`,`**’ *outparms* ‘**`)`**’ *methods*
operations ::= *operation* | *{operation* ‘**,**’ *}* operation*

ids ::= *id* | *{id* ‘**,**’ *}* id*
variabledec ::= *id* ‘**`:`**’ *sort* ‘**`=`**’ *initval*
variabledecs ::= *variabledec* | *{variabledec* ‘**,**’ *}* variabledec*

inparm ::= **‘`in`’** *id*
inparms ::= *inparm* | *{inparm* ‘**,**’ *}* inparm*
outparm ::= **‘`out`’** *id*
outparms ::= *iparm* | *{outparm* ‘**,**’*}* outparm*

method ::= ‘**`method`**’ *id* ‘**`enabling`**’ *boolexpr* ‘**`local`**’ *variabledecs*
methods ::= *{outparm* ‘**,**’*}+*

bhcomponent ::= *atomicbhcomponent* | *compositebhcomponent*
atomicbhcomponent ::= *skip* | *fail* | *assert* | *set* | *send* | *receive*
compositebhcomponent ::= *iterate* | *sequence* | *alternative* | *choice* | *local* | *xor*
bhcomponents ::= *bhcomponent* | *{ bhcomponent* ‘**,**’ *}* bhcomponent*

skip ::= ‘**`skip`**’
fail ::= ‘**`fail`**’
assert ::= ‘**`assert`**’ *boolexpr*
set ::= ‘**`set`**’ *id* ‘<<’ *expr*
send ::= ‘**`send`**’ *exprlist* ‘**`in`**’ *id*
receive ::= ‘**`receive`**’ *ids* **‘`in`’** *id*
iteration ::= ‘**`while`**’ *boolexpr* ‘**`do`**’ *bhcomponent*
sequence ::= *bhcomponent* ‘**`;`**’ *bhcomponent*
alternative ::= *bhcomponent* ‘**`alt`**’ *bhcomponent*
local ::= ‘**`local`**’ *variabledecs* ‘**`in`**’ *bhcomponent*
choice ::= ‘**`or(`**’ *bhcomponents* ‘**`)`**’
xor ::= ‘**`xor(`**’ *bhcomponent* ‘**,**’ *bhcomponents* ‘**`)`**’

Appendix B – Initial OBLOG specification of the ABP

```
// sndstatus = enum (NOTSENT = 0, SENTNOACK = 1, SENTANDACK = 2)
// rcvstatus = enum (NOMESSAGE = 0, NEWMESSAGE = 2, SAMEMESSAGE = 2)
// linestatus = enum (STATERROR = 0, STATGOOD = 1)
// message = enum (NIL = 0, SOMETHING = 1)
// bit = enum (0, 1)

object sender

      attribute bval : bit = 0;
      attribute statval : sndstatus  = NOTSENT;

      operation accept( in m : message )
            method sendMessage
            enabling true
            do
                  call transline.sendMessage( m, bval );

      operation getStatus( out s : status )
            method returnStatus
            enabling true
            local b : bit = 0;
            do
                  call ackline.receiveAcknowledge( b );
                  pre b = bval in
                        (
                              set bval << ¬bval;
                              set statval << SENTANDACK;
                        )
                  alt
                        set statval << SENTNOACK;
                  set s << statval;

end


object transline

      attribute buffMsg : message = NIL;
      attribute buffBval : bit = 0;
      attribute lStatus : linestatus = STATGOOD;

      operation sendMessage( in m : message, in b : bit )
            method send
            enabling true
            do
                  or
```

```
                (     (
                            set lStatus << STATGOOD;
                            set buffMsg << m;
                            set buffBVal << b
                      )
                      set lStatus << STATERROR;
                )

     operation getMessage( out m : message, out b : bit )
          method send
          enabling lStatus = STATGOOD;
          do
                set m << buffMsg;
                set b << buffBVal;
end

object ackline

     attribute buffBval : bit = 0;
     attribute lStatus : linestatus = STATGOOD;

     operation sendAcknowledge( in b : bit )
          method send
          enabling true
          do
                or
                (
                      (
                            set lStatus << STATGOOD;
                            set buffBVal << b
                      )
                      set lStatus << STATERROR;
                )

     operation receiveAcknowledge( out b : bit )
          method send
          enabling lStatus = STATGOOD
          do
                set b << buffBVal

end

object receiver

     attribute bval : bit = 1;
     attribute statval : rcvstatus = NOMESSAGE;

     operation deliver( out m : message )
          method deliverAndAcknowlege
          enabling status  = NEWMESSAGE
          local msgparm : message = NIL, bparm : bit = 0;
          do
```

```
                call transline.getMessage( msgparm, bparm );
                pre b ≠ bval in
                     (
                          call ackline.sendAcknowledge( b );
                          set m << msgparm;
                          set bval << bparm;
                          set statval << NEWMESSAGE;
                     )
                alt
                     set statval << SAMEMESSAGE

      operation getStatus( out s : rcvstatus )
           method returnStatus
           enabling true
           do
                set s << statval;

  end
```

Appendix C – Syntactic sugar free specification of the ABP

```
object sender

      attribute bval : bit = 0;
      attribute statval : sndstatus = NOTSENT;

      receive m : message in accept
            // sendMessage
            (
                  assert true;
                  send m, bval in transline.sendMessage;
                  receive in transline.sendMessage;
                  send in accept;
            )
            alt
                  failon accept;

      receive in getStatus; // out s : sndstatus
            local s : status = NOTSENT in
            (
            // returnStatus
                  assert true;
                  local b : bit in
                  (
                        send in ackline.getAcknowledge;
                        receive b : bit in ackline.getAcknowledge;
                        assert b = bval;
                        (
                              set bval << ¬bval;
                              set statval << SENTANDACK;
                        )
                        alt
                              set statval << SENTNOACK;
                        set s << statval;
                  );
                  send s in getStatus;
            )
            alt
                  failon getStatus;


end


object transline
```

```
    attribute buffMsg : message = NIL;
    attribute buffBval : bit = 0;
    attribute lStatus : linestatus = STATGOOD;


    receive m : message, b : bit in sendMessage;
        // send
        (
        assert true;
            (
                or
                (
                    (
                        set lStatus << STATGOOD;
                        set buffMsg << m;
                        set buffBVal << b
                    )
                    set lStatus << STATERROR;
                )
            )
        send in sendMessage;
        )
        alt
            failon sendMessage;


    receive in receiveMessage; //out m : message, out b : bit
    local m : message = NIL, b : bit = 0 in
        (
        //send
            assert lStatus = STATGOOD;
            (
                set m << buffMsg;
                set b << buffBVal
            );
        send m, b in receiveMessage
        )
        alt
            failon receiveMessage;

end


object ackline

    attribute buffBval : bit = 0;
    attribute lStatus : linestatus = STATGOOD;

    receive b : bit in sendAcknowledge
        // send
        (
```

```
                assert true;
                or
                (
                     (
                      set lStatus << STATGOOD;
                      set buffBval << b;
                     )
                     set lStatus << STATERROR;
                 )
                send in sendAcknowledge;
           )

           alt
                failon sendAcknowledge;

      receive in getAcknowledge //out b : bit
           local b : bit = 0 in
           (
           // receive
                assert lStatus = STATGOOD;
                set b << buffBval
      send b in getAcknowledge;
           )
           alt
                failon getAcknowledge;
end


object receiver

      attribute bval : bit = 1;
      attribute statval : rcvstatus = NOMESSAGE;

      receive in deliver; //out m : message
           local m : message = NIL in
           (
           // deliverAndAcknowlege
                assert true;
                (
                     send in transline.receiveMessage;
                     receive msgparm, bparm in
transline.receiveMessage;
                     assert bparm ≠ bval;
                     (
                          send bparm in ackline.sendAckowledge;
                          receive in Ackline.SendAcknowledge;
                          set m << msgparm;
                          set bval << bparm;
                          set statval << NEWMESSAGE;
                     )
                     alt
```

```
                              set statval << SAMEMESSAGE;
            )
            send m in deliver
            )
            alt
                  failon deliver;


      receive in getStatus; // out s : rcvstatus
            local s : rcvstatus = NOMESSAGE in
            (
            // returnStatus
                  assert true;
                  set s << statval;
                  send s in getStatus
            )
            alt
                  failon accept;

end
```

Appendix D – LOTOS specifications of the ABP

ABPLIB

```
library
   BOOLEAN,
   NATURAL
endlib

type
   dtype
is
   sorts
      dtype
   opns
      dval (*! constructor *) : -> dtype
endtype

type
   bit
is Boolean
   sorts
      bit
   opns
      0 (*! constructor *),
      1 (*! constructor *) : -> bit
    not : bit -> bit
    _eq_ : bit, bit -> Bool
    _neq_ : bit, bit -> Bool
   eqns
      forall x,y : bit
      ofsort bit
         not(0) = 1;
         not(1) = 0;
      ofsort Bool
         x eq x = true;
         0 eq 1 = false;
         1 eq 0 = false;
         x neq y = not(x eq y);
endtype

type
   sndstatus
is Boolean
   sorts
      sndstatus
   opns
      NOTSENT (*! constructor *),
      SENTNOACK (*! constructor *),
      SENTANDACK (*! constructor *) : -> sndstatus
   _eq_ : sndstatus, sndstatus -> Bool
   eqns
      forall x,y : sndstatus
      ofsort Bool
         x eq x = true;
         NOTSENT eq SENTNOACK = false;
         NOTSENT eq SENTANDACK = false;
         SENTNOACK eq SENTANDACK = false;
         x eq y = y eq x;
(*
SENTNOACK eq NOTSENT = false;
         SENTANDACK eq NOTSENT = false;
         SENTANDACK eq SENTNOACK = false;*)
endtype

type
   rcvstatus
is
   sorts
      rcvstatus
   opns
      NOMESSAGE (*! constructor *),
      NEWMESSAGE (*! constructor *),
      SAMEMESSAGE (*! constructor *) : -> rcvstatus
endtype

type
   linestatus
is Boolean
   sorts
      linestatus
   opns
      STATERROR (*! constructor *),
      STATGOOD (*! constructor *) : -> linestatus
      _eq_ : linestatus, linestatus -> Bool
   eqns
      forall x : linestatus
      ofsort Bool
         x eq x = true;
         STATERROR eq STATGOOD = false;
         STATGOOD eq STATERROR = false;
endtype

type
   message
is Boolean
   sorts
      message
```

```
   opns
      NIL (*! constructor *),
      SOMETHING (*! constructor *) : -> message
      _eq_ : message, message -> Bool
    eqns
      forall x : message
      ofsort Bool
         x eq x = true;
         NIL eq SOMETHING = false;
         SOMETHING eq NIL = false;
endtype

type
   access
is
   sorts
      access
   opns
      READ (*! constructor *),
      WRITE (*! constructor *) : -> access
endtype

type
   recover
is
   sorts
      recover
   opns
      NORECOVER (*! constructor *),
      GO (*! constructor *) : -> recover
endtype
```

