UMinho | 2011

João Martins

Formal Verification of Ada Programs: An Approach Based on Model Checking

Universidade do Minho
Escola de Engenharia

João Pedro Marques da Silva Martins

Formal Verification of Ada Programs: An Approach Based on Model Checking

Outubro de 2011

Universidade do Minho
Escola de Engenharia

João Pedro Marques da Silva Martins

Formal Verification of Ada Programs: An Approach Based on Model Checking

Tese de Mestrado
Mestrado em Informática

Trabalho efectuado sob a orientação do
Professor Doutor Jorge Sousa Pinto

Outubro de 2011

The EVOLVE (Evolutionary Validation, Verification and Certification) is a collaboration project, developed by the consortium leader Critical Software SA and the University of Minho. The consortium's major intention is to develop the sustained development of innovative know-how throughout the cooperation between R&D institutions and companies appointed for embedded systems. This is co-funded by EUR 258.335,56 (two hundred and fifty eight thousand, three hundred and thirty five euros and fifty six cents) by QREN, under the Operational Programme for Competitiveness Factors and contracted with ADI through the Contrato de Concessão Incentivos Financeiros n. 2008/001621 with the total amount of investment of EUR 404.131,97 (four hundred and four thousand, one hundred and thirty one euros and ninety seven cents) with the execution period planned for between the 01-06-2008 e 31-09-2011. This work was partially supported by the EVOLVE Project.

Acknowledgements

I would like first to thank my family, you are my great inspiration. I dedicate this thesis to my mother, my father, Hugo and Lili for their enthusiasm and expectations from me.

I also would like to thank my friends from Cabeceiras de Basto and Braga for providing me relaxing and funny moments, which gave me the strength i needed when things were not going well.

Lastly, a special thank to my advisor from Critical Software José Miguel Faria and to my advisor in University of Minho prof. Jorge Sousa Pinto. Thank you for all the guidance and help provided, but thank you especially for the support and motivation you gave me.

Resumo

O rápido crescimento da complexidade dos sistemas de software exige, agora mais do que nunca, uma validação rigorosa dos mesmos por forma a manter ou até mesmo aumentar a confiança nestes sistemas. Em particular nos sistemas críticos, onde as falhas podem ter consequências catastróficas podendo até incluir a perca de várias vidas humanas, é de extrema importâcia o desenvolvimento de técnicas capazes de garantir altos níveis de confiança para estes sistemas.

Nesta tese é proposta a utilização de uma técnica formal para a verificação de programas Ada, que pretende aumentar a confiança em sistemas cuja implementação seja realizada nesta linguagem de programação. Mais precisamente, pretende-se a aplicação da técnica de verificação de modelos para a análise do código fonte de programas concorrentes Ada, com especial foco para o domínio dos sistemas críticos.

A vericação de modelos é uma técnica bem sucedida no que diz respeito à garantia de um aumento de fiabilidade destes sistemas. No entanto, a aplicação desta técnica a sistemas de software enfrenta ainda vários obstáculos, e as ferramentas e técnicas para ajudar a ultrapassar estes obstáculos estão ainda a ser desenvolvidas. A ferramenta desenvolvida no contexto desta tese (ATOS) visa responder a problemas como (i) a construção de modelos a partir de programas e (ii) a especificação de propriedades para estes modelos de acordo com as pretendidas para os programas.

A construção manual de modelos que simulam o comportamento de programas é um processo complexo, temporalmente dispendioso, e sujeito a falhas devido à complexidade destes sistemas. De forma a ultrapassar este problema o ATOS propõe a extracção automática de modelos a partir de programas Ada. Por outro lado, o mapeamento das propriedades desejadas dos programas em propriedades dos modelos pode ser uma tarefa com um grau de complexidade elevado, pois requer entre outros a utilização de um formalismo lógico ao qual a maioria dos programadores não está acostumado. O ATOS ajuda no mapeamento destas propriedades, oferecendo vários mecanismos de suporte à sua especificação.

Palavras-chaves: Ada, Extração de Modelos, Lógica Temporal, Métodos Formais, Sistemas Críticos, SPIN, Verificação de Programas, Verificação Formal, Vericação de Modelos, Vericação de Modelos de Software.

Abstract

The rapid growth of the complexity of software systems demands, now more than ever, a rigorous validation of these systems in order to maintain or even increase their reliability. In particular in high-integrity systems, where failures may have catastrophic consequences which may even include the lost of human lives, the development of verification techniques capable of ensuring high degrees of confidence is seen as extremely important.

In this thesis the use of a formal technique for the verification of Ada programs is proposed, which aims to increase the reliability of systems whose implementation is based on this programming language. More precisely, we target the application of the model checking technique to the verification of source code of concurrent Ada programs, with a special focus on the critical systems domain.

Model checking is a well-succeeded technique for providing increased levels of assurance regarding system correctness. However, its application to software systems still faces several obstacles, and the necessary tools and techniques to help in the overcoming of these problems are still being developed. The tool presented in this thesis (ATOS) addresses the problems of (i) constructing models from programs and (ii) specifying properties for models corresponding to the ones desired for programs.

The manual construction of models that simulate the behavior of programs is a time-costly, complex and error-prone process, due to the complexity of these systems. In order to overcome this problem, ATOS proposes the automatic extraction of models from Ada programs. On the other hand, the mapping of the desired properties from programs to models can be a task of high complexity, because it requires among others that they are expressed in a logical formalism that most programmers are not acquainted with. ATOS helps in this mapping task by providing several mechanisms aiming to support the specification of properties.

Keywords: Ada, Critical Systems, Formal Methods, Formal Verification, Model Checking, Model Extraction, Software Verification , Software Model Checking, SPIN, Temporal Logic.

Contents

List of Acronyms

ANTLR Another Tool for Language Recognition

ASIS Ada Semantic Interface Specifications

AST Abstract Syntax Tree

ATOS Ada Translation to Spin

ATP Automated Theorem Provers

ASIL Automotive Safety Integrity Levels

BDD Binary Decision Diagram

BIR Bandera Intermediate Representation

BLAST Berkeley Lazy Abstraction Software Verification Tool

BSL Bandera Specification Language

CC Common Criteria

CFA Control Flow Automata

CFG Control Flow Graph

CIL C Intermediate Language

CPN Colored Petri Nets

CNES Centre National d'Etudes Spatiales

CTL Computation Tree Logic

DAG Direct Acyclic Graph

DAL Design Assurance Level

DbC Design by Contract

DFA Deterministic Finite Automata

DFD Data Flow Diagram

EAL Evaluation Assurance Level

ESA European Spatial Agency

EVOLVE Evolutionary Validation, Verification and Certification

GB Gigabyte

GPS Gnat Programming Studio

GSW Galileo Software Standard

GUI Graphical User Interface

HELENA High Level Net Analyzer

IDE Integrated Development Environment

IEC International Electrotechnical Commission

ISO International Organization for Standardization

JML Java Modeling Language

JPF Java Path Finder

LOC Lines Of Code

LTL Linear Temporal Logic

MB Megabyte

MILS Multiple Independent Levels of Security

MMR MILS Message Router

NFA Non-Deterministic Finite Automata

NuSMV New Symbolic Model Verifier

PROMELA Process or Protocol Meta Language

PIFP Partition Information Flow Policy

RAM Random Access Memory

SEDL S-Expression Design Language

SIL Safety Integrity Levels

SMV Symbolic Model Verifier

SPARK Spade Ada (?) Kernel

SPIN Simple PROMELA Interpreter

SRS Software Requirements Specification

TCTL Timed Computation Tree Logic

List of Figures

List of Tables

Listings

1. Introduction

The work presented in this document was developed in the context of the EVOLVE project, a project that in Portugal was promoted jointly between Critical Software SA and the University of Minho, and intended to propose techniques and tools capable of verifying and validating software products along all of the development phases. The target domain are the so-called reactive systems, i.e., systems that react to environment stimuli. The verification of these systems intends to focus particularly on concurrency aspects, due to the difficulty of stating the correctness of concurrent systems. Most embedded systems, commonly used in many critical missions, belong to the class of concurrent reactive systems.

In particular, this work intends to address the formal verification of critical systems at the implementation level. More specifically, we fous on the analysis of the source code of concurrent systems through the use of formal methods techniques. Formal methods have been used in different phases of the software development process, in particular at the system specification level, which can be considered already a mature topic. However, their usage in source code verification has only recently been addressed.

The present work describes a formal tool called Ada Translation to Spin (ATOS). The tool supports the verification of Ada programs using the model checking technique. More precisely, this work focuses on verifying the source code of concurrent programs written in Ada [TDB$^+$07, BA09], although it can also be useful for verifying sequential Ada programs.

The remainder of this chapter contains an introduction to several essential topics, aiming to frame this work in the formal verification context.

1.1. Formal Verification of Software Systems

Information systems have nowadays an extremely high importance in our lives, whether as an essential support for highly efficient communication systems, or in the support of systems essential for the evolution of medical and avionics sciences. The rapid growth of these systems, prompted by the evolution of several different economic sectors, have quickly led to the increase of the complexity of the computer programs underlying these systems. Currently, most software systems are characterized by such a high level of complexity that it is now more difficult than ever to have a reasonable degree of confidence in their reliability.

Attaining absolute certainty that a software system has no errors, i.e. ensuring that it will not produce any unexpected behavior, is clearly an utopia. However, increasing the degree of confidence in software systems is a central and justified concern, since errors (the so-called bugs) have consequences. The cost of a system failure is not the same in all systems; in *high integrity systems* or *critical systems* failures may be catastrophic.

On the fourth of July 1996, less that a minute after lunching, the rocket Ariane 5 was destroyed by its automated self-destruct system. After the accident inspection carried out by European Spatial Agency (ESA) and Centre National d'Etudes Spatiales (CNES), a report was produced explaining the reasons for this accident. The report revealed that the destruction of Ariane 5 was caused by an error in software design: the conversion of a 64-bit floating point number to a signed 16-bit integer was at the origin of the accident. In this accident no lives were lost, but the damages approached 260 million Euros.

Presently, in the majority of software systems deployed the certification is made essentially through the *testing* mechanism. This technique consists in passing a set of inputs to the system and observing if the obtained outputs correspond to the ones expected. In the attempt to increasing the efficiency of the input set, which leads to a higher certification degree, a huge broad of techniques have emerged, and more will certainly appear as a result of the big investments made in this research field. As an alternative to testing we find *formal methods*, which encompass techniques capable of providing high levels of assurance. Due to the high costs associated with their usage, formal methods have mainly been used in the development of critical software systems.

Formal methods Many different definitions of Formal Methods are given in different books and scientific documents. As an example, Formal Methods can be seen as "the applied mathematics for modeling and analyzing Information and Communication Technology systems" [BK08]. With the growth of systems' complexity and size the interest in formal methods has risen, since they provide a basis for the precise definition of consistency, completeness, specification, implementation and correctness of systems. Formal methods encompass several validation methods such as: *deductive verification* [Hoa69, Flo67], *model checking* [JGP99, BK08] or *theorem proving* [BM83].

Even though formal methods are seen as very promising in systems' validation, they are currently still seen as very costly and not suitable for the majority of systems. In fact, it is possible to conclude that formal methods are essentially used in the development of *critical systems*, where failures are seen as unacceptable. This vision is explained essentially by the cost-effectiveness relation, and by the difficulty of scaling up the use of formal methods for "big" projects. Nevertheless, a study in [WLBF09] has revealed that the effect of formal methods in several projects, considering time, cost and quality, were very positive as can be observed in Figure 1.1. This study was based on data collected from more than sixty projects (probably, most of them related to critical systems).

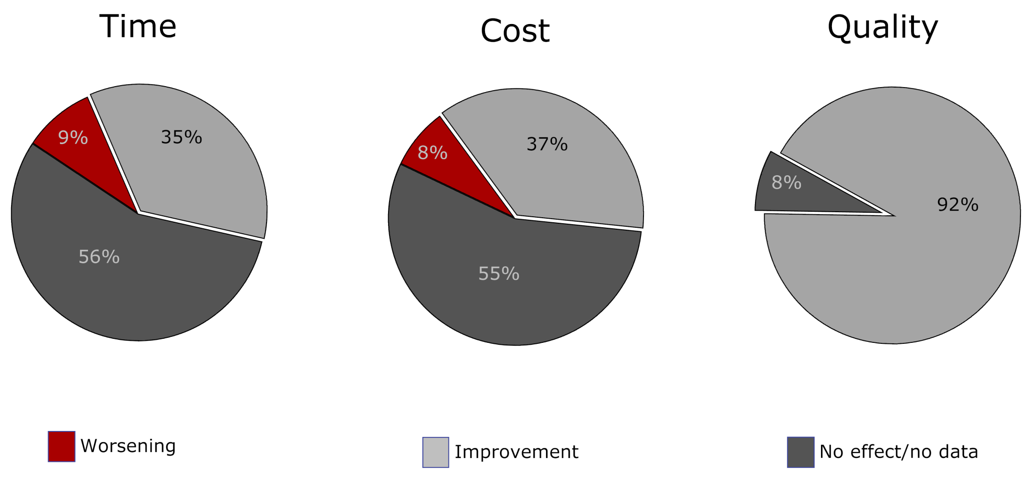


Figure 1.1.: Formal methods effect on projects: time, cost and quality

Even so, the time and cost of applying Formal Methods can be reduced through the use of automated techniques (e.g. model checking) and the development of new tools capable of reducing the time and knowledge required for their application. Formal methods techniques will only be suitable for the majority of software deployments when they are capable of providing answers in minutes rather than days.

1.2. Verification of Critical Systems in the Industrial Context

The software for critical systems is required to be certified with a high level of reliability, due to the cost associated with failures. Therefore, the industry responsibly for the development of these systems has been progressively increasing its interest in Formal Methods. However, this increasing interest is not simply explained by the potential profits of their application; another reason, probably with more impact, is the explicit recommendation of the application of formal methods in most critical systems standards.

The deployment of safety critical systems is regulated by several standards. These standards vary according to the domain they apply to (medical, defense, railway, aerospace, etc.), but have significant commonalities. In general, they define different assurance levels and different activities/objectives for achieving compliance with each level. For each assurance level, the different activities/objectives are classified as mandatory (M), not required (-), or an intermediate value, such as recommendable (R) or highly recommendable (HR), as illustrated in Table 1.1. Typically, formal methods are considered highly recommendable for obtaining the highest assurance levels. More details about the particular safety standards of different industrial sectors are given next.

Requirement \ Safety Level	Lowest Level	...	Highest level
1	R	...	M
...	...	...	...
N	-	...	HR

Table 1.1.: The standards generic table to assign the safety level of a system

Avionics Industry The standard DO-178B: Software Considerations in Airborne Systems and Equipment Certification has an important role in the certification of aerospace and defense-related systems. This standard defines five levels of certification known as Design Assurance Level (DAL), which go from DAL E to DAL A, where the Lowest/highest level is DAL E/A. Other standards have originated from DO-178B, like the ESA-Galileo Software Standard (GSW). The application of formal methods in avionics is about to be clarified in a new version of the standard, DO-178C, which shall contain a segment specifically dedicated to the subject.

Railway & Automotive Industry In this industry the standards EN 50128:Railway Applications - Communications, Signaling, and Processing systems stand out. This standard is part of a group of related standards containing the EN 50126 and EN 50129 defined by CENELEC, the European Committee for Electrotechnical Standardization. The EN 50128 specifies five levels of safety, denominated as Safety Integrity Levels (SIL) where SIL0 and SIL4 are the lowest and the highest levels, respectively. The use of formal methods is highly recommend for the highest assurance levels.

Automotive Industry The International Organization for Standardization (ISO) 26262, which was adapted from the International Electrotechnical Commission (IEC) 61508, is a functional safety standard that classifies the system functions in four Automotive Safety Integrity Levels (ASIL), which vary from ASIL–A to ASIL–D , where ASIL–A is the lowest safety-critical level. Formal methods are highly-recommended for ASIL–C and ASIL–D.

Medicine Industry The standard IEC 62304 has an important role in medicine industry, concerning the certification of software for medical devices. The manufacturer classifies software system as whole in three classes: A (No injury or damage to health is possible); B (Non serious injury is possible) and C (Death or serious injury is possible). Based on this classification, IEC 62304 imposes the accomplishment of several requirements along the software development life cycle.

Common Criteria The Common Criteria (CC), or IEC 15408, is one of the most commonly used international standards. Unlike the previously presented standards, CC is more oriented to security-critical, rather than safety-critical, systems. CC evaluates its target systems in seven Evaluation Assurance Level (EAL) ranging between the lowest level EAL1, and the highest level EAL7. The use of formal methods is mandatory for the highest assurance levels.

Formal Methods Tools in Standards Depending on the application and domain, a tool to be used in the certification of a system may have to be itself qualified, prior to its acceptance and usage. As such, a tool based on formal technologies may require self qualification, not necessarily using another formal methods-based tool (otherwise one would enter a vicious cycle). This process is also regulated in the different domains. Acceptable arguments for the qualification of tools are, for example, evidence of a rigorous development process (detailed documentation of the development and verification activities), and "proven in use" evidence, i.e. a track record illustrating that the tool has been successfully applied in several projects.

1.3. Formal Verification of Ada Programs

The Ada programming language was developed at the request of the United States Department of Defense, with the aim of using it as the universal programming language for its military systems. However, the programming language that was originally targeted for military systems quickly proliferated to critical systems in general. Henceforth, the use of Ada in the development of critical systems has progressively increased and remains extremely significant at the present, which is mainly explained by its careful and safe design (as explained in [Bar08]), and the existence of clear guidelines for building this kind of system.

The first standard of the Ada language was published in 1983, and therefore denominated as Ada 83 (but before this date the language proposal had already been published for scientific review). The next version of Ada appeared in 1995, bringing many extension and modifications with respect to the previous version. This new version was denominated Ada 95. The last version of Ada and the one used in the context of this thesis is Ada 2005. This version no longer has the participation of the United States Department of Defense, whose collaboration in this project ceased in 1998. The promotion of Ada is now undertaken by a group of commercial companies, while its standardization had continued under the ISO procedures. A new version of Ada is expected in 2012, which is expected to continue providing new mechanisms for the improvement of systems reliability.

SPARK The programming language SPARK [Bar03] is a restricted subset of the Ada language, with added annotations for the specification of the desired behavior of programs, resembling the principles of Design by Contract (DbC) [Mey92]. These annotations are embedded in programs as comments, thereby enabling SPARK and Ada programs to share the same compiler. In other words, a valid SPARK program is also a valid Ada program. SPARK additionally provides a toolset containing different types of tools for different purposes, such as a static analyzer or an automated theorem prover.

RavenSPARK RavenSPARK [SPA08] is an extension of SPARK compliant with the Ravenscar profile [BDV04, AD03], which aims to make possible the formal verification of concurrent and real-time Ada programs. The restrictions imposed by the Ravenscar profile are the key to make formal verification of concurrent and real-time Ada programs treatable; without these it would be difficult to perform this type of analysis, due to the high level of some Ada features. Beyond the extension of the SPARK subset, RavenSPARK also includes new annotations concerning the additional primitives added to SPARK.

Formal Verification of Concurrent Ada Programs As mentioned, Ada is a programming language endowed with mechanisms that make it very appealing for the development of safe and secure systems. Arguably, the SPARK programming language along with its toolset are the most important landmark of formal verification in Ada programs. RavenSPARK was introduced in an attempt to extend formal verification to concurrent Ada programs, but it is still very limited. There exist of course other approaches to the verification of concurrent programs written in Ada, more precisely through the modeling of Ada programs [HM03, EKPPR03b], but there is still a gap that needs to be filled.

1.4. Verification of Software Systems using Model Checking

Model Checking is a formal verification technique suitable for assessing the functional properties of systems. Model checking-based methods are among the most successful formal methods, at least in the industrial context. This is due to their automated nature, which can contribute to significantly reducing the cost of applying the methods, by decreasing the expertise and the time that are commonly required.

Given a model and a property, this technique explores in a brute force manner the states of the model, in order to prove its correctness against the desired property. When a property is not valid for a given model, an error trace is produced containing the path in the model that violates the property. The process of model checking consists in three main phases [JGP99]: Modeling, Specification and Verification. Different approaches to the model checking process are proposed by different methodologies [BK08].

Modeling is the first phase of the process; it consists in converting a design into a formalism, i.e. in translating system requirements into a mathematical model, typically a transition system. Constructing a model that represents exactly the behavior of a system is essential for the success of model checking. It is usually said that "the model checking technique is as good as the model of the system" [BK08].

After the construction of the model, it is necessary to state which properties the model enjoys (or is supposed to enjoy). The *specification* of properties is usually given in some temporal logic formalism, with Computation Tree Logic (CTL) [HR04] and Linear Temporal Logic (LTL) [HR04, Pnu77] having a predominant role. This kind of logic allows for the specification of ordered events, without explicitly using time.

Lastly, the *verification* phase is an automatic process, which checks a property against a model. When the property check fails, i.e. an error is encountered, a trace showing the error path is provided. When error traces are caused by errors or limitations of the models or of the specified properties (rather than of the modeled systems or of the intended properties), they are called false negatives. Even though the verification process is automatic, most often human assistance is needed to overcome some obstacles, such as the state space explosion. Many options are offered to this effect by model checking tools. Choosing the best option is not an easy task and requires some expertise from the user.

The application of the model checking technique to software systems is seen as very promising and has led to the creation of a new research area, known as *software model checking* [DKW08, JM09]. This technique is not simply the application of model checking to software; it involves also the design of solutions to some obstacles in its application, most notably:

Model construction: The manual construction of a model of a software system is an error-prone and time consuming process, due to the complexity of these systems. In addition, there is a gap between the semantics of programming languages for software systems (e.g. C, Ada, or Java) and the input languages of model checking tools. Programming languages have richer features with more complex semantics than modeling languages.

State explosion: This is recognizably the biggest problem of model checking. In software model checking the problem can become even more serious, due to the size of software systems, which leads to the generation of models with an enormous number of states. Thus, more aggressive abstractions must be considered.

Property specification: Typically, properties are specified in some variant of temporal logic. This creates two difficulties: firstly, it requires some level of expertise for expressing the desired system properties in temporal logic. Second, the mapping of these properties to the properties of the model may not be straightforward, since the typical specification languages are designed to state properties of mathematical models rather than of source code.

Output interpretation: When a property does not hold in a given model, the model checker reports a counter-example illustrating a trace that evidences the violation of the property. Large models can produce very long traces. As such, manually matching the provided trace to the model's source code can be a really hard task.

1.5. A Formal Approach to the Verification of Ada Programs

The approach presented in this section corresponds to the one embedded in ATOS. The main role of ATOS is to help overcoming some of the software model checking problems (as outlined above), in the application of the model checking technique to Ada programs. In particular, ATOS directly addresses two of these problems: given an Ada program, it is capable of automatically extracting a model from it, overcoming the difficulties expressed in the *model construction* problem, and by inferring properties directly from Ada programs or from annotations embedded in the programs, it tackles also the *property specification* problem.

ATOS uses the Simple PROMELA Interpreter (SPIN) model checker [Hol03] as back-end to perform the verification of the extracted models. SPIN is a model checking tool focused on verifying the correctness of models of concurrent systems, which clearly matches our intents. The models are described in PROMELA, the SPIN modeling language, and correctness claims can be stated through LTL formulas or PROMELA assertions. In short, ATOS generates PROMELA models from Ada programs, and LTL formulas or assertions from inferred and annotated properties.

The ATOS approach and related decisions Software model checking tools follow essentially two main approaches: (i) they either generate input models for one or more different existing model checkers, where ATOS is included, or (ii) they include their own model checking algorithms to check the correctness of the models. Selecting one of these two approaches was not straightforward and required the analysis of the pros and cons of each approach, as enumerated in Table 1.2.

Approach	Advantages	Disadvantages
(i)	• Less time required; • Profit from all the techniques already implemented in the target tool, from abstraction techniques to model checking algorithms; • Use a stable and recognized tool (of course it depends from the chosen tool).	• Little room for development of new techniques related with model checking theories, such as abstraction techniques, or the definition of new mathematical models which can provide less complex and faster algorithms; • Restricted to the temporal logic of the chosen tool;
(ii)	• Opportunity to developed new techniques and theories according to what is intended;	• Significant time and effort required; • One could being "reinventing the wheel";

Table 1.2.: Comparison of approaches

An analysis of the advantages and disadvantages of approaches (i) and (ii) revealed that both could be equally interesting to explore, but due to the time restrictions of this work, the approach (i) was judged to be more suitable, because it could guarantee results in the short term.

Another important decision in the definition of the ATOS approach was whether or not to use an intermediate representation between the programs and the models. On one hand, using such a representation would give us the possibility to then generate more easily models that could be exported to others model checkers, because the intermediate representation would be closest to a model than the programs. On the other hand, program details might be lost in this intermediate representation, and it was our intention to generate models that would be as accurate as possible. The decision of performing the extraction of models directly to PROMELA without any intermediate representation was due essentially to this fact. It also seemed that it might be relatively easy to generate input models for others model checkers directly from PROMELA, since it provides high-level mechanisms for the specification of models.

1.6. Contributions and Document Outline

Our most important contributions in this work are are:

1. a model extraction mechanism, which automatically generates PROMELA models from Ada programs;
2. several mechanisms supporting the extraction of properties from annotated Ada programs to PROMELA models; and
3. as a case study of significant size to help validate our approach, the verification of a Separation Kernel subsystem using these mechanisms.
4. The background material contained within this thesis, which surveys themes like model checking, software model checking and the Ada programming language.

A paper containing part of this work was presented at the *Embedded and Real-Time Systems* track of INForum 2011. A new paper containing all of ATOS' features including the new supported Ada primitives and the new case study (the Separation Kernel) is about to be prepared.

The rest of this document is organized as follows. First, some background information about software model checking is given in Chapter 2, concerning its different approaches, theories and tools. Chapter 3 introduces ATOS, and chapters 4 and 5 explain the model and property extraction mechanisms implemented in it, respectively. Then the application of ATOS is demonstrated in Chapter 6 through two case-studies: *Readers-Writers* and *Separation Kernel.* Lastly, a discussion of the developed work along with its limitations and future improvements is given in Chapter 7.

2. Model Checking - Approaches, Theories and Tools

A broad range of model checking tools are available. Even though these tools follow all the same technique, the approaches and theories implemented in each are very different. The application of the model checking technique, whether on software or hardware systems, requires some knowledge of these concepts. Therefore, in the context of this work, a deep study was performed with the aim of obtaining the sufficient knowhow for the development of a tool like ATOS.

Our research of model checking techniques began with the study of the different mathematical model theories, in order to understand their semantics and decide whether there were one most suitable for our intents. Along with the mathematical models, several different approaches to the logical formalism used in model checking (*Temporal Logic*) were studied. This formalism enables the specification of properties required for correctness of the models.

The next step of research was the study of model checking tools, realizing what approaches and theories were followed by each, and analyzing its application in systems verification. Despite software model checking having only recently emerged as a research field, there exist already several (software) model checkers targeted to support the verification of software systems. Therefore, a study of some software model checking tools was also carried out, with particular focus on the ones that support the verification of Ada programs.

The research described above was very important in the definition of the ATOS approach. Many questions such as whether to Use or not an existing model checker, or the choice of the most adequate model checker to use as ATOS back-end, were answered based on all the information collected along this research process. The research was also important to realize how we may contribute to the verification of Ada programs using the model checking technique.

In the remaining of this chapter an overview of mathematical models is first presented, along with details about the two main theories: Petri Nets and Automata. Next, an introduction to temporal logic is given, covering the details of some different temporal logic approaches. Lastly, several model checking and software model checking tools are analyzed.

2.1. Mathematical models

A formal model is an essential piece in the model checking technique puzzle, because it allows the representation and reasoning about the system behavior. Formal models have a mathematically defined syntax and semantics. Such models can be manipulated by a computer tool and can be used to verify system properties, i.e., to prove that certain desired properties are fulfilled, or that certain undesired properties are guaranteed to be avoided.

Our purpose here is to identify and analyze the characteristics of mathematical models, in order to understand which is the "ideal" model to represent a concurrent software system. Next, it will be analyzed two of the most used mathematical models in model checking: Automata and Petri Nets.

2.1.1. Automata

Automata models are transition systems, where nodes represent the state of a system in a given moment and the edges (transitions) specify how the system evolve. A Deterministic Finite Automata (DFA) is a 5-tuple (S, s0, L, F, T), where S is a finite set of states, s0 is the initial state, L is the set of labels (the actions executed in transitions), $F \subseteq S$ is the set of final states and $T \subseteq S \times L \times S$ is the transition relation.

A path[1] in the model is an ordered sequence of labels. In DFA paths are finite, which makes impossible the representation of non-terminating systems. This restriction can be solved with a simple method called Buchi-acceptance [Per90], where infinite runs are enabled. DFA's have another constraint, they can not represent non-deterministic actions contained in systems behavior. Transforming a DFA into a non-deterministic Automata (NFA) is a simple process [HMU06], in fact these two structures are equivalent, i.e, each DFA can be converted into a NFA and vice-versa. Finally, to represent concurrent systems it is only missing the simulation of concurrency, which can be achieved by the product of NFA's (i.e the joint of two NFA's).

The use of Automata models in model checking, is a stable matter. Many successful model checkers use Automata models as a basis for theirs transition system definition (e.g. SPIN uses NFA's as mathematical model).

2.1.2. Petri Nets

Petri Nets [PR08, DJE01] as graphical and mathematical model, provide a uniform environment for modeling concurrent systems. The transition system of Petri Nets are composed by nodes and arcs. There are two types of nodes: places and transitions, which are connected by arcs. Arcs only connect places and transitions or vice versa, they do not connect two places or two transitions. The hold of conditions is represented by tokens in places. Transitions are only activated if and only if there is at least one token at each of its input places.

