2017

Combining type checking with model checking for system verification

https://hdl.handle.net/2144/27171

Boston University
BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

COMBINING TYPE CHECKING
WITH MODEL CHECKING
FOR SYSTEM VERIFICATION

by

ZHIQIANG REN
B.S., Shanghai Jiao Tong University, 2003
M.S., Shanghai Jiao Tong University, 2006

Submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

2017
Approved by

First Reader

Hongwei Xi
Professor of Computer Science

Second Reader

Assaf Kfoury
Professor of Computer Science

Third Reader

Azer Bestavros
Professor of Computer Science
Acknowledgments

I would like to thank my parents for everything, especially for bearing my absence from them for so many years. I am so grateful to my primary advisor Hongwei Xi, who introduced me to the world of colourful types and later formal methods, supported me to explore various possibilities with advices and encouragement, and guided me through all the stages of my research. I would also like to thank Professor Assaf Kfoury and Professor Azer Bestavros for helping my research in the area of formal methods via courses and advices, as well as supporting me at different stages of my PhD study. Also I would like to thank Likai Liu, Matthew Danish, Ye Li, Hanwen Wu, William Blair, and many other students, staff and faculty at Boston University who have helped me along in countless ways over the years. Without their support, none of this is possible. Finally, I would like to thank my wife Jing, who always has faith in me during this journey.
COMBINING TYPE CHECKING
WITH MODEL CHECKING
FOR SYSTEM VERIFICATION

ZHIQIANG REN

Boston University, Graduate School of Arts and Sciences, 2017

Major Professor: Hongwei Xi
Professor of Computer Science

ABSTRACT

Type checking is widely used in mainstream programming languages to detect programming errors at compile time. Model checking is gaining popularity as an automated technique for systematically analyzing behaviors of systems. My research focuses on combining these two software verification techniques synergically into one platform for the creation of correct models for software designs.

This thesis describes two modeling languages ATS/PML and ATS/Veri that inherit the advanced type system from an existing programming language ATS, in which both dependent types of Dependent ML style and linear types are supported. A detailed discussion is given for the usage of advanced types to detect modeling errors at the stage of model construction. Going further, various modeling primitives with well-designed types are introduced into my modeling languages to facilitate a synergic combination of type checking with model checking.

The semantics of ATS/PML is designed to be directly rooted in a well-known modeling language PROMELA. Rules for translation from ATS/PML to PROMELA are designed and a compiler is developed accordingly so that the SPIN model checker can
be readily employed to perform checking on models constructed in ATS/PML. AT-
S/Veri is designed to be a modeling language, which allows a programmer to construct
models for real-world multi-threaded software applications in the same way as writing
a functional program with support for synchronization, communication, and schedul-
ing among threads. Semantics of ATS/Veri is formally defined for the development
of corresponding model checkers and a compiler is built to translate ATS/Veri into
CSP# and exploit the state-of-the-art verification platform PAT for model checking
ATS/Veri models. The correctness of such a transformational approach is illustrated
based on the semantics of ATS/Veri and CSP#.

In summary, the primary contribution of this thesis lies in the creation of a family
of modeling languages with highly expressive types for modeling concurrent software
systems as well as the related platform supporting verification via model checking.
As such, we can combine type checking and model checking synergically to ensure
software correctness with high confidence.
# Contents

1 Introduction ......................................................... 1  
  1.1 Problem Statement ............................................... 1  
  1.2 Model Checking and Type Checking Based Verification ........... 3  
  1.3 ATS/PML and ATS/Veri ........................................... 5  
  1.4 Thesis Organization ............................................. 7  

2 Introduction of ATS .................................................. 9  
  2.1 Overview of ATS .................................................. 11  
  2.2 Statics of ATS .................................................... 12  
      2.2.1 Sorts ...................................................... 12  
      2.2.2 Static Terms .............................................. 13  
      2.2.3 Types ..................................................... 13  
  2.3 Dynamics of ATS .................................................. 14  
      2.3.1 let-expression ........................................... 16  
      2.3.2 conditional-expression .................................. 17  
      2.3.3 case-expression ......................................... 17  
  2.4 Dependent Types .................................................. 18  
  2.5 Linear Types .................................................... 21  
  2.6 Programming with Theorem Proving ................................ 25  
  2.7 Concurrent Programming with ATS ................................ 27  
  2.8 Extending ATS with Abstract Types and Constant Functions ....... 28  
  2.9 Verification of ATS Programs .................................... 29
3 ATS/PML for Modeling Concurrent Systems

3.1 Overview of SPIN and PROMELA

3.1.1 Modeling in PROMELA

3.1.2 Semantics of PROMELA Models

3.2 PROMELA Meets Advanced Types

3.2.1 From ATS to ATS/PML

3.2.2 Leading Example

3.2.3 Combining Type Checking with Model Checking

3.2.4 Constructing Correct Models in ATS/PML

3.3 Translating ATS/PML to PROMELA

3.3.1 Processes

3.3.2 let Expressions

3.3.3 Simple Expression

3.3.4 Print Statement

3.3.5 Primitives

3.3.6 Expression Statements

3.3.7 Initiating Processes

3.3.8 Process Id

3.3.9 Initialization of Models

3.3.10 Assertions

3.3.11 Non-deterministic selection

3.3.12 Deterministic Branch

3.3.13 Inline Procedures

3.3.14 Repetition

3.3.15 Interaction with PROMELA

3.3.16 Channel Operations
3.3.17 Array Operations ........................................... 76
3.3.18 Mapping Types in ATS to PROMELA ............... 79
3.4 Using Advanced Types for Modeling: Case Study ........ 80
3.4.1 Dependent Types and Linear Types .................. 80
3.4.2 Session Types .............................................. 85
4 ATS/Veri: A Modeling Language with Advanced Types 96
4.1 Introduction .................................................. 96
4.2 Design of ATS/Veri ....................................... 97
  4.2.1 Syntax ..................................................... 98
  4.2.2 Types .................................................... 98
4.3 Primitives .................................................... 99
  4.3.1 Summary of Primitives ................................. 99
  4.3.2 Synchronization Primitives ......................... 100
  4.3.3 Global Variable Access Primitives ................. 105
  4.3.4 Thread Creation Primitives ....................... 109
  4.3.5 Model Checking Primitives ....................... 112
  4.3.6 Virtual Lock ......................................... 113
4.4 From Expression to Statement ............................ 118
  4.4.1 Intermediate Representation (IR) .................. 118
  4.4.2 Translation from ATS/Veri to IR ................. 119
4.5 Operational Semantics .................................. 125
  4.5.1 Formal Definition ................................... 125
  4.5.2 Create Instructions from a Model ................. 127
  4.5.3 Generation of State Transition Systems .......... 128
  4.5.4 Effects of Instructions .............................. 129
  4.5.5 Effects of Primitives ............................... 132
5 Model Checking ATS/Veri