acklineabst – An example of a behavioural specification used for debugging

```
specification acklineabst[asa,aga] : noexit

library
   ABPLIB
endlib

behavior
   acklinesafety[asa,aga](0 of Bit)
where

(*
 *
 * ACKLINESAFETY
 *
 *)

process acklinesafety[asa, aga](b:bit) : noexit :=
         asa?bval:Bit; asa!false; acklinesafety[asa,aga](bval)
       []
         aga;
          (
               aga!b!false; acklinesafety[asa,aga](b)
            []
               aga!(0 of bit)!true; acklinesafety[asa,aga](b)

          )
endproc

endspec
```

ackline

```
specification ackline[asa,aga] : noexit

library
   ABPLIB
endlib

behavior
   ackline[asa,aga]
where

(*
 *
 * ACKLINE
 *
 *)

process ackline[asa, aga] : noexit :=
hide buffBval, lStatus in
(
   acklineStatus[buffBval, lStatus ](0 of bit, STATERROR)
     |[buffBval, lStatus]|
   acklineOperations[asa, aga, buffBval, lStatus]
)
where
   process acklineStatus[buffBval, lStatus](buffBvalv:bit, lStatusv:linestatus) : noexit :=
         buffBval!WRITE?XbuffBval:bit; acklineStatus[buffBval, lStatus](XbuffBval, lStatusv)
      []
         lStatus!WRITE?XlStatus:linestatus; acklineStatus[buffBval, lStatus](buffBvalv, XlStatus)
      []
         buffBval!READ!buffBvalv; acklineStatus[buffBval, lStatus](buffBvalv, lStatusv)
      []
         lStatus!READ!lStatusv; acklineStatus[buffBval, lStatus](buffBvalv, lStatusv)
   endproc (* acklineStatus *)

    process acklineOperations[asa, aga, buffBval, lStatus] : noexit :=
         (
             opasa[asa, buffBval, lStatus]
           []
             opaga[aga, buffBval, lStatus]
         )
            >> accept rval: Bool in
               (* silence operation failures *)
               (* We can place a stub here *)
                acklineOperations[asa, aga, buffBval, lStatus]
      where
         process opasa[asa, buffBval, lStatus] : exit(Bool) :=
         (* receive in sendAcknowledge *)
            receiveinsendAcknowledge[asa] >> accept b:bit in
         (* or *)
            (
               orop1[buffBval, lStatus](b)
            []
               orop2[lStatus]
            )
            >>
         (* ; *)
            accept rval : Bool in
               [rval eq true] -> exit(true)
               []
               [rval eq false] ->
                 sendinsendacknowledge[asa] >> accept rval:Bool in
                    exit(rval)
         where
            process receiveinsendAcknowledge[asa] : exit(bit) :=
               asa?b:bit; exit(b)
            endproc (* receiveinsendAcknowledge *)
            process orop1[buffBval, lStatus](b:bit) : exit(Bool) :=
            (* sets *)
               setlStatus[lStatus] >> accept rval:Bool in
                  [rval eq true] -> exit(true)
                  []
                  [rval eq false] ->
                     setbuffBval[buffBval](b) >> accept rval:Bool in exit(rval)
            where
               process setlStatus[lStatus] : exit(Bool) :=
                  lStatus!WRITE!STATGOOD; exit(false)
               endproc (* setlStatus *)
               process setbuffBval[buffBval](b:bit) : exit(Bool) :=
                  buffBval!WRITE!b; exit(false)
               endproc (* sebufBval *)
            endproc (* orop1 *)
            process orop2[lStatus] : exit(Bool) :=
            (* set lStatus << STATERROR *)
               lStatus!WRITE!STATERROR; exit(false)
            endproc (* orop2 *)
            process sendinsendacknowledge[asa] : exit(Bool) :=
               asa!false; exit(false)
            endproc (* sendinsendaacknowledge *)
        endproc
        process opaga[aga, buffBval, lStatus] : exit(Bool) :=
         hide break, b in
            (
                  lvars[b, break](0 of Bit)
                |[b, break]|
                    (
                       opagabody[aga, b, buffBval, lStatus] >>
                       accept rval:Bool in
                        (
                          break!dval;
                          exit(rval)
                        )
                    )
             )
```

```
        where
            process lvars[b, break](bv:bit) : exit(Bool) :=
                b!WRITE?Xb:bit; lvars[b, break](Xb)
              []
                b!READ!bv; lvars[b, break](bv)
              []
                break?dummy:dtype; exit(false)
            endproc (* lvars *)
            process opagabody[aga, b, buffBval, lStatus] : exit(Bool) :=
              hide c,r1 in
              (
               (buffBval!READ?bval:bit; c!bval; exit(false))
               |||
                (
                  (
                    receiveingetAcknowledge[aga] >> accept rval:Bool in
                      [rval eq true] -> exit(true)
                      []
                      [rval eq false] ->
                        (* assert lStatus = STATGOOD *)
                        assert[lStatus] >> accept rval:bool in
                          [rval eq true] -> exit(true)
                          []
                          [rval eq false] ->
                             (* sets *)
                             sets[b, buffBval] >> accept rval:bool in
                             [rval eq true] -> exit(true)
                             []
                             [rval eq false] ->
                              (* send b in getAcknowledge *)
                               sendingetAcknowledge[aga, b] >> accept rval:bool in
                                  [rval eq true] ->
                                     (
                                       (* alt failon getacknowledge *)
                                       c?bval:bit; buffBval!WRITE!bval;
                                       failongetacknowledge[aga] >> accept rval2:Bool in
                                       exit(rval2)
                                     )
                                  []
                                  [rval eq false] -> exit(false)
                  )
                )
             )
            where
              process receiveingetAcknowledge[aga] : exit(Bool) :=
                aga; exit(false) (* perigo *)
              endproc (* receiveingetAcknowledge *)
              process assert[lStatus] : exit(Bool) :=
               pexpr[lStatus] >> accept lstatusv:linestatus in
                  (
                    [lstatusv eq STATGOOD] -> exit(false)
                    []
                    [lstatusv eq STATERROR] -> exit(true)
                  )
              where
                  process pexpr[lStatus] : exit(linestatus) :=
                     lStatus!READ?lval:linestatus; exit(lval)
                  endproc (* pexpr *)
              endproc
              process sets[b, buffBval] : exit(Bool) :=
                setb[b, buffBval]
              where
                 process setb[b, buffBval] : exit(Bool) :=
                  pexpr[buffBval] >> accept bval:bit in
                        b!WRITE!bval; exit(false)
                 where
                    process pexpr[buffBval] : exit(bit) :=
                       buffBval!READ?bval:bit; exit(bval)
                    endproc (* pexpr *)
                 endproc (* setb *)
              endproc (* sets *)
              process sendingetAcknowledge[aga, b] : exit(Bool) :=
                   pexpr[b] >> accept bval:bit in
                     aga!bval!false; exit(false)
              where
                  process pexpr[b] : exit(bit) :=
                       b!READ?bval:bit; exit(bval)
                  endproc (* pexpr *)
              endproc (* sendingetAcknowledge *)
              process failongetAcknowledge[aga] : exit(Bool) :=
                 aga!0 of bit!true; exit(false)
              endproc (* failongetAcknowledge *)
           endproc (* opagabody *)
        endproc (* optaga *)
      endproc (* acklineOperations *)
endproc (* ackline *)

endspec
```