[1]Paths are also named as runs

The representation of systems is usually done by high level Petri Nets like Colored Petri Nets (CPN) [Jen95]. CPN's combine the capabilities of Petri Nets (control structures, synchronization, communication and resource sharing) with the capabilities of a high programming language (data and data manipulations).

Every Petri Net can be converted into a CPN and vice versa, so there is no theoretical difference between them. However CPN is a much more compact and convenient modeling language than Petri Net. CPN's are to Petri Nets as high level programming languages are to assembly.

A bunch of verifications methods exist to provide Petri Nets analysis. Model checking algorithms are included in these methods. There are already some model checking tools using Petri Nets as mathematical model (e.g. Quasar).

2.2. Temporal Logic

The temporal logic allows the specification of properties over a given period of time. The time may be implicit or explicit, and enables the specification of an ordered sequence of events. The model checking technique is based on temporal logic, thereby enabling the specification of formulas (properties) which validate a set of model states rather than single points as it proposed by the traditional propositional or predicate logic. Therefore, in the context of software model checking, temporal logic is very useful and can really simplify the specification of properties due to its capability to state properties for regions of code rather than single points in implementation.

Several temporal logic approaches have been proposed until now, each with a different purpose. These different approaches may be distinguished by theirs *view of time*. Next, are presented two of this views denominated as LTL and CTL. These may be considered the most important ones, since are the basis for many others, such as CTL* and TCTL, also presented in this section.

2.2.1. LTL

The Linear Temporal Logic (LTL) views time as a set of individual paths, where a path is a sequence of events (states). An example of how LTL *views* a given transition system is illustrated in of Figure 2.1. LTL contains several connectives which encompass the "normal" operators (common to the traditional logic formalisms) such as: $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, but it also encompasses other connectives[2]. These connectives allow the reference of (implicit) time in models such as: F (in a future state), G (all future states), X (next state) and the binary operators U (Until) and W (Weak-Until). Where the formula $\phi\ U\ \psi$ holds on a path if ϕ

[2]There exist several different terminologies to represent the same connectives we depict one of these

holds on all states path *until* ψ holds and ψ must hold at some future state. The $\phi\ W\ \psi$ is similar to the previous however it is not required that ψ holds in some state.

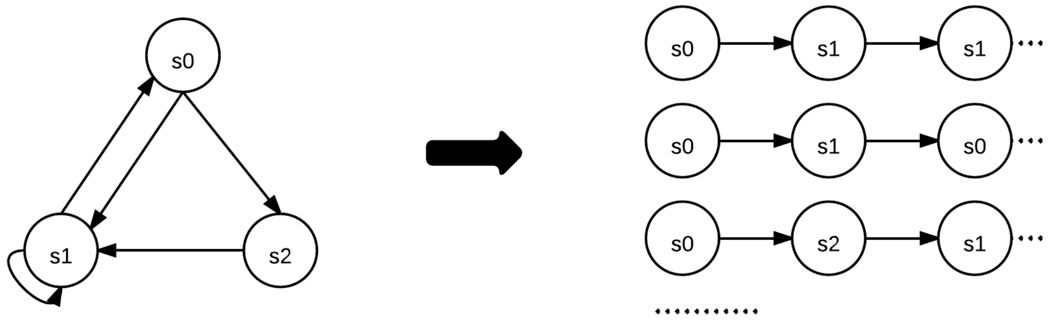


Figure 2.1.: The LTL view of time

2.2.2. CTL

The Computation Tree Logic (CTL) which belongs to the *branching-time logics*, views time as a tree of paths. A similar example to one used in LTL is utilized again (Figure 2.2), but now to illustrate the CTL view of time. The CTL view of time unlike the LTL, allows one to assert the *existence* of a path rather than just state properties over *all* paths.

The CTL has the same time operators as LTL but now they are quantified over the paths. Therefore, the temporal operators appear as pairs: the first part is the quantifier which can be A (all paths) or E (exists at least one path); while the second part may contain the already introduced X, F, G, and U operators (The W did not exist but it is derivable). As an example, the formula EF ϕ, means "*exists* a path in a *future* state where ϕ holds".

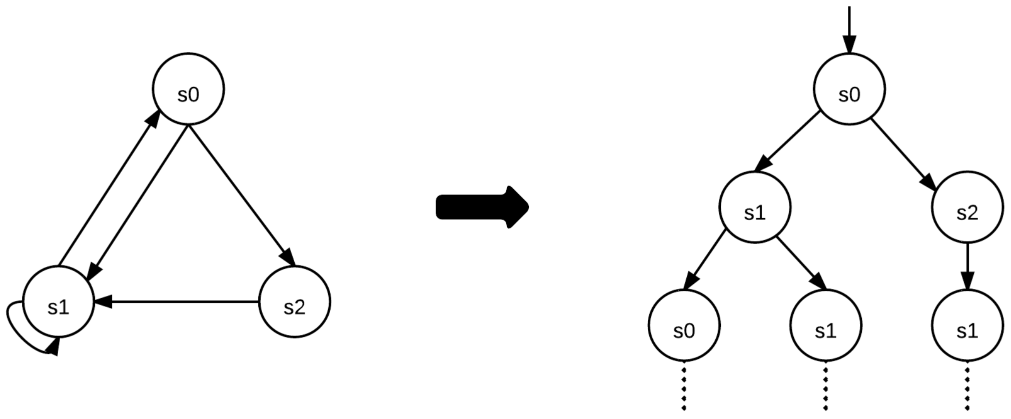


Figure 2.2.: The CTL view of time

2.2.3. CTL*

The CTL* [EC82] is a superset of CTL, that combines the expressive power of LTL and CTL as illustrated in Figure 2.3. As mentioned, LTL does not allow the quantifications over paths permitted in CTL . Nevertheless, CTL does not include all the expressive power of LTL too, since LTL permits the selection of range paths, which is impossible in CTL.

CTL* combines the expressive power of LTL and CTL by dropping the CTL constraint which requires that all temporal operators (X, F, G, and U) must be preceded by a quantifier (E and A). Thus, CTL* encompass two classes of formulas: *state formulas* (evaluated at state level) and *path formulas* (evaluated along paths).

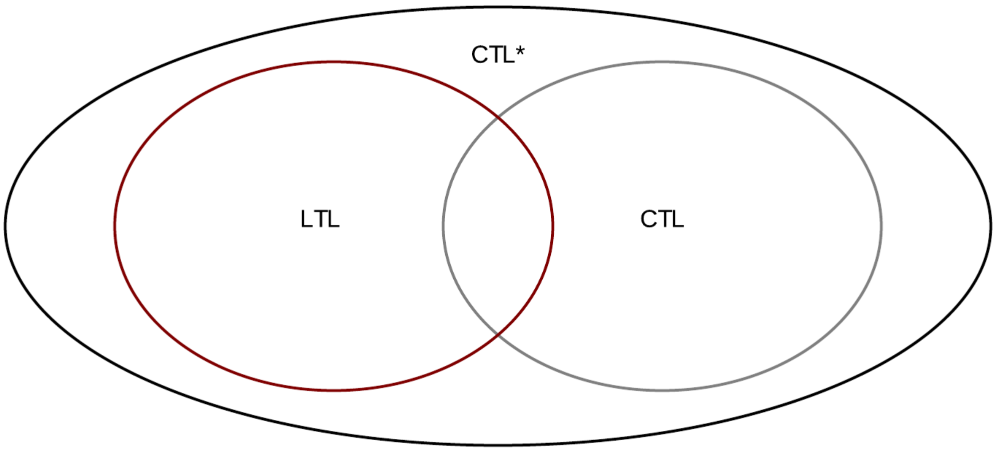


Figure 2.3.: CTL* expressive power

2.2.4. TCTL

The Timed Computation Tree Logic (TCTL) [ACD93, LPY95] is an extension of CTL, allowing the use of explicit time constraints in properties specification. For example, in the property "in all future paths p is valid" expressed as $AF\ p$ it is possible the addition of a time restriction. Thus, one can state in "all future paths p is valid within 10 units of time" as $AF_{\leq 10}\ p$.

TCTL requires a different transition system comparing to the one used previously (in LTL and CTL). The main difference between them is the addition of *clocks* to the (timed) models. A number of finite clocks defined in $\mathbb{R}_{\geq 0}$, may exist in a model, these evolve synchronously as if there was a global clock controlling the entire system, nevertheless, the clocks reset can be performed individually.

2.3. Model Checking - Tools

The first model checkers start to be developed in the early 80's, since then a significant number of model checking tools had appeared. Most of these tools are focused on the verification of concurrent systems, in fact these systems were in the model checking origin, from Edmund Clarke in [Cla08] we quote "Model Checking did not arise in a historical vacuum. There was an important problem that needed to be solved, namely Concurrent Program Verification".

Amongst the model checking tools we highlight three: SPIN; NuSMV; and Uppaal. The SPIN because was one of the first model checkers and it is probably the most successful one; NuSMV for being the first symbolic model checker[3]; and Uppaal for incorporating timed models (timed automata), allowing the use of explicit time, essential in the verification of real-time systems. These three tools were presented in the remaining of this section.

2.3.1. SPIN

Simple PROMELA Interpreter (SPIN) is a general tool for verifying the correctness of distributed software models and belongs to the category of model checker's[Hol03]. It was written by Gerard J. Holzmann and others in the original Unix group of the Computing Sciences Research Center at Bell Labs, beginning in 1980.

Modeling language and property specification SPIN targets efficient software verification and hardware verification. The tool supports a high level language to specify systems descriptions, called Process or Protocol Meta Language (PROMELA). SPIN is essentially used to trace logical design errors in distributed systems by supporting requirements specification in Linear Temporal Logic (LTL) or embedded assertions in models. LTL presented in SPIN has the following syntax:

$$\phi ::= \ true \mid false \mid 0 \mid 1 \mid p \mid \neg\, \phi \mid \phi \ \wedge \ \phi \mid \phi \ \vee \ \phi \mid \phi \rightarrow \phi \mid [\,]\, \phi \mid \ <> \ \phi \mid \phi \, U \, \phi$$

,

where p is propositional atom and the connectives [] (equivalent to G operator), <> (equivalent to F operator) and U are the temporal connectives. As previously explained, these connectives mean: [] all future states; <> some future state and U until the state.

Verification There are three main modes for the verification of models in SPIN tool: Simulator, allowing for rapid prototyping with a random, guided, or interactive simulations; Verifier, capable of rigorously validate of user specified correctness requirements (using partial order reduction, collapse compression and minimized automata theories to optimize the search) or

[3]The primary version of NuSMV (SMV) was the first symbolic model checker

validate even very large system models with maximal coverage of the state space using abstraction techniques like *hash compact* and *bitstate*. Driver for swarm verification (a new form of swarm computing), which can make optimal use of large numbers of available compute cores to leverage parallelism and search techniques, which may increase the chance of verifying large models by speeding up this process.

2.3.2. Uppaal

Uppaal [BDL04] is a toolbox for the verification of real-time systems jointly developed by Uppsala University and Aalborg University. The tool was designed to verify systems modeled as networks of timed automata extended with integer variables, structured data types, and channel synchronization. In Uppaal, the state of the system is given by the location of all automata, the clocks, and the values of the variables. A timed automaton is a finite-state machine extended with clock variables. It uses a dense-time model, i.e., when a clock variable evaluates all the clocks advance too, in a synchronously manner.

Modeling language and property specification The Uppaal modeling language is a mix between the graphically design of system entities behavior and the specification of data structures and functions described in a C similar language. The query language of Uppaal, used in property specification process is a subset of Timed Computation Tree Logic (TCTL). As mentioned, TCTL among others allows the use of time constraints in the specification of properties.

Verification The Uppaal tool has two main modes, one that presents a graphical visualizer for quickly and efficient simulation and another where properties may be specified and then validated. In the latter, different algorithms search (depth-first and breadth-first) and abstraction techniques (allowing different modes for the space state representation and techniques that reduce the state space) were provided.

2.3.3. NuSMV

The New Symbolic Model Verifier (NuSMV) is based on the SMV model checker, actually it is the reimplementation of the latter. The SMV model checker was the first symbolic model checker, and as such was in the origin of this new class of model checking tools [McM93]. The symbolic model checkers aim to tackle the main problem of model checking, the state explosion, by introducing the concept of symbolic representation. This concept allow the manipulation of sets of states rather than single states as in the classic model checking approach, where the symbolic techniques Binary Decision Diagram (BDD) and Satisfiability Solving (SAT) have a prominent role.

Modeling language and property specification The modeling language of NuSMV was inherited from SMV, and it describes finite state machines. The specification of a NuSMV model is made using modules, which have parameters and may be instantiated several times. There must be at least one *main* module per system (like in C language). The modules specification have a variable declaration part and an assignment part. The assignment part may be divided in two different parts: the variables initialization; and the definition of system transitions, which is made indicating the value (or the values, allowing non-determinism) of variables for the next state. The described systems specification process is very extensive and along with the obligation of array indexes and parameters having to be constants, make the NuSMV modeling language quite low-level. The properties specification is made in the models within the modules declaration. NuSMV supports the specification of formulas in both LTL and CTL formalism (including one predefined pattern of properties, the *invariants*).

Verification The verification of NuSMV models includes both model simulation and validation. In the latter, are offered the classic model checking algorithms (i.e, the exhaustive verification) for BDD's representation and the bounded model checking algorithms using a propositional SAT solver. The bounded model checking algorithms unroll the finite state machines only for a fixed number of steps, thereby reducing the resources required for the verification but also decreasing the effectiveness of verification.

2.4. Software Model Checking - Tools

The earlier model checking tools (e.g. SPIN) were not prepared to perform the verification of software systems. Many problems (already enumerated) emerged when it was tried the application of model checking technique to software. In order to solve these problems a new generation of model checkers came up (the software model checking tools). These new tools have among others automated the modeling phase by starting to receive programs as input instead of models. In Table 2.1 are illustrated some of these software model checking tools as well as their supported input programming language. In the rest of this section were analyzed four of these model checking tools: Quasar, Ada Translating Toolset, BLAST and BANDERA.

Model Checking Tools	Input Language
SPIN	PROMELA and C
BLAST	C
Quasar	Ada
BANDERA	Java
Ada Translating Toolset	Ada
SLAM	C
Java Path Finder	Java

Table 2.1.: Model checkers and translation tools input languages

2.4.1. Quasar

Quasar [BPP99, EKPPR03a]is a model checker tool, that enables the specification and verification of temporal properties on concurrent Ada programs. Quasar receives Ada programs as input, converting them into high level Petri Nets.

The translation process of an Ada program to a CPN performed in Quasar, can be divided in two phases.

1. The whole Ada program is converted into a set of generic Petri Nets, where each Petri Net is an Ada construction. These generic Petri Nets have abstract transitions that are then replaced by the correct sub net, this operation is called substitution.

2. The set of produced Petri Nets in (1) is then combined through merging operations. These operations combine two Petri Nets by merging places having the same name.

```
task body Client is
 Begin
   Loop
     server.service
   End loop
 End Client

task body Server is
 Begin
   Loop
     accept service
   End loop
 End Client
```

To clarify the translation process performed by Quasar, Figures[4] 2.4 and 2.5 illustrate the first and second phases, respectively, for the translation of the above Ada program (The $\prec$ and $\bowtie$ operators present in Figures 2.4 and 2.5, represent the substitution and merging operations, respectively).

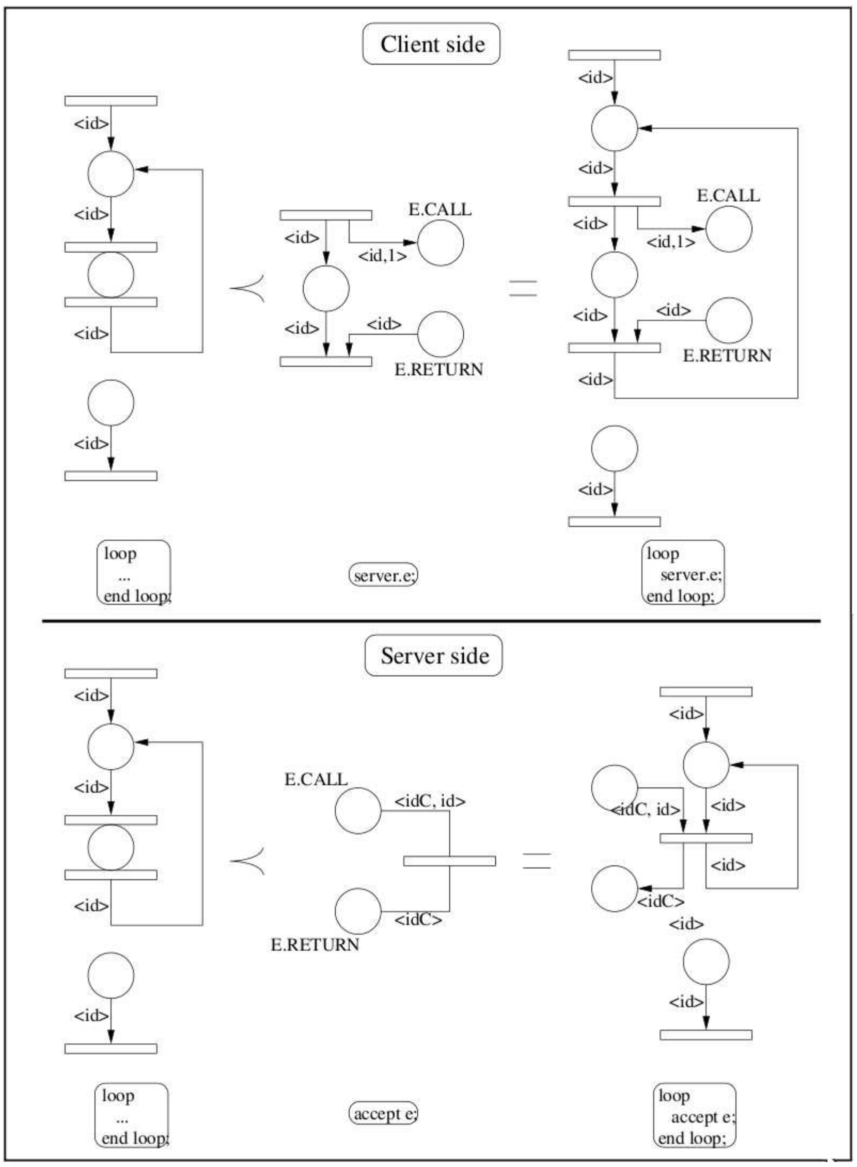


Figure 2.4.: The first phase of Quasar translation process

[4]Figure imported from [EKPPR03a]

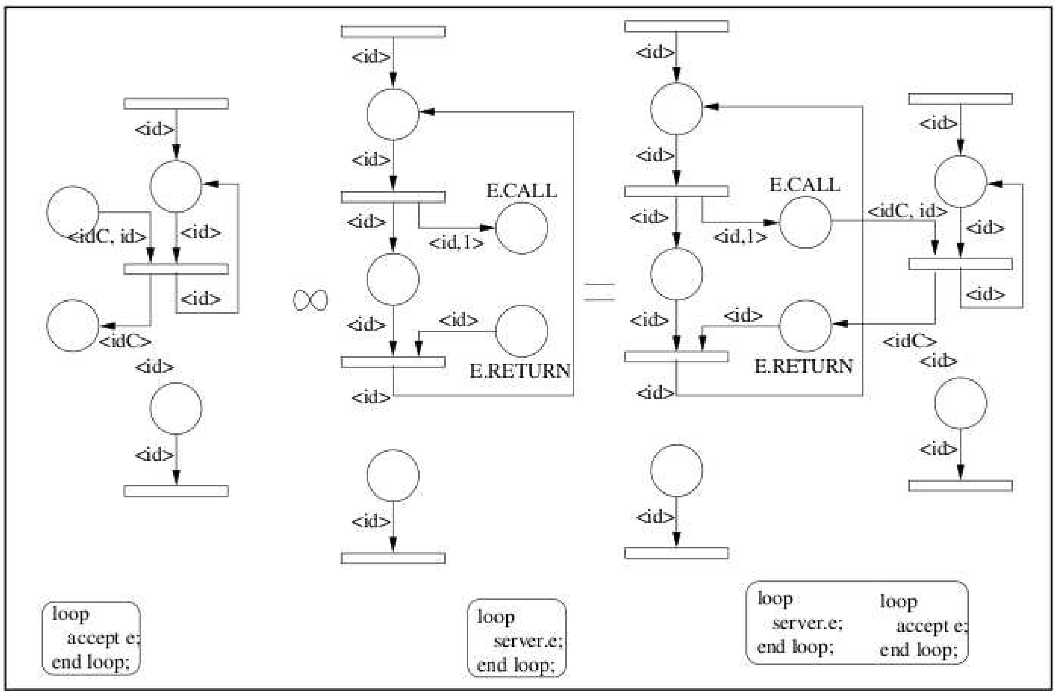


Figure 2.5.: The second phase of Quasar translation process

For the design of temporal properties, Quasar provides four templates: state accessibility, bounded wait, critical section and stable property. These templates are LTL formulas and correspond to the most common properties, for more information about them see [EKPPR03a]. The validation of Quasar extracted models is performed with High Level Net Analyzer (HELENA) [Eva05].

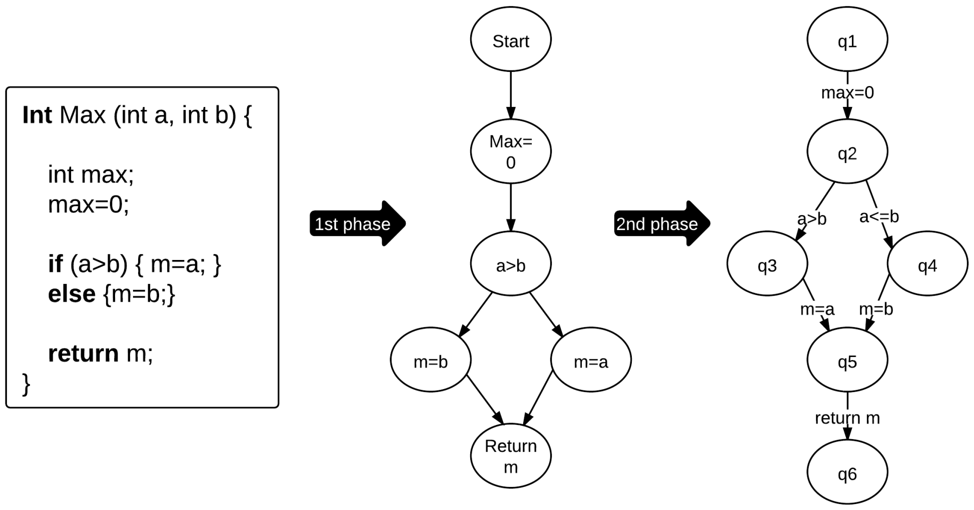
2.4.2. BLAST

Berkeley Lazy Abstraction Software Verification Tool (BLAST) [BHJM07] is an automatic verification tool for checking temporal safety properties of C programs. Given a C program and a property, Blast answers the question " Does the program satisfies the property?". If the answer is *no* it provides an execution path that exhibits a violation of the property (or, since the problem is undecidable, does not terminate).

BLAST uses C Intermediate Language (CIL) [NMRW02] to extract information from C programs, this information is crucial in the model extraction process. Programs are internally represented as a set of Control Flow Automata (CFA), one for each function. A CFA is a directed graph, where states correspond to control points of the program and edges correspond to program operations. There exist four types of program operations: blocks of assignments, assume predicates[5], function calls and return instructions. To translate a program to a set of

[5] An assume predicate is the condition that must hold to execute a certain transition

CFA's BLAST use an intermediate representation, which is the Control Flow Graph (CFG) of the correspondent program. Figure 2.6, demonstrates the extraction process of *Max* program performed by Blast. *Max* is a simple program which receives two integers and returns the bigger one.



```
Int Max (int a, int b) {

    int max;
    max=0;

    if (a>b) { m=a; }
    else {m=b;}

    return m;
}
```

Figure 2.6.: The translation of program `Max` done by Blast

The verification on BLAST can be done, through these three forms: reachability checking, assertion checking and temporal safety specifications. Reachability checking is made through the use of labels, the user may annotate labels in C programs enabling BLAST to check whether such labels are reachable or not in the source code. Assertions are checkpoints of programs, where a given property must hold and they can be annotated everywhere in C program. For the specification of temporal properties, BLAST has its own specification language, which is given in terms of program events.

2.4.3. Ada Translating Toolset

The Ada Translating Toolset [DPC98] converts Ada programs to input languages of different model checkers (SPIN and SMV). Thereby enabling, in an automatic way, the verification of an Ada program by different model checking tools.

Initially this toolset, converts the original Ada program to a safe finite-state variant by applying program abstractions, configuration, restriction and specialization enabling the creation of a tractable model to apply model checking technique. The transformed code is then converted to S-Expression Design Language (SEDL) [Cor93] using the IRIS-Ada toolset. SEDL allows the specification of concurrent systems in a compact language notation that can be easily converted into finite state Automata. The next step is the design of properties in INCA query language. Finally, with the concurrent system written in SEDL and the properties specified in INCA query language, the INCA tool [Sie02] creates the input model and property for a model checking tool. The described translation process is graphically illustrated in Figure[6] 2.7.

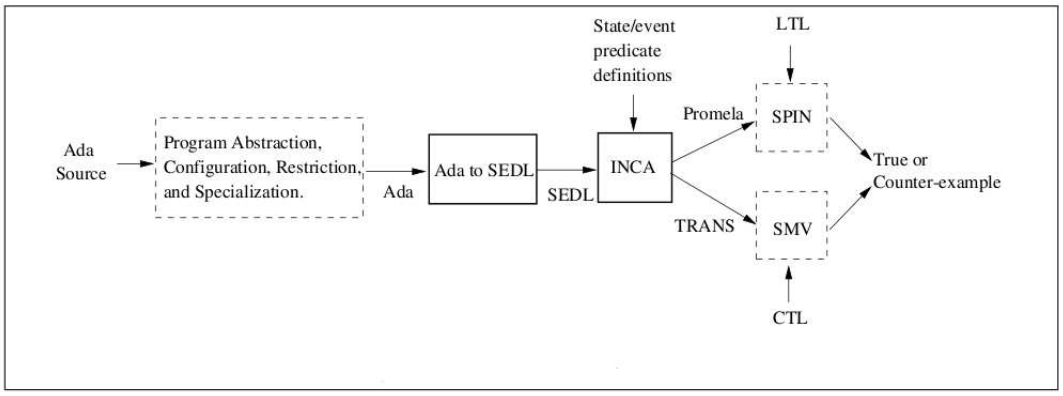

State/event predicate definitions
LTL
Promela
SPIN
Ada Source
Program Abstraction, Configuration, Restriction, and Specialization.
Ada
Ada to SEDL
SEDL
INCA
True or Counter-example
TRANS
SMV
CTL

Figure 2.7.: The Ada Translating Toolset approach

2.4.4. Bandera

The Bandera tool is the most complete of the tools presented in this section, because it tackles all of the software model checking main problems. The Bandera project was carried out by an experienced team, involving many people who had participated in the Ada translation Toolset development. However, the Bandera tool whose the first released was set up in 2005 is no longer supported.

Bandera receive Java programs as input and is able to extract from these, input models for several model checkers. The specification of Java programs properties is available through Bandera Specification Language (BSL), an own language based on source-code annotations, with a similar syntax to the already "dead" Java Modeling Language (JML) [BCC+03]. BSL enables the specification of temporal properties through the same high level mechanism of ATOS, which encompasses a set of predefined pattern properties (see Section 5.2.1). Assertion on Java programs were also allowed in BSL, as well as the specification of *observable* properties (commonly denominated as invariants) of different Java control points.

At first, Bandera converts Java programs into an intermediate representation using Jimple language as support. Jimple is essentially a language of control-flow graphs, targeted to be a Java decompiler. Next, with the intermediate representation and the properties specified in BSL, Bandera builds the Bandera Intermediate Representation (BIR) using a tool named BIR constructor. Lastly, from BIR Bandera is able to generate input models for four different model checkers: SPIN, dSPIN (a no-longer spin extension with functions and pointers) [DIS99], Symbolic Model Verifier (SMV) (the origin tool of New Symbolic Model Verifier (NuSMV)

[6]Figure imported from [DPC98]

tool [CCGR00]) and Java Path Finder (JPF) [Hav99] (which was initially a Java translator to SPIN that was then converted into a standalone model checker).

Bandera also encompasses a back-trace mechanism, enabling an easy mapping of the error traces provided by the enumerated model checkers, and two program abstraction techniques: *program slicing* and *abstract interpretation*, whose details are presented next.

Program slicing Program slicing is one example of an abstraction model technique. Based on a given property, all lines of a program which do not influence the verification of that property are removed. Thereby reducing the size of program and consequently the number of model states. For example, in a property which checks the value of a variable A, all variables in program that do not influence A's value, can be removed.

Abstract interpretation This technique reduces the model *via* data abstraction. When a property to be checked does not depend on the program's concrete values but instead depend on the properties of those values, some values could be abstracted. For example, a program may use a vector to store information about people (e.g age, date of married, number of brothers, etc), but if the property to be checked only depends on a particular information such as being married or not, it is possible to abstract the large number of vector states onto a small set:*{married, not married}*.

3. An Overview of ATOS

The tool presented in this chapter (ATOS) may be classified as a software model checker, a framework which helps in the application of the model checking technique to software systems (both definitions seem correct). The main goal of ATOS is to help in the overcoming of the software model checking problems identified previously, in an attempt to verify concurrent software systems.

ATOS integrates two mechanisms which are very helpful in taming the model construction and the property specification problems. The mechanism used to tackle the model construction problem is the model extraction whereas the property specification problem is tackled using the property extraction mechanism.

The model extraction consists in the automatic generation of models from programs. In particular ATOS receives as input Ada programs (only the main program is required) and outputs PROMELA models. The technical details of the model extraction process are given in chapter 4.