5.1 Crash course of CSP# ........................................... 135
  5.1.1 Syntax of CSP# ........................................... 135
  5.1.2 Operational Semantics ................................. 136

5.2 Model Checking ATS/Veri with CSP# ......................... 137
  5.2.1 The Application-Specific Part ....................... 137
  5.2.2 The Platform-Specific Part ......................... 140
  5.2.3 Correctness of the Transformational Approach .... 142

6 Related Work ......................................................... 146

6.1 Programming Languages with Advanced Types ............... 146

6.2 Modeling Languages and Model Checkers ..................... 147
  6.2.1 PROMELA and SPIN ................................. 148
  6.2.2 CSPM and FDR ........................................ 148
  6.2.3 CSP# and PAT ........................................ 149
  6.2.4 CSPM and ProB ........................................ 150
  6.2.5 NuSMV ................................................. 150
  6.2.6 SAL ................................................... 151
  6.2.7 SystemC and KRATOS ................................. 151
  6.2.8 IC3 ................................................... 152

6.3 Model Checking Main Stream Programming Languages ....... 152
  6.3.1 Haskell ............................................... 152
  6.3.2 Java .................................................. 153
  6.3.3 C ...................................................... 153
  6.3.4 Erlang ................................................. 154

7 Conclusions and Future Work .................................. 156

7.1 Conclusions ....................................................... 156
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2 Limitation and Future Work</td>
<td>157</td>
</tr>
<tr>
<td>References</td>
<td>160</td>
</tr>
<tr>
<td>Curriculum Vitae</td>
<td>166</td>
</tr>
</tbody>
</table>
List of Tables

1.1 Comparison of Type Checking and Model Checking . . . . . . . . . . 4

3.1 Basic Types for Data Object in PROMELA . . . . . . . . . . . . . . . 36

3.2 Operator Precedence, High to Low . . . . . . . . . . . . . . . . . . . 38
# List of Figures

2-1 Some formal syntax for the statics of ATS ........................................ 12  
3-1 Peterson’s algorithm modeled in RPOMELA ................................. 46  
3-2 Peterson’s algorithm modeled in ATS/PML ................................. 47  
3-3 System Overview ................................................................. 53  
3-4 PROMELA code with labels and \texttt{goto} statements .................. 65  
3-5 ATS/PML code with mutually tail-recursive functions .................. 65  
3-6 ATS/PML syntax for mutually recursive function group .............. 66  
3-7 Sample code for interaction with PROMELA .............................. 70  
3-8 Basic Channel Operations ...................................................... 71  
3-9 Change of States by Periodic Control Module ............................ 81  
3-10 Advanced Types for Basic Operations ..................................... 83  
3-11 Modeling Operation of Periodic Control Module ....................... 84  
3-12 Agent, Client, and Server Processes ..................................... 87  
3-13 ATS/PML model for \textit{Agent} process .................................. 93  
3-14 ATS/PML model for \textit{Server} process .................................. 94  
3-15 ATS/PML model for \textit{Client} process .................................. 95  
4-1 Synchronization Primitives ..................................................... 102  
4-2 Shared objects ........................................................................ 103  
4-3 Model of a Producer Solution .................................................. 106  
4-4 Model of a Consumer Solution ................................................ 107  
4-5 Global Variable Access Primitives of ATS/Veri ............................ 108
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-6</td>
<td>Implementation for Buffer</td>
<td>110</td>
</tr>
<tr>
<td>4-7</td>
<td>Thread Creation Primitives of ATS/Veri</td>
<td>111</td>
</tr>
<tr>
<td>4-8</td>
<td>Complete Model for Producer-Consumer Problem</td>
<td>111</td>
</tr>
<tr>
<td>4-9</td>
<td>Model Checking Primitives of ATS/Veri</td>
<td>112</td>
</tr>
<tr>
<td>4-10</td>
<td>Syntax for ATS/Veri IR</td>
<td>120</td>
</tr>
<tr>
<td>5-1</td>
<td>Syntax for CSP#</td>
<td>136</td>
</tr>
<tr>
<td>5-2</td>
<td>Content of a Translated Model in CSP#</td>
<td>138</td>
</tr>
<tr>
<td>5-3</td>
<td>Process in CSP# for function in ATS/Veri</td>
<td>140</td>
</tr>
<tr>
<td>5-4</td>
<td>Modeling of Scheduler</td>
<td>143</td>
</tr>
<tr>
<td>5-5</td>
<td>Process Capable of Scheduling</td>
<td>143</td>
</tr>
<tr>
<td>5-6</td>
<td>Bisimulation Relation between ATS/Veri and CSP#</td>
<td>145</td>
</tr>
</tbody>
</table>
List of Abbreviations

ATS .............. Applied Type System
PTS .............. Pure Type System
PwTP .............. Programming with Theorem Proving
SPIN .............. Simple Promela Interpreter
PROMELA .............. Process Meta-Language
LTL .............. Linear Temporal Logic
IR .............. Intermediate Representation
PAT .............. Process Analysis Toolkit
Chapter 1

Introduction

1.1 Problem Statement

Our society is unprecedentedly relying on the correct behavior of myriads of computing devices, i.e. the hardware and the software running on it should behave correctly according to their specifications. Consequently, we are seeing an increasing demand for the formal verification of mission-critical systems, such as autopilot in avionics, life-support systems, or ABS in your cars. Moreover, Moore’s law is approaching its demise and processor frequencies have plateaued in the previous decade. Meanwhile, the age of multi-core / many-core processors is emerging, as we have been witnessing an exponential increase in the number of processor cores in recent years. The downside of the trend is that our algorithms do not automatically profit from the newer generation of processors. Thus, it is now the programmers’ responsibility to design and implement efficient concurrent software systems to fully exploiting the powerful architecture with those cores. Building sequential software is already an error-prone process. Adding concurrency makes things worse. Hence, it is very important as well as challenging to tackle the problem of how to design, analyze, and implement efficient yet correct concurrent software systems in a cost effective manner. And this stimulates my research to employ formal methods in the development process in a flexible yet effective manner.