transline

```
specification transline[trm,tsm] : noexit

library
   ABPLIB
endlib


behavior
    transline[tsm,trm]
where

(*
 *
 * TRANSLINE
 *
 *)

process transline[tsm, trm] : noexit :=
hide buffMsg, buffBval, lStatus in
(
   translineStatus[buffMsg, buffBval, lStatus ](NIL, 0 of bit, STATGOOD)
     |[buffMsg, buffBval, lStatus]|
   translineOperations[tsm, trm, buffMsg, buffBval, lStatus]
)
where
   process translineStatus[buffMsg, buffBval, lStatus](buffMsgv:message, buffBvalv:bit, lStatusv:linestatus) : noexit
:=
         buffMsg?XbuffMsg:message; translineStatus[buffMsg, buffBval, lStatus](XbuffMsg, buffBvalv, lStatusv)
      []
         buffBval?XbuffBval:bit; translineStatus[buffMsg, buffBval, lStatus](buffMsgv, XbuffBval, lStatusv)
      []
         lStatus?XlStatus:linestatus; translineStatus[buffMsg, buffBval, lStatus](buffMsgv, buffBvalv, XlStatus)
      []
         buffMsg!buffMsgv; translineStatus[buffMsg, buffBval, lStatus](buffMsgv, buffBvalv, lStatusv)
      []
         buffBval!buffBvalv; translineStatus[buffMsg, buffBval, lStatus](buffMsgv, buffBvalv, lStatusv)
      []
         lStatus!lStatusv; translineStatus[buffMsg, buffBval, lStatus](buffMsgv, buffBvalv, lStatusv)
   endproc (* translineStatus *)

   process translineOperations[tsm, trm, buffMsg, buffBval, lStatus] : noexit :=
         (
             optsm[tsm, buffMsg, buffBval, lStatus]
          []
             optrm[trm, buffMsg, buffBval, lStatus]
         ) >> accept rval: Bool in
               (* silence operation failures *)
               (* We can place a stub here *)
           translineOperations[tsm, trm, buffMsg, buffBval, lStatus]
      where
         process optsm[tsm, buffMsg, buffBval, lStatus] : exit(Bool) :=
         (* receive in sendMessage *)
            receiveinsendmessage[tsm] >> accept m:message, b:bit in
         (* or *)
            (
               orop1[buffBval, buffMsg, lStatus](m, b)
            []
               orop2[lStatus]
            )
            >>
         (* ; *)
            accept rval : Bool in
               [rval eq true] -> exit(true)
               []
               [rval eq false] ->
                 sendinsendmessage[tsm] >> accept rval:Bool in
                    exit(rval)
          where
            process receiveinsendmessage[tsm] : exit(message, bit) :=
               tsm?m:message?b:bit; exit(m,b)
            endproc (* receiveinsendMessage *)
            process orop1[buffBval, buffMsg, lStatus](m:message, b:bit) : exit(Bool) :=
            (* sets *)
               setlStatus[lStatus] >>  accept rval:Bool in
                  [rval eq true] -> exit(true)
                  []
                  [rval eq false] -> setbuffMsg[buffMsg](m) >> accept rval:Bool in
                      [rval eq true] -> exit(true)
                      []
                      [rval eq false] -> setbuffMsg[buffMsg](m) >> accept rval:Bool in
                        setbuffBval[buffBval](b) >> accept rval:Bool in exit(rval)
            where
               process setlStatus[lStatus] : exit(Bool) :=
                  lStatus!STATGOOD; exit(false)
               endproc (* setlStatus *)
               process setbuffMsg[buffMsg](m:message) : exit(Bool) :=
                  buffMsg!m; exit(false)
               endproc (* setbuffMsg *)
               process setbuffBval[buffBval](b:bit) : exit(Bool) :=
                  buffBval!b; exit(false)
               endproc (* sebufBval *)
            endproc (* orop1 *)
            process orop2[lStatus] : exit(Bool) :=
            (* set lStatus << STATERROR *)
               lStatus!STATERROR; exit(false)
            endproc (* orop2 *)
            process sendinsendmessage[tsm] : exit(Bool) :=
               tsm!false; exit(false)
            endproc (* sendinsendmessage *)
        endproc
        process optrm[trm, buffMsg, buffBval, lStatus] : exit(Bool) :=
         hide break, m,b in
            (
                   lvars[m, b, break](NIL, 0 of Bit)
```

```
        |[m, b, break]|
          (
            optrmbody[trm, m, b, buffMsg, buffBval, lStatus] >>
             accept rval:Bool in
            (
               break!dval;
               exit(rval)
            )
          )
     )
where
    process lvars[m, b, break](mv:message, bv:bit) : exit(Bool) :=
        m?Xm:message; lvars[m, b, break](Xm, bv)
      []
        b?Xb:bit; lvars[m, b, break](mv, Xb)
      []
        m!mv; lvars[m, b, break](mv, bv)
      []
        b!bv; lvars[m, b, break](mv, bv)
      []
        break?dummy:dtype; exit(false)
    endproc (* lvars *)
    process optrmbody[trm, m, b, buffMsg, buffBval, lStatus] : exit(Bool) :=
      hide c, r1 in
      (
        (buffMsg?mval:message; buffBval?bval:bit; c!mval!bval; exit(false) )
       |||
        (
            (* receive in receivemessage *)
            receiveinreceivemessage[trm] >> accept rval:Bool in
            (* assert lStatus = STATGOOD *)
              [rval eq true] -> exit(true)
              []
              [rval eq false] ->
                assert[lStatus] >> accept rval:bool in
                  [rval eq true] -> exit(true)
                  []
                  [rval eq false] ->
                     (* sets *)
                     sets[m, b, buffMsg, buffBval] >> accept rval:bool in
                     [rval eq true] -> exit(true)
                     []
                     [rval eq false] ->
                      (* send b in getAcknowledge *)
                         sendinreceivemessage[trm, m, b] >> accept rval:bool in
                          [rval eq true] ->
                             (
                               (* alt failon getacknowledge *)
                               c?bval:bit; buffBval!bval;
                               failonreceivemessage[trm] >> accept rval2:Bool in
                               exit(rval2)
                             )
                          []
                          [rval eq false] -> exit(false)
          )
       )
    where
    process receiveinreceivemessage[trm] : exit(Bool) :=
       trm; exit(false)
    endproc (* receiveinsendMessage *)
      process assert[lStatus] : exit(Bool) :=
         pexpr[lStatus] >> accept lstatusv:linestatus in
          (
            [lstatusv eq STATGOOD] -> exit(false)
            []
            [lstatusv eq STATERROR] -> exit(true)
          )
      where
          process pexpr[lStatus] : exit(linestatus) :=
             lStatus?lval:linestatus; exit(lval)
          endproc (* pexpr *)
      endproc (* assert *)
      process sets[m, b, buffMsg, buffBval] : exit(Bool) :=
       setm[m, buffMsg] >> accept rval:Bool in
          [rval eq true] -> exit(true)
          []
          [rval eq false] -> setb[b, buffBval] >> accept rval:Bool in
             exit(rval)
      where
         process setm[m, buffMsg] : exit(Bool) :=
            pexpr[m] >> accept mval:message in
                buffMsg!mval; exit(false)
         where
            process pexpr[m] : exit(message) :=
               m?mval:message; exit(mval)
            endproc (* pexpr *)
         endproc (* setm *)
         process setb[b, buffBval] : exit(Bool) :=
          pexpr[b] >> accept bval:bit in
                buffBval!bval; exit(false)
         where
            process pexpr[b] : exit(bit) :=
               b?bval:bit; exit(bval)
            endproc (* pexpr *)
         endproc (* setb *)
      endproc (* sets *)
      process sendinreceivemessage[trm, m, b] : exit(Bool) :=
         pexpr1[m] >> accept mval:message in
          pexpr2[b] >> accept bval:bit in
             trm!mval!bval!false; exit(false)
      where
          process pexpr1[m] : exit(message) :=
               m?mval:message; exit(mval)
          endproc (* pexpr1 *)
          process pexpr2[b] : exit(bit) :=
               b?bval:bit; exit(bval)
          endproc (* pexpr2 *)
      endproc (* sendinreceivemesssage *)
```

```
               process failonreceivemessage[trm] : exit(Bool) :=
                  trm!NIL!0 of Bit!true; exit(false)
               endproc (* failonreceivemessage *)
            endproc (* optrmbody *)
         endproc (* optrm *)
      endproc (* translineOperations *)
endproc (* transline *)

endspec
```