The properties extraction mechanism results from joining two different mechanisms, one that automatically generates properties from programs and another which builds properties from annotations in Ada programs. The annotation language provided is inspired by SPARK. The properties extracted by these mechanisms can either be in the form of LTL formulas or assertions embedded in the extracted models. More details about these mechanisms are given in chapter 5.

The models and properties extracted by ATOS are then verified with the SPIN model checker. Although the SPIN verification step is an automatic process, the user should have some experience with and knowledge of the SPIN model checker. This experience and knowledge are even more important in the presence of the state explosion problem, since SPIN offers several abstraction techniques and different search algorithms. Knowing how to take advantage of these functionalities could be the difference between being able or not to verify a model.

In summary, ATOS receives Ada programs as input, generating a model of these programs along with properties to be verified against this model. The verification is then performed by SPIN, which receives as input the previously generated model and properties. The graphical representation of this process is illustrated in Figure 3.1.

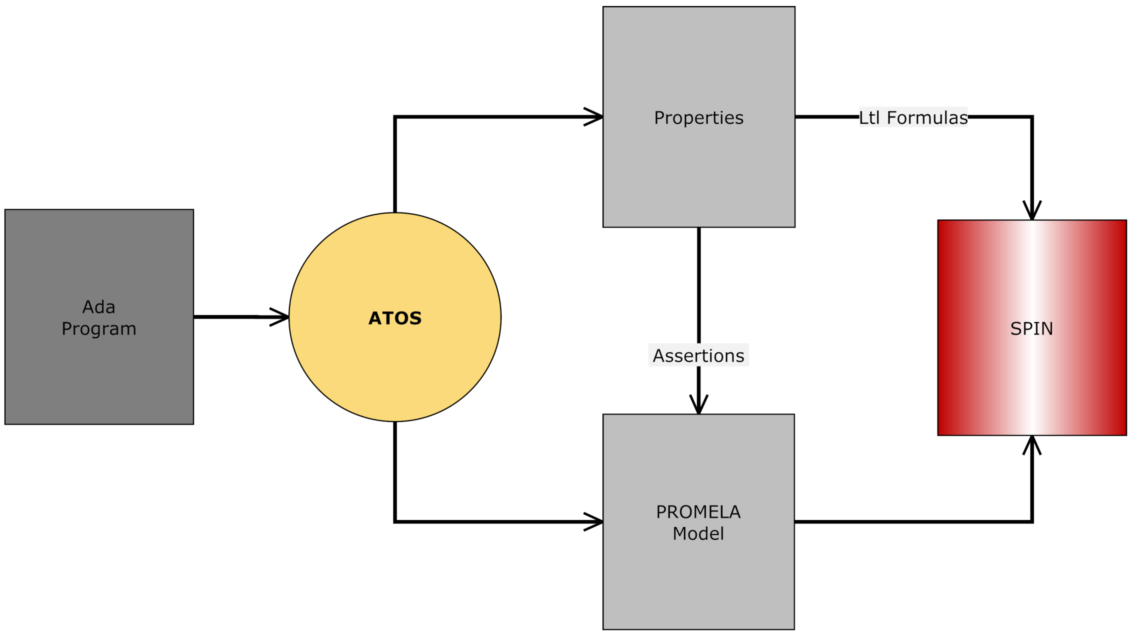


Figure 3.1.: ATOS process

ATOS was fully implemented in Ada, using the Gnat Programming Studio (GPS), an Integrated Development Environment (IDE) for Ada development. GPS was very helpful in the maintenance, debugging and testing along the software development process. No less important was the Ada Semantic Interface Specifications (ASIS) [BSB91], a library that offers an excellent interface to the Ada syntax tree (AST) of programs. This library has provided all the syntax and semantic information required by ATOS.

In the remaining of this chapter it is given an overview of ASIS and explained how ATOS is capable of converting an Ada program by just receiving the main of a program. The simple ATOS GUI is also explained in this chapter, by giving an overview and illustrating its principal functionalities.

3.1. ASIS

The packages of ASIS library offer an excellent interface to the Ada syntax tree programs. The access to the environment[1] of Ada programs provides valuable syntactic and semantic information. The design of ASIS tries to be independent of Ada environments, thus supporting portability of software engineering tools.

ASIS packages explore the abstract syntax tree (AST) of an Ada program. Each node from AST can be accessed through the package *Element* and other packages allow the access to all information of a node, such as: kind (declaration, expression, statement), related elements

[1]Ada AST can be referred as Ada environment

and components.

ASIS offers two ways to access the AST, through an iterator that transverses the whole tree, using the algorithm depth-first [CSRL01] and through queries. The iterator is used to access generic information, such as: list of tasks, list of variables, list of protected objects, etc. While queries are used to obtain specific information, like the upper and lower bound of a cycle clause. The general use of ASIS is explained in its website[2].

Many tools use ASIS as support to perform code analysis (data flow analysis, safety and security compliance tools, static correctness verifiers, tasking analysis tools). The successful use of ASIS by these tools and also in ATOS demonstrates the importance of ASIS concerning the analysis of Ada programs.

3.2. How To Compile from Main?

The translation of Ada programs performed by ATOS requires only the main file as input, even if this depends from other library units declared in separate files. However, in order to allow ATOS to process separate library units there are two conditions which must be valid: 1) all library units must be in the same context and 2) the library units names and the names of the files which contain them must be exactly the same.

The process performed by ATOS to convert an Ada program from main can be divided in two phases, a first phase where it processes all the units in the correct order and a second phase where it puts together all the generated files containing the library units.

Processing Library Units

ATOS uses the with clauses of a main file to obtain and translate all of the library units that is dependent from. Nevertheless, the units must be processed in the correct order because they can be dependent from one another. Therefore, ATOS generates a dependency graph, which allows it to convert the library units in the correct order.

As it was said previously, the dependency graph is built based on the with clauses of main declaration, and on the with clauses of its dependent library units and so on. The generated directed acyclic graph (DAG) is an ordered pair G = (V, A), with:

- V a set of vertices, which in this case represent library units;
- A a set of ordered pairs of vertices, that represent the library units dependency relation.

In order to ensure that there are no circular dependencies between the library units, the generated graph must be a directed acyclic graph, i.e, let P be a path of G:

[2]Http://www.sigada.org

$$(a, Z) \in P \Rightarrow (Z, a) \notin P,\ where\ a \in V\ and\ Z \equiv V \setminus \{a\}$$

In Figure 3.2 is shown an example of a generated graph by ATOS.

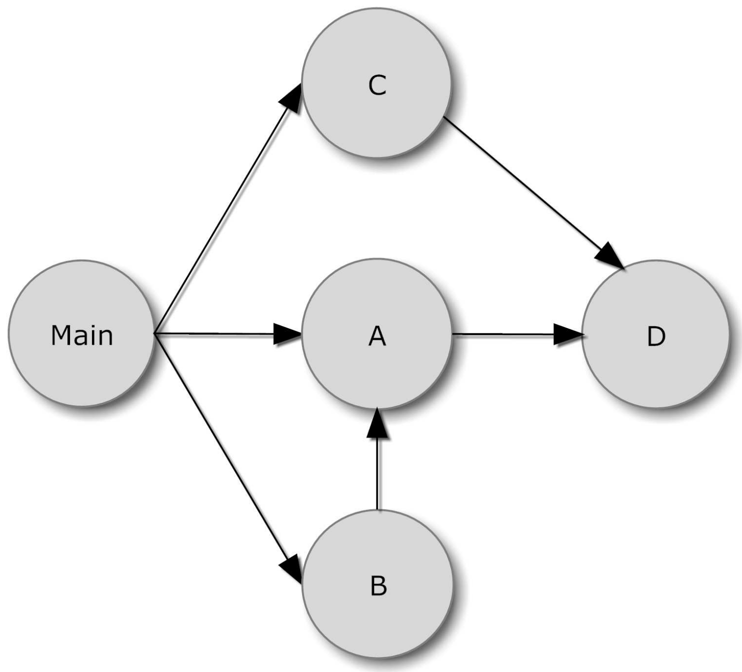


Figure 3.2.: An example of a generated graph by ATOS

After the construction of the dependency graph, ATOS transverses it, starting with a vertex that does not point to other vertex (a library unit that does not depend from others library units). In the next steps this procedure is repeated until the main unit is processed. As an example, ATOS would transverses the graph represented in Figure 3.2 as follows:

1. At first, there is only one library unit that does not depend from others library units which is the D library, so this is the first one to be preprocessed and then translated;
2. Next, there are two library units that can be chosen after D, which are A and C, ATOS randomly chooses one, for hypothesis admit that ATOS choose C;
3. Now, there is only library that did not depend from others, which is A, so is the chosen by ATOS;
4. At this phase, remains only Main and B, but Main depends on B, so in this step the library B is chosen;
5. Finally, is chosen the Main unit.

Order File Inclusion

The translation of an Ada program may origin several different files, ATOS generates one per each library unit declared in a separate file (the specification and the body parts of a unit are put together in the same file). However, PROMELA does not has the same reference mechanisms of Ada, i.e., in PROMELA it does not exist the *with* clause primitive to reference other units. The only way for a declaration from a unit being available for other unit is being declared above of it. Thus, ATOS uses a different mechanism to allow the same visibility rules of Ada, which is the directive *include*. This primitive allows the include of all declarations of a file in a certain point of other file.

ATOS through the directive *include* generate a PROMELA model containing different files as it was a single file, where the order of files inclusion is given according to the library units dependencies. A file (library unit) can only be included once otherwise would occur a redefinition of its declarations. Thus, the inclusion order is given in a list where the first element did not include any of the others, the second element includes the first one, the third element includes the second and so on. This list is generated through the previous generated dependency graph and the algorithm is also the same that generates the processing order.

3.3. GATOS - a graphical user interface for ATOS

ATOS has a simple Graphical User Interface (GUI) designed in Java using the API swing [HWL+02]. The decision of implementing the GUI in Java rather than in Ada, which would seem the obvious choice since it was used in the development of ATOS, was due to the facilities provided by the use of the *swing gui builder* provided by Netbeans IDE.

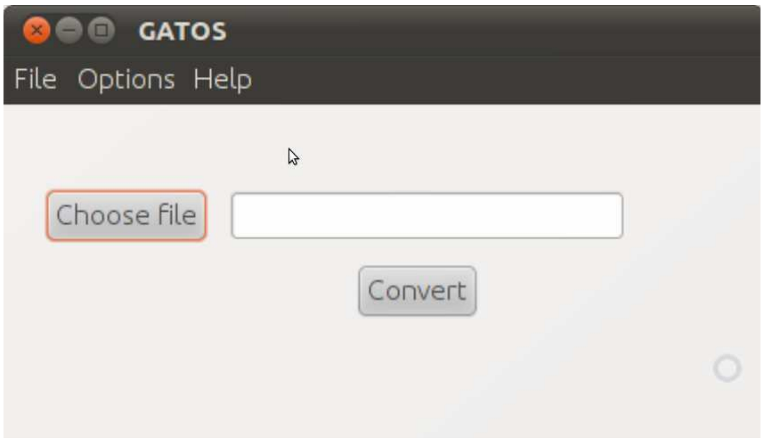


Figure 3.3.: GATOS main window

GATOS is composed by a simple design with colors black and gray as main background colors, the main window is shown in Figure 3.3. The creation of GATOS had as main goal making the execution of ATOS simpler, without requiring knowledge at the command line level. The execution of ATOS can be performed in two simple steps with GATOS:

1. **Loading the Ada program** – The selection of a file (Ada program) may be performed through the **file selector** illustrated in Figure 3.4, which contains a filter for Ada programs (*.adb and *.ads) in order to facilitate the search of Ada files. Loading an Ada program may also be performed by writing directly in the white box (next to the button *Choose File*) the path of the file;

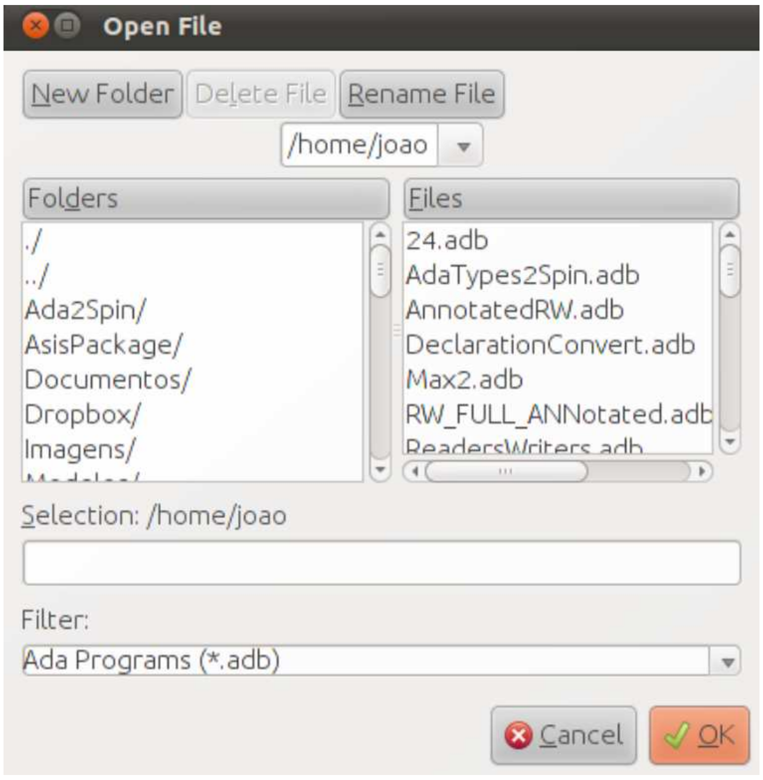


Figure 3.4.: The file selector of GATOS

2. **Executing ATOS** – The ATOS execution with the selected file as input can be performed by simply pushing the button *Convert*, which will generate the PROMELA model as well as several properties for this.

4. Model Extraction

The ATOS tool is capable of translating a subset of the Ada language into the PROMELA language. The targeted subset of Ada intends to be as inclusive as possible. Knowing beforehand that would be almost impossible to cover all of Ada's subset, trying to reach the coverage of other subsets like SPARK or RavenSPARK would be an error. Since these subsets were built aiming to permit a specific type of verification, completely different from the one presented in this thesis, so there were restrictions required for this verification that do not make sense in this one and vice-versa. Therefore, the intention is allowing that these different approaches complement each other at coverage level and not make them comparable.

ATOS makes a special effort to extract accurate models, in particular by choosing an approach which does not use intermediate representations of programs, because this could lead to the loss of program details and consequently to the generation of inaccurate models. It is very important that the extracted models indeed simulate the behavior of the Ada programs, otherwise they are useless – in particular when the main goal is the verification of high integrity systems where failures are unacceptable.

The model extraction performed by ATOS tries to map the Ada primitives into similar PROMELA primitives. Nevertheless, when an Ada primitive does not have a correspondent similar primitive in PROMELA, ATOS tries to convert these so that the semantics of the Ada primitive is respected. As mentioned, all the syntactic and semantic information about the Ada programs required by ATOS in the translation process is fully provided by ASIS.

The model extraction process is performed by traversing the AST of the input program twice. In the first traversal ATOS preprocesses information that will be required in the second traversal, when the model is actually extracted. In the preprocessing phase, ATOS collects information such as the number of tasks, which is required for example to declare the array of channels associated to each process (in particular to obtain the range of the array). The preprocessed information is identified and explained along with extraction details.

ATOS handles the following Ada declarations: subprograms (procedures and functions), packages, concurrency primitives (tasks, protected objects, and entries), variable declarations (including arrays and basic records), integer constants, and new integer types and subtypes. The translation of many statements as well as the conversion of logical operators or support for the Ada inheritance mechanism are also provided by ATOS. Throughout this chapter, these and other details about the model extraction performed by ATOS will be explained.

4.1. Encapsulation

Encapsulation is a well-known mechanism used by most programming languages, including Ada, which restricts and hides object's data. This mechanism is not directly matched in the PROMELA semantics; nevertheless, it is partially assured by the ATOS translation.

ATOS converts all the declarations, whether global or not, as global declarations in the PROMELA models. This solution came up after encountering difficulties in the use of PROMELA scopes, namely due to the fact that PROMELA *processes* (tasks) do not allow for the declaration of *inlines* inside its "body", which are used, for example, to translate procedures and functions.

The PROMELA restriction explained above forced the development of encapsulation mechanisms in ATOS that would not be necessary if there were no such restriction. A list of problems created by this restriction and mechanisms provided by ATOS to maintain encapsulation is given next:

1. The declarations of process types were not scoped anymore, so if a process type (used for instance in the conversion of task types) was instantiated twice, these two new processes would share the same declarations because they were global (e.g. they would share the same variables). To overcome this problem, ATOS generates many declarations as process instantiations, replicating them. These declarations are suffixed with a number indicating the instantiation, which ranges from $0..N$, where N is the number of process instantiations. The replication of process declarations forced the replication of processes themselves (since they were now actually accessing different declarations with different names): a new process is created for each instantiation, once more suffixed with the instantiation number. The instantiations of a process are counted in the preprocessing and if for example a task type is declared but not instantiated then it is not represented in the extracted model, which reduces its complexity;

2. Not only were process declarations considered globally, but all other declarations were global to. So, for example if two processes call the same procedure of a package, they would also share the same declarations. Once more the solution proposed by ATOS is the replication of the procedure twice, one for each process. The procedures are differentiated by prefixing their names with the name of the process that will execute it. In order to know which processes execute which procedures and functions, ATOS generates a Data Flow Diagram (DFD) in the preprocessing phase. With the DFD ATOS not only solves an encapsulation problem, but it also reduces the complexity of the extracted models because it does not generate declarations which are not used by any process;

3. ATOS preserves the encapsulation of variables by declaring them globally in the model, prefixed with the name of its "mother entity". For example, the variable with the name

Var from the protected object PO is declared globally as PO_Var. The variable renaming process is performed automatically by ATOS. This encapsulation mechanism guarantees only the maintenance of variable encapsulation, all other declarations must have different names in order to avoid the redefinition of these along the extracted model.

The solutions introduced by ATOS make all declarations available to all PROMELA primitives, which in principle could in turn lead to new encapsulation errors. However this is not the case in practice, since all the Ada programs considered for translation must be correct, so the encapsulation is guaranteed already by the Ada compiler. In other words, ATOS just has to preserve the referential integrity that is present in Ada programs; it is not required to detect possible referential integrity errors in the input programs. The maintenance of encapsulation is provided by the mechanisms explained above.

4.2. Types

A good type system allows for the construction of powerful abstractions, which provide valuable information to the compiler, so that many logic or design errors can be found at an early stage of the software developing process. The Ada and PROMELA type systems are very different, the first one offers a much more powerful and wider type system.

The wide Ada type system is categorized in a hierarchy where types inherits properties from other types that stand above in the hierarchy. Figure 4.1 illustrates the Ada type hierarchy. The SPIN type system is much more primitive: it has only a few predefined types as shown in table 4.1, and a simple mechanism for the definition of new types that permits the aggregation of several variables in a single structure, i.e. it allows for the declaration of record structure types.

The declaration of task and protected object types are explained in Sections 4.6 and 4.7, respectively. The details of the translation of all other Ada types are given in the remainder of this section.

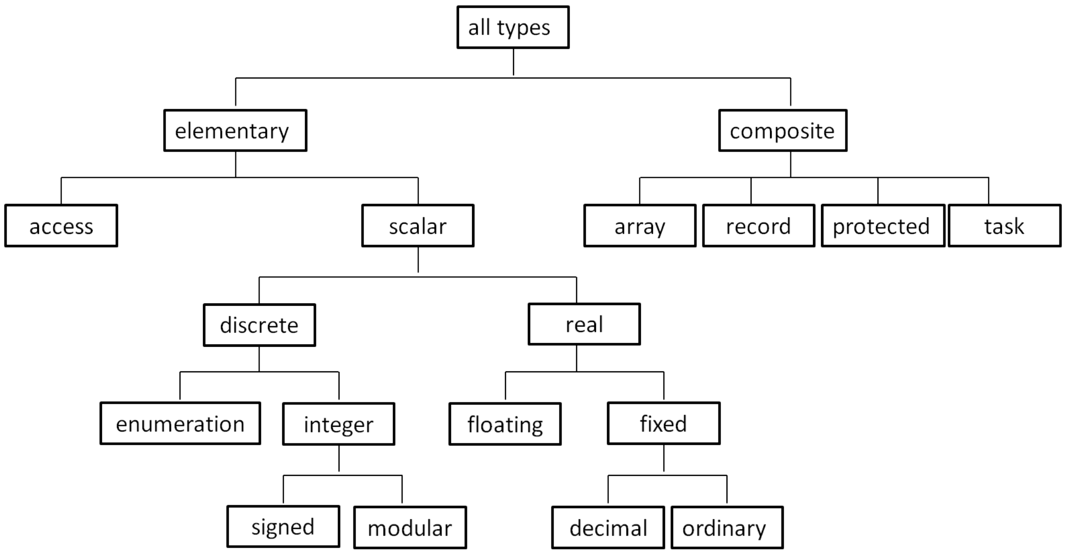


Figure 4.1.: Ada type Hierarchy

Type PROMELA	Range
bit	$0, 1$
bool	false, true
byte	$0..255$
chan	$1..255$
mtype	$1..255$
pid	$0..255$
short	$-2^{15}..2^{15}-1$
int	$-2^{31}..2^{31}-1$
unsigned	$0..2^{n}-1$

Table 4.1.: PROMELA basic types

Numeric Types

The numeric types correspond to the following path in the Ada hierarchy types: *all types* → *elementary* → *scalar* → *discrete* → *integer* → *signed*. In order to provide a more intuitive explanation, *signed* types will be referred to simply as numeric types.

ATOS is capable of converting all the numeric types whose range is defined in $\mathbb{Z}$, where the *minimum* and the *maximum* values allowed are -2^{15} and $2^{15}-1$, respectively. The definition of *maximum* and *minimum* come from the assumption that all Ada programs will run on a 16-bit machine. All Ada types which belong to this range of values are converted to some PROMELA predefined type, the translation is performed by ATOS according to the rules defined in table 4.2. The ranges for PROMELA numeric types (see table 4.1) are defined assuming that the verification of PROMELA models will be performed in 32-bit machines.

Ada Type		SPIN Type
LowerBound	UpperBound	
< 0	–	Short
≥ 0	UpperValue	Unsigned with n bits , where $2^{n-1} - 1 < UpperValue \leq 2^n - 1$

Table 4.2.: Numeric Types conversion rules

The first rule of table 4.2 concerns negative lower bounds. There exist only two PROMELA types which support negative numbers: `integer` and `short`. For the range of values supported by Ada types, `short` is more appropriate, since it supports the range of values defined by the Ada integer types, and it saves memory when compared to the `integer` type (it uses only two bytes instead of four).

The second rule tries like the previous to reduce as much as possible the memory required by the extracted models. The value of `n` corresponds to the number of bits that the unsigned type represents, and is defined according to the smallest number of bits required to represent the upper bound of the Ada type. For instance, a type which represents values between 0..100 is translated to an `unsigned` with 7 bits, whereas a type defined between 0..50 is converted into an unsigned with 6 bits.

Predefined Numeric Types – The Ada programming language offers several predefined types; ATOS is capable of converting some of these types. The predefined numeric types supported by ATOS and the correspondence between these is demonstrated in table 4.3. The matching between the types is done according to the rules of table 4.2.

Type Ada	Range	Type PROMELA	Range
integer	$-2^{15} + 1..2^{15} - 1$	short	$-2^{15}..2^{15} - 1$
positive	$1..2^{15} - 1$	unsigned	$0..2^{n=15} - 1$
natural	$0..2^{15} - 1$	unsigned	$0..2^{n=15} - 1$

Table 4.3.: Correspondence between Ada predefined types and PROMELA types

The range of values that PROMELA types can represent is always greater than or equal to the corresponding Ada type ranges. As such, when corresponding types have different ranges, overflow errors could stay undetected in the models. For example, a variable with type `short` in PROMELA can represent the number -2^{15}, which would not be possible with the corresponding Ada type. This is easily avoided by adding to the model an LTL formula which asserts that a variable respects its range values (see Section 5.1.1).

ATOS allows for range constraints to be used in variables declarations and takes advantage of this by reducing (if possible) the memory needed for the models, which will make easier its subsequent verification. The translation of a variable type declared with a range constraint is performed according to its “new” range of values. Say, a natural variable with range 0..45 is not declared as an unsigned with 15 bits, but is instead declared as an unsigned with 6 bits.

Numeric Types and Subtypes Declaration – ATOS supports the declaration of new subtypes of integer types, event though the PROMELA language does not support this functionality. The information of an integer type declaration is kept in a data structure, which contains the name of the type and its range. Thus, every time the type is used, for instance as the type of a variable, ATOS looks for its range and converts it according to the rules defined in Table 4.2.

Booleans

The boolean type belongs to Ada's enumeration types. Although ATOS is not capable of converting types of this kind, it is capable of translating the boolean predefined type. This type is defined as an enumeration with two values (true or false), and ATOS converts it to the *bool* type in PROMELA. The conversion is direct (and obvious), see table 4.4.

Type Ada	Range	Type PROMELA	Range
boolean	false, true	bool	false, true

Table 4.4.: The translation of the Ada predefined boolean type to PROMELA

Array Types

Ada allows for the declaration of array types. PROMELA does not support the declaration of this kind of types. However, ATOS is capable of translating array types through a strategy similar to the one used for the conversion of numeric type declarations, which is to store the information concerning the type declaration and use it when the type is instantiated. The information required to simulate the declaration of an array type consists of the range of the array; the type of its elements; and the type name. An example of a variable declaration whose type is a previously defined array type is given next[1].

[1]The listing style is different in order to distinguish examples from rules

```
    Declaration of Type
Type ExArrayType is ArrayRange of ElementType

    Declaration of variable using the declared type
VarName: ExArrayType
```

```
  The type is processed

  The variable declaration
ElementType VarName [ArrayRange]
```

Record Types

The record type is a composite type, which aggregates one or more fields (elements) into a single object. Ada offers some alternatives for the declaration of records, such as the declaration of a null record or the declaration of a record with a variant part (for more information about record declaration see [TDB+07]). ATOS supports only the declaration of “simple” records, which are record type declarations defined with just a name and a list of elements.

The PROMELA modeling language provides only a simple mechanism for introducing new types of record structures, very similar to structs in the C programming language. The translation performed by ATOS uses this mechanism to convert Ada “simple” records to PROMELA. The translation rule used by ATOS is illustrated below.

```
Type TypeName
   Record
       ElementsDeclaration
   End Record
```

```
typedef TypeName {
      ConvertedElementsDeclaration
}
```

4.3. Variables

The declaration of a variable is composed by the name of the variable and its type; variables with non-limited types may be initialized. The translation of variables is not static and depends on the variable type. Variables of numeric, record and boolean types have the same translation rule while for the other types there exists a new rule for each type. Variables may also be declared as constants, meaning that their values are never altered during program execution. The translation of constant variables are covered by specific rules too.

The remaining of this section explains the translation rules for variables with numeric, record, boolean and array types, and also the conversion performed for constant variables. The translation of variables whose type is a task type or a protected object type is explained in Sections 4.6 and 4.7, respectively.

Numeric, Boolean and Record Variables

A variable with a numeric type (see Section 4.2) may be defined within a range, which determines its new upper and lower bounds. This range is not explicitly present in the models, but it is used in the translation of variable types as well as to generate range checking properties (see Section 5.1.1).

As was said previously, variables with numeric, boolean and record types share the same translation rule, which may be observed below.

```
VariableName: TypeName [:= Initialization]
```

⬇

```
TypeName VariableName [= Initialization]
```

Array Variables

An array variable is a data structure which aggregates a list of elements, all of the same type. This data structure exists both in Ada and PROMELA. However, in Ada it is more powerful, since it allows the range of an array to be defined using complex expressions. In PROMELA, the range of an array can only be defined within $0..\mathtt{N}-1$, where $\mathtt{N} \in \mathbb{N}$ represents the number of elements. In Ada, the range of an array is defined within `N..M`, where $\mathtt{N},\mathtt{M} \in \mathbb{Z}$ and $\mathtt{N} \leq \mathtt{M}$, thus allowing for both a lower and an upper bound for an array range, unlike PROMELA arrays for which only the upper bound is defined.

ATOS can convert all declarations of arrays in Ada, even those with a lower bound different from 0. In the conversion of an Ada array to a PROMELA array, ATOS first calculates the number of elements defined in the original array, which is given by the formula:

$$Nr\_Of\_Arrays\_Elements = Upper\_Bound - Lower\_Bound + 1,$$

where `Upper_Bound` and `Lower_Bound` are the upper and lower bound, respectively, of an Ada array range. With the number of elements ATOS can declare the array in PROMELA, but the translation work is not over yet, since PROMELA arrays can only be accessed from indexes between 0..`Nr_Elements` $- 1$. This problem is solved by calculating the difference between 0 (the lower bound of a PROMELA array) and the lower bound of an Ada array; the calculated offset is then added to the indexes of the PROMELA array every time it is accessed.

The translation rule for array variables is given next, together with the expression rule to access its indexes.

```
  Array declaration
VariableName: array range (Lower_Bound..Upper_Bound) of ElementsType
                                                    [:= Initialization]

  Array's index access
VariableName(Pos)
```

⬇

```
  Array declaration
ElementsType VariableName[Upper_Bound - Lower_Bound + 1] [= Initialization]

  Array's index access
VariableName[Pos + (Offset)]
```

Multidimensional arrays are not allowed in PROMELA, at least in an explicit way. This restriction forced the creation in ATOS of a special translation rule when the type of the array's elements is an array type (i.e., when a bi-dimensional array is declared). The type of the elements is declared through the typedef mechanism, a new record type is created containing an array variable with the array type, this new type is then used as the type of the multidimensional array. The translation performed by ATOS is clarified by the following rule.

```
  Array declaration
VariableName: array range (Lower_Bound..Upper_Bound) of ArrayType
                                                    [:= Initialization]

  Array's index access
VariableName(Pos1)(Pos2)
```

⬇

```
Typedef SubVariableName {
  SubVariableName    ArraytypeDeclaration
}

  Array declaration
SubVariableName VariableName[Upper_Bound − Lower_Bound + 1]

  Array's index access
VariableName[Pos1 + (Offset1)].SubVariableName[Pos2 + (Offset2)]
                                                  [= Initialization]
```

Constants

A constant is a variable with an initialization whose value is never modified along the program execution (attempts to modify the value of a constant are detected statically at compilation time). ATOS does not need to worry about possible errors caused by constant variables being modified, because the absence of these errors is guaranteed by the Ada compiler.