Generally speaking, formal methods are a particular group of mathematically based techniques for the specification, development and verification of software and
hardware systems. First and foremost, they include various techniques for modeling complex systems as mathematical entities. By building a mathematically rigorous model of a complex system, various techniques in formal methods can be adopted to verify a system’s properties in a more thorough fashion based on mathematical proofs than empirical testing. As systems become more complicated, and safety becomes a more important issue, such a formal approaches to system design offer another level of assurance.

One of the main barriers to the practical application of formal methods is that it is difficult to write or obtain high quality models (formal specifications). This is a problem engineers face in industry today, and is a major problem I address in this thesis by exploiting the technique of type checking. Type systems along with corresponding type checking techniques, widely used in practical programming languages (e.g. C/C++, Java, ML, Haskell), are perhaps the most pervasive of all software verification techniques. A plethora of evidence has demonstrated their capabilities of detecting programming errors at compile-time. Considering that writing programs is a similar process to modeling software systems, it is appealing to adopt the advancement in type systems developed for programming languages to help identify errors in reasoning when writing models. Going one step further, model checking, an automated technique in formal methods for system verification, is gaining more and more attention than ever before from the software verification community due to the increasing number of successful applications. Yet large-scale application of model checking is still limited due to the initial complexity of constructing faithful models as well as the size of the state space which may increase exponentially with the number of the components of a system.

In this thesis, I address the issue of combining the two prevailing techniques, type checking and model checking, synergically to better support the modeling and
verification of concurrent software systems.

1.2 Model Checking and Type Checking Based Verification

Before revealing details of my research, I give out some informal description about type checking and model checking as follows with Table 1.1 outlining their characteristics.

Type systems are widely used in practical programming languages to detect programming errors at compile-time. They involve a set of types and type constructors along with the rules that govern whether or not a program is legal with respect to types. Informally, a type system provides a discipline, which programmers must follow when constructing programs with typed entities, and therefore guarantees the correctness of the programs. With advanced type systems e.g. dependent types or linear types, programmers can specify fine-grained program invariants via types, and the type checking procedure would verify these invariants with the help of user-supplied proofs as well as automated tools such as theorem prover or SMT solver. Advanced abstraction of system behaviors via types is enabled by human creativity, and this makes type checking a scalable technique for verifying program invariants, which is especially true for systems with complicated data structures. However, the aforementioned advanced type systems are still not sophisticated enough to support the verification of rich temporal properties common for concurrent systems. Moreover, even if we manage to create a type system with such capability, its usage would be prohibitive to human programmers due to the system’s inherent complexity.

Model checking, a push-button technology, uses computers to exhaustively analyze the behaviour of a system that is finite state or has finite state abstraction. It is especially useful for the verification of concurrent systems with intensive control flow against rich temporal properties, which is extremely challenging to handle solely rely-
Model checking technique consists of 1. language formalisms allowing designers to construct models of systems to be built; 2. logic formalisms (e.g. process algebra, temporal logic) for specifying temporal behaviours of such models; 3. algorithms for checking the validity of property specifications against models. The shortcoming of model checking is that as the number of system variables and components increases, the size of system state space to be checked grow exponentially. This is called the "state explosion problem".

In this research, I combine two aforementioned software verification techniques synergically in one platform for the creation of correct models for software designs. The proposed platform consists of a modeling language equipped with advanced type systems, as well as supports for applying model checking algorithms on models written in such modeling language. I believe that the proposed platform can benefit system designers from both type checking and model checking and mitigate the weakness of either one.

- Model designers can use type checking to eliminate, bugs which are not intrinsic to the concurrent nature of systems. Such bugs may be introduced due to mistakes in local reasoning within one process as well as in resource management across multiple processes. Some examples include out-of-bounds array access, resources leaking, etc. In short, we can have a more correct model, which would cut the cost of the model checking stage.

- Type systems enable designers to specify requirements inside the model in a

\[\begin{array}{|c|c|c|}
\hline
\text{Characteristics} & \text{Type Checking} & \text{Model Checking} \\
\hline
\text{Human creativity} & \text{Programmer guided abstraction / reasoning} & \text{Push-button technology} \\
\text{Scalability} & \text{Scalable} & \text{State explosion} \\
\text{Verified Properties} & \text{Program Invariants} & \text{Temporal behavior} \\
\text{Targeting system} & \text{Complicated dynamic data structure} & \text{Control-intensive system} \\
\hline
\end{array}\]

\textbf{Table 1.1:} Comparison of Type Checking and Model Checking
more natural manner. And some of the resulting type constraints, which may not be solved by type checking procedure alone, can be discharged via model checking (using primitives with well-designed types). On one hand, this offers a disciplined approach to exploiting special features of model checking (e.g. assertion) at the stage of model construction. On the other hand, the capability of verifying global properties offered by model checking makes it feasible to type check sophisticated models for concurrent software systems.

A more detailed description of my research goes as follows.