sender

```
specification sender[sam, sgs, tsm, trm, asa, aga] : noexit

library
   ABPLIB
endlib

behavior
      sender[sam, sgs, tsm, aga]
where

(*
 *
 * SENDER
 *
 *)

process sender[sam, sgs, tsm, aga] : noexit :=
hide bval, statval in
(
   senderStatus[bval, statval](0 of bit, NOTSENT)
     |[bval, statval]|
   senderOperations[sam, sgs, tsm, aga, bval, statval]
)
where
   process senderStatus[bval, statval](bvalv:bit, statvalv:sndstatus) : noexit :=
         bval!WRITE?Xbval:bit; senderStatus[bval, statval](Xbval, statvalv)
      []
         statval!WRITE?Xstatval:sndstatus; senderStatus[bval, statval](bvalv, Xstatval)
      []
         bval!READ!bvalv; senderStatus[bval, statval](bvalv, statvalv)
      []
         statval!READ!statvalv; senderStatus[bval, statval](bvalv, statvalv)
   endproc (* senderStatus *)

   process senderOperations[sam, sgs, tsm, aga, bval, statval] : exit :=
         (
             opsam[sam, tsm, bval, statval]
           []
             opsgs[sgs, aga, bval, statval]
         )
            >> accept rval: Bool in
               (* silence operation failures *)
               (* We can place a stub here *)
                senderOperations[sam, sgs, tsm, aga, bval, statval]

      where
         process opsam[sam, tsm, bval, statval] : exit(Bool) :=
         (* receive in accept *)
            receiveinaccept[sam] >> accept m:message in
         (* parte1 do alt *)
            op1[sam, tsm, bval](m) >> accept rval:Bool in
               (
                  [rval eq true] ->
                     (
                        failonAccept[sam] >> accept rval:Bool in
                        exit(false)
                     )

                  []
                  [rval eq false] -> exit(false)
               )
         where
           process receiveinaccept[sam] : exit(message) :=
             sam?m:message; exit(m)
           endproc (* receiveinaccept *)
           process op1[sam, tsm, bval](m:message) : exit(Bool) :=
              (* ; send m, bval in transline.sendMessage *)
              sendintranslineSendMessage[tsm, bval](m) >> accept rval : Bool in
               ( [rval eq true] -> exit(true)
                 []
                 [rval eq false] ->
              (* ; receive in transline.sendMessage *)
                 receiveintranslSendMessage[tsm] >> accept rval:Bool in
                 (
                   [rval eq true] -> exit(true)
                   []
                   [rval eq false] ->
              (* ; send in accept *)
                    (
                     sendinAccept[sam] >> accept rval:Bool in
                        exit(rval)
                    )
                 )
               )
            where
               process sendintranslineSendMessage[tsm, bval](m:message) : exit(Bool) :=
                  pexpr[bval] >> accept bvalv:bit in
                        tsm!m!bvalv; exit(false)
               where
                  process pexpr[b] : exit(bit) :=
                       b!READ?bval:bit; exit(bval)
                  endproc (* pexpr *)
               endproc
               process receiveintranslSendMessage[tsm] : exit(Bool) :=
                  tsm?rval:Bool; exit(rval)
               endproc
               process sendinAccept[sam] : exit(Bool) :=
                  sam!false;exit(false)
               endproc
            endproc (* op1 *)
            process failonAccept[sam] : exit(Bool) :=
               sam!true;exit(true)
            endproc
         endproc (* opsam *)
```

```
process opsgs[sgs, aga, bval, statval] : exit(Bool) :=
 hide break, s in
    (
          lvars[s, break](NOTSENT)
         |[s, break]|
             (
                opsgsbody[sgs, aga, s, bval, statval] >>
                accept rval:Bool in
                 (
                   break!dval;
                   exit(rval)
                 )
             )
     )
where
    process lvars[s, break](sv:sndstatus) : exit(Bool) :=
        s!WRITE?Xs:sndstatus; lvars[s, break](Xs)
      []
        s!READ!sv; lvars[s, break](sv)
      []
        break?dummy:dtype; exit(false)
    endproc (* lvars *)
    process opsgsbody[sgs, aga, s, bval, statval] : exit(Bool) :=
           (* receive in getStatus *)
           receiveingetStatus[sgs] >> accept rval:Bool in
              [rval eq true] -> exit(true)
              []
              [rval eq false] ->
              (
              (* ; send in ackline.getAcknowledge *)
                 sendinacklinegetAcknowledge[aga] >> accept rval : Bool in
                 [rval eq true] -> exit(true)
                 []
                 [rval eq false] ->
              (* ; receive b in ackline.getAcknowledge *)
                  ( receiveinacklinegetAcknowledge[aga] >> accept b:Bit, rval:Bool in
                    [rval eq true] -> exit(true)
                    []
                    [rval eq false] ->
               (* assert b = bval *)
                    ( assert[bval](b) >> accept rval:Bool in
                      [rval eq true] ->  (*trocar*)
                       (
                          setstatvalnoack[statval]
                       )
                      []
                      [rval eq false] ->
                       (
                          sets[bval, statval]
                       )
                    ) >> accept rval:Bool in
               (* set s << statlval *)
                      (  [rval eq true] -> exit(true)
                         []
                         [rval eq false] -> setsstatval[s, statval] >> accept rval:Bool in
               (* sendingetstatus *)
                         (
                           [rval eq true] -> exit(true)
                           []
                           [rval eq false] -> sendingetStatus[sgs, s]
                         )
                      )

                   )
               (* fail on getstatus *)
               (* -- ignored --*)
                ) >> accept rval:Bool in
                (
                   [rval eq true] -> failongetstatus[sgs]
                   []
                   [rval eq false] -> exit(false)
                )
   where
      process receiveingetStatus[sgs] : exit(Bool) :=
         sgs; exit(false) (* isto poderia //// dar porcaria!!! *)
      endproc (* receiveingetStatus *)
      process sendinacklinegetAcknowledge[aga] : exit(Bool) :=
        aga; exit(false)
      endproc (* sendinacklinegetAcknowledge *)
      process receiveinacklinegetAcknowledge[aga] : exit(Bit, Bool) :=
        aga?b:Bit?rval:Bool; exit(b,rval)
      endproc (* receiveinacklinegetAcknowledge *)
      process assert[bval](b:Bit) : exit(Bool) :=
       pexpr[bval] >> accept bvalv:Bit in
          (
            [bvalv neq b] -> exit(true)
            []
            [bvalv eq b] -> exit(false)
          )
      where
          process pexpr[bval] : exit(Bit) :=
             bval!READ?bvalv:Bit; exit(bvalv)
          endproc (* pexpr *)
      endproc
      process sets[bval, statval] : exit(Bool) :=
        setbval[bval] >> accept rval:Bool in
          [rval eq true] -> exit(true)
          []
          [rval eq false] ->
            setstatval[statval]  >> accept rval:Bool in exit(rval)
      where
         process setbval[bval] : exit(Bool) :=
            pexpr[bval] >> accept bvalv:Bit in
                bval!WRITE!not(bvalv); exit(false)
         where
           process pexpr[bval] : exit(Bit) :=
              bval!READ?bvalv:Bit; exit(bvalv)
           endproc (* pexpr *)
```

```
                endproc (* setbval *)
                process setstatval[statval] : exit(Bool) :=
                  statval!WRITE!SENTANDACK; exit(false)
                endproc (* setstatval *)
           endproc (* sets *)
           process setstatvalnoack[statval] : exit(Bool) :=
                  statval!WRITE!SENTNOACK; exit(false)
           endproc (* setstatvalnoack *)
           process setsstatval[s, statval] : exit(Bool) :=
             pexpr[statval] >> accept svalv:sndstatus in
                s!WRITE!svalv; exit(false)
           where
              process pexpr[sval] : exit(sndstatus) :=
                 sval!READ?svalv:sndstatus; exit(svalv)
              endproc (* pexpr *)
           endproc (* setsstatval *)
           process sendingetStatus[sgs, s] : exit(Bool) :=
             pexpr[s] >> accept svalv:sndstatus in
                sgs!svalv!false; exit(false)
           where
              process pexpr[sval] : exit(sndstatus) :=
                 sval!READ?svalv:sndstatus; exit(svalv)
              endproc (* pexpr *)
           endproc (* sendingetStatus *)
           process failongetStatus[sgs] : exit(Bool) :=
               sgs!NOTSENT!true; exit(false)
           endproc (* failongetStatus *)
         endproc (* opsgsbody *)
       endproc (* optsgs *)
     endproc (* senderOperations *)
endproc (* sender *)

endspec
```