The translation of constant variables could be exactly the same as for "normal" variables if there were no problems with the complexity of the extracted models. In an attempt to save resources, ATOS has different rules for constant variables, once more selected according to the type of the variable. The details of these rules are presented next.

Numeric Type Constant variables with numeric types are translated into a preprocessor macro. A macro is defined according two entities substituted and substitute, where all occurrences of the first one (substituted) are automatically replaced for the second entity (substitute) by the compiler. In this case the macro substituted is the variable name and the macro substitute corresponds to the variable initialization value, i.e., all occurrences of constant variable name will be replaced by its initialization value. Thus, memory resource savings are optimal compared to other possible translations. The translation rule for constant variables is showed below.

```
VariableName:NumericTypeName constant [Range] [:= Initialization]
```

$$\Downarrow$$

```
#define VariableName Initialization
```

Record Types Constant variables with this type are converted equally to normal variables; the only difference concerns their type declaration. A new type is declared and its elements are initialized with the corresponding initialization values. The elements whose type is numeric may have a different type according to the rules defined in Section 4.2, where the upper and lower bound correspond to the initialization value.

```
VariableName:RecordTypeName constant Initialization
```

⬇

```
Typedef ConstRecordTypeName {
   Elements Initialization

}

ConstRecordTypeName VariableName
```

Array types The translation performed by ATOS to allow for constant arrays is very primitive, assuming that the initialization value is equal for all of its elements, the elements with numeric types are redefined according to their new range.

Derefered Constants This mechanism allow for the declaration of a constant variable in the visible part of a package, while its initialization is performed only in the private part or imported from other languages. Derefered constants are translated as if they were "normal" variables, i.e., they are declared and then initialized.

4.4. Subprograms

Subprograms encompass functions and procedures, whose execution is invoked through a function call and a procedure call, respectively. A function call is an expression and returns a value, whereas a procedure call is a statement and does not return any value. The definition of a subprogram can be given in two parts: the declaration that defines its interface and the body that defines its execution.

A subprogram can have parameters, each with its associated modes. There are three types of mode:

- **In** – A parameter with this mode is considered to be a constant (see Section 4.3), which means that it cannot be modified by the subprogram, and must be initialized. This mode is the default mode if no mode is specified. Functions can only have parameters with the In mode, in order to ensure side-effects freedom.

- **Out** – The parameter with this mode is an uninitialized variable. The variable is then initialized during the subprogram execution. This mode is typically used to pass information from the subprogram to the calling program.

- **In Out** – This mode is very similar to the Out mode; the main difference is the fact that in the In Out mode, the parameter (i.e., the variable) must be initialized.

A parameter can be passed either by-copy or by-reference. A parameter passed by-copy denotes a separate object from the actual parameter and the information transferred between the two happens only in two moments: immediately before and after the subprogram execution. A parameter passed by-reference on the other hand is updated along the subprogram execution.

The parameters passed by-reference are not a problem since they are updated along the correspondent PROMELA primitive execution, but the parameters passed by-copy must undergo a special processing. Firstly, it is important to identify which parameters are passed by-copy or by-reference. This distinction is made based on the parameter's type and is clearly explained in [TDB$^+$07]. After a careful look at this definition it follows that the types that require a special treatment are elementary types and composite types whose components all have an elementary type. The rest of the types do not need a special treatment, because either i) they have by-reference types (i.e. they are parameters passed by-reference); or ii) their translation is not supported by ATOS.

The mode of a parameter also has to be considered in the translation of by-copy parameters, since if a parameter has mode In its value is never altered, so it does not matter which type it has. Thus, parameters with mode In do not undergo special processing, even if they are considered to be passed by-copy.

The solution for parameters passed by-copy is the creation of an auxiliary variable for each, which is assigned with the value of the parameter before the beginning of the procedure execution (note that functions only have parameters with In mode). All occurrences of this parameter inside the procedure are replaced by the corresponding auxiliary variable. At the end of the procedure execution, the parameter is assigned with the corresponding auxiliary variable.

In the remaining of the section the translation of procedures and functions will be explained in more detail.

4.4.1. Procedures

A procedure is a subprogram which may have side-effects, i.e., it can have parameters with the three modes. Unlike functions, procedures do not return any value to the calling program:

they are simply a couple of declarations and statements encapsulated in a single object.

The specification part of a procedure declaration it is not important for the translation because the body contains all the required information, so it will be omitted. There is no support for the direct translation of procedures in PROMELA. However there exit three PROMELA primitives which could possibly be used to represent procedures: Processes, Inline and Macro. Both structures have advantages and disadvantages, which will now be considered.

Processes are able to encapsulate a sequence of statements, and allow for the definition of parameters. However, a PROMELA model can have only 255 processes, so translating procedure into processes could quickly result in this value being reached. Another problem is the asynchronous execution of processes: nothing guarantees that a process which makes a procedure call (creating a new process) will continue its execution only after the process responsible for the procedure execution has finished. Due to these reasons, in particular the latter one, translating procedures into processes does not seem to be a good choice.

Macro and Inline primitives are very similar: both amount to automatic inlining a procedure call into the body of a process. Compared to inline, macro has a disadvantage, which is that line-number references will be restricted to the location of a macro call, not a line number within a macro definition itself. This disadvantage led to the choice of inline to represent procedures in PROMELA, rather than macro.

Nevertheless, the parameters of an inline primitive do not have types or modes associated: unlike parameters of subprograms, inline parameters are simply names. This does not create a problem, since ATOS only converts Ada programs that compile successfully, and therefore the parameters' type and mode checking are guaranteed up-front by the Ada compiler.

The rule used by ATOS for the translation of an Ada procedure into a PROMELA inline is illustrated below.

```
Procedure ProcedureName (Type: Mode ParamName,...) is

        Declarations

    Begin

        Statements

End ProcedureName
```

⬇

```
        Declarations

inline ProcedureName (ParamName,...) {

        Statements

}
```

4.4.2. Functions

A function is a subprogram that can be invoked as part of an expression and has no side-effects on the calling program. Every call to a function produces a new copy of any object declared within it. The body of a function must have at least one return statement (see Section 4.9). As is the case in procedures, the specification part of a function is not relevant in the translation process because the body contains all the required information.

Functions are converted by ATOS into inline primitive for the same mentioned reasons as procedures. However, the semantics of functions brings a few extra problems, since a function can be recursively invoked as part of an expression. An inline can not be called recursively nor can it return a value, and consequently, cannot be part of an expression either.

The solution proposed by ATOS to overcome these problems cannot fully represent the behavior of Ada functions. ATOS adds an extra parameter to the function parameters, which will contain the return value. However, this mechanism requires a few assumptions: the expression containing the function can only contain a single element, which is the function call, and the expression must correspond to the right-side expression of an assignment (see Section 4.9), i.e. the value returned by the function must always be directly assigned to a variable.

The rule used by ATOS for the translation of an Ada function into a PROMELA inline is given below.

```
Function FunctionName (Type: In ParamName,...) is

        Declarations

    Begin

        Statements

end FunctionName
```

⬇

```

        Declarations

inline FunctionName (ParamName,... ,RetValue) {

        Statements

}
```

Automatic Inline

We must remark at this point that a SPIN restriction (or bug) forces ATOS to simulate the inline mechanism without explicitly using this primitive. The restriction is the impossibility of passing array variables as parameters of inlines: SPIN only allows parameterization with individual array positions, not entire arrays. So, instead of converting subprograms which have arrays as parameters into inlines, ATOS generates only their correspondent declarations and keeps the statements apart in a data structure. Thus, when a subprogram call is made, all the occurrences of the parameters are replaced by the instantiated parameters, and the statements are written directly in the process which made the call.

4.5. Packages

Packages are the *unit design* of Ada, they separate the interface from the implementation. The declaration of a package consists of two parts: *specification* and *body.* The specification part is also divided in two parts, the *private* part and the *visible* part. The basic declarations provided in the latter are visible outside the package, while the ones declared in the private part are only available within the package declarative region. The body region gives the completion of the declarations specified in the specification part, and can also contain new declarations. The package body defines a sequence of statements. In the case that no statements have been given by the user, this sequence of statements is implicitly built with the null statement (see Section 4.9).

As was to be expected from the semantic differences between Ada and PROMELA, the latter does not have a primitive that resembles a package, so the translation performed by ATOS is once more based on the mapping of its semantics to PROMELA models. The key idea behind the translation of packages is the fact that the visibility rules defined by packages are not relevant for their translation, since the Ada compiler ensures statically that they were respected by the Ada program inputs. Thus, the translation of packages is simply the translation of its declarations and statements. In order to avoid problems with the redefinition of declarations (which are global in the model), all the declaration names from packages are prefixed with the package name, for instance the function with name **func** declared in a package with the name **pack** is declared in a PROMELA model as **pack_func** and consequently all this function calls were also renamed. This encapsulation mechanism is performed automatically by ATOS.

Regarding package statements, they are encapsulated through the inline primitive and executed in the main program (see Section 4.10). When a package does not contain statements the correspondent inline is not generated unlike what happens in Ada. Where even if a package is declared with no statements it contains implicitly the *null* statement. The rule for the translation of packages is illustrated above.

```
Package PackageName is

        Declarations

    [Private]

        Private Declarations

end PackageName

Package body PackageName is

        Body Declarations

    [Begin]

        Statements

end PackageName
```

```
        Declarations

        Private Declarations

        Body Declarations

inline PackageName {

        Statements

}
```

4.6. Tasks

Ada tasks represent different threads of control, which execute independently and concurrently. This primitive may contain several interaction points, which allow the communication with other tasks. Beyond these interaction points tasks can interact with / affect other tasks in many ways (e.g. through shared variables).

The definition of an Ada task is divided in two parts: the specification, which describes the interface with other tasks, and a body that contains the code defining the task's behavior. A task can either be declared as a single task or as a type task. The first becomes active from the moment it is declared; whereas the type task simply creates a new type which can be instantiated later. The interaction points are declared first in the specification part through entries declarations which correspond to accept statements in the body (see Section 4.9).

Tasks are translated to PROMELA through the process primitive. Similarly to Ada tasks, processes in PROMELA are the only primitive that can represent parallel activities. Simple tasks are converted into active proctypes, which become active from the moment they are declared (just like simple tasks in Ada). Nevertheless, when a simple task is declared in the main program it will only become active after the statements of packages have been executed. The primitive *provide* can be used to suspend a process, by specifying a condition that must be valid for a process to execute. The condition included in active processes is given by the boolean variable `Start`, which has the value true (i.e., the condition is valid) in the main process only (see Section 4.10). Consequently, until the model gets to this point active processes are suspended.

Type tasks are translated into PROMELA proctypes, which simply creates a new process type that can be instantiated later. However, ATOS creates a new type for each instantiation due to encapsulation issues as explained in Section 4.1. The translation rule for both simple tasks and task types are in listings 4.1 and 4.2, respectively.

Listing 4.1: Simple task conversion rule

```
Task SimpleTaskName is
      Declarations
    [Private]
      Private Declarations
end SimpleTaskName

Task body SimpleTaskName is
      Body Declarations
    [Begin]
      Statements
End SimpleTaskName
```

⬇

```
        Declarations
        Private Declarations
        Body Declarations

active proctype SimpleTaskName(Id,Params...) provided Start {
        Statements
}
```

Listing 4.2: Task type conversion rule

```
Task type TaskTypeName is
      Declarations
    [Private]
      Private Declarations
end TaskTypeName

Task body TaskTypeName is
      Body Declarations
    [Begin]
      Statements
End TaskTypeName
```

⬇

```
        Declarations
        Private Declarations
        Body Declarations

proctype TaskTypeName0..N(Id,Params...) {
        Statements
}
```

A process has an associated ID number that univocally identifies it, and is used when a process makes an entry call (see Section 4.9). Processes also have an associated channel through which they can receive messages from other processes, used for example in accept statements (see Section 4.9). Channels are declared as an array where each array position corresponds to a channel belonging to the process whose ID corresponds to the index. The range of this array is calculated in the preprocessing phase, by counting the number of tasks in a model thereby the dynamic creation of tasks is not allowed.

4.7. Protected Objects

The protected object primitive is a structured mechanism that provides mutually-exclusive access to shared data. A protected object is relatively similar to a package, the main difference being the fact that all operations of a protected object are mutually-exclusive. The protected operations encompass three different declarations: procedures, functions and entries.

The semantics of protected objects are commonly explained using the eggshell model. Figure 4.2 illustrates graphically a protected object.

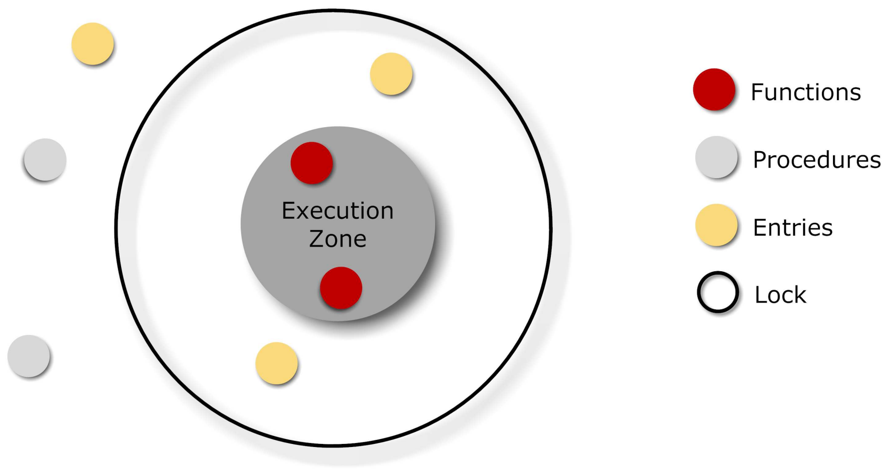


Figure 4.2.: Graphical representation of a protected object

The eggshell (black circle) represents a *lock* primitive. Each task must acquire the lock in order to execute a protected operation (the behavior of a lock is similar to that of a semaphore). Only one task can be inside the eggshell if it is executing an entry or a procedure operation; several tasks can be inside if they are executing function operations. Protected objects allow several functions to be executed concurrently because functions (supposedly) do not have side effects (this is not completely true because functions can in fact change the value of shared variables, influencing the other tasks' behavior).

When a task has accessed the eggshell (lock), if it is trying to execute a procedure or a function it goes directly inside the eggshell (i.e., starts its execution). If on the other hand it is trying to execute an entry operation, after having acquired the lock, the task must evaluate its entry barrier before execution starts. If the barrier is open (i.e. it has been evaluated to true) the task goes to the execution zone (i.e. inside the eggshell), otherwise it is queued in the associated queue of protected objects (i.e. stays within the eggshell). The tasks that have been blocked are enqueued one-by-one; each time a task terminates an operation, the barrier of the entry operation is reevaluated and if it is now open the task goes to the execution zone; otherwise it is queued again.

Since no similar primitive is available in PROMELA, the conversion of protected objects consists in the mapping of its semantics to the PROMELA models. The first step of the translation is the modeling of locks in PROMELA. A lock is modeled using a record structure with two variables: a boolean variable named `Critical_Section`, that indicates if the critical section is occupied (i.e., if a task is executing) and a variable `Functions`, which counts the number of functions in the execution zone. The semaphores are declared as an array whose elements have the previously explained structure as type. The number of array elements corresponds exactly to the number of protected objects. The declaration of semaphores in PROMELA is shown below.

```
typedef Semaphore{

byte Functions;
bool Critical_Section;

};

Semaphore Sem[**Nr. of Protected Objects**]
```

The state of a semaphore is altered through four operations: *Acquire, Release, AcquireF* and *ReleaseF*. More details about these four operations are given next:

- *Acquire* – change the value of `Critical_Section` to true, but only if its value is false, i.e, if no process is already in the critical section. Only entries and procedures execute this operation;

```

inline Acquire(SemIndex) {
atomic{

!Sem[SemIndex].Critical_Section →
            Sem[SemIndex].Critical_Section = true

}
}
```

- *Release* – alters the value of `Critical Section` to false when a process leaves the critical section. Only entries and procedures execute this operation;

```

inline Release(SemIndex) {
atomic{

        Sem[SemIndex].Critical_Section = false

}
}
```

- *AcquireF* – if there are no processes in the critical section or the process(es) is (are) executing a function operation, this changes the value of `Critical Section` to true and increases the value of `Functions`. Only functions execute this operation;

```
inline AcquireF(SemIndex) {
atomic{

(!Sem[SemIndex].Critical_Section || Sem[SemIndex].Functions > 0) →
                                   Sem[SemIndex].Critical_Section = true
                                   Sem[SemIndex].Functions++

}
}
```

- *ReleaseF* – decreases the value of `Functions`, and if the new value is equal to 0, i.e. if no processes remain in the execution zone, the value of `Critical_Section` is altered to false. Only functions execute this operation.

```
inline ReleaseF(SemIndex) {
atomic{

Sem[SemIndex].Functions-- →
                 if
                   ::Sem[SemIndex].Functions == 0 →
                            SemSem[SemIndex].Critical_Section = false
                   ::else → skip;
                 fi;

}
}
```

The next step in the translation of protected objects is the creation of a process that manages the functioning of protected operations. This process is responsible, among others, for the queue of entries. When a process finishes the execution of a protected operation, it communicates this to the protected object process (except if it is a function operation), which then activates the queued processes, in order for them to test again theirs entries barriers. If a barrier is now open then the enqueued task may execute, otherwise it is enqueued again. The lock is released if none of the entries has its barrier open. The PROMELA pseudo-code implemented by ATOS to simulate the protected object process is listed below.

```
do
   :: Waiting for a task termination  →
        do
          :: There are blocked tasks (entries operation)
               enqueue a task
               activate the enqueued task
               waiting for the reevaluation of entrie's barrier →
                   if
                    :: barrier is close → queue the task again
                    :: barrier is open (had executed)
                   fi
          :: else → Open the semaphore
                    break;
        od
od
```

A final element is necessary to understand the full translation of protected objects, concerning the interaction between the protected operations and the previously explained protected object process. As was said previously, there exist three different types of protected operations: entries, procedures and functions.

Procedures and Functions – The translation of procedures and functions as protected operations is almost equal to the translation of their 'normal' declarations. The only difference is that the protected operations have two extra statements, one at the beginning and the other at the end. The statement at the beginning concerns the acquisition of the lock for both operation, whereas the end statement corresponds in the case of function to the lock release, and in the case of procedures to the communication of operation termination to the protected object process.

Entries – The declaration of an entry in the specification part of a protected object, unlike its declaration in a task's specification, originates an entry body in the body part. The semantics of an entry as a protected operation is similar to the one for procedures (the execution of a sequence of statements). Entries differ from procedures just in that they have a barrier before the execution of their statements. This barrier should avoid referring to variables outside the protected object, so that the underlying assumptions of the state of the protected object are not violated.

The translation of entries to PROMELA is similar to the translation of procedures; both are converted into an *inline* and their parameters to *inline parameters*. However, the inline containing an entry's statements have a few extra statements to simulate the barrier of an entry, and the interaction with the protected object operation. The translation rule used by ATOS to convert an entry body to PROMELA is given next (the specification is omitted, because it is not relevant). In order to facilitate the understanding of the rule the corresponding SPIN code is given as pseudo-code.

```
Entry body EntryName(Params...) is
        Declarations
    Begin
        Statements
end EntryName
```

⬇

```
        Declarations


inline EntryName(Params...) {

     Acquire the lock

     TestCondition:
       atomic{
        if
         :: barrier is open →
                        goto Execute
         :: barrier is close →
                        if
                          ::Had already been blocked → goto InsideEgg
                          ::else → Is blocked
                                    Releases the lock
                                    goto InsideEgg
                        fi
        fi
       }
      InsideEgg:
         atomic{

         Wait until it is activated → goto TestCondition

         }

      Execute:
       Statements
       Inform protected object process that it has successfully finished
}
```

4.8. Expressions

Expressions in Ada can be explained as a combination of operators and terms whose evaluation retrieves a value. Each expression has a type, which defines the type of the retrieval value. Expressions are present in almost all programming languages, including Ada and PROMELA.

Ada gathers many different types of expressions. A semantic analysis of these expressions is performed by ATOS in the extraction process. The rules used in the conversion of expressions are almost direct, making this translation process trivial. Its discussion will be omitted in what follows.

One of the most important aspects of expressions is the evaluation process. This is performed according to a given set of precedence and association rules; the use of different rules may result in the retrieval of different values for the same expression. A careful analysis of the evaluation order of operators is given in the remaining of this section, as well as the explanation of the package *Expression_ Resolver* that allows ATOS to solve expressions in a static way.

Operators' Evaluation Order

Expressions are evaluated to produce a value; the precedence and associativity of operators affect this evaluation. The semantic and syntax of Ada expressions are not that far from PROMELA expressions; they share most of the operators and evaluation rules. Therefore, the conversion of operators is almost trivial, as shown in Table 4.5.

Ada operator	PROMELA operator
and	&&
or	||
=	==
/=	!=
mod	%
not	!
xor	^
<	<
<=	<=
>	>
>=	>=
+	+
-	-
*	*
/	/

Table 4.5.: Operators precedence level

As mentioned, evaluation order is defined by precedence and associativity rules. Precedence rules are not the same for Ada and PROMELA (which has the same rules as in the C programming language), whereas the associativity rules are equal for both languages. A deeper analysis of precedence rules is thus required in order to avoid errors in the evaluation of expressions.

In Table 4.6 it can be observed that Ada groups its operators in five levels of precedence (shown from lowest to highest), whereas PROMELA has more levels. The general distribution of operators is almost the same in Ada and PROMELA, but some operators which are grouped in a single level in Ada are divided in two or more levels in PROMELA. This difference in operator precedence may have disastrous consequences in expression evaluation, because two different results may be obtained for the same expression.

Ada operators Precedence	PROMELA operators Precedence
and, or, xor	||
	&&
	^
=, !=, <, >, <=, >=	==, !=
	<, >, <=, >=
+, -	+, -
*, /, mod	*, /, %
not	!

Table 4.6.: Operators precedence order

The operators *and* and *or* belong to the same level in Ada but are divided in two different levels in PROMELA, where && (the corresponding *and* operator in PROMELA) has higher precedence than the || operator. Thus, the evaluation of the expression

$$A \textit{ or } B \textit{ and } C$$

in Ada would be:

$$(A \textit{ or } B) \textit{ and } C,$$

whereas in PROMELA, after the translation, the evaluation would be:

$$A \mid\mid (B \ \&\& \ C).$$

The problem is that:

$$(A \textit{ or } B) \textit{ and } C \neq A \mid\mid (B \ \&\& \ C).$$

In order to avoid problems in the evaluation of expressions we advise users to use brackets to delimit subexpressions.

Expression Resolver

Along the translation process, ATOS needs to evaluate some expressions in order to perform the translation of several Ada primitives. The declaration of an array is an example of an Ada primitive which may require the evaluation of an expression, in order to calculate the array range. To help ATOS in situations similar to this, ATOS uses the package *ASIS_Expression_Resolver*, which calculates the numeric/boolean value of an expression through semantic analysis only. Without this package, expressions containing attribute reference such as: 'Last, 'First or 'Range could not be evaluated or converted.

4.9. Statements

Statements are a familiar concept to most programming languages, including Ada. This concept, also known as *instruction*, is also present in PROMELA. ATOS is able of converting the following Ada statements: if, null, assignment, case, loop, exit, while loop, for loop, goto, procedure call, accept, selective accept, return and entry call.

A specific rule is used for each of the supported statements. Some of these rules are trivial due to the common primitives of Ada and PROMELA. However, there are other translation rules which require some creativity in mapping their semantics to PROMELA. In the remaining of this section the conversion rules of statements are explained and illustrated.

If

The *if* statement for conditional execution is common in most programming languages, including Ada. PROMELA also supports this statement, however its semantics is a little bit different from the common *if* instruction semantics. In PROMELA the conditions of an *if* statement need not be mutually exclusive: one of the branches is chosen non-deterministically if more than one condition is valid. Another semantic difference is the fact that in PROMELA if none of the *if* conditions are valid the system is suspended, unlike what happens in programming languages like Ada.

ATOS uses the *If* mechanism provided by PROMELA in the conversion of Ada's *If* statement, despite the semantic differences between them. ATOS uses the *if* statements with only two conditions, the first is a "normal" condition and the second is the primitive *else.* This prevents the nondeterministic choice of conditions and processes from becoming suspended. The translation of the *if* statement is provided by three different rules:

1. **A simple *if* statement without an elsif or else path** – ATOS adds the *else* condition followed by the *skip* statement, to prevent a process from remaining suspended in case the *if* condition is not valid.

```
 If condition Then
     statements
End If
```

⬇

```
  if
    :: condition →    statements
    :: else → skip
  fi
```

2. **An *if* statement with an elsif path** – a new if statement with the elsif condition is executed after the else condition;

```
If condition1 Then
      statements
[Elsif condition2 Then]
      other statements
End If
```

$$\Downarrow$$

```
  if
    :: condition1 →    statements
    :: else → if
                 :: condition2 →    other statements
                 :: else → skip
               fi

 fi
```

3. **The existence of an else path** – the else statements are executed after the else condition instead of using the *skip* statement;

```
If condition1 Then
      statements
[Elsif condition2 Then]
      other statements
[Else]
      else statements
End If
```

```
  if
    :: condition1  →    statements
    :: else  →  if
                     :: condition2  →    other statements
                     :: else  →    else statements
                  fi

fi
```

Null

The *null* statement has no effect in the program execution; it is however a very useful statement. Some points in Ada programs require at least the presence of one statement, and the *null* statement can always be used if no other statements are algorithmically appropriate.

PROMELA has a similar statement to the *null* instruction named *skip*, which has exactly the same semantics. So ATOS translates Ada's *null* statement into the *skip* statement of PROMELA.

Assignment

Assignment statements set a new value for a variable (the left-hand side of an assignment). The assigned value results from an expression evaluation (the right-hand side of an assignment). PROMELA has an *assignment* statement too, however it only allows the assignment of variables with numeric types, whereas in Ada this statement can be used with variables of unrestricted types.

Numeric Types The assignment of variables with numeric type is represented in models similarly to Ada programs. The left-hand side contains the name of the variable and the right-hand side the expression. Only the assignment operator is different (:= is converted into =).

Record Types As mentioned, PROMELA does not allow the assignment of variables with record types, as such the assignment is not made to the whole variable but instead to its elements individually. The assignments of elements are translated as an atomic statement, to simulate the assignment of the original variable. This translation rule is given below.

```
VariableName:= Expression
```

⬇

```
atomic{
   VariableName.Elem1 = Expression.Elem1
     ...
   VariableName.ElemN = Expression.ElemN
}
```

Array Types Similarly to the assignment of variables with record types, variables with array types in PROMELA cannot be assigned in a single assignment, unlike what happens in Ada. Once more the solution is the assignment of the array's elements individually in a single step as an *atomic* statement.

```
VariableName:= Expression
```

⬇

```
atomic{
   VariableName[0] = Expression [0]
     ...
   VariableName[N] = Expression [N]
}
```

Function Call When the expression (the right-hand side) of an assignment is a function call, the translation follows a different rule. ATOS takes the variable to assign and includes it as a parameter of the function, following the conversion of functions (see Section 4.4.2).

```
VariableName:= FunctionCallName(CallParams)
```

⬇

```
  FunctionCallName(CallParams, VariableName)
```

Case

The *case* statement selects one alternative from a list of possible choices for the value of an expression. This list must encompass all the possible expression results, and the conditions defined in each alternative must be mutually exclusive. ATOS converts this statement to a PROMELA *If* statement. Although in PROMELA conditions are chosen nondeterministically if more than one is valid, in this case this is not a problem since all the conditions are by translation mutually exclusive. The problem of a process getting stuck in this statement does not exist either, because all possible values of expression are covered by the conditions.

```
Case Expression is

   When Value1 =>    statements
      ...
   When ValueN =>    statements

End Case
```

```
 if
   :: Expression==Value1 →    statements
   ...
   :: Expression==ValueN →    statements
 fi
```

Loop Statements

Loop statements allow the repetitive execution of statements. Each repetition is called an iteration and its execution may depend or not on a certain condition. Ada provides three different loop primitives: *loop; while loop* and *for loop.* ATOS converts all these primitives into the unique repetition primitive in PROMELA, the *do* primitive. The specific translation details about each of these primitives are given next.