1.3 ATS/PML and ATS/Veri

In contrast to developing a type theory with expressive types including both dependent type and linear types and then designing upon it a modeling language, I choose to exploit existing programming languages with advanced type systems to support modeling concurrent systems with the capability of model checking. ATS is a statically typed programming language that unifies implementation with formal specification. It is equipped with a highly expressive type system rooted in the framework Applied Type System, which gives the language its name. In particular, both dependent types and linear types are available in ATS, and the specification of a system can be encoded in the form of types when programming in ATS. ATS also has a theorem-proving subsystem which enables programmers to facilitate the type checking process by manually constructing proofs along side the actual code. This programming paradigm is referred to as programming with theorem-proving (PwTP) (Chen and Xi, 2005). Based on the aforementioned paradigm, we formalized a methodology of programmer-centric software verification (Ren and Xi, 2013) for sequential programs. In this methodology, programmers can introduce abstract concepts, state assumptions about these concepts, and write programs as well as proofs
based on these assumptions. This lightweight methodology towards program verification helps increase development efficiency. Based on my previous research related to ATS, I decided to derive a new modeling language from ATS by reusing its type system as well as the core syntax so that all the aforementioned type related features can be inherited, in the sense that we can use the original ATS type checker to type check models written in the proposed modeling language. Thus the aforementioned programmer-centric methodology can be applied to the design and verification of concurrent systems by exploiting model checking techniques to verify those assumptions not provable solely via types, which in turn increases the credibility of the correctness verification. The research is two pronged based on the capability of the new modeling language, the formalization of its dynamic semantics, and its target applications.

First, I choose to base the semantics of the new modeling language on a state-of-the-art modeling language Promela (Holzmann, 2003), supporting dynamic creation of concurrent processes as well as communication between processes via global variables and message channels. On one hand, I choose a subset of the syntactic core of ATS and introduce into it primitives with special operational semantics to cover features that have direct roots in Promela. Such primitives are assigned well-designed types to facilitate a synergic combination of type checking with model checking. On the other hand, models in the new modeling language is compiled into Promela, which defines the semantics of the original models. This modeling language is referred to as ATS/PML to indicate its tight bond with Promela. Under such setting, models in ATS/PML can be model checked by applying the SPIN (Holzmann, 2003) model checker to check the generated models in Promela. Considering its direct root in Promela, ATS/PML can be viewed as a front-end of Promela with an enhanced type system. It is expected that such a lightweight method would attract Promela users to start taking advantage of advanced type systems for detecting errors at compile time.
while constructing models targeting the SPIN model checker under the consideration of verification efficiency.

Second, I extend the core of ATS into an independent modeling language, in which programmers can construct models just like writing functional programs running on a virtual machine which explores all possibilities resulting from non-determinisms (e.g. scheduling). The extension consists of various primitives, supporting modeling concurrency (e.g. synchronization, communication) and non-determinism. These primitives also facilitate the specification of properties (e.g. assertion of global states) as well as enable the application of model checking for such properties. Well-designed types are assigned to these primitives to increase the confidence of their correct usage. This modeling language is referred to as ATS/Veri, indicating its usage for software verification via both model checking and type checking. I design ATS/Veri to be a conservative extension over ATS, i.e. a program consisting of only core features of ATS is a valid model in ATS/Veri. Meanwhile, I define the operational semantics of ATS/Veri based on state transition systems. With formal semantics at hand, I build a compiler to translate models in ATS/Veri into CSP#, the modeling language supported by the state-of-the-art model checker PAT (Sun et al., 2009a). With PAT (Program Analysis Toolkit), we can apply many model checking techniques including LTL model checking and refinement checking to verify various properties of models in ATS/Veri including deadlock-freeness, reachability, and LTL properties with fairness assumptions.

1.4 Thesis Organization

The remainder of this document is organized as follows:

Chapter 2 illustrates major parts of ATS including both static and dynamic semantics with focus on its advanced type system. It discusses how dependent types, linear
types, and the paradigm of programming with theorem proving can be exploited to verify functional properties of programs.

Chapter 3 gives an overview of the modeling language PROMELA including its features, semantics, usages, and the insufficiency of its type system. It introduces a new modeling language, ATS/PML, which has direct root in PROMELA from the perspective of semantics, but is equipped with advanced types including dependent types and linear types grafted from the language ATS. Detailed rules for the transformation between PROMELA and ATS/PML are given. It demonstrates, via examples, usages of ATS/PML and how modeling primitives with well-designed types can be used to facilitate a synergic combination of type checking with model checking.

Chapter 4 illustrates the design of ATS/Veri, a modeling language allowing a programmer to write models in the same way as writing a functional program. It includes multiple examples and discussion to illustrate the usage of primitives, ATS/Veri provides for modeling concurrent systems. It also gives out a formal definition of the operational semantics of ATS/Veri, which serves the foundation for incorporating model checking techniques for the verification of models in ATS/Veri.

Chapter 5 presents a transformational approach for model checking ATS/Veri. It begins with an introduction of a popular modeling language CSP# with corresponding state-of-the-art verification platform PAT, then illustrates the relation between a model in ATS/Veri and the translated model in CSP#. It concludes by giving an argument for the correctness of this transformational approach.

Chapter 6 discusses some related works in the field of exploiting type checking and model checking for system verification.

Chapter 7 summarizes the results of current research, suggests avenues of future work.
Chapter 2

Introduction of ATS

The focus of this research is the development of modeling languages equipped with highly expressive types (including both dependent types and linear types), as well as the study of practical methods for exploiting these types in order to detect modeling errors at the stage of model construction. With this goal in mind, I choose to exploit existing research work related to type system designs. There is already a framework Pure Type System \( \mathcal{PTS} \) (Barendregt, 1992) that offers a simple and general approach to designing and formalizing type systems. However, it is very difficult to set up a type system based on \( \mathcal{PTS} \), to support dependent types, which allows programmers to verify functional properties of their models, as well as accommodate realistic programming features common to modeling languages, e.g. recursion, effects, and exceptions. Though possible, modeling languages with such type system would be too difficult for programmers, who may not be experts in type systems, to use in practical applications. Therefore I choose to exploit another framework, Applied Type System (\( \text{ATS} \)) (Xi, 2004), which offers an approach to design type systems supporting dependent types in the presence of effects such as references and exceptions. Such capability of \( \text{ATS} \) comes from its design of a complete separation between statics, in which types are formed and reasoned about, and dynamics, in which programs are constructed and evaluated.