receiver

```
specification sender[rdr, rgs] : noexit

library
   ABPLIB
endlib

behavior
          receiver[rdr, rgs, trm, asa]
where

(*
 *
 * RECEIVER
 *
 *)

process receiver[rdr, rgs, trm, asa] : noexit :=
hide bval, statval in
(
   receiverStatus[bval, statval](0 of bit, NOMESSAGE)
     |[bval, statval]|
   receiverOperations[rdr, rgs, trm, asa, bval, statval]
)
where
   process receiverStatus[bval, statval](bvalv:bit, statvalv:rcvstatus) : noexit :=
         bval!WRITE?Xbval:bit; receiverStatus[bval, statval](Xbval, statvalv)
      []
         statval!WRITE?Xstatval:rcvstatus; receiverStatus[bval, statval](bvalv, Xstatval)
      []
         bval!READ!bvalv; receiverStatus[bval, statval](bvalv, statvalv)
      []
         statval!READ!statvalv; receiverStatus[bval, statval](bvalv, statvalv)
   endproc (* receiverStatus *)

   process receiverOperations[rdr, rgs, trm, asa, bval, statval] : exit :=
         (
             oprdr[rdr, trm, asa, bval, statval]
           []
             oprgs[rgs, statval]
         )
            >> accept rval: Bool in
               (* silence operation failures *)
               (* We can place a stub here *)
                receiverOperations[rdr, rgs, trm, asa, bval, statval]
      where
        process oprdr[rdr, trm, asa, bval, statval] : exit(Bool) :=
          hide break, m in
            (
                     lvars[m, break](NIL)
                 |[m, break]|
                     (
                        oprdrbody[rdr, trm, asa, m, bval, statval] >>
                        accept rval:Bool in
                         (
                           break!dval;
                           exit(rval)
                         )
                     )
             )
        where
            process lvars[m, break](mv:message) : exit(Bool) :=
                m!WRITE?Xm:message; lvars[m, break](Xm)
              []
                m!READ!mv; lvars[m, break](mv)
              []
                break?dummy:dtype; exit(false)
            endproc (* lvars *)
            process oprdrbody[rdr, trm, asa, m, bval, statval] : exit(Bool) :=
              (* receive in deliver *)
                 receiveindeliver[rdr] >> accept rval:Bool in
              (* ; send in transline.receiveMessage *)
                 sendintranslineReceiveMessage[trm, bval] >> accept rval : Bool in
                 (
                    [rval eq true] -> exit(true)
                    []
                    [rval eq false] ->
              (* ; receive msgparm, bparm in transline.receiveMessage *)
                    (
                     receiveintranslineReceiveMessage[trm] >> accept msgparm:message, bparm:bit, rval:Bool in
                     [rval eq true] -> exit(true)
                     []
                     [rval eq false] ->
                (* ; assert bparm =/= bval *)
                       ( assert[bval](bparm) >> accept rval:Bool in
                          (
                              [rval eq true] ->
                              (
                                (* set statval << samemessage *)
                                setstatvalsamemessage[statval]
                              )
                            []
                              [rval eq false] ->
                              (
                                (* send bparm in ackline.sendAcknowledge *)
                                sendinacklinesendack[asa](bparm) >> accept rval: Bool in
                                (
                                  [rval eq true] -> exit(false)
                                  []
                                  [rval eq false] -> receiveinacklinesendack[asa] >> accept rval:Bool in
                                  (
                                    [rval eq true] -> exit(false)
                                    []
                                    [rval eq false] -> sets[m, bval, statval](msgparm, bparm)
                                   )
```

```
                                 )
                              )  >> accept rval:Bool in
                                      (
                                        [rval eq true] -> exit(true)
                                        []
                                        [rval eq false] -> sendminDeliver[m, rdr] >> accept rval:Bool in
                                            exit(rval)
                                      )
                            )

                       )
                     )
                   )
            where
              process receiveinDeliver[rdr] : exit(Bool) :=
                rdr; exit(false)
              endproc (* receiveinDeliver *)
              process sendintranslineReceiveMessage[trm, bval] : exit(Bool) :=
                trm; exit(false)
              endproc (* sendintranslineReceiveMessage *)
              process receiveintranslineReceiveMessage[trm] : exit(message, bit, Bool) :=
                trm?m:message?b:bit?rval:Bool; exit(m,b,rval)
              endproc (* receiveintranslineReceiveMessage *)
              process assert[bval](b:Bit) : exit(Bool) :=
               pexpr[bval] >> accept bvalv:Bit in
                  (
                    [bvalv eq b] -> exit(true)
                    []
                    [bvalv neq b] -> exit(false)
                  )
              where
                  process pexpr[bval] : exit(Bit) :=
                     bval!READ?bvalv:Bit; exit(bvalv)
                  endproc (* pexpr *)
              endproc (* assert *)
              process sets[m, bval, statval](msgparm:message, bparm:bit) : exit(Bool) :=
                setm[m](msgparm) >> accept rval:Bool in
                  [rval eq true] -> exit(true)
                  []
                  [rval eq false] ->  setbval[bval](bparm) >> accept rval:Bool in
                    [rval eq true] -> exit(true)
                    []
                    [rval eq false] ->
                       setstatval[statval]  >> accept rval:Bool in exit(rval)
               where
                 process setm[m](msgparm:message) : exit(Bool) :=
                    m!WRITE!msgparm; exit(false)
                 endproc (* setm *)
                 process setbval[bval](bparm:bit) : exit(Bool) :=
                   bval!WRITE!bparm; exit(false)
                 endproc (* setbval *)
                 process setstatval[statval] : exit(Bool) :=
                   statval!WRITE!NEWMESSAGE; exit(false)
                 endproc (* setstatval *)
              endproc (* sets *)
              process setstatvalsamemessage[statval] : exit(Bool) :=
                   statval!WRITE!SAMEMESSAGE; exit(false)
              endproc (* setstatvalsamemessage *)
              process sendinacklinesendack[asa](bparm:bit) : exit(Bool) :=
                   asa!(bparm of bit); exit(false)
              endproc (* pexpr *)
              process receiveinacklinesendack[asa] : exit(Bool) :=
                   asa?rval:Bool; exit(rval)
              endproc (* pexpr *)
              process sendminDeliver[m, rdr] : exit(Bool) :=
                  pexpr[m] >> accept mv:message in
                        rdr!mv!false; exit(false)
              where
                  process pexpr[m] : exit(message) :=
                       m!READ?mval:message; exit(mval)
                  endproc (* pexpr *)
             endproc (* sendminDeliever *)
            endproc (* oprdrbody *)
         endproc (* oprdr *)

         process oprgs[rgs, statval] : exit(Bool) :=
          (* receive in getStatus *)
           receiveingetStatus[rgs] >> accept rval:Bool in
             [rval eq true] -> exit(true)
             []
             [rval eq false] ->
              (* ; send s in getStatus *)
               sendsingetStatus[rgs, statval]
           where
             process receiveinGetStatus[rgs] : exit(Bool) :=
                rgs; exit(false)
              endproc (* receiveinDeliver *)
            process sendsingetStatus[rgs, s] : exit(Bool) :=
              pexpr[s] >> accept svalv:rcvstatus in
                 rgs!svalv!false; exit(false)
            where
               process pexpr[sval] : exit(rcvstatus) :=
                  sval!READ?svalv:rcvstatus; exit(svalv)
               endproc (* pexpr *)
           endproc (* sendingetStatus *)
        endproc (* optrgs *)
      endproc (* receiverOperations *)
endproc (* receiver *)

endspec
```

Appendix E – Summary of properties verifyed with CADP

Property	CADP concret Syntax
EF(*ackline.sendacknowledge*(a_i) $\Rightarrow$ EF(*ackline.getAcknowledge.fail*=true))	`pot( <"ASA !0">T => pot( <"AGA !0 !TRUE">T ) )`
	`pot( <"ASA !1">T => pot( <"AGA !0 !TRUE">T ) )`
EF((*ackline.sendacknowledge*(a_I) $\wedge$ AF(*ackline.getAcknowledge.fail*=false)) $\Rightarrow$ AF(*ackline.getAcknowledge*(a_I))	`pot( (<"ASA !0">T) AND inev(<"AGA !0 !FALSE”>) ) => inev(<”AGA !0 !FALSE”>)`
	`pot( (<"ASA !1">T AND inev(<"AGA !0 !FALSE”>) ) => inev(<”AGA !1 !FALSE”>) )`
EF(*sender.accept*(m_i) $\Rightarrow$ AF(*receiver.deliver*(m_j) $\wedge$ AF(*receiver.getStatus*(SAMEMESSAGE)))	`pot( <"SAM !NIL">T => inev( <"RGM !SOMETHING !FALSE">T AND inev(<"RGS !SAMEMESSAGE !FALSE">T )))`
	`pot( <"SAM !SOMETHING">T => inev( <"RGM !SOMETHING !FALSE">T AND inev(<"RGS !NIL !FALSE">T )))`
AF((*sender.accept*(m_i) $\wedge$ AF(*transline.receiveMessage.fail*=false)) $\Rightarrow$ AF(*receiver.deliver*(m_i)))	`inev((<"SAM !NIL">T AND inev(<"TRM !NIL !FALSE">T)) => inev(<"RDR !NIL !FALSE">T)))`
	`inev((<"SAM !SOMETHING">T AND inev(<"TRM !SOMETHING !FALSE">T)) => inev(<"RDR !SOMETHING !FALSE">T)))`
AF(*ackline.getAcknowledge.fail*=false) $\Rightarrow$ AF(*sender.getStatus*(*SENTANDACK*))	`(<"SAM !NIL">T AND inev(<"RDR !NIL">T)) => inev(<"SGS !SENTANDACK !FALSE">T))`