Loop In this statement a sequence of instructions is executed infinitely, without requiring a condition to hold. There is only one way to jump out of a loop, which is by using the *exit* statement.

```
Loop
    statements
End Loop
```

$\Downarrow$

```
do
  ::    statements
od
```

While Loop This primitive executes a sequence of instructions repeatedly, but before each execution a certain condition is tested, and if it is not true the program jumps to the first statement after the loop. As in the *if* statement, a process which executes the *do* instruction may get suspended if none of its conditions are valid. In order to avoid this, ATOS uses the same strategy used in the *if* statement, that is, the addition of an *else* condition.

```
While condition Loop
    statements
End Loop
```

$\Downarrow$

```
do
  :: condition →    statements
  :: else → break
od
```

For Loop This statement is similar to the *while loop* statement, but the number of cycles is determined according to a range of values, where each single value corresponds to a new cycle. A high level mechanism for *for loop* statements is provided in Ada and PROMELA, which only requires a variable and the lower/upper bounds. However, unlike in Ada, where the variable is implicit (it does not need to be declared), in PROMELA the variable declaration is mandatory. Therefore, ATOS declares the variable before the statement's execution, and its type is given according to the range specified in the statement. In order to avoid the redefinition of for loop variables, if two or more *for loop* statements are declared within the same entity (e.g. procedure, function) then they must have different loop variable names.

```
For VarName in Range Loop
    statements
End Loop
```

⬇

```
VarType VarName;

for(VarName: Range){
      statements
}
```

Exit

The *exit* statement allows abandoning a loop statement by jumping to the first subsequent instruction. A loop abandon may be conditional or unconditional, if there is a condition or not to enclose the statement. Therefore, to deal with these two possibilities ATOS has two different translation rules:

- **Unconditional:** there is no condition for the abandoning of a loop. The conversion of unconditional *exit* statement (declared simply as `exit;`) is simply made through the declaration of a *break* statement in PROMELA.
- **Conditional:** in this case the *exit* statement requires a certain condition to be true in order to be executed. ATOS simulates this through an *if* statement in PROMELA, as follows.

```
Exit [LoopName] When  condition
```

⬇

```
if
  :: condition → break
  :: else → skip
fi
```

Goto

This statement corresponds to a jump from the point where it is being executed to another point in the program, with the destination point identified through a label. The *goto* mechanism exists both in Ada and in PROMELA with the same semantics, and the syntax is the same in both, so the translation is direct.

Procedure Call

A procedure call is an instantiation of a declared procedure. This statement is constructed using the procedure name along with the procedure's parameters (if any). According to the restriction presented in 4.4.2, procedure calls are converted according to the following two different rules.

1. **No arrays as parameters** – if a procedure does not have arrays as parameters, this means that it has been converted into an inline already, so instantiating the procedure corresponds simply to instantiating the corresponding inline primitive.

2. **At least an array as parameter** – in this case, the statements declared within the procedure are converted directly to the process which makes the procedure call, i.e., ATOS simulates internally the behavior of an inline without explicitly use this primitive.

Accept

Accept statements are declared in the specification part of a task (as an entry declaration) and identify the interaction points of a task. Unlike the entries declared in protected objects, when an entry is declared in the specification part of task ,the correspondent primitive in the body part is given by an *accept* statement and not by an entry body.

The translation of this statement is done by just mapping its semantics into PROMELA. In PROMELA, processes (tasks) communicate with each other through channels, so an accept statement is mapped simply as an execution point where a task is listening on a channel that will eventually receive a message from another task, where the message contains the sender identification and the parameters of the accept statement.

The communication between tasks follows the *rendezvous* mechanism. The basic principle of rendezvous is that the first party to reach the "rendezvous point" must wait for the other party to make the communication. So when a task calls, an entry is suspended until the communication finishes, i.e., it is suspended until the requested task finishes the sequence of statements within the *accept* statement. A task is also suspended if it reaches a communication point and no caller tasks are present. In this case, the process listening on the channel is suspended, whereas in the first case the caller task suspension is simulated, by making it wait for a message on its process channel.

```
Accept AcceptName (Parameters) do
      statements
End AcceptName
```

```
chan AcceptName [1] = [0] of {Byte, ParameterTypes}
   ParametersDeclaration
AcceptName ? SenderId, Parameters →
        statements
     Processes[SenderId]!AcceptName
```

Selective accept

The *selective accept* statement allows for the nondeterministic selection of one of multiple alternatives if their conditions are valid. These alternatives can be either an *accept, terminate* or *delay* statement. In none of the conditions to select an alternative is valid, the task executing the statement is suspended.

ATOS translates this statement using the *if* primitive of PROMELA, which has exactly the same semantics of the*selective accept* statement in Ada. However, the statements *terminate* and *delay* are not supported by ATOS, so only selection alternatives containing *accept* statements are considered.

```
Select
    [When condition1 =>]    accept statement
    or
    ...
    [When conditionN =>]    accept statement
End Select
```

```
 if
    :: [condition1 →]    accept statement
    ...
    :: [conditionN →]    accept statement
fi
```

Return

In Ada a return statement encloses a subprogram body or an accept body, and if it is within a function body it additionally returns a value. ATOS only allows the use of this statement in the function's body. As was explained in Section 4.4.2, the variable which will contain the return value is passed as parameter, named `RetValue`. So a return statement is nothing more than an assignment to this variable, where the value assigned is the return value. The assignment of the return value is done according to the assignment rules explained in Section 4.9.

```
Return Expression
```

$\Downarrow$

```
RetValue = Expression
```

Entry call

An *entry call* statement has two different targets: 1) an entry declared within a task which corresponds to an *accept* statement in the task body, or 2) an entry declaration belonging to a protected object which corresponds an entry body. In order to simulate these two types of entry calls ATOS performs two distinct translation rules:

1. In this case the entry call is converted to PROMELA by sending a message to the corresponding *accept* statement channel (created in the processing of the *accept statement*) with the *pid* of sender and the parameters to be instantiated. The sender task has to wait then for the reply of the requested task, which occur at the end of the correspondent *accept* statement execution.

```
EntryName (Parameters)
```

$\Downarrow$

```
EntryName! _pid, Parameters
Processes[_pid]?_
```

2. In this case the entry call conversion is similar to a procedure call translation, i.e. in both statements ATOS instantiates the corresponding *inline* declaration.

4.10. Main Program

The notion of *main* program exists in Ada, despite not being identified with a special name as happens in other programming languages (e.g. Java or C). The *main* program in Ada can be either a subprogram or a package.

The main of an Ada program is translated into an active proctype main , which is parameterless. The `main` process contains a sequence of statements in the following order:

1. **Package statements** – A package does not originate a process in the translation performed by ATOS, so its statements are executed in the `main` process before any other process is active;
2. **Process activations** – The processes declared in PROMELA remain suspended until all package statements have been executed. They are then activated by assigning the value *true* to the boolean variable `Start` which is blocking them.
3. **Process instantiations** – If there is any task instantiation declared in the declaration part of the main, they are performed in the `init` process. The process instantiations in PROMELA are performed inside other processes because they are executed through the `run` statement, rather than by a declaration as in Ada.
4. **Main program statements** – Finally, main statements are executed.

```
MainName body is
   declarations
 Begin
   statements
End
```

⬇

```
  declarations

init{
  packages statements
  Start = true
  tasks instantiations
  statements
  }
```

5. Property Specification

A model checker verifies whether a model fulfills a given (set of) property(ies). Hence, the specification of properties is a crucial step in the verification of programs when the objective is to use the model checking technique. These properties arise, essentially, from Software Requirements Specification (SRS), which produces a set of properties necessary for system correctness. However, mapping these properties into a "model checker's language" is not always straightforward, in particular when some of these properties must be specified in temporal logic: most software developers are not familiarized with this logical formalism.

In order to support the specification of properties, and helping in the mapping of source code properties to model properties, ATOS offers high-level mechanisms based on (SPARK-inspired) annotations in the source code, and on the automatic inference of properties from Ada programs. However, and although the annotation and inference mechanisms are very helpful in the property specification process, at least some basic knowledge of logic is still required.

Several difficulties were found while testing the inference and annotation mechanisms, most of them concerning SPIN bugs. For example, the expressions used in assertions (which are one of the ways of stating the correctness of models) can not have occurrences of the $\rightarrow$ connector, otherwise an error in the model is produces. This restriction can be overcome by using an equivalent formula with the connector $\vee$:

$$\phi \;\rightarrow\; \phi \;\equiv\; \neg\, \phi \;\vee\; \phi$$

Other difficulties are related with the fact that annotations are not semantically analyzed by ATOS: non-valid expressions are accepted within the annotations, and errors may occur in the conversion of expressions since is not being performed according to the expression translation rules used in programs. For example, an expression containing the value of an array position whose array has an offset (see Section 4.3) is not correctly translated because the offset will not be included in its conversion to SPIN.

The rest of the chapter contain the technical details of the inference and annotation mechanisms and all the techniques encompassed by these (range checking; temporal properties; invariants; etc).

5.1. Inferred Properties

Inferred properties are the ones extracted without user interference / annotations. The inference mechanism assembles the specification of properties which guarantee:

1. that the range of a variable with a numeric type is not violated (Range Checking), and
2. that deadlock configurations are not possible in the system (Deadlock).

The two enunciated inference mechanisms refer to *safety properties*, specified in SPIN as LTL formulas. The technical details of these two types of inferred properties are given in the remaining of this section.

5.1.1. Range Checking

The numeric types of PROMELA are very primitive, not allowing a precise conversion from the Ada types, as explained in Section 4.2. Although PROMELA is able to detect overflow errors in variables (without the specification of any property), if the range of a variable in PROMELA is bigger than the range of its correspondent variable in Ada, possible overflow errors may stay undetected.

In order to overcome this problem an LTL formula is automatically extracted, checking if the upper and lower bounds of variables are respected. This mechanism is activated when variables have range constraints, or the converted type of a variable has a bigger range of values than its correspondent Ada Type (e.g. integer to short). The generic LTL formula for the verification of variable range is illustrated below. All names for LTL formulas start with RC and are suffixed with a unique number.

$$\mathit{ltl}\ \mathit{RC}0..N\ \{\ [\ ](VarName \geq Var\_LowerBound\ \&\&\ VarName \leq Var\_UpperBound)\}$$

5.1.2. Deadlock

One of the typical problems in concurrent systems is the *deadlock* problem, which occurs when two or more entities are waiting for resources that will never be released by other entities. In order to overcome possible *deadlock* configurations ATOS automatically extracts a property which guarantees its freedom; this property is always the same independently from the program.

The property is specified as an LTL formula which checks deadlock freedom by verifying if none of the processes in the model can execute. The deadlock formula is expressed through a predefined variable of SPIN (the *np_* variable), which contains the number of active processes in a model.

$$\mathit{ltl}\ \mathit{Deadlock}\ \{\ [\ ](np\_ > 0)\}$$

The above formula does not detect when an individual entity is stuck; it just detects when the whole system can not evolve. This formula suffers from a false deadlock detection problem. This situation occurs when the system is supposed to finish, i.e., when at a certain point none of the processes should execute. In this situation the formula is not valid, despite the system being correctly designed.

5.2. Annotations

The annotation language provided by ATOS for the specification of properties in Ada programs is inspired by SPARK. Actually, this annotation language it is an extended subset of SPARK, and uses some of the annotations provided in SPARK (e.g. for the pre-postcondition annotations) and adds a few new ones (such as temporal properties and invariant annotations).

The syntax analysis of annotations is fully performed by ATOS, through a simple implemented in it. This mechanism allows ATOS to extract the individual elements of an annotation and (through string-matching) to perform the correct translation. Thus, ATOS does not perform any semantic analysis of expressions in annotations and so bad elaborated formulas will not be detected. The absence of semantic analysis of expressions does not permit a correct translation of all expressions, i.e., the translation may not be the same performed for the expressions in an Ada program.

Although ATOS does not perform any semantic analysis of expressions, it allows the use of old values in expressions. The old value of a variable contains its value before the beginning of an operation (function, procedure and entry) execution, which is very useful for the post-condition annotations. The old value of a variable is simulated in ATOS through the creation of an auxiliary variable that is assigned with the value of the corresponding variable at the beginning of the Ada operation. This new variable can then be used in the corresponding postcondition assertion as the old value. An old variable is annotated using SPARK notation, as *VarName~*, and it is converted as *VarName_Old*.

In the remaining of this section more details are given about the different types of annotations that may be specified in ATOS: *temporal properties, pre-postconditions, invariant, return, assert* and *special labels*.

5.2.1. Temporal properties

The specification of temporal properties in ATOS is restricted by the temporal logic allowed in SPIN, which is LTL. ATOS offers a high-level mechanism for the specification of temporal properties, based on the properties pattern for LTL defined in [DAC98]. These are composed by five basic patterns:

- **Universal** – the property is true in the execution;

- **Absence** – the property is never true in the execution;
- **Response** – a property which states that a designated state/event must be followed by another event;
- **Existence** – the property is valid at a certain point of the execution;
- **Precedence** – a property which states that an event is preceded by the first occurrence of another event.

The five patterns have variations which are defined in terms of five basic *pattern scopes*:

- **Globally** – a pattern holds *globally* along the program execution;
- **After** – a pattern holds *after* the first execution of a specified event;
- **Before** – a pattern holds *before* the first execution of a specified event;
- **Between** – a pattern holds *between* the occurrence of a designated event and the occurrence of another specified event;
- **After until** – a pattern holds *after* the occurrence of a specified event and *until* the next occurrence of another event, or throughout the rest of the program execution if there are no further occurrences of that event before the end of the program.

The pattern properties specified above can be annotated in Ada programs; these annotations then originate an LTL formula. The syntax of these annotations, as well as the corresponding LTL formulas, are given table 5.1. In addition to these sets of patterns, users can also specify their own temporal properties. ATOS provides automatic operator conversion and variable renaming.

Type	Annotation Syntax		LTL formula
Universality	P **is_true**	**globally**	$[\,]\ (P)$
		before R	$<>\ R \rightarrow (P\ U\ R)$
		after Q	$[\,](Q \rightarrow [\,](P))$
		between Q **andOp** R	$[\,]((Q\ \&\&\ !R\ \&\&\ <>\ R) \rightarrow (P\ U\ R))$
		after Q **until** R	$[\,]\ (Q\ \&\&\ !R \rightarrow (P\ W\ R))$
Absence	P **is_false**	**globally**	$[\,]\ (!P)$
		before R	$<>\ R \rightarrow (!P\ U\ R)$
		after Q	$[\,](Q \rightarrow [\,](!P))$
		between Q **andOp** R	$[\,]((Q\ \&\&\ !R\ \&\&\ <>\ R) \rightarrow (!P\ U\ R))$
		after Q **until** R	$[\,]\ (Q\ \&\&\ !R \rightarrow (!P\ W\ R))$
Existence	P **becomes_true**	**globally**	$<>\ (P)$
		before R	$!R\ W\ (P\ \&\&\ !R)$
		after Q	$[\,](!Q)\ \Vert\ <>(Q\ \&\&\ <>P))$
		between Q **andOp** R	$[\,](Q\ \&\&\ !R \rightarrow (!R\ W\ (P\ \&\&\ !R)))$
		after Q **until** R	$[\,](Q\ \&\&\ !R \rightarrow (!R\ U\ (P\ \&\&\ !R)))$
Response	S **responds_to** P	**globally**	$[\,](P \rightarrow <> S)$
		before R	$<> R \rightarrow (P \rightarrow (!R\ U\ (S\ \&\&\ !R)))\ U\ R$
		after Q	$[\,](Q \rightarrow [\,](P \rightarrow <> S))$
		between Q **andOp** R	$[\,]((Q\ \&\&\ !R\ \&\&\ <>\ R) \rightarrow (P \rightarrow (!R\ U\ (S\ \&\&\ !R)))\ U\ R)$
		after Q **until** R	$[\,](Q\ \&\&\ !R \rightarrow ((P \rightarrow (!R\ U\ (S\ \&\&\ !R)))\ W\ R)$
Precedence	S **precedes_to** P	**globally**	$!P\ W\ S$
		before R	$<> R \rightarrow (!P\ U\ (S\ \Vert\ R))$
		after Q	$[\,]\ !Q\ \Vert\ <>(Q\ \&\&\ (!P\ W\ S))$
		between Q **andOp** R	$[\,]((Q\ \&\&\ !R\ \&\&\ <>\ R) \rightarrow (!P\ U\ (S\ \Vert\ R)))$
		after Q **until** R	$[\,](Q\ \&\&\ !R \rightarrow (!P\ W\ (S\ \Vert\ R)))$

Table 5.1.: Pattern property annotations and their corresponding LTL formulas

5.2.2. Asserts, Preconditions and Postconditions

These annotations allow for the verification of conditions at a certain point during a program execution, and are converted into PROMELA asserts. The expressions within the annotations are not kept and reused to prove other assertions, as in deductive verification.

Assert An annotation corresponding to an **assert** can be specified anywhere in an Ada program where statements are allowed. This annotation allows checking a given expression at a certain point of a program. The syntax of an **assert** annotation can be given as follows

- -# **assert** *Expression*

This is mapped in PROMELA as

assert (*Expression*)

Pre- and postcondition These annotations are defined only in the body of the following Ada primitives: functions, procedures and entries. Precondition statements (asserts) appear at the beginning of the corresponding primitives, while postconditions appear at the end. The syntax of these annotations is given next, as well as their generic translation into PROMELA through an example operation.

```
OperationName is

     annotation of Precondition
  --# pre PreExpression
     annotation of Postcondition
  --# post PostExpression

    Begin

      Statements

end OperationName
```

⬇

```
     Operation begin
  assert(PreExpression)

     statements

  assert(PostExpression)
    Operation end
```

5.2.3. Return

The return annotation stating whether a function return value is correct or not. Functions are translated to inlines primitives with an additional parameter (`RetValue`) that contains its return value, as explained in Section 4.4.2. Therefore, stating if the return value is correct or not amounts to verifying if `RetValue` contains the desired value at end of the function execution. This verification is made through an assert statement at the end of functions, similar a postcondition. The generic annotation of a return property and its correspondent translation are given next:

$$- -\# \textbf{ return } \mathit{Expression}$$

and the translation is

$$\textbf{assert}(RetValue == Expression)$$

5.2.4. Invariants

Invariant properties permit checking if a given logic expression is valid along the execution of one of these Ada primitives: procedures, functions and entries. An invariant annotation is given in the body of the enunciated Ada primitives like preconditions and postconditions, and may be annotated as follows:

$$- -\#\textbf{invariant } \mathit{Expression}$$

ATOS converts the invariant annotations in several LTL formulas, one for each task that possibly executes the Ada primitive. ATOS generates LTL formulas equivalent to this pattern annotation:

$$\mathit{Expression} \textbf{ is\_true between } Q \textbf{ andOp } R$$

where Q/R corresponds to the begin/end states of an operation execution. These states are specified using specific labels for execution states: ATOS needs to create previously two labels (automatically created), one at the beginning and another at the end of the operation, so it can then specify

$$Q \equiv proc@OperationBeg$$

and

$$R \equiv proc@OperationEnd$$

The invariant is verified if all the generated LTL formulas are valid, i.e., if none of the processes which execute violates the invariant expression.

5.3. Special Labels

There are three labels in SPIN that may have a special meaning when verifying models: *end, accept* and *progress.* ATOS converts Ada label statement into PROMELA labels, so it is possible to use these special labels of SPIN as (if they were) special labels of Ada. These labels only have a special meaning in the verification mode, and even in this mode correctness checking involving the labels can be activated/ or disabled individually. There is thus no problem if using them as normal labels. An important detail is that all the labels starting with special label prefixes are considered special. For example, the label *endAll* has the same meaning of the special label *end.*

The use of special labels in an Ada program is not so intuitive: it requires the understanding of some parts of the translation process, and of course, of the particular meaning of each of these labels, so it is not recommended to rookie users. An overview of each special label is given next.

End The *end* label permits a distinction between valid and invalid end states of processes. A process which does not finish is not always incorrect, because it could be meant not to finish execution. With the label *end* it is possible to specify that a process that never finishes execution is not incorrect. This label is commonly used to ensure the absence of deadlock situations in a model, for such the user just needs to indicate the statements where a process may become stuck, and then SPIN will verify if it only these statements that are causing the process to block. An example of an incorrect model, followed by a correct one using the *end* label, is given below.

Listing 5.1: Incorrect model due the absence of *end* label

```
  active proctype B () {

    if
       ::false→ skip;
    fi;
}
```

Listing 5.2: Correct model using *end* label

```
 active proctype B () {
   end:
    if
       ::false→ skip;
    fi;
}
```

Progress SPIN is able to detect non-progress cycles, i.e., loops which are executed infinitely often along the model execution. The non-progressing of cycles could be an error or not, depending on the desired behavior of the model. The label *progress* equips the user with a way to state that a cycle which should be executed infinitely often is not a non-progress cycle. The example below presents two models, one where a non-progress cycle is detected, and another that is correct due to the addition of the *progress* label.

Listing 5.3: Incorrect model due the absence of *progress* label

```
active proctype B () {

  do
      ::skip;
  od;
}
```

Listing 5.4: Correct model using *progress* label

```
active proctype B () {

  do
      ::skip; progress: skip;
  od;
}
```

Accept This label is complementary to the *progress* label, in the sense that in contrast with *progress*, it is meant that a process cannot achieve this label infinitely often. Thus, an *accept* label may be used to detect undesired infinite loops. Two examples of incorrect/correct models are given below, using the *accept* label.

Listing 5.5: Incorrect model using *accept* label

```
active proctype B () {

  do
      ::skip; accept: skip;
  od;
}
```

Listing 5.6: Correct model using *accept* label

```
active proctype B () {

  do
      ::skip; accept: break;
  od;
}
```

6. Experimental Validation

The experimental validation of a tool is very important (almost indispensable) to establish its usefulness in practice, but also to identify its weaknesses. In this chapter, we report on the experimental validation of ATOS, performed through two case studies: *Readers-Writers* and *Separation Kernel*. ATOS was also used in a third case study (*Producers-Consumers*), but due to its similarity with Readers-Writers we include it only in Appendix A.

The Readers-Writers is a small case study, but it contains many of the supported Ada features in ATOS. The use of ATOS in this case study aims primarily to demonstrate the ATOS approach and some of its main features. The Separation Kernel on the other hand is a bigger case study than Readers-Writers, of considerable complexity, and intends to demonstrate the performance and the potential of ATOS in a context that resembles more the industrial setting.

The two Ada programs concerning each of the case studies had been previously developed by people external to ATOS. This is a relevant issue, in order to avoid the typically error-prone process of validating your approach and tools by applying them in the verification of programs that you have implemented yourself. The verification of the models from the two case studies was performed in a machine with 4 Gigabyte (GB) of Random Access Memory (RAM) and two cores running at 2,5 GHertz each. This information is relevant because of the state explosion problem.

The details of the Readers-Writers and Separation Kernel case studies are given in the remaining of the chapter. The details of the case studies are given in a similar manner: first we give an overview of the case study, followed by an overview of its implementation, and then the details of the verification performed with ATOS. The last section of this chapter contains some feedback and conclusions drawn from the ATOS validation process.

6.1. Readers-Writers

The case-study presented in this section is a solution to the *First Readers-Writers* problem, a well known concurrency computing problem: two different threads (readers and writers) try to obtain at the same time a shared data (read and write) with the constraint that these two processes cannot access the data at the same time (cannot read or write at the same time). The solution considered along this section was extracted from [BA06] and covers some of the Ada features that ATOS can translate. An overview of its implementation as well as the verification performed with ATOS are given next.

Implementation

The Ada program which solves the First Readers-Writers problem is composed by: the main `Readers_Writers` procedure; the declaration of a single protected object `RW`; a single `Writer` task; and a task type `Reader`, which is instantiated twice. The main procedure has only one statement (mandatory), which is the *null* statement. The details of the other components will be discussed next.

The protected object RW is composed by four protected operations and two data structures declared in the private part as illustrated in listing 6.1. The variable `Readers` contains the number of readers along the program execution, while the boolean `Writing` signals whether there is a writer writing. The values of these two variables are altered by four protected operations: two entries (`StartWrite` and `StartRead`) and two procedures (`EndWrite` and `EndRead`). The entries have a barrier, which in the case of `StartWrite` means that in order for a writer to start writing there can be no readers reading and no writers writing, while in the case of `StartRead` it indicates that there can be no writers writing in order for readers to start reading.

Listing 6.1: The protected object RW

```
Protected RW is
    procedure EndRead;
    procedure EndWrite;
    entry StartRead;
    entry StartWrite;
 private
    Readers:natural range 0..2:=0;
    Writing:boolean :=false;
 end RW;

Protected body RW is
    procedure EndRead is
      begin
        Readers:=Readers   1;
    end EndRead;

   procedure EndWrite is
      begin
        Writing:=false;
   end EndWrite;

    entry StartWrite when not Writing and Readers = 0 is
      begin
        Writing:=true;
    end StartWrite;

    entry StartRead when not Writing is
      begin
       Readers:=Readers + 1;
    end StartRead;
end RW;
```

The Writer and Reader Tasks The `Writer` task is a single task, while the `Reader` task is a task type, which means that several instantiations of it may be created during the program execution. In our example, as mentioned, it is instantiated twice. Lists 6.2 and 6.3 correspond to the bodies of `Reader` and `Writer` tasks, respectively. The behavior defined by these two body tasks is similar: it is a loop trying to start and end the read/write of the protected object.

Listing 6.2: The body of task Reader

```
task body Reader is
   begin
      loop
        RW.StartRead;
        RW.EndRead;
      end loop;
   end Reader;
```

Listing 6.3: The body of task Writer

```
task body Writer is
   begin
     loop
       RW.StartWrite;
       RW.EndWrite;
    end loop;
  end Writer;
```

Verification

The previously introduced Ada program was verified with ATOS. In order to exemplify the verification performed, we start by enumerating a set of expected requirements (properties) for the Readers-Writers example:

1. The numbers of readers lies between 0 and 2 (the max number of readers).
2. Readers and writers cannot execute simultaneously.
3. Before a reader finishes to read there must be at least one reader reading.
4. After a reader ends to read, the number of readers is decreased in one.
5. While a reader is finishing its execution the writers status must always be *not writing*.
6. The system is deadlock free.

The requirement 1 can be translated at code level, checking if the variable `Readers` does not violate its range constraint. The specification of this property is done automatically by ATOS through the *Range Checking* mechanism, which generates the following LTL formula:

$$ltl\ RC0\ \{\ [\ ](Readers\ \geq\ 0\ \&\&\ Readers\ \leq\ 2)\}$$

The next requirement is the most important and one of the less trivial. This requirement is mapped to source code by stating that along the program execution, the variables `Readers` and `Writing` can not simultaneously be *bigger than 0* and have the *true* value, respectively. This property is annotated in the program with following *absence* pattern property:

$$--\#\textbf{property}\ Readers > 0\ \textbf{and}\ Writing\ \textbf{is\_false globally}$$

This annotation is then automatically converted by ATOS into following the LTL formula:

$$ltl\ prop0\ \{[\ ]\ (\ !(Readers > 0\ \&\&\ Writing)\ )\}$$

The requirements 3 and 4 can be expressed through a precondition and a postcondition annotation, respectively, in procedure `EndRead`. The precondition states that the value of `Readers` must be greater than zero:

$$--\#\textbf{pre}\ Readers > 0,$$

while the post condition checks if the value of `Readers` at the end of the procedure is equal to the value of `Readers` at the beginning of procedure (old value), less one:

$$--\#\textbf{post}\ Readers = Readers \sim -1$$

Requirement 5 can be verified stating that the boolean `Writing` is always false along the procedure `EndRead` execution:

$$--\#\textbf{invariant not}\ Writing$$

ATOS converts the invariant annotations in several LTL formulas, one for each task that possibly executes the Ada primitive. In this example, there are two processes (tasks) which can possibly execute the procedure `EndRead` (the two instantiations of `Reader` task), so ATOS generates two LTL formulas equivalent to this pattern annotation:

$$\textbf{not}\ Writing\ \textbf{is\_true between}\ Q\ \textbf{andOp}\ R$$

where Q/R corresponds to the states whereupon the `Reader` processes begin/end the `EndRead` execution. The invariant is verified if the two LTL formulas are valid.

Results Analysis

As expected, the Ada *Readers-Writers* program respects all the requirements, i.e., the properties corresponding to the requirements were validated by the extracted model. Table 6.1 illustrates some of the information output by SPIN in the properties verification, including memory and time spent in the verification. These are very small, and so the state explosion problem was not a concern.

Requirement	Validation	Memory (Megabytes)	Time (seconds)
1	✓	7,289	0,02
2	✓	6,312	0,02
3,4	✓	7,289	0,02
5[1]	✓	7,484	0,02
6	✓	6,312	0,02

Table 6.1.: Result analysis of Readers-Writers

In order to understand how new readers and writers tasks may affect the program verification, in particular how these new tasks could lead to state explosion, two more Readers-Writers programs were constructed, one with two readers and two writers, and the other with three readers and two writers. In both we tried to verify requirement 6 again, in the first one it took 0,59 seconds and 25,160 Megabytes were spent, whereas in the second 2023,967 Megabytes were spent, and the verification took 192 seconds. It was not possible to verify a third program with one more reader due to memory and time restrictions. It can thus be concluded that in this case (and probably in other cases), due to the arbitrary interleaving execution implemented by SPIN, the number of processes has enormous impact on the resources required for its verification.

6.2. Separation Kernel

The concept of separability was first introduced by Rushby [Rus81]; he stated that computer systems which are completely separate from each other are safe from one another. He also argued that the separation between the components (computer systems) does not have to be necessarily physical, as long as the components are separated at some level of abstraction. The Multiple Independent Levels of Security (MILS) [AFTO04] was proposed a few years later. This is an architecture with a high level of assurance, based on the separation of its components and on controlled information flow, using a separation kernel.

The MILS Message Router (MMR) is one of the two subsystem that implement communication security inMILS. This subsystem is responsible for controlling message flow between the components (and is also known as Partition Information Flow Policy (PIFP)). The Ada program described along the rest of this section, extracted from [ROAf$^+$06], implements an MMR system and it was. In order to test this program (system), it was created a main which simulates the interaction between the MMR components.

[1]The time and memory presented concern to each LTL formula of the two encompassed by the invariant

Implementation

The Ada program that implements MMR (in Appendix B) contains six packages, with all the operations and data structures required for the simulation of a MMR system. The main data structures and operations are illustrated in Figure 6.1 and are briefly described below:

- **Message** – this is a record structure composed by three fields: `origin, destiny` and `data`. The latter is not very relevant for the verification.