Based on \( \text{ATS} \), a functional programming language was designed and implemented and received its name AT S ((Xi, 2008)). The type system of AT S supports not only
dependent types (in DML style (Xi, 2007)), but also guarded recursive datatypes and linear types. My study and practice of both the programming language ATS and some state-of-the-art modeling languages (e.g. PROMELA (Holzmann, 2003) and CSP# (Sun et al., 2009a).) suggest that all these types can be of great use for model construction. Also there is great similarity between the semantics of ATS and those modeling languages. Therefore instead of designing and implementing a new type system from the framework ATS directly, I choose to reuse the type system of ATS and part of its syntax features as the start point for the development of new modeling languages.

The type system of ATS is developed gradually, starting from its initial support of dependent types (Xi, 1998) (Xi and Pfenning, 1999) (Xi, 2007), to the addition of guarded recursive datatypes (Xi et al., 2003), to the accommodation of linear types (Zhu and Xi, 2005) (Shi and Xi, 2009), and later to the adoption of Programming with Theorem Proving (Chen and Xi, 2005). Formal development of the mathematical proof for the type soundness property was conveyed in multiple researches, each of which targets certain newly added types it focuses. Also the successful employment of the programming language ATS in various application development over years has circumstantially shown its correctness. Hence, in this section, I will not formally develop the complete type system and semantics of ATS from a theoretical perspective. Instead, I will illustrate via examples various features of its type system as well as their usage in practice. Also I will give out intuitive interpretation of those types and explain their connection with the semantics of ATS. The discussion here can help readers understand the type system of proposed modeling languages introduced in later chapters as well as relate the semantics of ATS with such modeling languages.
2.1 Overview of ATS

ATS is a statically typed programming language that unifies implementation with formal specification. It is equipped with a highly expressive type system rooted in the framework \textit{Applied Type System} (ATS), which gives the language its name. In particular, both dependent types and linear types are available in ATS.

The core of ATS is a ML-like functional language based on call-by-value evaluation. It supports features including high order functions (closure), pattern matching, lazy evaluation, and parametric polymorphism. The availability of linear types in ATS often makes functional programs written in it to run not only with surprisingly high efficiency (when compared to C) but also with surprisingly small (memory) footprint (when compared to C as well).

Going further, full-fledged ATS is a practical programming language supporting both functional programming and imperative programming. It includes features supporting writing programs effectively such as module and template system. Moreover, some features for constructing efficient programs such as explicit pointer arithmetic and explicit memory allocation/deallocation, which are considered dangerous in other languages, can be safely supported in ATS due to its type system. Last but not least, from ATS source code, ATS compiler generates C code, which can then be compiled and linked with garbage collector to run without additional support of runtime on platform including Linux and Mac.

The language ATS has a dynamic component (dynamics) in which programs and proofs are constructed, as well as a static component (statics) in which sophisticated types are constructed for both programs and proofs as specifications of their properties. (Here, we use the word “type” in its most common sense as contrast to our definition of \textit{type} as shown in section 2.2). The compiler of ATS checks whether a program in the dynamics can be assigned certain type according to a set of typing
rules. As such, a program is verified to have the property encoded by its type. Such programming paradigm of constructing both programs and proofs with certain types, which can be verified by the compiler is called Programming with Theorem Proving (PwTP).

In this thesis, my research exploits the core of ATS as well as its unique support for imperative programming via the concept of variables. Thus, my discussion about ATS in the sequel mainly focuses on these areas of ATS and simply refer to them as ATS.

2.2 Statics of ATS

Though inspired by dependent types in Martin-Löf’s development of constructive type theory, the dependent types adopted by the ATS programming language are of a restricted form. To be more specific, in ATS, types are not allowed to be dependent on program terms (i.e. expressions in the dynamics) directly. Instead they are constructed purely in the statics. The statics in ATS is a simply-typed language without any side effect (e.g. recursion). Such design decision not only makes it practical for programmers to reason about the types they want to use, but also leads to an efficient type checking procedure.

2.2.1 Sorts

Figure 2.1 gives out a formal definition of the statics of ATS, the essence of which is a simply typed \(\lambda\)-calculus augmented with a set of built-in constants. We use the
name static terms to refer to the terms in the statics and name sort to refer to the
type for static terms. For instance, ATS has built-in sorts such as bool, int, addr,
type, prop, viewtype, and etc. Also we can define new algebraic sort using datasort.
For example, the code below shows an inductive definition of sort ilist which can be
viewed as a mathematical definition of a finite list of integers.

datasort ilist =
| inil of ()
| icons of (int, ilist)

2.2.2 Static Terms

ATS provides a lot of built-in static constructors and constants to help construct
sophisticated terms in the statics. For instance, the statics contains constructors
corresponding to all the integers in mathematics. These constructors are of sort
() → int. (For clarity, I will use n instead of n() to represent static terms of sort int.)
Similarly, true and false of sort () → bool are provided in the statics for boolean
values. Going further, common operators for such integers and boolean values in
mathematics have their counterparts as constant functions of appropriate sorts in the
statics, e.g. + of sort int → int → int, < of sort int → int → bool.

2.2.3 Types

Among various static terms, those of sort type, prop, view, and viewtype are of par-
ticular interest because they can be used as types (in its most general sense) for
expressions in the dynamics (a.k.a. dynamic expression). To be more precise, we
may refer to static terms of sort type types, static terms of sort prop props, static
terms of sort view views, and static terms of sort viewtype viewtypes. In particular,
we may refer to static terms of any of these sorts types. when there is no ambiguity
in the context. In common programming languages, a type is usually understood as a
set of elements. In ATS, however, it is more natural to interpret types as the meaning of the dynamic expressions, which are assigned the types. And the statics of ATS supports the construction of sophisticated static terms used as types to describe the meaning of a program in a precise manner.