Appendix F – UPPAAL .ta specification of the ABP

```
//
// UPPAAL automata version of the alternating bit protocol
//

//sndstatus = enum : NOTSENT = 0, SENTNOACK = 1, SENTANDACK = 2
//rcvstatus = enum : NOMESSAGE = 0, NEWMESSAGE = 1, SAMEMESSAGE = 2
//linestatus = enum : STATERROR = 0, STATGOOD = 1
//message = enum : NIL = 0, SOMETHING = 1
//bit = enum : false = 0, true = 1
//recover = enum : no = 0, yes = 1

/////////////////////////////////////////////////////////////////////
// SENDER data area                                              //
/////////////////////////////////////////////////////////////////////

// Attributes
int sr_bval;
int sr_statval;

// Channels
urgent chan sam_call, sam_parm;
urgent chan sgs_call, sgs_parm;

// AssocMem(sender.accept)
// IN-parms
int in_sam_m;
// OUT-parms
int out_sam_recover;

// AssocMem(sender.getStatus)
// IN-parms
// OUT-parms
int out_sgs_s;
int out_sgs_recover;

// LocalMem(sender.getStatus)
// LOCAL-parms
int local_sgs_s;
int local_sgs_b;
int local_sgs_recover;

/////////////////////////////////////////////////////////////////////
// TRANSLINE data area                                           //
/////////////////////////////////////////////////////////////////////

// Attributes
int tl_buffMsg;
int tl_buffBVal;
int tl_lstatus;

// Channels
urgent chan tsm_call, tsm_parm;
urgent chan trm_call, trm_parm;

// AssocMem(transline.sendMessage)
// IN-parms
int in_tsm_m;
int in_tsm_b;
// Out-parms
int out_tsm_recover;

// AssocMem(transline.receiveMessage)
// IN-parms
// OUT-oparms
int out_trm_m;
int out_trm_b;
int out_trm_recover;

// LocalMem(transline.receiveMessage)
// LOCAL-parms
int local_trm_m;
int local_trm_b;

// For the test boxes
//
int tl_recover_status;


/////////////////////////////////////////////////////////////////////
// ACKLINE data area                                             //
/////////////////////////////////////////////////////////////////////

// Attributes
int al_buffBval;
int al_lstatus;

// Channels
urgent chan asa_call, asa_parm;
urgent chan aga_call, aga_parm;

// AssocMem(ackline.sendAcknowledge)
// IN-parms
int in_asa_b;
// Out-parms
```

```
int out_asa_recover;

// AssocMem(ackline.getAcknowledge)
// IN-parms
// OUT-oparms
int out_aga_b;
int out_aga_recover;

// LocalMem(ackline.getAcknowledge)
// LOCAL-parms
int local_aga_b;

// For the test boxes
//
int al_recover_status;

/////////////////////////////////////////////////////////////////////
// RECEIVER data area                                              //
/////////////////////////////////////////////////////////////////////

// Attributes
int rr_bval;
int rr_statval;

// Channels
urgent chan rdr_call, rdr_parm;
urgent chan rgs_call, rgs_parm;

// AssocMem(receiver.deliver)
// IN-parms
// OUT-parms
int out_rdr_m;
int out_rdr_recover;

// AssocMem(receiver.getStatus)
// IN-parms
// OUT-parms
int out_rgs_s;
int out_rgs_recover;

// LocalMem(receiver.deliver)
// LOCAL-parms
int local_rdr_m;
int local_rdr_msgparm;
int local_rdr_bparm;
int local_rdr_recover;

// LocalMem(receiver.getStatus)
// LOCAL-parms
int local_rgs_s;


/////////////////////////////////////////////////////////////////////
// SENDER object module                                             //
/////////////////////////////////////////////////////////////////////

process sender {
   state
          sr_init,
          sr_main,
        sam_entry_l1, sgs_entry_l1,
          sam_entry_l2, sgs_entry_l2,
          sam_entry_l3, sgs_entry_l3,
          sam_send_l1 , sam_send_l2, sam_send_l3,
          sam_receive_l2, sam_receive_l3,
          sgs_send_l1, sgs_send_l2, sgs_send_l3,
          sgs_receive_l1, sgs_receive_l2, sgs_receive_l3,
          sgs_receive_l4, sgs_receive_l5,
          sgs_assert_l1, sgs_assert_l2, sgs_assert_l3,
          sgs_setbval , sgs_setstatval,
          sgs_sets_l1, sgs_sets_l2, sgs_setstatval_noack,
          sgs_sendgs_l1 , sgs_sendgs_l2, sgs_sendgs_l3,
          sgs_failon_l1, sgs_failon_l2, sgs_failon_l3,
          sam_sendinaccept_l1, sam_sendinaccept_l2,
          sr_end;

   init sr_init;
   trans

          sr_init -> sr_main {
                    assign
                              sr_bval := 0,
                              sr_statval := 0;
          },
        // operation dispacher
        sr_main -> sam_entry_l1 {
        },
        sr_main -> sgs_entry_l1 {
        },

        ///////////////////
        // ACCEPT
        ///////////////////

          // receive in SendMessage
          sam_entry_l1 -> sam_entry_l2 {
                    sync sam_call?;
        },
        sam_entry_l2 -> sam_entry_l3 {
                    sync sam_parm!;
        },
        // assert true -- simplifyed!
          sam_entry_l3 -> sam_send_l1 {
          },
          // send m, bval in transline.sendMessage
          sam_send_l1 -> sam_send_l2 {
                    sync tsm_call!;
                    assign in_tsm_m := in_sam_m, in_tsm_b := sr_bval;
```

```
  },
  sam_send_l2 -> sam_send_l3 {
          sync tsm_parm?;
  },
  // receive in transline.sendMessage
  sam_send_l3 -> sam_receive_l2 {
          sync tsm_call?;
  },
  sam_receive_l2 -> sam_receive_l3 {
          sync tsm_parm!;
  },
  sam_receive_l3 -> sam_sendinaccept_l1 {
          sync sam_call!;
  },
  sam_sendinaccept_l1 -> sam_sendinaccept_l2 {
          sync sam_parm?;
  },
  sam_sendinaccept_l2 -> sr_end {
  },

  ////////////////////
  // GETSTATUS
  ////////////////////

  // receive in getMessage
  sgs_entry_l1 -> sgs_entry_l2 {
          sync sgs_call?;
},
sgs_entry_l2 -> sgs_entry_l3 {
          sync sgs_parm!;
},
  sgs_entry_l3 -> sgs_send_l1 {
  },
  // send in ackine.getAcknowledge
  sgs_send_l1 -> sgs_send_l2 {
          sync aga_call!;
},
sgs_send_l2 -> sgs_send_l3 {
          sync aga_parm?;
},
  sgs_send_l3 -> sgs_receive_l1 {
  },
  // receive b in ackline.getAcknowledge
  sgs_receive_l1 -> sgs_receive_l2 {
          sync aga_call?;
},
sgs_receive_l2 -> sgs_receive_l3 {
          sync aga_parm!;
          assign
                    local_sgs_b := out_aga_b,
                    local_sgs_recover := out_aga_recover;
},
  // Its OK!
  sgs_receive_l3 -> sgs_receive_l4 {
          guard local_sgs_recover == 0; // NOERROR
  },
  // Gotta recover!
  sgs_receive_l3 -> sgs_receive_l5 {
          guard local_sgs_recover == 1;
  },
  sgs_receive_l5 -> sgs_failon_l1 {
  },
  // assert b = bval
  sgs_receive_l4 -> sgs_assert_l1 {
  },
  // good !
  sgs_assert_l1 -> sgs_assert_l2 {
          guard local_sgs_b == sr_bval;
  },
  sgs_assert_l1 -> sgs_assert_l3 {
          guard local_sgs_b > sr_bval;
  },
  sgs_assert_l1 -> sgs_assert_l3 {
          guard local_sgs_b < sr_bval;
  },
  // set bval << not bval
  sgs_assert_l2 -> sgs_setbval {
          assign sr_bval := 1 - sr_bval;
  },
  // set statval << SENTANDACK
  sgs_setbval -> sgs_setstatval {
          assign sr_statval := 2; // SENTANDACK
  },
  sgs_setstatval -> sgs_sets_l1 {
  },
  // alt part ...
  //   set statval << SENTNOACK
  sgs_assert_l3 -> sgs_setstatval_noack {
          assign sr_statval := 1; // SENTNOACK
  },
  sgs_setstatval_noack -> sgs_sets_l1 {
  },
  // set s << statval
  sgs_sets_l1 -> sgs_sets_l2 {
          assign local_sgs_s := sr_statval;
  },
  // send s in getStatus
  sgs_sets_l2 -> sgs_sendgs_l1 {
  },
  sgs_sendgs_l1 -> sgs_sendgs_l2 {
          sync sgs_call!;
          assign out_sgs_s := local_sgs_s;
},
sgs_sendgs_l2 -> sgs_sendgs_l3 {
          sync sgs_parm?;
},
  // goto end:
  sgs_sendgs_l3 -> sr_end {
```

```
          },
          // alt
          // failon getStatus
          sgs_failon_l1 -> sgs_failon_l2 {
                    sync sgs_call!;
                    assign out_sgs_recover := 1; // The system is in failure state!
        },
        sgs_failon_l2 -> sgs_failon_l3 {
                    sync sgs_parm?;
          },
          sgs_failon_l3 -> sr_end {
          },
          // Loopback
          sr_end -> sr_main {
          };
}


/////////////////////////////////////////////////////////////////////
// TRANSLINE object module                                          //
/////////////////////////////////////////////////////////////////////

process transline {
   state
          tl_init,
          tl_main, tl_end,
        tsm_entry_l1, trm_entry_l1,
        tsm_entry_l2, tsm_entry_l3,
        trm_entry_l2, trm_entry_l3,
          tsm_or_l1, tsm_or_l2,
        tsm_setlstatus1_l1, tsm_setlstatus2_l1, tsm_setlstatus2_l2,
          tsm_setBuffMsg_l1,
          tsm_setBuffBVal_l1, tsm_setBuffBVal_l2,
          tsm_failon_l1, tsm_failon_l2, tsm_failon_l3,
        tsm_send_l1, tsm_send_l2, tsm_send_l3,
          trm_assert_l1, trm_assert_l2, trm_assert_l3,
        trm_setm_l1, trm_setb_l1, trm_setb_l2,
        trm_send_l1, trm_send_l2, trm_send_l3,
          trm_failon_l1, trm_failon_l2, trm_failon_l3,
          tr_end;


   init tl_init;
   trans

          tl_init -> tl_main {
                    assign
                              tl_buffBVal := 0,
                              tl_lstatus := 1;
          },
        // operation dispacher
        tl_main -> tsm_entry_l1 {
        },
        tl_main -> trm_entry_l1 {
        },

        ///////////////////
        // SENDMESSAGE
        ///////////////////

          // receive in SendMessage
          tsm_entry_l1 -> tsm_entry_l2 {
                    sync tsm_call?;
        },
        tsm_entry_l2 -> tsm_entry_l3 {
                    sync tsm_parm!;
        },
        // assert true -- simplifyed!
          // OR
        tsm_entry_l3 -> tsm_or_l1 {
          },
          //   -- To member 1
        tsm_or_l1 -> tsm_setlstatus1_l1 {
          },
          //   -- To member 2
        tsm_or_l1 -> tsm_setlstatus2_l1 {
          },
          //
        // OR
          //    FirstMember
        //
        // set lstatus << STATGOOD
        tsm_setlstatus1_l1 -> tsm_setBuffMsg_l1 {
                    assign tl_lstatus := 1; // 1 = STATGOOD
        },
          // set buffMessage << m
        tsm_setBuffMsg_l1 -> tsm_setBuffBVal_l1 {
                    assign tl_buffMsg := in_tsm_m;
        },
        // set buffBval << m
        tsm_setBuffBVal_l1 -> tsm_setBuffBVal_l2 {
                    assign tl_buffBVal := in_tsm_b;
        },
        tsm_setBuffBVal_l2 -> tsm_or_l2 {
        },
        //
        // OR
        //    Second Member
        //
        // set lstatus << STATERROR
        tsm_setlstatus2_l1 -> tsm_setlstatus2_l2 {
                    assign tl_lstatus := 0; // 0 = STATERROR
          },
          tsm_setlstatus2_l2 -> tsm_or_l2 {
          },
          //
          tsm_or_l2 -> tsm_send_l1 {
          },
```

```
        // send in SendMessage
        tsm_send_l1 -> tsm_send_l2 {
              sync tsm_call!;
                assign out_tsm_recover := 0;
        },
        tsm_send_l2 -> tsm_send_l3 {
              sync tsm_parm?;
        },
        tsm_send_l3 -> tl_end {
        },


          ////////////////////
          // GETMESSAGE
          ////////////////////

          // receive in getMessage
          trm_entry_l1 -> trm_entry_l2 {
                    sync trm_call?;
        },
        trm_entry_l2 -> trm_entry_l3 {