- **Memory** – this is represented as an array where each of its positions is a memory cell (or memory space) and it is named in source code as `Mem_Space`. Each memory position contains a message, and belongs to a certain partition. All partitions have the same memory space, which is exactly the number of partitions. Thus, if the system has three partitions then each has three memory cells. In order to read/write messages from memory, partitions can execute the operations (procedures) read/write.

- **Pointers** – pointers indicate which are the memory spaces belonging to each partition at a given moment. This data structure is represented as a square matrix, where each column contain the pointers to the memory space of a certain partition. The value of position *i,j* in the matrix `Pointers` indicates that the memory position corresponding to this value belongs to partition `i`.

- **Mailbox** – this is a data structure designed as a square matrix (named `Flags`), where each column represents the mailbox of a partition. The mailbox is simultaneously an inbox and an outbox, thereby the position `i,j` in the matrix indicates (through a boolean value) either if a partition `i` has received a message from partition `j` or if a partition `i` had sent a message to partition `j` depending on the message that it is pointing out. The message corresponding to the (slot) matrix position `(i,j)` is in the memory cell (an index of `Mem_Space`) whose the position (index) is given by the value of position `(i,j)` from the matrix `Pointers`.

- **Policy** – this is the data structure which contains the partition information flow policy. This structure is designed once more as square matrix, named `Policy_Space`. Each matrix position contains a boolean value, indicating a communication permission, where the position `(i,j)` states if the partition `i` can send information to the partition `j`. The function `Is_Allowed`, given an origin and a destination partition returns a boolean value indicating if the communication is permitted or not. In this particular example, only the partition with the ID equal to 1 can send information to partition with the ID equal 2, thereby all other communications are not allowed.

- **Route** – this is an operation (procedure) responsible for the message flow between partitions. When a partition sends a message to another, it puts it in a memory space (in a memory cell). Then when the kernel looks for new messages in the system, it will see this one, and if the sender partition is allowed to send messages to the recipient partition, the memory space (memory position) which contains the message starts to belong to the recipient partition (by exchanging the pointers).

- **Send Message** – this is the operation that simulates the process of sending messages. Firstly, the message is built, then it is written in memory, and lastly it is placed in the sender partition outbox.
- **Read Message** – this is the operation which simulates the process of reading messages. All the received messages of a partition are initially copied to an auxiliary data structure, then the mailbox is cleared, and lastly the messages are erased from memory.

In order to test the MMR system, a *main* program was created containing two *partitions* tasks (named `user_tasks` in source code) and one *kernel* task, whose behavior is:

- **Partition** – defined by sending and receiving messages to/from other partitions. This behavior is repeated twice by each partition; a message cannot be sent from a partition to itself.
- **Kernel** – for routing messagesm and in parallel ensuring that the information flow policy is respected, by executing the `route` operation. This behavior is repeated twice by the kernel task.

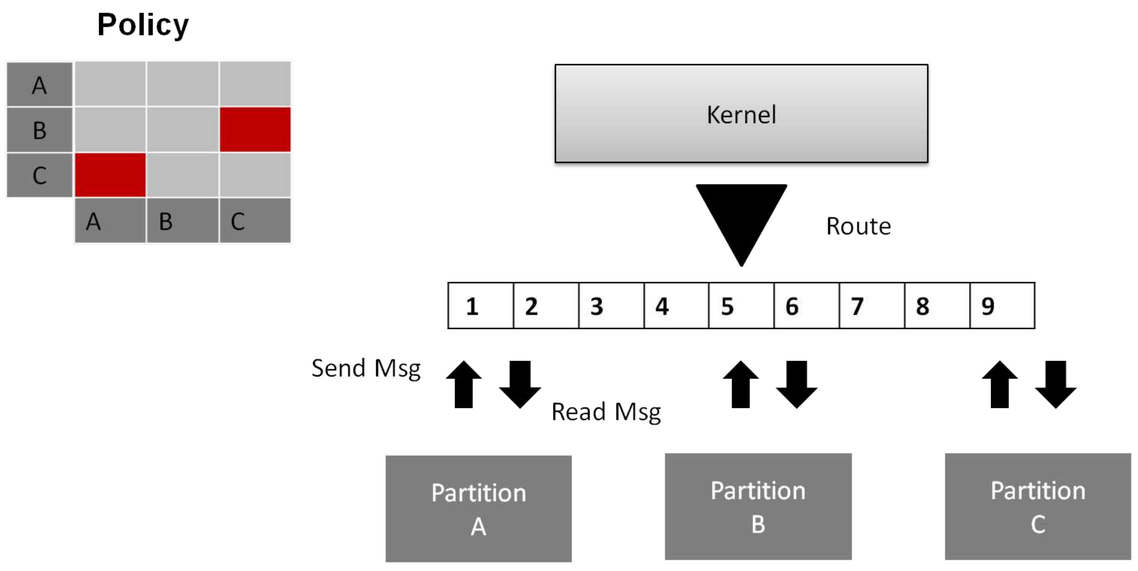


Figure 6.1.: The MMR system

Verification

The main goal of the MMR verification is to assess its correctness according to several functional requirements. Some of the requirements are identified in [ROAf$^+$06], but new requirements were also taken into account in the verification performed with ATOS. These requirements are described next, along with their corresponding annotation in the program:

1. **The information flow policy is immutable** – this requirement can be mapped to the code level by ensuring that all values of the policy matrix are not modified once this data structure has been initialized, i.e., after the execution of policy package statements. The specification of this property through an annotation is intuitive using this pattern property:

   – –#**property** *PolicyIsImmutable* **is_true after** *PolicyStatementsEnd*

   The complexity of this property specification concerns the specification of expressions *PolicyIsImmutable* and *PolicyStatementsEnd*. The expression *PolicyIsImmutable* is built specifying the values for `policy` matrix, which can be made looking for the package `policy`. In this case, as was said previously, only one communication is allowed, so the expression is specified as follows:

   PolicyIsImmutable $\equiv$ $Policy\_Space(1)(2)$ **and not** $Policy\_Space(x)(y)$

   where $\{x, y\} \in N$ and $x \neq 1 \wedge y \neq 2$, with N containing the number of partitions (in this case contains only two ID's 1 and 2). The specification of expression *PolicyStatementsEnd* is trickier, because it requires the identification of a point in the program (the end of the policy statements). To identify a certain point in an Ada program the primitive Label can be used, so first it is necessary to add the label `PolicyEnd` at the end of the policy statements. However, in SPIN it is not possible to refer a label point; it is only permitted referencing a process in a label. Thus, it is required to indicate the process that has arrived at that label. As such, and because all package statements are executed in the `main`, the expression *PolicyStatementsEnd* is specified as follows:

   PolicyStatementsEnd $\equiv$ $main@PolicyEnd$

2. **The memory space allocated to each partition is not shared among the other partitions** – this requirement is translated at the code level ensuring that the matrix `pointers` have different values for all their positions. The expression which states this

requirement is:

NoRepPointers $\equiv$ $Pointers(x)(y)\ /= Pointers(z)(w)$

where $\{x, y, z, w\} \in N$ and $x \neq z \vee y \neq w$, with N containing the partition's ID's. This property should only be valid after the initialization of pointers, and while the pointers are not being swapped by the *route* operation. Thus, firstly it is important to ensure that *route* performs the swapping of pointers well, so the *NoRepPointers* expression is annotated as a precondition and postcondition of this procedure:

$--\#$**pre** $NoRepPointers$

$--\#$**post** $NoRepPointers$

These pre- and postconditions may also be annotated in the procedures `read_msg` and `send_msg`, to state that these processes do not affect the pointers matrix.

3. **The memory space of a partition can only contain messages whose sender is authorized** – the mapping of this requirement to code level is performed stating that if a partition x has received a message (i.e., there is a message in its memory space, which contains its ID in the destination field) from a partition y, then this is because the position *(x,y)* in the policy matrix contains the value true (i.e., the message is allowed to be there). This property should be valid along the whole program execution, and is therefore is annotated as:

   $--\#$**property** $(Flags(x)(y)$ **and** $Mem\_Space(Pointers(x)(y)).Dest == x$
   $--\#$ **and** $Mem\_Space(Pointers(x)(y)).Dest == y) \rightarrow$
   $--\#Policy\_Space(y)(x)$ **is_true globally**

   where $\{x, y\} \in N$ and N is the set of partition ID's.

4. **Inbox well formed** – This property ensures that the inbox of partitions is correctly formed. In order to verify this property two expressions are used, one for the slots with true value and another for the empty ones. Thus, if a given slot of a mailbox is flagged (i.e., if a position `(x,y)` of matrix `Flags` contains the value *true*) then the correspondent message for that slot (which would be in `Mem_Space(Pointers(x,y))`) must have the correct destination and origin values (i.e., the `origin` field must be `y` and `destination`

must be `x`). This may be expressed as:

$$\textbf{SlotsWithMessages} \equiv \begin{array}{l} \mathit{Flags}(x)(y) \;\rightarrow\; (\mathit{Mem\_Space}(\mathit{Pointers}(x)(y)).\mathit{Dest} = x \\ \textbf{and } \mathit{Mem\_Space}(\mathit{Pointers}(x)(y)).\mathit{Origin} = y) \end{array}$$

The empty slots of mailbox (i.e., the positions of matrix `Flags` which contain the value *false*) must point to a default message. This condition is expressed as:

$$\textbf{EmptySlots} \equiv (\textbf{not } \mathit{Flags}(x)(y)) \;\rightarrow\; \mathit{Mem\_Space}(\mathit{Pointers}(x)(y)) \;=\; \mathit{DefMsg}$$

where $\{x, y\} \in N$, with N containing the partitions ID's. Thus, a single expression is built to guarantee the **Inbox well formed** property, which results from the conjunction of the two previous expressions:

$$\textbf{InboxWellFormed} \equiv (\mathit{SlotsWithMessages}) \textbf{ and } (\mathit{EmptySlots})$$

This property should then be annotated as a precondition of `Read_Msg` because a partition should not read sent messages or incorrect messages. The property is also annotated as a postcondition of the `Route` operation, to ensure that the routing is well performed.

5. **Outbox well formed** – this is very similar to **inbox well formed**; the difference is that rather then verifying if the inbox is correct it verifies if the outbox is well formed. Thus, the slots with messages would be tested with the following expression:

$$\textbf{OutSlotsWithMessage} \equiv \begin{array}{l} \mathit{Flags}(x)(y) \;\rightarrow \\ (\mathit{Memory\_Space}(\mathit{Pointers}(x)(y).\mathit{Dest} = y \\ \;\textbf{and } \mathit{Memory\_Space}(\mathit{Pointers}(x)(y).\mathit{Origin} = x) \end{array}$$

while the empty slots are verified with the **EmptySlots** expression used in the previous property. The expression corresponding to this requirement is then:

$$\textbf{OutboxWellFormed} \equiv (\mathit{OutSlotsWithMessage}) \textbf{ and } (\mathit{EmptySlots})$$

This property is annotated as a precondition of `Route`, to ensure that all messages that will be routed hane not been previously routed.

6. **Messages are correctly sent by partitions** – this property is mapped to the source code level ensuring that the `Send_Msg` procedure has a correct behavior, i.e, that produces the expected results. Thus, there are two different expected results from this operation: 1) the input message is sent correctly; and 2) no extra messages were produced. The first one may be expressed as:

   SendMessageCorrectly $\equiv$ $M.Dest = x$ **and** $M.Origin = y \rightarrow$ $((Memory\_Space(Pointers(y,x)).Dest = x$ **and** $(Memory\_Space(Pointers(y,x)).Origin = y))$ **and** $Flags(y)(x)$

   whereas the second expected result can be expressed as :

   NoExtraMessages $\equiv$ $M.Dest \ / = \ x$ **or** $M.Origin \ / = \ y \ \rightarrow$ $(Memory\_Space(Pointers(y,x)) \ =$ $Memory\_Space \sim (Pointers(y,x))$ **and** $Flags(y)(x) = Flags \sim (y)(x))$

   where `x` and `y` are partition ID's. The conjunction of these two expressions allows for the specification of this property, which is annotated as postcondition of `Send_Msg` procedure.

   $- -\#$**post** $SendMessageCorrectly$ **and** $NoExtraMessages$

7. **Messages are correctly read by partitions** – the reading operation should build an array containing all new received messages, so this requirement may be specified at code level by ensuring the correctness of the outputted array. The array of messages will be built in the input variable `A`, which should contain: the new income messages (**DesiredMessages**); and no unexpected messages (**NoUnexpectedMsg**). This condition may be expressed through the following expressions:

   DesiredMessages $\equiv$ $Flags \sim (x)(y) \rightarrow A(y) \ = \ Mem\_Space \sim (Pointers(x)(y)))$

   and

$$\textbf{NoUnexpectedMsg} \equiv \quad \textbf{not}\ \mathit{Flags} \sim (x)(y) \rightarrow A(y) \ = \ \mathit{DefMsg}$$

where `x` is ID of the partition which is executing the operation, and `y` is the ID of another partition. These expressions are then annotated as `Read_Msg` postconditions, as follows:

$- -\#\textbf{post}\ \mathit{DesiredMessages}\ \textbf{and}\ \mathit{NoUnexpectedMsg}$

8. **Messages are correctly routed** – this requirement can be ensured by guaranteeing that the procedure `Route` has a correct behavior. A correct routing behavior is defined through the verification of three distinct cases: if there is a message and the communication is allowed then the pointer of the memory position where the message is contained is swapped and the inbox of the recipient partition is activated (**AllowedMsg**); if there is a message but the communication is not allowed then the message is erased, as well as the message indication in sender's inbox (**NotAllowedMsg**); all the mailbox slots which were not flagged should not produce unexpected modifications in the memory or in the mailbox of partitions (**NoAlterations**).

$$\begin{aligned}\textbf{AllowedMsg} \equiv\ & \mathit{Flags} \sim (x)(y)\ \textbf{and}\ \mathit{Policy\_Space}(x)(y) \rightarrow \\ & \mathit{Mem\_Space}(\mathit{Pointers}(y)(x)) = \mathit{Mem\_Space} \sim (\mathit{Pointers}\ (x)(y)) \\ & \textbf{and}\ \mathit{Flags}(y)(x))\end{aligned}$$

$$\begin{aligned}\textbf{NotAllowedMsg} \equiv\ & \mathit{Flags} \sim (x)(y)\ \textbf{and}\ \ \textbf{not}\ \mathit{Policy\_Space}(x)(y) \rightarrow \\ & (\mathit{Mem\_Space}(\mathit{Pointers}(y)(x)) = \mathit{DefMsg}\ \textbf{and} \\ & \textbf{not}\ \mathit{Flags}(y)(x))\end{aligned}$$

$$\begin{aligned}\textbf{NoAlterations} \equiv\ & \textbf{not}\ \mathit{Flags} \sim (x)(y) \rightarrow \\ & \mathit{Mem\_Space}(\mathit{Pointers}(y)(x)) = \mathit{DefMsg}\ \textbf{and} \\ & \textbf{not}\ \mathit{Flags}(y)(x))\end{aligned}$$

From the conjunction of these different expected behaviors results a property which can then be annotated as a `Route` postcondition:

$- -\#\textbf{post}\ \mathit{AllowedMsg}\ \textbf{and}\ \mathit{NotAllowedMsg}\ \textbf{and}\ \mathit{NoAlterations}$

9. **The Kernel eventually executes the routing operation** – this requirement is mapped to the code level by stating that the `kernel` task eventually reaches the end of

the `Route` procedure. A pattern property may be annotated specifying this property:

– –#**property** $KernelTask@RouteEnd$ **becomes_true globally**

Beyond ensuring that the kernel task eventually executes the routing operation, this property also ensures that the precondition and postcondition of `Route` are asserted at least once (for e.g., if the precondition of route was incorrect but the model never reaches it, then no errors will be detected in the model). Although SPIN outputs the unreachable code, this kind of error is very common, and can be easily detected with liveness properties similar to this one.

Results Analysis

After the specification phase, these properties were verified against the extracted model. As can be seen in Table 6.2, the requirements 2, 4, 5, 6, 7, 8 are not fulfilled by the input Ada program. An individual analysis of these errors is given next:

- Although the requirement 2 is valid as pre- and postcondition of the `Route` operation, it is not valid as pre- and postcondition of operations `Read_msg` and `Send_msg`. This happens because there is a moment in the route operation where two positions of matrix pointers momentarily have the same value, as is shown in listing 6.4. As such, and because the `Route` operation is not mutually exclusive from operations `Read_msg` and `Send_msg`, this property is not valid.

Listing 6.4: Route operation

```
...
T := Pointers(I)(J);
B := Flags(I)(J);
Pointers(I)(J) := Pointers(J)(I);
Two pointers, pointing for the same memory position
Flags(I)(J) := Flags(J)(I);
Pointers(J)(I) := T;
Flags(J)(I) := B;
...
```

- The properties concerning the requirement 4 and 5 are both invalid for the same reason. The inbox and outbox are represented in the same mailbox (i.e., the matrix `Flags`) so if, for example, a partition sends a message, and after this checks its mailbox (i.e., read the messages in its mailbox) without a route operation being executed in the middle, the message that it had sent will still be in the mailbox, and it will be read. Therefore, for these requirements to be valid it would be necessary that after the send or read operation execution, was performed the route operation, and vice-versa.

- The requirements 6, 7, 8 were not fulfilled, due the fact that the operations `route`, `send_msg` and `read_msg` are not mutually exclusive, which was already the problem for requirement 2. For example, in the postcondition of `route` it cannot be ensured that the expression *NoAlteration* is valid, because after the beginning of this operation a new message may be sent by a partition. Similar errors to this one explain the failure of the attempts to ensure the other two requirements.

Requirement	Validation	Memory (Megabytes)	Time (seconds)
1	✓	2.756,441	322
2	x	2,189	0
3	✓	2.756,195	321
4,5	x	2,195	0
6	x	2,195	0
7	x	2,195	0
8	x	20,066	2,46
9	✓	2,195	0

Table 6.2.: Result analysis of Separation Kernel

In order to solve the problems observed for requirements 2, 6, 7, 8 it was decided to declare the operations `route`, `send_msg` and `read_msg` within a protected object, as can be observed in listing 6.5. Therefore, these operations become now mutually exclusive, which allowed testing if other problems were hidden behind this one. Table 6.3 contains the results of verification performed with this new Ada program. As expected the requirements 4, 5 are still not fulfilled by these program, but all the others are now valid. As was said previously, in order for the input program to respect requirements 4 and 5 it would be necessary to synchronize `route`, `send_msg` and `read_msg` so that after each sending or reading operation a routing operation would be performed, as well as a sending or a reading operation after each routing operation.

Listing 6.5: The specification part of protected object containing the MMR operations

```
protected Operations is
    procedure Route
    procedure Send_Msg(M: in Msg)
    procedure Read_Msgs(P: in Proc_Id; A: out Mem_Row)
end Operations
```

Requirement	Validation	Memory (Megabytes)	Time (seconds)
1	✓	207,773	125
2	✓	13,281	1,16
3	✓	207,773	125
4,5	x	2,539	0
6	✓	13,281	1,16
7	✓	13,281	1,16
8	✓	13,281	1,16
9	✓	2,539	0

Table 6.3.: Result analysis of Separation Kernel with a protected object

An interesting error in Route procedure A bug in ATOS (which has already been corrected) forced the verification of a counter-example provided by SPIN for requirement 8. Although this was not a problem of the input program, but a problem in its translation, this error could be contained in the program. For instance, imagine that the piece of code from the route operation presented in listing 6.6 is replaced by the one in listing 6.7.

Listing 6.6: Route operation - original code

```
for I in Lbl_t.Proc_ID loop
        for J in Lbl_t.Proc_ID range I .. Lbl_t.Proc_ID'Last loop
```

Listing 6.7: Route operation - error code

```
for I in Lbl_t.Proc_ID loop
        for J in Lbl_t.Proc_ID loop
```

Thus, after the analysis of the counter-example given by SPIN, it was realized that the route operation was badly conceived. The problem is that in fact the route operation might erase allowed messages which should be routed. The counter-example is defined step-by-step as follows:

1. The partition with the ID 1 sends a message to the partition with ID 2, and therefore the position (1,2) of `Flags` become true and message is placed in `Mem_Space(Pointers(1)(2))`.
2. Next, the kernel starts the routing operation, which iterates over all positions of `Flags` checking for new messages. The iterations over the mailbox reserved for the messages sent from a partition to itself will be omitted. Thus, there are only two iterations possible: the first is when $I = 1$ and $J = 2$ and second is for $I = 2$ and $J = 1$ (The second iteration is not possible in the original program). In the first iteration (see listing 6.8) the message which was sent by partition 1 is routed to the mailbox of partition 2;

however, this message is erased in the second iteration and consequently lost (see listing 6.9).

Listing 6.8: Route operation - first iteration

```
    I=1 and J=2
  ...
                the condition is false
               if not Policy.Is_Allowed(1,2) then
                  Memory.Write(Msg_t.Def_Msg, Pointers(1)(2));
                  Flags(1)(2) := FALSE;
              end if;

                the condition is true
              if not Policy.Is_Allowed(2,1) then
                  Memory.Write(Msg_t.Def_Msg, Pointers(2)(1));
                  Flags(2)(1) := FALSE;
              end if;

                the condition is true
              if Flags(1)(2) or Flags(2)(1) then
                  T := Pointers(1)(2);
                  B := Flags(1)(2);
                  Pointers(1)(2) := Pointers(2)(1);
                  Flags(1)(2) := Flags(2)(1);
                  Pointers(2)(1) := T;
                  Flags(2)(1) := B;
              end if;
               the message has been routed
  ...
```

Listing 6.9: Route operation - second iteration

```
  I=2 and J=1
...
                  the condition is true and the message
                  which was routed is erased
                 if not Policy.Is_Allowed(2,1) then
                   Memory.Write(Msg_t.Def_Msg, Pointers(2)(1));
                   Flags(2)(1) := FALSE;
                end if;

                  the condition is false
                if not Policy.Is_Allowed(1,2) then
                   Memory.Write(Msg_t.Def_Msg, Pointers(1)(2));
                   Flags(1)(2) := FALSE;
                end if;

                  the condition is false
                if Flags(2)(1) or Flags(1)(2) then
                   T := Pointers(2)(1);
                   B := Flags(2)(1);
                   Pointers(2)(1) := Pointers(1)(2);
                   Flags(2)(1) := Flags(1)(2);
                   Pointers(1)(2) := T;
                   Flags(1)(2) := B;
                end if;
                ...
```

6.3. An overview of the experimental validation process

This section presents the most interesting conclusions from the experimental validation of ATOS. In particular we give feedback on how the typical software model checking problems affect the verification, and also a small comparison between the verification that ATOS and Bakar Kiasan have performed, using the separation kernel example.

How do the typical software model checking problems affected the verification with ATOS? With the mechanisms for extracting models and help in the property specification, the model construction and property specification problems almost disappeared. Still missing is a deeper semantic analysis of expressions contained in annotations, which forced some of the expressions to be expressed directly in SPIN.

The state explosion problem manifested in both examples. Although in the first example (*Readers-Writers*) it was forced on purpose, in the second example verifying the corresponding

Ada program with three partitions was already not possible. In the attempt to verify this example a machine with 24 GB of RAM was used, but verification was not possible due to memory and time restrictions. A verification with *collapse compression* was attempted, but this technique did not make the verification possible. We then tried to use *minimized automata*, which reduces considerably the memory spent in the verification, but after running for two weeks the verification had still not finished.

The output interpretation problem did not cause many problems during the verification of the two examples. The *Readers-Writers* is a small example, but the *Separation Kernel* already has considerable complexity, and even in the second example the mapping of counter-examples given by SPIN was not so difficult as could be expected.

A simple comparison between ATOS and Bakar Kiasan using the Separation Kernel example Bakar Kiasan performs the verification of SPARK programs, using the symbolic execution technique, aiming to scale SPARK for "big" industrial projects. After inspecting the verification of Separation Kernel performed in Bakar Kiasan [BHR+11] based on the SPARK annotations, and the verification performed with ATOS, several differences stand out.

One of the main differences is the expressiveness provided by the temporal logic used in ATOS, which allows the specification of properties not for particular program points as SPARK annotations do, but for regions of code (regions of programs). An example of this difference may be observed in the property specified to ensure the requirement 1 of Separation Kernel presented in Section 6.2. ATOS specifies this property ensuring that the partition information policy is never modified once it has been initialized, whereas in Bakar Kiasan this property is specified stating that in the beginning (precondition) and in the end (postcondition) of the principal operations (`Read_Msg`, `Send_Msg` and `Route`) the partition information policy remains unaltered. Both properties seem correct in order to fulfill the requirement 1, however it seems also correct to state that the property specified in ATOS is simpler and more efficient than the one specified with SPARK based annotations. Another difference is the fact that in Bakar Kiasan a sequential version of Separation Kernel is verified (SPARK is sequential), whereas in ATOS is verified a concurrent one, which, as was amply demonstrated, brings new challenges.

7. Conclusions and Future Work

In this chapter we present the conclusions concerning the goals and achievements of this thesis, followed by the limitations of the work developed, and ideas for future work.

7.1. Conclusions

In this section we present a summary of the work developed in this thesis, and some of the results obtained. A comparison with similar tools to the one developed in this thesis is also made.

The main goal of this thesis was the development of an artifact that would enable the formal verification of Ada programs, having the verification of critical systems as main background. The formal technique chosen to support the formal verification was the model checking technique, a completely automated formal technique. A tool named ATOS was developed to support the application of the model checking technique to Ada programs. ATOS tries to bridge the gap in the application of model checking to software by automating the model construction from Ada programs, and by extracting properties either by inferring them, or by extracting them from annotations contained in the programs.

The direct extraction of PROMELA models from a subset of Ada programs is performed through the semantic mapping of Ada's declarations, statements, and expressions, to similar semantic primitives in PROMELA. In spite of the semantic differences between a programming language (Ada) and a modeling language (PROMELA), this approach can be considered well-succeeded. The approach imposes a few restrictions, as is the case with functions, but in general all the Ada features covered have a complete correspondent PROMELA feature. This is the case in particular for the main concurrency primitives (tasks, protected objects and shared variables), which are all completely covered by ATOS. The model extraction process executed by ATOS tries to be as rigorous as possible, in the attempt to generate models that are closely related with the corresponding Ada programs. As an example of this rigor one can highlight the distinction between by-pass and by-reference parameters.

The specification of properties is made through annotations and complemented by the inference of safety properties, such as range checking of numeric variables and freedom of deadlock. The annotations are inspired by the SPARK annotation language, encompassing a few similar syntactic and semantic annotations like pre- and postconditions, but also containing new annotations like the specification of operation invariants, or the ones related to the construction of temporal properties. The use of the special labels mechanism proposed by ATOS is not so straightforward, since it requires not only a deep understanding of their meaning, but also of the context where the labels are used. In general, our evaluation is that the mechanisms provided by ATOS to support the specification of properties seem quite reasonable and helpful.

The validation of ATOS was performed through the case-studies *Readers-Writers* and *Separation Kernel*. The use of ATOS in these two case studies aimed primarily to improve the confidence on the tool itself, and secondly to confirm in practice the potential of ATOS in the verification of Ada programs. The use of ATOS in *Readers-Writers* and *Separation Kernel* have also revealed the simplicity and efficiency of properties specified in temporal logic, when compared for example with the logic used in SPARK, essentially due to the capability to state properties for a region of code, rather than a single point in the source code. Particularly in the separation kernel example, the results were very interesting, firstly because the example has approximately 500 Lines Of Code (LOC) which is already a significant complexity, and secondly due to the capacity demonstrated to find concurrent system problems.

According to a published study [WLBF09], the complexity of *Separation Kernel* is considerable (perhaps not so different from many critical systems). This study reports that the majority of industrial projects where formal analysis was applied have between 1.000 and 1.000.000 of LOC. So it can be concluded that the application of ATOS in the industrial context is at least partially demonstrated by this example. *Separation Kernel* has also revealed that the implementation of MMR, when tested in a concurrent configuration, has in fact many errors that are detected by ATOS, thereby reinforcing our confidence in its adequacy for the verification of concurrent Ada programs.

A comparison between three model checkers which support program verification for Ada
We now turn to a comparison between ATOS, Quasar and Ada Translating Toolset, to justify the development of a new software model checking tool (ATOS), when there were already two existing tools (Quasar and Ada Translating Toolset). The main reason is the fact that from the work developed in Quasar and Ada Translating Toolset, all that remains are a few papers, i.e, the corresponding tools are no longer available. This fact also prevents a deeper comparison between these three tools. Nevertheless, we would like to point out some differences, starting by a small comparison between the underlying approaches of each.

The three tools use other model checker(s) to perform the verification of the extracted models: ATOS uses SPIN, while Quasar has Helena (developed as an autonomous tool in the context of the Quasar project) and Ada Translating Toolset exports models to both SPIN and SMV. However, the extraction processes of these tools are completely different. While Quasar generates models for transition systems based on Petri nets, the other two tools use modeling languages to describe transition systems based on automata theory.

The extraction of models from Ada programs is made directly in the case of Quasar and ATOS, whereas in Ada Translating Toolset an intermediate representation is used between the programs and the models. In the latter, although the extraction process is known in general, no further details could be found (not even a single complete example), thereby making the task of analyzing in detail its extraction process very hard. The approaches followed by ATOS and Quasar, have some common properties, in particular the translation rules used in both tools may be valid for other programming languages than Ada, because declarations like procedures or functions and statements such as if, loop or case are common to almost all programming languages. In terms of the supported subset, it is hard to compare these three tools because of the scarce information concerning this matter for both Quasar and Ada Translating Toolset.