ATS provides various constant constructors for creating basic types. For example, \texttt{int} is overloaded for two type constructors of sort \((\_ \rightarrow \textit{type}) \rightarrow \textit{type}\) and sort \(\texttt{int} \rightarrow \textit{type}\). Thus \texttt{int} is a type, so as \texttt{int}(1) and \texttt{int}(x) given that \(x\) is a variable of sort \texttt{int}. Similarly, \texttt{bool} is overloaded for two type constructors of sort \((\_ \rightarrow \textit{type}) \rightarrow \textit{type}\) and sort \(\texttt{bool} \rightarrow \textit{type}\). Thus \texttt{bool} is a type, so as \texttt{bool} \((\text{true})\) and \texttt{bool} \((b)\) given that \(b\) is a variable of sort \texttt{bool}. Some other basic type constructors shall be illustrated later when corresponding types are discussed.

Going further, ATS also provides various type constructors to form complex types including guarded types, asserting types, universal types, and existential types, which shall be explained later based on examples of their usages.

Lastly, ATS allows programmers to introduce new type constructors into statics either by defining new algebraic datatypes or simply declare abstract types without concrete definition. Both of these two methods are useful in my proposed modeling languages, and shall be explained when encountered later.

### 2.3 Dynamics of ATS

The dynamic semantics of a multi-threaded ATS program can be formally defined based on a form of parallel reduction with side-effects on a set of resources (e.g. memory, lock), which is discussed in (Xi, 2000) (Shi, 2008) (Shi et al., 2010). However, such semantics indeed resembles the evaluation of multi-threaded programs on a multi-core machine equipped with resources. Another way to put it, reductions in the formal semantics corresponds to evaluation steps on the multi-core machine, which
is more intuitive for comprehension. Thus I choose the later method to illustrate the
dynamic semantics of ATS.

Usually, a program in ATS consists of several function definitions including one
main function. The syntax of ATS is similar to that of ML programming language.
For example, the following code shows a simple function definition in ATS.

```plaintext
fun foo (x: int, y: int): int = x + y
```

Although ATS supports a limited form of type inference that allows you to omit
certain type annotations, it is a good practice to put all of them on function definitions
because the truly advanced types supported by ATS are not compatible with full type
inference.

As a functional programming language in its core, ATS supports the usage of
functions as first-class objects. That is functions are just normal values, which can
be used as arguments, return values, and be assigned to variables or stored in data
structures. Functions can be defined inside other functions with or without names
(lambda expression). In ATS, closures are formed if the function bodies refer to names
outside the current scope. One thing worth noting is that ATS supports a special type
of closures which contain linear resources and can be invoked once and only once. And
such type of closures are called linear closures.

ATS is an expression based language, i.e. the body of a function is an expression in
the dynamics of ATS. An expression of ATS is sometimes called a dynamic expression
since it is constructed in the dynamics of ATS, as opposed to a static term, which is
built in the statics of ATS. A dynamic expression can be a primitive constant, such
as an integer or a character, a name for a value or variable, a lambda expression, or
a function invocation with expressions as arguments, e.g. `f (1, g (1, 'c'))` where
f and g are names for functions. Besides these expressions, there are three special
structural expressions: let-expression, conditional expression, and case-expression,
which are discussed in the sequel.

2.3.1 let-expression

A let-expression has the following syntax:

```
let
    statement1
    statement2
    ...
in
    exp
end
```

A let-expression is evaluated as follows: First, all the statements are evaluated in the same order as they appears. Then the final expression is evaluated, and the resulting value is used as the value of the let-expression. A statement can be a pattern match, a variable definition, or a inner function (closure) definition.

A variable definition in ATS is similar to those in C programming language. For example, in the following code:

```
var x: int = 1 + 2
```

The expression `1 + 2` on the right hand side of the equation is evaluated first and the result is stored in the variable `x`.

A pattern match has the following syntax:

```
val pattern = expression
```

To evaluate a pattern match, the expression on the right side of the equation gets evaluated first. Then the evaluation result is matched against the pattern, which involves binding names in the pattern to the content of the evaluation result. (Each name can be thought as a variable whose content cannot be changed after the binding.) If the pattern match fails, then an exception is thrown to indicate such error. Note
that variable assignments can be viewed as a special form of function invocation and can be treated as a normal expression. The following code is for updating the content of the variable x.

```plaintext
val () = x := x + 1
```

The void-pattern on the left hand side of the equation is for checking that the type of the expression on the right hand side is actually void.

### 2.3.2 conditional-expression

A condition-expression has the following syntax:

```plaintext
if exp1
  then exp2
  else exp3
```

A conditional-expression is evaluated as follows: First `exp1` is evaluated. If this evaluates to `true` then the value of the conditional-expression is the value obtained by evaluating the expression `exp2`. Otherwise, the value obtained by evaluating `exp3` is the final value of the whole expression.

### 2.3.3 case-expression

A case-expression has the following syntax:

```plaintext
case exp of
  | pattern1 when guard1 => exp1
  | pattern2 when guard2 => exp2
  ...
```

A case-expression is evaluated as follows: First, `exp` is evaluated, assume this evaluates to `value`. Thereafter `value` is matched against `pattern1`. If there is a match, then `guard1`, which is a predicate and may involve names appearing in `pattern1`, is evaluated. If `guard1` is evaluated to `true`, then the result of the evaluation of `exp1`
is the value of the case-expression. If pattern match fails or the predicate is evaluated to \texttt{false}, the whole matching process is applied to the second clause (involving \texttt{pattern2} and \texttt{guard2}). An exception is thrown if the matching process fails for all the clauses.

### 2.4 Dependent Types

As a ML-like functional programming language at heart, ATS supports programming with no usage of advanced types at all. For example, the type of the following function

\begin{verbatim}
fun foo (x: int): int = x + 2
\end{verbatim}

simply indicates that it takes an integer (more precisely, a value of type \texttt{int}) as argument and returns an integer.