                    sync trm_parm!;
        },
          trm_entry_l3 -> trm_assert_l1 {
          },
          // assert lstatus = STATGOOD
          trm_assert_l1 -> trm_assert_l2 {
                    guard tl_lstatus == 1; // STATGOOD
          },
          trm_assert_l1 -> trm_assert_l3 {
                    guard tl_lstatus < 1; //
          },
          // Its ok!
          trm_assert_l2 -> trm_setm_l1 {
          },
          // It fails
          trm_assert_l3 -> trm_failon_l1 {
          },
          // set m << buffMessage
        trm_setm_l1 -> trm_setb_l1 {
                    assign local_trm_m := tl_buffMsg;
        },
        // set b << buffBval
        trm_setb_l1 -> trm_setb_l2 {
                    assign local_trm_b := tl_buffBVal;
        },
          trm_setb_l2 -> trm_send_l1 {
        },
          // send in SendMessage
        trm_send_l1 -> trm_send_l2 {
                    sync trm_call!;
                    assign out_trm_m := local_trm_m,
                           out_trm_b := local_trm_b,
                           out_trm_recover := 0;
        },
        trm_send_l2 -> trm_send_l3 {
                    sync trm_parm?;
        },
        trm_send_l3 -> tr_end {
        },
          // failon receiveMessage
          trm_failon_l1 -> trm_failon_l2 {
                    sync trm_call!;
                    assign out_trm_recover := 1;
          },
        trm_failon_l2 -> trm_failon_l3 {
                    sync trm_parm?;
        },
        trm_failon_l3 -> tr_end {
        },
          tr_end -> tl_end {
          },
          // LoopBack
          tl_end -> tl_main {
          };
}


/////////////////////////////////////////////////////////////////////
// ACKLINE object module                                                //
/////////////////////////////////////////////////////////////////////

process ackline {
   state
          al_init,
          al_main, al_end,
        asa_entry_l1, aga_entry_l1,
        asa_entry_l2, asa_entry_l3,
          aga_entry_l2, aga_entry_l3,
          asa_or_l1, asa_or_l2,
        asa_setlstatus1_l1, asa_setlstatus2_l1, asa_setlstatus2_l2,
          asa_setBuffMsg_l1,
          asa_setBuffBVal_l1, asa_setBuffBVal_l2,
          asa_failon_l1, asa_failon_l2, asa_failon_l3,
        asa_send_l1, asa_send_l2, asa_send_l3,
          aga_assert_l1, aga_assert_l2, aga_assert_l3,
        aga_setm_l1, aga_setb_l1, aga_setb_l2,
        aga_send_l1, aga_send_l2, aga_send_l3,
          aga_failon_l1, aga_failon_l2, aga_failon_l3,
          asa_end;


   init al_init;
   trans
          al_init -> al_main {
                    assign
                              al_buffBval := 0,
                              al_lstatus := 1;
```

```
  },
// operation dispacher
al_main -> asa_entry_l1 {
},
al_main -> aga_entry_l1 {
},

////////////////////////
// SENDACKNOWLEDGE
////////////////////////

  // receive in SendAcknowledge
  asa_entry_l1 -> asa_entry_l2 {
            sync asa_call?;
},
asa_entry_l2 -> asa_entry_l3 {
            sync asa_parm!;
},
// assert true -- simplifyed!
  // OR
asa_entry_l3 -> asa_or_l1 {
  },
  //   -- To member 1
asa_or_l1 -> asa_setlstatus1_l1 {
  },
  //   -- To member 2
asa_or_l1 -> asa_setlstatus2_l1 {
  },
  //
// OR
  //    FirstMember
//
// set lstatus << STATGOOD
asa_setlstatus1_l1 -> asa_setBuffBVal_l1 {
            assign al_lstatus := 1; // 1 = STATGOOD
},
// set buffBval << b
asa_setBuffBVal_l1 -> asa_setBuffBVal_l2 {
            assign al_buffBval := in_asa_b;
},
asa_setBuffBVal_l2 -> asa_or_l2 {
},
//
// OR
//    Second Member
//
// set lstatus << STATERROR
asa_setlstatus2_l1 -> asa_setlstatus2_l2 {
            assign al_lstatus := 0; // 0 = STATERROR
  },
  asa_setlstatus2_l2 -> asa_or_l2 {
  },
  //
  asa_or_l2 -> asa_send_l1 {
  },
// send in SendAcknowledge
asa_send_l1 -> asa_send_l2 {
      sync asa_call!;
        assign out_asa_recover := 0;
},
asa_send_l2 -> asa_send_l3 {
      sync asa_parm?;
},
asa_send_l3 -> asa_end {
  },
  asa_end -> al_end {
},



  ////////////////////
  // GETACKNOWLEDGE
  ////////////////////

  // receive in getMessage
  aga_entry_l1 -> aga_entry_l2 {
            sync aga_call?;
},
aga_entry_l2 -> aga_entry_l3 {
            sync aga_parm!;
},
  aga_entry_l3 -> aga_assert_l1 {
  },
  // assert lstatus = STATGOOD
  aga_assert_l1 -> aga_assert_l2 {
            guard al_lstatus == 1; // STATGOOD
  },
  aga_assert_l1 -> aga_assert_l3 {
            guard al_lstatus < 1; // not good
  },
  // Its ok!
  aga_assert_l2 -> aga_setb_l1 {
  },
  // It fails
  aga_assert_l3 -> aga_failon_l1 {
  },
  // set b << buffBval
aga_setb_l1 -> aga_setb_l2 {
            assign local_aga_b := al_buffBval;
},
aga_setb_l2 -> aga_send_l1 {
},
  // send in GetAcknowledge
aga_send_l1 -> aga_send_l2 {
            sync aga_call!;
            assign out_aga_b := local_aga_b,
                   out_aga_recover := 0;
},
```

```
        aga_send_l2 -> aga_send_l3 {
                    sync aga_parm?;
        },
        aga_send_l3 -> al_end {
        },
          // failon getAcknowledge
          aga_failon_l1 -> aga_failon_l2 {
                    sync aga_call!;
                    assign out_aga_recover := 1;
          },
        aga_failon_l2 -> aga_failon_l3 {
                    sync aga_parm?;
        },
        aga_failon_l3 -> al_end {
        },
          // LoopBack
          al_end -> al_main {
          };
}


////////////////////////////////////////////////////////////////////
// RECEIVER object module                                          //
////////////////////////////////////////////////////////////////////

process receiver {
   state
          rr_init,
          rr_main,
        rgs_entry_l1, rdr_entry_l1,
          rgs_entry_l2, rdr_entry_l2,
          rgs_entry_l3, rdr_entry_l3,
          rgs_send_l1 , rgs_send_l2, rgs_send_l3,
          rgs_receive_l2, rgs_receive_l3,
          rdr_send_l1, rdr_send_l2, rdr_send_l3,
          rdr_receive_l1, rdr_receive_l2, rdr_receive_l3,
          rdr_receive_l4, rdr_receive_l5,
          rdr_assert_l1, rdr_assert_l2, rdr_assert_l3,
          rdr_setbval , rdr_setstatval,
          rdr_sendm_l1, rdr_sendm_l2, rdr_sendm_l3, rdr_setstatval_samemsg,
          rdr_sendb_l2 , rdr_sendb_l3, rdr_setmsgparm,
          rdr_failon_l1, rdr_failon_l2, rdr_failon_l3,
          rgs_sets,
          rr_end;

   init rr_init;
   trans

          rr_init -> rr_main {
                    assign
                              rr_statval := 0,
                              rr_bval := 1;
          },
        // operation dispacher
        rr_main -> rgs_entry_l1 {
        },
        rr_main -> rdr_entry_l1 {
        },

          ////////////////////
          // DELIVER
          ////////////////////

          // receive in deliver
          rdr_entry_l1 -> rdr_entry_l2 {
                    sync rdr_call?;
        },
        rdr_entry_l2 -> rdr_entry_l3 {
                    sync rdr_parm!;
        },
          rdr_entry_l3 -> rdr_send_l1 {
          },
          // send in transline.receiveMessage
          rdr_send_l1 -> rdr_send_l2 {
                    sync trm_call!;
        },
        rdr_send_l2 -> rdr_send_l3 {
                    sync trm_parm?;
        },
          rdr_send_l3 -> rdr_receive_l1 {
          },
          // receive msgparm, bparm in transline.receiveMessage
          rdr_receive_l1 -> rdr_receive_l2 {
                    sync trm_call?;
        },
        rdr_receive_l2 -> rdr_receive_l3 {
                    sync trm_parm!;
                    assign
                              local_rdr_msgparm := out_trm_m,
                              local_rdr_bparm := out_trm_b,
                              local_rdr_recover := out_trm_recover;
        },
          // Its OK!
          rdr_receive_l3 -> rdr_receive_l4 {
                    guard local_rdr_recover == 0; // NOERROR
          },
          // Gotta recover!
          rdr_receive_l3 -> rdr_receive_l5 {
                    guard local_rdr_recover == 1;
          },
          rdr_receive_l5 -> rdr_failon_l1 {
          },
          // assert bparm =/= bval
          rdr_receive_l4 -> rdr_assert_l1 {
          },
          // Bad !
          rdr_assert_l1 -> rdr_assert_l2 {
                    guard local_rdr_bparm == rr_bval;
```

```
          },
          // Good !
          rdr_assert_l1 -> rdr_assert_l3 {
                    guard local_rdr_bparm > rr_bval;
          },
          rdr_assert_l1 -> rdr_assert_l3 {
                    guard local_rdr_bparm < rr_bval;
          },
          // send b in ackline.sendAcknowledge
          rdr_assert_l3 -> rdr_sendb_l2 {
                    sync asa_call!;
                    assign in_asa_b := local_rdr_bparm;
          },
          rdr_sendb_l2 -> rdr_sendb_l3 {
                    sync asa_parm?;
          },
          // set m << msgparm
          rdr_sendb_l3 -> rdr_setmsgparm {
                    assign local_rdr_m := local_rdr_msgparm;
          },
          // set bval << bparm
          rdr_setmsgparm -> rdr_setbval {
                    assign rr_bval := local_rdr_bparm;
          },
          // set statval << NEWMESSAGE
          rdr_setbval -> rdr_setstatval {
                    assign rr_statval := 1; // NEWMESSAGE
          },
          rdr_setstatval -> rdr_sendm_l1 {
          },
          // alt part ...
          //   set statval << SAMEMESSAGE
          rdr_assert_l2 -> rdr_setstatval_samemsg {
                    assign rr_statval := 2; // SAMEMESSAGE
          },
          rdr_setstatval_samemsg -> rdr_sendm_l1 {
          },
          // send m in deliver
          rdr_sendm_l1 -> rdr_sendm_l2 {
                  sync rdr_call!;
                    assign out_rdr_m := local_rdr_m;
        },
        rdr_sendm_l2 -> rdr_sendm_l3 {
                    sync rdr_parm?;
        },
          // goto end:
          rdr_sendm_l3 -> rr_end {
          },
          // alt
          // failon deliver
          rdr_failon_l1 -> rdr_failon_l2 {
                    sync rdr_call!;
                    assign out_rdr_recover := 1; // The system is in failure state!
        },
        rdr_failon_l2 -> rdr_failon_l3 {
                    sync rdr_parm?;
          },
          rdr_failon_l3 -> rr_end {
          },

          //////////////////
        // GETSTATUS
        //////////////////

          // receive in SendMessage
          rgs_entry_l1 -> rgs_entry_l2 {
                    sync rgs_call?;
        },
        rgs_entry_l2 -> rgs_entry_l3 {
                    sync rgs_parm!;
        },
        // assert true -- simplifyed!
          // set s << statval
          rgs_entry_l3 -> rgs_sets {
                    assign local_rgs_s := rr_statval;
          },
        rgs_sets -> rgs_send_l1 {
        },
          // send s in getStatus
          rgs_send_l1 -> rgs_send_l2 {
                    sync rgs_call!;
                    assign out_rgs_s := local_rgs_s;
          },
          rgs_send_l2 -> rgs_send_l3 {
                    sync rgs_parm?;
          },
          rgs_send_l3 -> rr_end {
          },
          // Loopback
          rr_end -> rr_main {
          };
}
```