The specification of properties is arguably easier and more intuitive in Quasar and ATOS than in Ada Translating Toolset, which has no high-level mechanisms to tackle the property specification problem. Quasar proposes several templates of LTL formulas, thereby easily enabling the specification of the properties which are encompassed by these templates, but does not propose solutions for the ones that are outside of this scope. By the latter we are not only referring to different required LTL formulas, but also to other elements like assertions. With the pattern annotations mechanism, ATOS proposes a high-level mechanism for the specification of LTL formulas which is complemented by the other forms of annotations, enabling a wide variety of mechanisms to specify properties at code level.

In terms of usability no comparisons can be made because neither Quasar nor Ada Translating Toolset are available. However, it can be said that only ATOS and Quasar have a GUI which, theoretically, makes them more usable than Ada Translating Toolset.

7.2. Limitations and Future Work

Let us now consider the limitations of this work along with possible solutions for some of them, to be implemented in future work.

A parser for annotations ATOS parses annotations in the code through a very basic mechanism, which does not perform any semantic analysis of the expressions present within an annotation. This incapacity implied that some expressions could not be correctly translated to SPIN according to the conversion performed for models. In order to solve this problem, a parser for the annotation language would be required. A preliminary study of this matter has already been carried out, and the use of Another Tool for Language Recognition (ANTLR) [Par07] seems to be a good solution for this parser implementation.

Objects quantification SPIN does not allow the use of quantifiers in formulas. Thus, for stating simple facts like "all positions of an array have a positive value" which could be simply expressed as

$$\forall i,\ MinArr \leq\ i\ < MaxArr \rightarrow Arr(i) > 0$$

one has to explicitly state this property for all of the array's positions as follows

$$Arr(MinArr) \textbf{ and ... and } Arr(MaxArr)$$

The quantification of objects (e.g. quantifying over tasks or data types) would enable a simple and less error-prone property specification process. In fact, what we are proposing is an extension of the SPIN propositional logic with first order quantifications, which would be supported by ATOS at the translation level. This mechanism is a primary objective for future work.

The absence of an Ada scheduler Concurrent Ada programs may be executed on a single processor (interleaved) or on a multiprocessor. Ada proposes several different priority scheduling policies, which define the order by which the tasks are executed when they are competing for a resource (a processor). The concurrency model of SPIN is different from Ada's: the behavior of SPIN models is defined simply by arbitrarily interleaving the processes' statements. In an abstract way it is possible to state that all the configurations defined by Ada's priority scheduling policies are represented in the single "scheduling policy" of SPIN. Thus, the absence of a scheduler in SPIN means that more configurations are being tested than those which could in fact occur in an Ada program. Therefore, false-positives may be detected, i.e., some error configurations may be detected in SPIN that do not occur in the Ada program being studied.

A preliminary study of a scheduler implementation in SPIN was already made. In our sketch, the scheduler would be a PROMELA process with two queues associated, one for the ready tasks and one for the non-ready. The scheduler process would control the execution tasks through a condition placed in the *provided* expression of processes (recall that a process can only execute if the expression within the *provided* primitive is true). All processes would have the same condition, that would be

$$Execute == \_pid,$$

where the `Execute` variable contain the *id* of the selected task (process) to execute, and this selection would be controlled by the scheduler process. Lastly, the task dispatching points (the point where a task become blocked) would be identified through the *timeout* primitive which is only executed when no other process can execute in a PROMELA model. Nevertheless, this is just a sketch and unforeseen problems may prevent its implementation.

Even though representing all scheduling policies in SPIN is something that seems impossible to achieve, the implementation of a few policies seems not only possible but also advisable, to improve the confidence in the tool and the associated verification technique. With the implementation of the presented scheduler draft, we think is possible to represent the *non-preemptive* and *preemptive* dispatching policies in SPIN. However, this is just a sketch, thereby these or other scheduler policies still remain as future work.

Extending the Ada subset covered by ATOS The semantic gap between Ada and PROMELA is huge, so we knew in advance that it would be almost impossible to cover all the Ada features in the context of this work. Nevertheless, the Ada subset covered by ATOS is in our opinion already reasonable for our goals, since most critical systems typically use only a small subset of Ada. However, there are still some Ada features that it would be interesting to support (e.g. enumeration types), so extending the Ada subset covered by ATOS, while not a main priority, is an interesting goal for the future.

Addressing the state explosion and output interpretation problems These problems, which could be foreseen, manifested in practice in the case studies of ATOS. In order to enable the verification of more complex models, and consequently more complex programs, one would have to introduce additional abstraction mechanisms. The abstraction techniques provided by SPIN are already helpful for verifying large models; however, the models generated from software systems tend to have a huge amount of states, which would require abstraction techniques capable of reducing their complexity. One of the commonly used techniques is *program slicing*[1] (presented in Section 2.4.4), which slices all the parts of a program that do not influence the satisfaction of a certain property. The implementation of this or other abstraction techniques would require an in-depth study of the research fields related with these type of techniques.

The mapping of the error traces provided by the model checker back to the code is not always obvious, and is often prone to errors. In our experience with ATOS this task did not raise so many problems as we anticipated. Nevertheless, with more complex programs than the ones tested in ATOS so far, the mapping problem will certainly raise much more difficulties, thereby it is important, as future work, the implementation of artifacts to tackle this problem.

[1]Program slicing may have other meanings in different contexts

Bibliography

[ACD93] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. Information and Computation, 104:2–34, 1993.

[AD03] Peter Amey and Brian Dobbing. High integrity ravenscar. In 8th International Conference on Reliable Software Technologies – Ada-Europe 2003 (AE03), 2003.

[AFTO04] Jim Alves-Foss, Carol Taylor, and Paul Oman. A multi-layered approach to security in high assurance systems. Hawaii International Conference on System Sciences, 9:90302b, 2004.

[BA06] Mordechai Ben-Ari. Principles of Concurrent and Distributed Programming (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[BA09] Mordechai Ben-Ari. Ada for Software Engineers - second edition with Ada 2005 (2. ed.). Springer, 2009.

[Bar03] John Barnes. High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Bar08] John Barnes. Safe and Secure Software: An invitation to Ada 2005. AdaCore, 2008.

[BCC$^+$03] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joeseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In Thomas Arts and Wan Fokkink, editors, Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03), volume 80 of Electronic Notes in Theoretical Computer Science (ENTCS), pages 73–89. Elsevier, June 2003.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[BDV04] Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems. Ada Lett., XXIV(2):1–74, 2004.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast: Applications to software engineering. Int. J. Softw. Tools Technol. Transf., 9:505–525, October 2007.

[BHR$^+$11] Jason Belt, John Hatcliff, Robby Robby, Patrice Chalin, David Hardin, and Xianghua Deng. Bakar kiasan: flexible contract checking for critical systems using symbolic execution. In Proceedings of the Third international conference on NASA Formal methods, NFM'11, pages 58–72, Berlin, Heidelberg, 2011. Springer-Verlag.

[BK08] Christel Baier and Joost-Pieter Katoen. Principles of model checking. MIT Press, 2008.

[BM83] Robert S. Boyer and J Strother Moore. Proof-checking, theorem-proving and program verification. Technical report, 1983.

[BPP99] Eric Bruneton and Jean-François Pradat-Peyre. Automatic verification of concurrent ada programs. In Proceedings of the 1999 Ada-Europe International Conference on Reliable Software Technologies, Ada-Europe '99, pages 146–157, London, UK, 1999. Springer-Verlag.

[BSB91] James B. Bladen, David Spenhoff, and Steven J. Blake. Ada semantic interface specification (asis). In Proceedings of the conference on TRI-Ada '91: today's accomplishments; tomorrow's expectations, TRI-Ada '91, pages 6–15, New York, NY, USA, 1991. ACM.

[CCGR00] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. International Journal on Software Tools for Technology Transfer, 2:2000, 2000.

[Cla08] Edmund M. Clarke. The birth of model checking. In 25 Years of Model Checking, pages 1–26, 2008.

[Cor93] James C. Corbett. An sedl translator. Technical report, 1993.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to Algorithms. McGraw-Hill Higher Education, 2nd edition, 2001.

[DAC98] Matthew Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In Proceedings of the Second Workshop on Formal Methods in Software Practice, pages 7–15. ACM Press, 1998.

[DIS99] Claudio Demartini, Radu Iosif, and Riccardo Sisto. dspin: A dynamic extension of spin. In SPIN, pages 261–276, 1999.

[DJE01] Jörg Desel, Gabriel Juhás, and Katholische Universität Eichstätt. What is a petri net? informal answers for the informed reader. In Unifying Petri Nets, LNCS 2128, pages 1–27. Springer, 2001.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 27(7):1165–1178, July 2008.

[DPC98] Matthew B. Dwyer, Corina S. Pasareanu, and James C. Corbett. Translating ada programs for model checking : A tutorial. Technical report, 1998.

[EC82] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. Sci. Comput. Program., 2(3):241–266, 1982.

[EKPPR03a] S. Evangelista, C. Kaiser, J. F. Pradat-Peyre, and P. Rousseau. Verifying linear time temporal logic properties of concurrent ada programs with quasar. Ada Lett., XXIV:17–24, December 2003.

[EKPPR03b] Sami Evangelista, Claude Kaiser, Jean-François Pradat-Peyre, and Pierre Rousseau. Quasar: a new tool for concurrent ada programs analysis. In Proceedings of the 8th Ada-Europe international conference on Reliable software technologies, pages 168–181, Berlin, Heidelberg, 2003. Springer-Verlag.

[Eva05] Sami Evangelista. High level petri nets analysis with helena. In ICATPN, pages 455–464. Springer, 2005.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, Mathematical Aspects of Computer Science, volume 19 of Proceedings of Symposia in Applied Mathematics, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[Hav99] Klaus Havelund. Java pathfinder user guide. NASA Ames Research, 1999.

[HM03] Douglas J. Howe and Stephen Michell. An approach to formal verication of real time concurrent ada programs. Ada Lett., XXIII:87–92, September 2003.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576–580, October 1969.

[Hol03] Gerard Holzmann. Spin model checker, the: primer and reference manual. Addison-Wesley Professional, first edition, 2003.

[HR04] Michael Huth and Mark Dermot Ryan. Logic in computer science - modelling and reasoning about systems (2. ed.). Cambridge University Press, 2004.

[HWL$^+$02] Marc Hoy, Dave Wood, Marc Loy, James Elliot, and Robert Eckstein. Java Swing. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2 edition, 2002.

[Jen95] Kurt Jensen. Coloured Petri nets: basic concepts, analysis methods and practical use, vol. 2. Springer-Verlag, London, UK, 1995.

[JGP99] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. Model Checking. The MIT Press, 1999.

[JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. ACM Comput. Surv., 41:21:1–21:54, October 2009.

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-Checking for Real-Time Systems. In Proc. of Fundamentals of Computation Theory, number 965 in Lecture Notes in Computer Science, pages 62–88, August 1995.

[McM93] Kenneth L. McMillan. Symbolic model checking. Kluwer, 1993.

[Mey92] Bertrand Meyer. Applying "Design by Contract". Computer, 25(10):40–51, October 1992.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In CC '02: Proceedings of the 11th International Conference on Compiler Construction, pages 213–228, London, UK, 2002. Springer-Verlag.

[Par07] Terence Parr. The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[Per90] D. Perrin. Finite automata. In J. van Leeuwen, editor, Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics, pages 1–57. Elsevier, Amsterdam, 1990.

[Pnu77] Amir Pnueli. The temporal logic of programs. In Proceedings of the 18th Annual Symposium on Foundations of Computer Science, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[PR08] Carl Adam Petri and Wolfgang Reisig. Petri net. Scholarpedia, 3(4):6477, 2008.

[ROAf+06] Bryan Rossebo, Paul Oman, Jim Alves-foss, Ryan Blue, and Paul Jaszkowiak. Using spark-ada to model and verify a mils message router. In Proceedings of the International Symposium on Secure Software Engineering, 2006.

[Rus81] John Rushby. The design and verification of secure systems. In Eighth ACM Symposium on Operating System Principles (SOSP), pages 12–21, Asilomar, CA, December 1981. (ACM Operating Systems Review, Vol. 15, No. 5).

[Sie02] Stephen F. Siegel. The inca query language. Technical report, 2002.

[SPA08] SPARK Team. SPARK Examiner: The SPARK Ravenscar Profile, January 2008.

[TDB+07] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, Erhard Ploedereder, and Pascal Leroy. Ada 2005 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1 (Lecture Notes in Computer Science). Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. ACM Comput. Surv., 41:19:1–19:36, October 2009.

A. Producers-Consumers

A.1. Ada Program

Listing A.1: The Ada program of Producers-Consumers

```
Procedure ProducersConsumers is
task Producer;
task type Consumer;



bufferArr: array (0..99) of integer;

protected Buffer is
   entry Consume(cons: out integer);
   entry Produce(prod: in integer);
private
  se o i for 100; iremos ter um problema no array k ira ser detectado
      BufferSize: integer:=99;
      i: integer range 0..99 :=0;
end Buffer;

   protected body Buffer is


   entry Consume(cons:out integer) when i>0 is
    begin
      cons:=bufferArr(i);
      i:= i 1;
   end Consume;

   entry Produce(prod:in integer) when i<BufferSize is
   begin
      bufferArr(i):=prod;
      i:= i+1;
   end Produce;

end Buffer;
Cons1, Cons2:Consumer;
```

```
   task body Consumer is
         consume:integer;
      begin

        loop
         Buffer.Consume(consume);

         end loop;

      end Consumer;
   task body Producer is



         product:integer:=3;
         begin

            loop
            Buffer.Produce(product);

            end loop;
         end Producer;




   begin

   null;

end ProducersConsumers;
```

A.2. PROMELA Model

Listing A.2: The extracted model from the Producers-Consumers program

```
#include "Semaphores.pml"
Semaphore Sema[ 1];
chan Processes[ 3] = [0] of {mtype};

#include "ControlProtectedObjects.pml"
chan End_Operation[ 1] = [0] of {bit};
chan Blocked_Entries[ 1] = [5] of {byte};
chan Active_Blocked_Entries[ 1] = [0] of {byte};
```

```
byte Wait[ 1];
bool Start=false;


/*ProducersConsumers*/

/*                    Declarations                    */
short  ProducersConsumers_bufferArr  [ 100 ]   ;

short  BufferSize    =    100   ;

unsigned  i  :  7    =    0   ;





inline  Consume  (  cons)  {

FirstTime=true;
Acquire ( 0);
TestCondition:  atomic{


if
::   i > 0   → goto  Execute;
:: else → if
::FirstTime→ FirstTime=false;
Blocked_Entries[ 0]!_pid;
Wait[ 0]++;
Release( 0);
goto InsideEgg;
::!FirstTime → End_Operation[ 0]!0;
goto InsideEgg;
fi;
fi;
}

InsideEgg:
atomic{

Active_Blocked_Entries[ 0]?_pid→
goto TestCondition;
```

```
}
Execute:



cons = ProducersConsumers_bufferArr [ i + ( 0 ) ] ;


i = i 1 ;

InfExec:

atomic{
if
::FirstTime→ End_Operation[ 0]!1;
::!FirstTime → Wait[ 0] ;
End_Operation[ 0]!1;
fi;
FirstTime=true;
}
}



inline Produce ( prod) {

FirstTime=true;
Acquire ( 0);
TestCondition: atomic{


if
:: i < BufferSize → goto Execute;
:: else → if
::FirstTime→ FirstTime=false;
Blocked_Entries[ 0]!_pid;
Wait[ 0]++;
Release( 0);
goto InsideEgg;
::!FirstTime → End_Operation[ 0]!0;
goto InsideEgg;
fi;
```

```
fi;
}

InsideEgg:
atomic{

Active_Blocked_Entries[ 0]?_pid->
goto TestCondition;

}
Execute:



ProducersConsumers_bufferArr [ i + ( 0 ) ] = prod ;


i = i + 1 ;

InfExec:

atomic{
if
::FirstTime-> End_Operation[ 0]!1;
::!FirstTime -> Wait[ 0] ;
End_Operation[ 0]!1;
fi;
FirstTime=true;
}
}




active proctype Buffer ( ) provided (Start) {

ControlProtectedObjects( 0)

}
short Consumer0_consume ;
```

```
proctype  Consumer0(  byte ProcNumber) provided (Start)  {

local byte SenderPid;
local bool FirstTime=true;
/*                  Statements                  */

do
::  Consume (  Consumer0_consume  );
od
}
short Consumer1_consume  ;




proctype  Consumer1(  byte ProcNumber) provided (Start)  {

local byte SenderPid;
local bool FirstTime=true;
/*                  Statements                  */

do
::  Consume (  Consumer1_consume  );
od
}
short Producer_product   =   3  ;




active proctype  Producer  (  ) provided (Start)  {

local byte SenderPid;
local bool FirstTime=true;

/*                  Statements                  */

do
::  Produce (  Producer_product  );
od
}

init {
```

```
Start=true;



run Consumer0( 1);


run Consumer1( 2);

/*                    Statements                    */

skip;

}
```

B. Separation Kernel

B.1. Ada Program

Listing B.1: The specification of package *Lbl_t*

```
 package Lbl_t
is
  subtype Proc_Id is integer range 1 .. 2;

  Mem_Size: constant integer := Proc_Id'Last * Proc_Id'Last;
  subtype Pointer is integer range Proc_Id'First .. Mem_Size;
end Lbl_t;
```

Listing B.2: The specification of package *Msg_t*

```
with Lbl_t;use Lbl_t;

package Msg_t
is
    0 represents a null message
   type Data_Range is range 0 .. 2;

   type Msg is
      record
         Data: Data_Range;
         Origin: Proc_Id;
         Dest: Proc_Id;
      end record;

  Def_Msg: constant Msg;

   procedure Set_Origin(aMsg: in out Msg; O: in Proc_Id);

   function Get_Origin(aMsg: in Msg) return Lbl_t.Proc_Id;

   procedure Set_Dest(aMsg: in out Msg; D: in Lbl_t.Proc_Id);

   function Get_Dest(aMsg: in Msg) return Lbl_t.Proc_Id;

   procedure Set_Data(aMsg: in out Msg; B: in Data_Range);
   function Get_Data(aMsg: in Msg) return Data_Range;

   function Is_Default_Message(aMsg: in Msg) return Boolean;

private
   Def_Msg: constant Msg := Msg'(Data => 0,
                                  Origin => Lbl_t.Proc_Id'First,
                                  Dest => Lbl_t.Proc_Id'First);
end Msg_t;
```

Listing B.3: The body of package *Msg_t*

```
package body Msg_t
is
   procedure Set_Origin(aMsg: in out Msg; O: in Lbl_t.Proc_Id) is
   begin
      aMsg.Origin := O;
   end Set_Origin;

   function Get_Origin(aMsg: in Msg) return Lbl_t.Proc_Id is
   begin
      return aMsg.Origin;
   end Get_Origin;


   procedure Set_Dest(aMsg: in out Msg; D: in Lbl_t.Proc_Id) is
   begin
      aMsg.Dest := D;
   end Set_Dest;

   function Get_Dest(aMsg: in Msg) return Lbl_t.Proc_Id is
   begin
      return aMsg.Dest;
   end Get_Dest;


   procedure Set_Data(aMsg: in out Msg; B: in Data_Range) is
   begin
      aMsg.Data := B;
   end Set_Data;

   function Get_Data(aMsg: in Msg) return Data_Range is
   begin
      return aMsg.Data;
   end Get_Data;

   function Is_Default_Message(AMsg: in Msg) return Boolean
   is
     Z: boolean ;
   begin
      Z:=false;
      if AMsg.Data = 0 and AMsg.Origin = Lbl_t.Proc_Id'First and AMsg.Dest =
          Lbl_t.Proc_Id'First
     then
         Z := True;
      end if;
      return Z;
   end Is_Default_Message;
end Msg_t;
```

Listing B.4: The specification of package *Mem_t*

```
with Lbl_t, Msg_t;use Lbl_t, Msg_t;

package Mem_t
is
   pragma elaborate_body(Mem_t);

  type Mem_Row is array (Proc_Id) of Msg;
  Def_Mem_Row:Mem_Row;
end Mem_t;
```

Listing B.5: The body of package *Mem_t*

```
package body Mem_t is

begin

  Def_Mem_Row:= Mem_Row'(others => Msg_t.Def_Msg);

end Mem_t;
```

Listing B.6: The specification of package *Memory*

```
with Lbl_t, Msg_t;use Lbl_t,Msg_t;

package Memory

is
  type Mem_Space_T is array (Proc_Id'First..Mem_Size) of Msg;
  Mem_Space: Mem_Space_T;

  procedure Write( M: in Msg_t.Msg; S: in Lbl_t.Pointer);

  function Read(S: in Lbl_t.Pointer) return Msg_t.Msg;
```

```
end Memory;
```

Listing B.7: The body of package *Memory*

```
package body Memory
is
  procedure Write(M: in Msg_t.Msg; S: in Lbl_t.Pointer) is

  begin
    Mem_Space(S) := M;
  end Write;

  function Read(S: in Lbl_t.Pointer) return Msg_t.Msg is
  begin
    return Mem_Space(S);
  end Read;

begin
  Mem_Space := Mem_Space_T'(others => Def_Msg);
end Memory;
```

Listing B.8: The specification of package *Policy*

```
with Lbl_t;use Lbl_t;

package Policy

is
  type Policy_Row is array (Proc_Id) of boolean;
  type Policy_Matrix is array (Proc_Id'First..Proc_Id'Last) of Policy_Row;
  Policy_Space: Policy_Matrix;

  function Is_Allowed(Origin: in Lbl_t.Proc_Id; Dest: in Lbl_t.Proc_Id)
                      return Boolean;

  function Is_Satisfied
                      return Boolean;

 end Policy;
```

Listing B.9: The body of package *Policy*

```
package body Policy
is
  function Is_Allowed(Origin: in Lbl_t.Proc_Id; Dest: in Lbl_t.Proc_Id)
                      return Boolean
```

```
     is
  begin
    return Policy_Space(Origin)(Dest);
  end Is_Allowed;

  function Is_Satisfied return Boolean is
  begin
    return
    (
     Policy_Space(1)(1) = false and
     Policy_Space(1)(2) = true  and
     Policy_Space(2)(1) = false and
     Policy_Space(2)(2) = false
  );
  end Is_Satisfied;

  begin

   Policy_Space := Policy_Matrix'(others =>
                              (Policy_Row'(others => false)));
    Policy_Space(1)(2) := true;

end Policy;
```

Listing B.10: The specification of package *Mmr*

```
with Msg_t, Lbl_t, Mem_t, Memory, Policy;use Msg_t, Lbl_t, Mem_t, Memory,
    Policy;

package Mmr

is
  type Pointer_Row is array (Proc_Id) of Pointer;
  type Pointer_Matrix is array (Proc_Id) of Pointer_Row;
  Pointers: Pointer_Matrix;

  type Flags_Row is array (Proc_Id) of boolean;
  type Flags_Matrix is array (Proc_Id) of Flags_Row;
  Flags: Flags_Matrix := Flags_Matrix'(others =>
                                  Flags_Row'(others => false));



  procedure Route;

  procedure Send_Msg(M: in Msg_t.Msg);

  procedure Read_Msgs(P: in Lbl_t.Proc_Id; A: out Mem_t.Mem_Row);

```

```
end Mmr;
```

Listing B.11: The body of package *Mmr*

```
    ==================================================================
    Procedure: Fill_Mem_Row

    Retrieve messages for process P and store them in the memory row
    data structure M.  The data structure M acts as a "mailbox" for P
    holding messages destined to P.  When a process Q has not sent a message
    to P (as indicated by Flags), Q's slot in P's row holds a default message
     .

    ==================================================================
procedure Fill_Mem_Row(P: in Lbl_t.Proc_Id; M: out Mem_Row)

is
begin
     initialize mem row to default message (using an aggregate here will
      allow us
     to class M as an "out" only variable
  M := Mem_t.Mem_Row'(others => Def_Msg);
     loop through each (sending) process I
  for L in Proc_Id range Lbl_t.Proc_Id'First .. Lbl_t.Proc_Id'Last loop
    if Flags(P)(L) = True then
        if Flags indicates that P has been sent a message from I,
              then copy that message into T(I) via temp variable M.
      M(L) := Read(Pointers(P)(L));
    end if;
  end loop;
end Fill_Mem_Row;

    ==================================================================
    Procedure: Zero_Flags

    Sets all of P's flags to false

    ==================================================================
procedure Zero_Flags(P: in Proc_Id)

is
begin
  for K in Proc_Id loop
    Flags(P)(K) := false;
  end loop;
end Zero_Flags;
```

```
   ==========================================================================
   Procedure: Zero_Mem_Row

   Clear (write default message to) all the memory slots "owned" by the
   mailbox of P (ownership is indicated by Pointers data structure)

   ==========================================================================
procedure Zero_Mem_Row(P: in Proc_Id)

is

begin
  for I in Proc_Id range Proc_Id'First .. Proc_Id'Last loop
    Write( Def_Msg, Pointers(P)(I));
  end loop;
end Zero_Mem_Row;


procedure Route

is
   T: Pointer;
   B, allow: boolean;
begin
     for I in Proc_Id loop
         for J in Proc_Id range I .. Lbl_t.Proc_Id'Last loop
                allow:= Is_Allowed(I,J);
                if not allow then
                   Write(Def_Msg, Pointers(I)(J));
                   Flags(I)(J) := false;
                end if;
                allow:= Is_Allowed(J,I);
                if not allow then
                   Write(Def_Msg, Pointers(J)(I));
                   Flags(J)(I) := false;
                end if;

                   note: it may not be necessary to swap the flags or even
                    pointers below,
                   but the rest of the code currently depends on Flags being
                    swapped
                if Flags(I)(J) or Flags(J)(I) then
                   T := Pointers(I)(J);
                   B := Flags(I)(J);
                   Pointers(I)(J) := Pointers(J)(I);
                   Flags(I)(J) := Flags(J)(I);
                   Pointers(J)(I) := T;
```

```
                    Flags(J)(I) := B;
                end if;
            end loop;
        end loop;
    end Route;


        Send_Msg procedure is called by System when the system is in the
        "sending" state. The MMR places the
         message in memory and sets the flag to indicate there is a
         message waiting to be delivered. It is the responsibility
         of Route to check to see if the communication is allowed by
         the policy.
    procedure Send_Msg(M: in Msg)

   is
        Origin: Proc_Id;
        Dest: Proc_Id;
    begin
        Origin := Get_Origin(M);
        Dest := Get_Dest(M);
        Write(M, Pointers(Origin)(Dest));
        Flags(Origin)(Dest) := True;
    end Send_Msg;


        Read_Msgs will be called from System for each Proc_Id. Route
         should have already been called so all the messages are
         valid. We just need to transfer P's messages into A and then
         do clean up.
         In the use of Flags and Pointers matrices, the first process
         used in indexing is the destination process (e.g., P), and
         the second index represents the source of the message.
    procedure Read_Msgs(P: in Proc_Id; A: out Mem_Row)
    is
        Temp_Mem_Row : Mem_Row;
    begin
            copy all flagged messages for P into Temp_Mem_Row
        Fill_Mem_Row(P, Temp_Mem_Row);
            clear flags (set to false) associated with P's mailbox
        Zero_Flags(P);
            clear memory "owned" by P's mailbox
        Zero_Mem_Row(P);
            move the result to the output parameter
        A := Temp_Mem_Row;
    end Read_Msgs;
```

```
begin
  Pointers := Pointer_Matrix'(others =>
                                Pointer_Row'(others => 1));

     initialize the Pointers array with the numbers 1 .. Lbl_t.Proc_Id'Last
  for I in Proc_Id range Lbl_t.Proc_Id'First .. Lbl_t.Proc_Id'Last loop
     for J in Proc_Id range Lbl_t.Proc_Id'First .. Lbl_t.Proc_Id'Last loop
        Pointers(I)(J) := J + (Lbl_t.Proc_Id'Last * (I    1));
     end loop;
  end loop;

end Mmr;
```

Listing B.12: The system *main* program

```
 with Lbl_t, Msg_t, Mem_t, Memory, Policy, Mmr; use Lbl_t, Msg_t, Mem_t, Memory,
      Policy, Mmr;

procedure Main is

   task KernelTask;
   task type UserTask(TaskID: Proc_Id);

   UT1 : UserTask(1);
   UT2 : UserTask(2);

   task body KernelTask is
      i: natural range 0..3:= 0;
   begin
      while i < 2 loop
         Route;
         i:=i+1;

      end loop;
   end KernelTask;

   task body UserTask is
      i: natural range 0..3 := 0;
      dest : Proc_Id;
      locMsg : Msg;
      A: Mem_Row;
   begin
      while i < 2 loop

         dest :=  (i mod 2) + 1;
          locMsg.Data:=0;
          locMsg.Origin:=TaskID;
          locMsg.Dest:=dest;
         loop
            if (locMsg.Origin /= locMsg.Dest) then exit;
            else locMsg.Dest := (locMsg.Dest mod 2) +1;
            end if;
         end loop;

          Read_Msgs(TaskID, A);

          Send_Msg(locMsg);

          i:=i+1;
      end loop;
   end UserTask;

begin

   null;
end Main;
```

B.2. PROMELA Model

Listing B.13: Package *Lbl_ t* translation

```
chan Processes[ 3] = [0] of {mtype};
bool Start=false;
/*                                                          Lbl_t
                                                             */

#define   Lbl_t_Mem_Size     4
```

Listing B.14: Package *Msg_ t* translation

```
# include "Lbl_t.pml"
/*                                                            Msg_t
                                                             */

typedef Msg{
unsigned Data : 2   ;

unsigned Origin : 2   ;

unsigned Dest : 2   ;

};
```

```
typedef ConstMsg{
unsigned Data   : 1 = 0;
unsigned Origin   : 1 =  1;
unsigned Dest   : 1 =  1;
};
ConstMsg Def_Msg;








inline  UserTask0_Get_Origin  (  aMsg,  Ret  )  {
/*                    Statements                    */

Ret = aMsg  .Origin  ;
}




inline  UserTask1_Get_Origin  (  aMsg,  Ret  )  {
/*                    Statements                    */

Ret = aMsg  .Origin  ;
}




inline  UserTask0_Get_Dest  (  aMsg,  Ret  )  {
/*                    Statements                    */

Ret = aMsg  .Dest  ;
}




inline  UserTask1_Get_Dest  (  aMsg,  Ret  )  {
/*                    Statements                    */
Ret = aMsg .Dest ;
}
```