Based on dependent types, ATS provides programmers with several special forms of types including guarded types, asserting types, universal types, existential types, and singleton types. Using such types, the function in the previous example can be rewritten as follows:

\begin{verbatim}
fun foo {n:int | n >= 0} (x: int n): [y: int | y > n] int y
\end{verbatim}

The type can be formally written as the following:

\[
\forall n : \text{int}. \ n \geq 0 \supset \text{int}(n) \rightarrow \exists y : \text{int}. \ y > n \wedge \text{int}(y)
\]

Note that, \(\supset\) and \(\wedge\) are two constant type constructors of sort \((\text{bool}, \text{type}) \rightarrow \text{type}\) in the statics. Also the details about encoding universal type and existential type using constant constructors are omitted here.
Intuitively, the type above can be interpreted as that the function takes as argument a value of type \texttt{int}(n) given that \( n \) is of sort \texttt{int} and \( n \geq 0 \), and then returns a value of type \texttt{int}(y) where \( y \) is certain static value of sort \texttt{int} and \( y > x \). In essence, the setting up of the typing rules of ATS enables programmers to treat types as formulae in first-order logic as well as write code in a way similar to proof construction in intuitionistic logic. Going one step further, singleton types, which are each a type for only one specific value, set up one-to-one correspondence between types in the statics and values in the dynamics. For instance, \texttt{bool}(B) is a singleton type for the boolean value equal to \( B \), and \texttt{int}(I) is a singleton type for the integer equal to \( I \), and \texttt{ptr}(L) is a singleton type for the pointer that points to the address (or location) \( L \).

Based on the aforementioned types, the soundness of ATS’ type system guarantees that in a well-typed program, the argument for invoking the function \texttt{foo} is a non-negative integer, and the return value of the function is an integer bigger than the input argument.

In the body of the function, the + operator is overloaded with the function declared as follows:

\[
\text{fun add \{i,j:int\} (x: int (i), y: int (j)): int (i + y)}
\]

When type checking the body of the function, the type checker of ATS can smartly figure out that the type of the value 2 is \texttt{int}(2), and in turn that the type of the return value is \texttt{int}(n + 2). Constraints including \( y = n + 2 \) and \( y > n \) are generated and then solved given the premise \( n \geq 0 \), leading to the well-typedness of the function definition.

In the example above, the type of the function can be interpreted as a formula on static values in the first-order logic. And singleton types correlate the static values and dynamic values so that the type can be further interpreted as a specification of
the pre-condition and post-condition of the dynamic behavior of the function. At first glance, it may seem redundant that we have similar concepts in both statics and dynamics. But the extra level of abstraction provides programmers with flexibility in describing properties of programs. For example, we can use a static integer to represent the length of an array in the dynamics. We can introduce into ATS a type constructor \texttt{arrayref} for such an array type as follows:

\begin{verbatim}
abstype arrayref (a: t@ype, n: int)
\end{verbatim}

Note that the sort \texttt{t@ype} is a superset of the sort \texttt{type}. A static value of sort \texttt{t@ype} is a type for unboxed values while a static value of sort \texttt{type} is a type for boxed values. Static values of sort \texttt{t@ype} or \texttt{type} are simply referred to as types when there is no ambiguity, though the former are used more often in code examples considering that the type \texttt{int} in ATS is actually of sort \texttt{t@ype}.

Given a type \texttt{a} and an integer \texttt{n} in the statics, the type \texttt{arrayref} \texttt{(a, n)} is for an array whose elements are of type \texttt{a} and whose length is \texttt{n}. The corresponding function for creating, subscripting, and updating such arrays are declared as follows:

\begin{verbatim}
fun arrayref_create {a:t@ype} {len: nat} (len: int len, x: a): arrayref (a, len)

fun arrayref_get {a:t@ype} {n, len: nat | n < len} (arr: arrayref (a, len), pos: int n): a

fun arrayref_set {a:t@ype} {n, len: nat | n < len} (arr: arrayref (a, len), pos: int n, x: a): void
\end{verbatim}

Note that \{\texttt{n:nat}\} is a short cut for \{\texttt{n:int | n >= 0}\}. It is straightforward to see that the types for the array subscript and update functions indicate such pre-conditions that the input argument for the position inside an array has to be a natural number smaller than the length of the array. If a program using such functions for manipulating arrays is well-typed in ATS, we are guaranteed by the type system that
there would be no out-of-bound error for accessing arrays as long as those constant
function implementations, which may be done external to ATS, actually possess those
properties specified by their types.

In summary, the type system of ATS enables programmers to specify properties
for program in the dynamics using advanced types in the statics. Such types can be
interpreted as formulae in first-order logic. When type checking a program, the type
checker of ATS first generates all of the constraints needed to be verified in order
to ensure the well-typedness of the program and then passes these constraints to an
SMT-solver to check for their satisfiability. Going further, a programmer can help
the type checker discharge constraints by manually inserting proof code, which shall
be illustrated later in this chapter.

2.5 Linear Types

The dependent types in ATS allows programmers to describe states of resources (e.g.
the length of an array) in a program in an abstract yet precise manner in order to
prevent misuse of such resources. For example, a well-typed program constructed in
ATS with the usage of arrayref cannot cause array out-of-bound error at run-time.
Going further, ATS offers a whole collection of types enabling programmers to track
and safely manipulate resources (e.g. the memory holding the array). Such types
are called linear types, where the word linear comes from linear logic (Girard et al.,
1989).

In ATS, a static term of sort view is called a view or a linear prop. A dynamic
expression whose type is a view is called a linear proof. Similarly, a static term of
sort viewtype is called a viewtype or linear type. A dynamic expression whose type
is a viewtype is called a linear object. In the most general sense, both linear prop
and linear type are called linear types, and both linear proof and linear object are
called linear objects. Intuitively, a linear type is for a value containing some resources, which cannot be arbitrarily discarded or duplicated, and shall be consumed once and only once eventually. I will use the following example to illustrate the usage of linear types in ATS.