Appendix G – UPPAAL .ta specification of test automata and properties

```
//
// Test automata for the verification of progress properties
//

////////////////////////////////////////////////////////////////////
// PROGRAM                                           //
//        call ackline.sendAcknowledge(0);           //
//        call ackline.getAcknowledge( out_aga_b, out_aga_recover )  //
////////////////////////////////////////////////////////////////////

process call_ackline_send0_get {
        state
                call_l1, call_l2, call_l3, call_l4, transmit0,
                call_l6, call_l7, call_l8, call_performed, call_bad;
        init
                call_l1;
        trans
                call_l1 -> call_l2 {
                        sync asa_call!;
                },
                call_l2 -> call_l3 {
                        sync asa_parm?;
                        assign
                                in_asa_b := 0;
                },
                call_l3 -> call_l4 {
                        sync asa_call?;
                },
                call_l4 -> transmit0 {
                        sync asa_parm!;
                },
                transmit0 -> call_l6 {
                        sync aga_call!;
                },
                call_l6 -> call_l7 {
                        sync aga_parm?;
                },
                call_l7 -> call_l8 {
                        sync aga_call?;
                },
                call_l8 -> call_performed {
                        sync aga_parm!;
                        //assign
                                //al_recover_status := out_aga_recover;
                },
                call_performed -> call_bad {
                        guard out_aga_b > 0, // ack =/= ack2 c/ ack=0
                        al_lstatus == 1;     // ackline.lStatus == STATGOOD
                };
}

////////////////////////////////////////////////////////////////////
// PROGRAM                                                              //
//        call ackline.sendAcknowledge(1);                          //
//        call ackline.getAcknowledge( out_aga_b, out_aga_recover )  //
////////////////////////////////////////////////////////////////////

process call_ackline_send1_get {
        state
                call_l1, call_l2, call_l3, call_l4, transmit0,
                call_l6, call_l7, call_l8, call_performed, call_bad;
        init
                call_l1;
        trans
                call_l1 -> call_l2 {
                        sync asa_call!;
                },
                call_l2 -> call_l3 {
                        sync asa_parm?;
                        assign
                                in_asa_b := 1;
                },
                call_l3 -> call_l4 {
                        sync asa_call?;
                },
                call_l4 -> transmit0 {
                        sync asa_parm!;
                },
                transmit0 -> call_l6 {
                        sync aga_call!;
                },
                call_l6 -> call_l7 {
                        sync aga_parm?;
                },
                call_l7 -> call_l8 {
                        sync aga_call?;
                },
                call_l8 -> call_performed {
                        sync aga_parm!;
                        //assign
                                //al_recover_status := out_aga_recover;
                },
                call_performed -> call_bad {
                        guard out_aga_b < 1, // ack =/= ack2 c/ ack=1
                        al_lstatus == 1;     // ackline.lStatus == STATGOOD
                };
```

```
}

//
// PROPERTIES
//        E<>(call_ackline_send0_get.call_performed and
//               (out_aga_recover==1))
//        E<>(call_ackline_send1_get.call_performed and
//                   (out_aga_recover==1))
//

//
// PROPERTIES
//        E<>(call_ackline_send0_get.call_performed)
//        E<>(call_ackline_send1_get.call_performed
//
//        A[](not call_ackline_send0_get.call_bad)
//      A[](not call_ackline_send1_get.call_bad)
//


/////////////////////////////////////////////////////////////////////
// PROGRAM                                                                       //
//        call sender.accept(1);                                                 //
//        call receiver.deliver( out_rdr_m, out_rdr_recover );           //
//      call receiver.getStatus( out_sgs_s )                             //
/////////////////////////////////////////////////////////////////////

process call_accept_deliver_getstatus {
          state
                    call_l1, call_l2, call_l3, call_l4, call_l5,
                    call_l6, call_l7, call_l8, call_l9, call_l10,
                    call_l11, call_l12, call_performed, call_bad;
          init
                    call_l1;
          trans
                    call_l1 -> call_l2 {
                              sync sam_call!;
                    },
                    call_l2 -> call_l3 {
                              sync sam_parm?;
                              assign
                                        in_sam_m := 1;
                    },
                    call_l3 -> call_l4 {
                              sync sam_call?;
                    },
                    call_l4 -> call_l5 {
                              sync sam_parm!;
                    },
                    call_l5 -> call_l6 {
                              sync rdr_call!;
                    },
                    call_l6 -> call_l7 {
                              sync rdr_parm?;
                    },
                    call_l7 -> call_l8 {
                              sync rdr_call?;
                    },
                    call_l8 -> call_l9 {
                              sync rdr_parm!;
                    },
                    call_l9 -> call_l10 {
                              sync rgs_call!;
                    },
                    call_l10 -> call_l11 {
                              sync rgs_parm?;
                    },
                    call_l11 -> call_l12 {
                              sync rgs_call?;
                    },
                    call_l12 -> call_performed {
                              sync rgs_parm!;
                    },
                    call_performed -> call_bad {
                        guard
                           out_rgs_s < 1,   // s=/=NEWMESSAGE <
                           tl_lstatus == 1; // tl_lStatus=STATGOOD
                    },
                call_performed -> call_bad {
                        guard
                           out_rgs_s > 1,   // s=/=NEWMESSAGE >
                           tl_lstatus == 1; // tl_lStatus=STATGOOD
                    };
}

//
// PROPERTIES
//        E<>(call_accept_deliver_getstatus.call_performed)
//      A[](not call_accept_deliver_getstatus.call_bad)
//

/////////////////////////////////////////////////////////////////////
// PROGRAM                                                                       //
//        call sender.accept(1);                                                 //
//        call receiver.deliver( out_rdr_m, out_rdr_recover );           //
//      call sender.getStatus( out_sgs_s )                               //
/////////////////////////////////////////////////////////////////////

process call_send_deliver_acknowledge {
          state
                    call_l1, call_l2, call_l3, call_l4, call_l5,
                    call_l6, call_l7, call_l8, call_l9, call_l10,
                    call_l11, call_l12, call_performed, call_bad;
          init
                    call_l1;
          trans
                    call_l1 -> call_l2 {
                              sync sam_call!;
```

```
                    },
                    call_l2 -> call_l3 {
                              sync sam_parm?;
                              assign
                                        in_sam_m := 1;
                    },
                    call_l3 -> call_l4 {
                              sync sam_call?;
                    },
                    call_l4 -> call_l5 {
                              sync sam_parm!;
                    },
                    call_l5 -> call_l6 {
                              sync rdr_call!;
                    },
                    call_l6 -> call_l7 {
                              sync rdr_parm?;
                    },
                    call_l7 -> call_l8 {
                              sync rdr_call?;
                    },
                    call_l8 -> call_l9 {
                              sync rdr_parm!;
                    },
                    call_l9 -> call_l10 {
                              sync sgs_call!;
                    },
                    call_l10 -> call_l11 {
                              sync sgs_parm?;
                    },
                    call_l11 -> call_l12 {
                              sync sgs_call?;
                    },
                    call_l12 -> call_performed {
                              sync sgs_parm!;
                    },
                    call_performed -> call_bad {
                        guard
                           out_sgs_s < 1,   // s=/= SENTNOACK <
                           al_lstatus == 1, // al_lStatus=STATGOOD
                           tl_lstatus == 1; // tl_lStatus=STATGOOD
                    },
                call_performed -> call_bad {
                        guard
                           out_sgs_s > 1,   // s=/= SENTNOACK >
                           al_lstatus == 1, // al_lStatus=STATGOOD
                           tl_lstatus == 1; // tl_lStatus=STATGOOD ;
}
//
// PROPERTIES
//        E<>(call_send_deliver_acknowledge.call_performed)
//      A[](not call_send_deliver_acknowledge.call_bad)
//

system sender, receiver, transline, ackline, call_send_deliver_acknowledge;
```