Listing B.15: Package *Mem_t* translation

```
#include "Msg_t.pml"
/*                                                    Mem_t
                                                      */

Msg Mem_t_Def_Mem_Row [ 2] ;

inline Mem_t (){
atomic {
Mem_t_Def_Mem_Row [ 0].Data =  Def_Msg .Data;
Mem_t_Def_Mem_Row [ 0].Origin =  Def_Msg .Origin;
Mem_t_Def_Mem_Row [ 0].Dest =  Def_Msg .Dest;
Mem_t_Def_Mem_Row [ 1].Data =  Def_Msg .Data;
Mem_t_Def_Mem_Row [ 1].Origin =  Def_Msg .Origin;
Mem_t_Def_Mem_Row [ 1].Dest =  Def_Msg .Dest;
}

};
```

Listing B.16: Package *Memory* translation

```
#include "Mem_t.pml"
/*                                                    Memory
                                                      */

Msg Memory_Mem_Space [ 4] ;







inline  UserTask0_Write  (  M, S)  {

/*              Statements              */
atomic {
Memory_Mem_Space [ S + (  1  ) ].Data = M .Data;
Memory_Mem_Space [ S + (  1  ) ].Origin = M .Origin;
Memory_Mem_Space [ S + (  1  ) ].Dest = M .Dest;

}
```

```
 }


inline UserTask1_Write ( M, S) {

/*              Statements              */
atomic {
Memory_Mem_Space [ S + ( 1 ) ].Data = M .Data;
Memory_Mem_Space [ S + ( 1 ) ].Origin = M .Origin;
Memory_Mem_Space [ S + ( 1 ) ].Dest = M .Dest;

}
 }


inline KernelTask_Write ( M, S) {

/*              Statements              */
atomic {
Memory_Mem_Space [ S + ( 1 ) ].Data = M .Data;
Memory_Mem_Space [ S + ( 1 ) ].Origin = M .Origin;
Memory_Mem_Space [ S + ( 1 ) ].Dest = M .Dest;

}
 }




inline UserTask0_Read ( S, Ret ) {
/*              Statements              */

atomic {
Ret.Data = Memory_Mem_Space [ S + ( 1 ) ].Data;
Ret.Origin = Memory_Mem_Space [ S + ( 1 ) ].Origin;
Ret.Dest = Memory_Mem_Space [ S + ( 1 ) ].Dest;

}
}
```

```
inline UserTask1_Read ( S, Ret ) {
/*                    Statements                    */

atomic {
Ret.Data = Memory_Mem_Space [ S + ( 1 ) ].Data;
Ret.Origin = Memory_Mem_Space [ S + ( 1 ) ].Origin;
Ret.Dest = Memory_Mem_Space [ S + ( 1 ) ].Dest;

}
}



inline Memory (){
atomic {
Memory_Mem_Space [ 0].Data = Def_Msg .Data;
Memory_Mem_Space [ 0].Origin = Def_Msg .Origin;
Memory_Mem_Space [ 0].Dest = Def_Msg .Dest;
Memory_Mem_Space [ 1].Data = Def_Msg .Data;
Memory_Mem_Space [ 1].Origin = Def_Msg .Origin;
Memory_Mem_Space [ 1].Dest = Def_Msg .Dest;
Memory_Mem_Space [ 2].Data = Def_Msg .Data;
Memory_Mem_Space [ 2].Origin = Def_Msg .Origin;
Memory_Mem_Space [ 2].Dest = Def_Msg .Dest;
Memory_Mem_Space [ 3].Data = Def_Msg .Data;
Memory_Mem_Space [ 3].Origin = Def_Msg .Origin;
Memory_Mem_Space [ 3].Dest = Def_Msg .Dest;
}

};
```

Listing B.17: Package *Mmr* translation

```
# include "Policy.pml"
/*                                                Mmr
                                                  */


typedef SubMmr_PointersType {
short subPointers[ 2];
};
SubMmr_PointersType Mmr_Pointers[ 2]   ;

typedef SubMmr_FlagsType {
bool subFlags[ 2];
};
```

```
SubMmr_FlagsType Mmr_Flags[ 2]   =   false  ;


inline  UserTask0_Zero_Flags  ( P)  {

/*                    Statements                    */


byte K;  for(K: 1.. 2) {



Mmr_Flags [ P + ( 1 ) ].subFlags [ K + ( 1 ) ]  =  false  ;



};
 }


inline  UserTask1_Zero_Flags  ( P)  {

/*                    Statements                    */


byte K;  for(K: 1.. 2) {



Mmr_Flags [ P + ( 1 ) ].subFlags [ K + ( 1 ) ]  =  false  ;



};
 }


inline  UserTask0_Zero_Mem_Row  ( P)  {

/*                    Statements                    */


byte I;  for(I: 1.. 2) {

UserTask0_Write ( Def_Msg  ,  Mmr_Pointers [ P + ( 1 ) ].subPointers [
    I + (  1  ) ]   );

};
 }


inline   UserTask1_Zero_Mem_Row   (  P)  {

/*                   Statements                   */


byte I;  for(I: 1.. 2) {

UserTask1_Write (  Def_Msg  ,  Mmr_Pointers [ P + (  1  ) ].subPointers [
    I + (  1  ) ]   );

};
 }

unsigned KernelTask_Route_T : 3  ;

bool KernelTask_Route_B  ;

bool KernelTask_Route_allow  ;

inline   KernelTask_Route   (  )  {

/*                   Statements                   */
SubMmr_FlagsType Mmr_Flags_Old[ 2] ;
SubMmr_PointersType Mmr_Pointers_Old[ 2]  ;
Msg Memory_Mem_Space_Old[4];
 d_step{
Mmr_Flags_Old[  0].subFlags[0] =Mmr_Flags[  0].subFlags[0];
Mmr_Flags_Old[  0].subFlags[1] =Mmr_Flags[  0].subFlags[1];
Mmr_Flags_Old[  1].subFlags[0] =Mmr_Flags[  1].subFlags[0];
Mmr_Flags_Old[  1].subFlags[1] =Mmr_Flags[  1].subFlags[1];

Mmr_Pointers_Old[0].subPointers[0]=Mmr_Pointers[0].subPointers[0];
Mmr_Pointers_Old[0].subPointers[1]=Mmr_Pointers[0].subPointers[1];
Mmr_Pointers_Old[1].subPointers[0]=Mmr_Pointers[1].subPointers[0];
Mmr_Pointers_Old[1].subPointers[1]=Mmr_Pointers[1].subPointers[1];

Memory_Mem_Space_Old[0].Origin= Memory_Mem_Space[0].Origin;
```

```
Memory_Mem_Space_Old[0].Dest=Memory_Mem_Space[0].Dest;
Memory_Mem_Space_Old[0].Data=Memory_Mem_Space[0].Data;

Memory_Mem_Space_Old[1].Origin= Memory_Mem_Space[1].Origin;
Memory_Mem_Space_Old[1].Dest=Memory_Mem_Space[1].Dest;
Memory_Mem_Space_Old[1].Data=Memory_Mem_Space[1].Data;

Memory_Mem_Space_Old[2].Origin= Memory_Mem_Space[2].Origin;
Memory_Mem_Space_Old[2].Dest=Memory_Mem_Space[2].Dest;
Memory_Mem_Space_Old[2].Data=Memory_Mem_Space[2].Data;

Memory_Mem_Space_Old[3].Origin= Memory_Mem_Space[3].Origin;
Memory_Mem_Space_Old[3].Dest=Memory_Mem_Space[3].Dest;
Memory_Mem_Space_Old[3].Data=Memory_Mem_Space[3].Data;
};



byte I;  for(I: 1.. 2) {




byte J;  for(J: I.. 2) {



KernelTask_Is_Allowed ( I , J , KernelTask_Route_allow );



if
:: ( ! KernelTask_Route_allow ) -> KernelTask_Write ( Def_Msg ,
    Mmr_Pointers [ I + ( 1 ) ].subPointers [ J + ( 1 ) ] );

Mmr_Flags [ I + ( 1 ) ].subFlags [ J + ( 1 ) ] = false ;
::else -> skip;
fi;

KernelTask_Is_Allowed ( J , I , KernelTask_Route_allow );



if
:: ( ! KernelTask_Route_allow ) -> KernelTask_Write ( Def_Msg ,
    Mmr_Pointers [ J + ( 1 ) ].subPointers [ I + ( 1 ) ] );
Mmr_Flags [ J + ( 1 ) ].subFlags [ I + ( 1 ) ] = false ;
::else -> skip;
fi;


if
:: ( Mmr_Flags [ I + ( 1 ) ].subFlags [ J + ( 1 ) ] || Mmr_Flags [ J
    + ( 1 ) ].subFlags [ I + ( 1 ) ] ) ->

KernelTask_Route_T = Mmr_Pointers [ I + ( 1 ) ].subPointers [ J + ( 1
     ) ] ;


KernelTask_Route_B = Mmr_Flags [ I + ( 1 ) ].subFlags [ J + ( 1 ) ]
    ;


Mmr_Pointers [ I + ( 1 ) ].subPointers [ J + ( 1 ) ] = Mmr_Pointers [
     J + ( 1 ) ].subPointers [ I + ( 1 ) ] ;


Mmr_Flags [ I + ( 1 ) ].subFlags [ J + ( 1 ) ] = Mmr_Flags [ J + ( 1
     ) ].subFlags [ I + ( 1 ) ] ;


Mmr_Pointers [ J + ( 1 ) ].subPointers [ I + ( 1 ) ] =
    KernelTask_Route_T ;


Mmr_Flags [ J + ( 1 ) ].subFlags [ I + ( 1 ) ] = KernelTask_Route_B
    ;
::else -> skip;
fi;



};

};


 }
```

```
unsigned UserTask0_Send_Msg_Origin : 2 ;

unsigned UserTask0_Send_Msg_Dest : 2 ;


inline UserTask0_Send_Msg ( M) {

 d_step{
Mmr_Flags_Old[ 0].subFlags[0] =Mmr_Flags[ 0].subFlags[0];
Mmr_Flags_Old[ 0].subFlags[1] =Mmr_Flags[ 0].subFlags[1];
Mmr_Flags_Old[ 1].subFlags[0] =Mmr_Flags[ 1].subFlags[0];
Mmr_Flags_Old[ 1].subFlags[1] =Mmr_Flags[ 1].subFlags[1];



Memory_Mem_Space_Old[0].Origin= Memory_Mem_Space[0].Origin;
Memory_Mem_Space_Old[0].Dest=Memory_Mem_Space[0].Dest;
Memory_Mem_Space_Old[0].Data=Memory_Mem_Space[0].Data;

Memory_Mem_Space_Old[1].Origin= Memory_Mem_Space[1].Origin;
Memory_Mem_Space_Old[1].Dest=Memory_Mem_Space[1].Dest;
Memory_Mem_Space_Old[1].Data=Memory_Mem_Space[1].Data;

Memory_Mem_Space_Old[2].Origin= Memory_Mem_Space[2].Origin;
Memory_Mem_Space_Old[2].Dest=Memory_Mem_Space[2].Dest;
Memory_Mem_Space_Old[2].Data=Memory_Mem_Space[2].Data;

Memory_Mem_Space_Old[3].Origin= Memory_Mem_Space[3].Origin;
Memory_Mem_Space_Old[3].Dest=Memory_Mem_Space[3].Dest;
Memory_Mem_Space_Old[3].Data=Memory_Mem_Space[3].Data;
};


/*                    Statements                    */

UserTask0_Get_Origin ( M , UserTask0_Send_Msg_Origin );

UserTask0_Get_Dest ( M , UserTask0_Send_Msg_Dest );
UserTask0_Write ( M , Mmr_Pointers [ UserTask0_Send_Msg_Origin + ( 1
    ) ].subPointers [ UserTask0_Send_Msg_Dest + ( 1 ) ] );

Mmr_Flags [ UserTask0_Send_Msg_Origin + ( 1 ) ].subFlags [
    UserTask0_Send_Msg_Dest + ( 1 ) ] = true ;
 }

unsigned UserTask1_Send_Msg_Origin : 2 ;

unsigned UserTask1_Send_Msg_Dest : 2 ;


inline UserTask1_Send_Msg ( M) {


 d_step{
Mmr_Flags_Old[ 0].subFlags[0] =Mmr_Flags[ 0].subFlags[0];
Mmr_Flags_Old[ 0].subFlags[1] =Mmr_Flags[ 0].subFlags[1];
Mmr_Flags_Old[ 1].subFlags[0] =Mmr_Flags[ 1].subFlags[0];
Mmr_Flags_Old[ 1].subFlags[1] =Mmr_Flags[ 1].subFlags[1];


Memory_Mem_Space_Old[0].Origin= Memory_Mem_Space[0].Origin;
Memory_Mem_Space_Old[0].Dest=Memory_Mem_Space[0].Dest;
Memory_Mem_Space_Old[0].Data=Memory_Mem_Space[0].Data;

Memory_Mem_Space_Old[1].Origin= Memory_Mem_Space[1].Origin;
Memory_Mem_Space_Old[1].Dest=Memory_Mem_Space[1].Dest;
Memory_Mem_Space_Old[1].Data=Memory_Mem_Space[1].Data;

Memory_Mem_Space_Old[2].Origin= Memory_Mem_Space[2].Origin;
Memory_Mem_Space_Old[2].Dest=Memory_Mem_Space[2].Dest;
Memory_Mem_Space_Old[2].Data=Memory_Mem_Space[2].Data;

Memory_Mem_Space_Old[3].Origin= Memory_Mem_Space[3].Origin;
Memory_Mem_Space_Old[3].Dest=Memory_Mem_Space[3].Dest;
Memory_Mem_Space_Old[3].Data=Memory_Mem_Space[3].Data;
};


/*                    Statements                    */



UserTask1_Get_Origin ( M , UserTask1_Send_Msg_Origin );

UserTask1_Get_Dest ( M , UserTask1_Send_Msg_Dest );
```

```
UserTask1_Write ( M , Mmr_Pointers [ UserTask1_Send_Msg_Origin + ( 1
    ) ].subPointers [ UserTask1_Send_Msg_Dest + ( 1 ) ] );

Mmr_Flags [ UserTask1_Send_Msg_Origin + ( 1 ) ].subFlags [
    UserTask1_Send_Msg_Dest + ( 1 ) ] = true ;



 }

Msg UserTask0_Read_Msgs_Temp_Mem_Row [ 2] ;





Msg UserTask1_Read_Msgs_Temp_Mem_Row [ 2] ;






inline Mmr (){
atomic {
Mmr_Pointers [ 0]. subPointers [ 0 ]= 1;
Mmr_Pointers [ 0]. subPointers [ 1 ]= 1;
Mmr_Pointers [ 1]. subPointers [ 0 ]= 1;
Mmr_Pointers [ 1]. subPointers [ 1 ]= 1;
}


byte I; for(I: 1.. 2) {



byte J; for(J: 1.. 2) {



Mmr_Pointers [ I + ( 1 ) ].subPointers [ J + ( 1 ) ] = J + ( 2 * ( I
      1 ) ) ;



};
};
};
```

Listing B.18: The *main* translation

```
# include "Mmr.pml"



/*   Main   */

/*                    Declarations                    */



unsigned KernelTask_i : 2   =   0  ;





active proctype  KernelTask  (  ) provided (Start)  {

local byte SenderPid;
local bool FirstTime=true;

/*                    Statements                    */

do
:: KernelTask_i < 2→ KernelTask_Route ( );

KernelTask_i  =  KernelTask_i + 1  ;
:: else → break;
od
}
unsigned UserTask0_i : 2   =   0  ;

unsigned UserTask0_dest : 2  ;

Msg UserTask0_locMsg  ;

Msg UserTask0_A [ 2]  ;





proctype  UserTask0(  unsigned TaskID:  2;byte ProcNumber) provided (
    Start)  {

local byte SenderPid;
local bool FirstTime=true;
/*                    Statements                    */

do
:: UserTask0_i < 2→

 do
 ::
UserTask0_dest  =  ( UserTask0_i % 2 ) + 1  ;


UserTask0_locMsg .Data  =  0  ;


UserTask0_locMsg .Origin  =  TaskID  ;


UserTask0_locMsg .Dest  =  UserTask0_dest  ;

if
::
(
UserTask0_locMsg .Origin != UserTask0_locMsg .Dest
)
→
break;
::else → skip;
fi;

od;

SubMmr_FlagsType Mmr_Flags_Old[ 2] ;

Msg Memory_Mem_Space_Old[4];
 atomic{
Mmr_Flags_Old[ 0].subFlags[0] =Mmr_Flags[ 0].subFlags[0];
Mmr_Flags_Old[ 0].subFlags[1] =Mmr_Flags[ 0].subFlags[1];
Mmr_Flags_Old[ 1].subFlags[0] =Mmr_Flags[ 1].subFlags[0];
Mmr_Flags_Old[ 1].subFlags[1] =Mmr_Flags[ 1].subFlags[1];
```

```
Memory_Mem_Space_Old[0].Origin= Memory_Mem_Space[0].Origin;
Memory_Mem_Space_Old[0].Dest=Memory_Mem_Space[0].Dest;
Memory_Mem_Space_Old[0].Data=Memory_Mem_Space[0].Data;

Memory_Mem_Space_Old[1].Origin= Memory_Mem_Space[1].Origin;
Memory_Mem_Space_Old[1].Dest=Memory_Mem_Space[1].Dest;
Memory_Mem_Space_Old[1].Data=Memory_Mem_Space[1].Data;

Memory_Mem_Space_Old[2].Origin= Memory_Mem_Space[2].Origin;
Memory_Mem_Space_Old[2].Dest=Memory_Mem_Space[2].Dest;
Memory_Mem_Space_Old[2].Data=Memory_Mem_Space[2].Data;

Memory_Mem_Space_Old[3].Origin= Memory_Mem_Space[3].Origin;
Memory_Mem_Space_Old[3].Dest=Memory_Mem_Space[3].Dest;
Memory_Mem_Space_Old[3].Data=Memory_Mem_Space[3].Data;
};


Msg Aux_A [ 2 ];
atomic {
Aux_A [ 0 ] .Data =
UserTask0_A [ 0 ] .Data;
Aux_A [ 0 ] .Origin =
UserTask0_A [ 0 ] .Origin;
Aux_A [ 0 ] .Dest =
UserTask0_A [ 0 ] .Dest;
Aux_A [ 1 ] .Data =
UserTask0_A [ 1 ] .Data;
Aux_A [ 1 ] .Origin =
UserTask0_A [ 1 ] .Origin;
Aux_A [ 1 ] .Dest =
UserTask0_A [ 1 ] .Dest;
};
Msg Aux_M [ 2 ];
atomic {
Aux_M [ 0 ] .Data =
UserTask0_Read_Msgs_Temp_Mem_Row [ 0 ] .Data;
Aux_M [ 0 ] .Origin =
UserTask0_Read_Msgs_Temp_Mem_Row [ 0 ] .Origin;
Aux_M [ 0 ] .Dest =
UserTask0_Read_Msgs_Temp_Mem_Row [ 0 ] .Dest;
Aux_M [ 1 ] .Data =
UserTask0_Read_Msgs_Temp_Mem_Row [ 1 ] .Data;
Aux_M [ 1 ] .Origin =
UserTask0_Read_Msgs_Temp_Mem_Row [ 1 ] .Origin;
Aux_M [ 1 ] .Dest =
UserTask0_Read_Msgs_Temp_Mem_Row [ 1 ] .Dest;
};
atomic {
Aux_M [ 0].Data =
Def_Msg .Data;
Aux_M [ 0].Origin =
Def_Msg .Origin;
Aux_M [ 0].Dest =
Def_Msg .Dest;
Aux_M [ 1].Data =
Def_Msg .Data;
Aux_M [ 1].Origin =
Def_Msg .Origin;
Aux_M [ 1].Dest =
Def_Msg .Dest;
}
byte L;
for(L: 1.. 2) {
if
::
(
Mmr_Flags [ TaskID + ( 1 ) ].subFlags [ L + ( 1 ) ] == true
)
->
UserTask0_Read ( Mmr_Pointers [ TaskID + ( 1 ) ].subPointers [ L + ( 1
    ) ] , Aux_M [ L + ( 1 ) ] );
::else -> skip;
fi;
};
atomic{
UserTask0_Read_Msgs_Temp_Mem_Row [ 0].Data =
Aux_M[ 0].Data;
UserTask0_Read_Msgs_Temp_Mem_Row [ 0].Origin =
Aux_M[ 0].Origin;
UserTask0_Read_Msgs_Temp_Mem_Row [ 0].Dest =
Aux_M[ 0].Dest;
UserTask0_Read_Msgs_Temp_Mem_Row [ 1].Data =
Aux_M[ 1].Data;
UserTask0_Read_Msgs_Temp_Mem_Row [ 1].Origin =
Aux_M[  1].Origin;
UserTask0_Read_Msgs_Temp_Mem_Row  [  1].Dest =
Aux_M[  1].Dest;
}
UserTask0_Zero_Flags (
TaskID
);
UserTask0_Zero_Mem_Row (
TaskID
);
atomic {
Aux_A  [  0].Data =
UserTask0_Read_Msgs_Temp_Mem_Row  [  0].Data;
Aux_A  [  0].Origin =
UserTask0_Read_Msgs_Temp_Mem_Row  [  0].Origin;
Aux_A  [  0].Dest =
UserTask0_Read_Msgs_Temp_Mem_Row  [  0].Dest;
Aux_A  [  1].Data =
UserTask0_Read_Msgs_Temp_Mem_Row  [  1].Data;
Aux_A  [  1].Origin =
UserTask0_Read_Msgs_Temp_Mem_Row  [  1].Origin;
Aux_A  [  1].Dest =
UserTask0_Read_Msgs_Temp_Mem_Row  [  1].Dest;
}

atomic{
UserTask0_A  [  0].Data =
Aux_A[  0].Data;
UserTask0_A  [  0].Origin =
Aux_A[  0].Origin;
UserTask0_A  [  0].Dest =
Aux_A[  0].Dest;
UserTask0_A  [  1].Data =
Aux_A[  1].Data;
UserTask0_A  [  1].Origin =
Aux_A[  1].Origin;
UserTask0_A  [  1].Dest =
Aux_A[  1].Dest;
}

UserTask0_Send_Msg (   UserTask0_locMsg   );

UserTask0_i  =  UserTask0_i + 1   ;
```

```
:: else → break;
od;




}
unsigned UserTask1_i : 2   =   0  ;

unsigned UserTask1_dest : 2  ;

Msg UserTask1_locMsg  ;

Msg UserTask1_A [ 2]  ;





proctype  UserTask1(  unsigned TaskID:  2;byte ProcNumber) provided (
    Start)  {

local byte SenderPid;
local bool FirstTime=true;
/*                  Statements                   */

do
:: UserTask1_i < 2→

do

::UserTask1_dest  =  ( UserTask1_i % 2 ) + 1  ;


UserTask1_locMsg .Data  =  0  ;


UserTask1_locMsg .Origin  =  TaskID  ;


UserTask1_locMsg .Dest  =  UserTask1_dest  ;

if
 :: UserTask1_locMsg .Origin != UserTask1_locMsg .Dest →
```

```
break;

  :: else→
skip;

fi;

od;

SubMmr_FlagsType Mmr_Flags_Old[ 2] ;

Msg Memory_Mem_Space_Old[4];
 atomic{
Mmr_Flags_Old[ 0].subFlags[0] =Mmr_Flags[ 0].subFlags[0];
Mmr_Flags_Old[ 0].subFlags[1] =Mmr_Flags[ 0].subFlags[1];
Mmr_Flags_Old[ 1].subFlags[0] =Mmr_Flags[ 1].subFlags[0];
Mmr_Flags_Old[ 1].subFlags[1] =Mmr_Flags[ 1].subFlags[1];


Memory_Mem_Space_Old[0].Origin= Memory_Mem_Space[0].Origin;
Memory_Mem_Space_Old[0].Dest=Memory_Mem_Space[0].Dest;
Memory_Mem_Space_Old[0].Data=Memory_Mem_Space[0].Data;

Memory_Mem_Space_Old[1].Origin= Memory_Mem_Space[1].Origin;
Memory_Mem_Space_Old[1].Dest=Memory_Mem_Space[1].Dest;
Memory_Mem_Space_Old[1].Data=Memory_Mem_Space[1].Data;

Memory_Mem_Space_Old[2].Origin= Memory_Mem_Space[2].Origin;
Memory_Mem_Space_Old[2].Dest=Memory_Mem_Space[2].Dest;
Memory_Mem_Space_Old[2].Data=Memory_Mem_Space[2].Data;

Memory_Mem_Space_Old[3].Origin= Memory_Mem_Space[3].Origin;
Memory_Mem_Space_Old[3].Dest=Memory_Mem_Space[3].Dest;
Memory_Mem_Space_Old[3].Data=Memory_Mem_Space[3].Data;
};


Msg Aux_A [ 2 ];
atomic {
Aux_A [ 0 ] .Data =
UserTask1_A [ 0 ] .Data;
Aux_A [ 0 ] .Origin =
UserTask1_A [ 0 ] .Origin;
```

```
Aux_A [ 0 ] .Dest =
UserTask1_A [ 0 ] .Dest;
Aux_A [ 1 ] .Data =
UserTask1_A [ 1 ] .Data;
Aux_A [ 1 ] .Origin =
UserTask1_A [ 1 ] .Origin;
Aux_A [ 1 ] .Dest =
UserTask1_A [ 1 ] .Dest;
};
Msg Aux_M [ 2 ];
atomic {
Aux_M [ 0 ] .Data =
UserTask1_Read_Msgs_Temp_Mem_Row [ 0 ] .Data;
Aux_M [ 0 ] .Origin =
UserTask1_Read_Msgs_Temp_Mem_Row [ 0 ] .Origin;
Aux_M [ 0 ] .Dest =
UserTask1_Read_Msgs_Temp_Mem_Row [ 0 ] .Dest;
Aux_M [ 1 ] .Data =
UserTask1_Read_Msgs_Temp_Mem_Row [ 1 ] .Data;
Aux_M [ 1 ] .Origin =
UserTask1_Read_Msgs_Temp_Mem_Row [ 1 ] .Origin;
Aux_M [ 1 ] .Dest =
UserTask1_Read_Msgs_Temp_Mem_Row [ 1 ] .Dest;
};
atomic {
Aux_M [ 0].Data =
Def_Msg .Data;
Aux_M [ 0].Origin =
Def_Msg .Origin;
Aux_M [ 0].Dest =
Def_Msg .Dest;
Aux_M [ 1].Data =
Def_Msg .Data;
Aux_M [ 1].Origin =
Def_Msg .Origin;
Aux_M [ 1].Dest =
Def_Msg .Dest;
}
byte L;
for(L: 1.. 2) {
if
::
(
Mmr_Flags [ TaskID + ( 1 ) ].subFlags [ L + ( 1 ) ] == true
)
→
UserTask1_Read ( Mmr_Pointers [ TaskID + ( 1 ) ].subPointers [ L + ( 1
    ) ] , Aux_M [ L + ( 1 ) ] );
::else → skip;
fi;
};
atomic{
UserTask1_Read_Msgs_Temp_Mem_Row [ 0].Data =
Aux_M[ 0].Data;
UserTask1_Read_Msgs_Temp_Mem_Row [ 0].Origin =
Aux_M[ 0].Origin;
UserTask1_Read_Msgs_Temp_Mem_Row [ 0].Dest =
Aux_M[ 0].Dest;
UserTask1_Read_Msgs_Temp_Mem_Row [ 1].Data =
Aux_M[ 1].Data;
UserTask1_Read_Msgs_Temp_Mem_Row [ 1].Origin =
Aux_M[ 1].Origin;
UserTask1_Read_Msgs_Temp_Mem_Row [ 1].Dest =
Aux_M[ 1].Dest;
}
UserTask1_Zero_Flags (
TaskID
);
UserTask1_Zero_Mem_Row (
TaskID
);
atomic {
Aux_A [ 0].Data =
UserTask1_Read_Msgs_Temp_Mem_Row [ 0].Data;
Aux_A [ 0].Origin =
UserTask1_Read_Msgs_Temp_Mem_Row [ 0].Origin;
Aux_A [ 0].Dest =
UserTask1_Read_Msgs_Temp_Mem_Row [ 0].Dest;
Aux_A [ 1].Data =
UserTask1_Read_Msgs_Temp_Mem_Row [ 1].Data;
Aux_A [ 1].Origin =
UserTask1_Read_Msgs_Temp_Mem_Row [ 1].Origin;
Aux_A [ 1].Dest =
UserTask1_Read_Msgs_Temp_Mem_Row [ 1].Dest;
}
```

```
atomic{
UserTask1_A [ 0].Data =
Aux_A[ 0].Data;
UserTask1_A [ 0].Origin =
Aux_A[ 0].Origin;
UserTask1_A [ 0].Dest =
Aux_A[ 0].Dest;
UserTask1_A [ 1].Data =
Aux_A[ 1].Data;
UserTask1_A [ 1].Origin =
Aux_A[ 1].Origin;
UserTask1_A [ 1].Dest =
Aux_A[ 1].Dest;
}

   UserTask1_Send_Msg (   UserTask1_locMsg   );

UserTask1_i  =  UserTask1_i + 1   ;
:: else → break;
od ;



}

active proctype main() {

Mem_t();
Memory();
Policy();
Mmr();
Start=true;

run UserTask0(1, 1);



run UserTask1(2, 2);

/*                    Statements                    */

skip;

}
```