In previous section, dependent types are used to help check array-bounds statically. However, the array itself, with type $arrayref(a, n)$, cannot be released manually (no constant function provided), thus must have its memory managed automatically (e.g. by a garbage collector). Adding such function for releasing the array may lead to many programming bugs, which are well-known by programmers with experience of manipulating allocated memory manually. Luckily, linear types in ATS can come to help. New types for such arrays as well as corresponding constant functions can be rewritten as follows:

```plaintext
absview arrayview (a: t@ype, len: int, l: addr)

fun arrayview_create {a:t@ype} {len: nat} (len: int len, x: a):
[l: addr | l >= null] (arrayview (a, len, l) | ptr l)

fun arrayview_get {a:t@ype} {n, len: nat | n < len} {l:addr | l >= null} (pf: arrayview (a, len, l) | p: ptr l, pos: int n):
(arrayview (a, len, l) | a)

fun arrayview_set {a:t@ype} {n, len: nat | n < len } {l:addr | l >= null} (pf: arrayview (a, len, l) | p: ptr l, pos: int n, x: a):
(arrayview (a, len, l) | void)

fun arrayview_destroy {a:t@ype} {len: nat} {l:addr | l >= null} (pf: arrayview (a, len, l) | p: ptr l): void
```
In the above code, I introduce an abstract view constructor `arrayview`. Given a static term `a` of sort `t@ype`, a static term `len` of sort `int`, and a static term `l` of sort `addr`, the view `arrayview (a, len, l)` can be interpreted as a linear proposition stating that there is an array residing at the address `l`, whose elements are of type `a`, and whose length is equal to `len`. For `arrayref` related functions, a parameter of type `arrayref (a, len)` is used as a reference to the array. In contrast, in `arrayview` related functions, we need two parameters to access the array: 1. a pointer `p` whose value is the address `l` of the array; 2. a linear proof `pf` showing that that an expected array actually exists at the address `l`.

Syntactically, views (e.g. `arrayview (a, len, l)`) and props are written on the left side of the vertical bar `|`, while `t@ype` (e.g. `ptr l`) and `viewtype` on the right side, as shown in the example above. Note that dynamic expressions that are on the left side of the vertical bar are only of interest to the type checker, but erased after type checking. And only dynamic expression on the right side of the vertical bar `|` will be part of the final, compiled program. If the type `arrayref(a, len)` is indeed implemented in the form of pointer, the implementations of `arrayref` related functions can be reused without any change as implementations for functions related to `arrayview` except that the newly added `arrayview_destroy` has to be implemented to release the memory allocated for the array.

The function `arrayview_create` returns a linear proof proving the existence of the array. Both function `arrayview_get` and `arrayview_set` takes a linear proof as their argument and returns a new linear proof of the same view, which can be interpreted as that these two functions do not consume the input linear proof in net. In contrast, function `arrayview_destroy` simply consumes the linear proof, indicating the destruction of the array. The following code shows the usage of all these functions.

```plaintext
fun foo () : void = let
```
It is easy to see that using these functions, we cannot create two linear proofs of the same view. Also invoking `arrayview_destroy` is the only way to consume a linear proof. Moreover, the typing rules of ATS guarantees that the evaluation process, except for invocations of those constant functions (a.k.a. ad hoc reduction in (Shi and Xi, 2009)), neither produces nor consumes any linear resources (linear proof in the example here). Therefore, when evaluating a well-typed program using these constant functions, it is impossible to have two linear proofs of the same view, thus there is no issue of pointer-alias common in C programming language. Also, linear proof has to be presented in order to subscript the array, thus preventing the error of accessing released memory. Going further, since function `foo` does not return any linear resource, the linear proof generated inside the function body has to be consumed by the `arrayview_destroy` once and only once, thus eliminating bugs of memory leak (resource not being consumed).
2.6 Programming with Theorem Proving

In previous section, linear proofs are used to justify that arrays of certain length exist at addresses indicated by pointers. In ATS, proofs (dynamic expressions of type prop and view) can be constructed inline with programs in a syntactically intwined manner though their existence is completely erased after type checking. Proofs serve as a medium for encoding a programmer’s reasoning inline with the program as well as help discharge constraints, which otherwise cannot be solved by type checker alone. Such programming paradigm of constructing proofs as well as programs together is called Programming with Theorem Proving, which is a signatory feature of ATS.

In the previous example related to arrayview, the two functions arrayview_get and arrayview_set consume a linear proof and generate a new linear proof, which indicates the possible change of states of the arrays. However, such state transition of the underlining array due to function invocations cannot be seen on the type of the function since the two proofs involved are of the same view.

The following code shows the type for an array-based buffer as well as constant functions for manipulating it. (To simplify the discussion, I assume that there is only one such buffer in the program and its elements are of type int. Thus the parameters of sort addr and t@ype are omitted when defining the view constructor arraybufview.)

```plaintext
absview arraybufview (len: int, beg:int, tail: int)

fun arraybufview_get
  {len, beg, tail | beg <= tail; tail <= len} (pf: !arraybufview (len, beg, tail)
   >> arraybufview (len, beg + 1, tail)
   | pos: int beg): int
```

fun arraybufview_set
  {len, beg, tail: nat | beg <= tail; tail <= len} (pf: !arraybufview (len, beg, tail)
    >> arraybufview (len, beg, tail + 1)
    | pos: int tail, x: int): void

prfun arraybufview_rewind
  {len, beg, tail: nat | beg == tail; tail <= len} (pf: !arraybufview (len, beg, tail)
    >> arraybufview (len, 0, 0)): void

In the code above, arraybufview is used to construct views for describing states of
the buffer. Intuitively, a proof of type arraybufview (len, beg, tail) represents
that the array is of length len, and elements located at positions between beg (in-
clusive) and tail constitute the content of the corresponding buffer. The type of
arraybufview_get indicates that we can only access the first element (at location
beg) in the buffer. And once we get it, we can now only access the the rest of the ele-
ments (starting from beg + 1) in the buffer. Note that the syntax !view1 >> view2
is for indicating the type checker that the type of the input argument is changed from
view1 to view2 due to the invocation. Such syntax can help save the trouble of using
a new name for each newly generated proof, which is very verbose as shown in the
sample code related to arrayview in the previous section. It is worth noting that I
use the keyword prfun to declare the function arraybufview_rewind. It indicates
that the function is purely for manipulating proofs and does not change the underling
array at all. Such function corresponds to the design that after all the elements in
the buffer have been accessed, we can add elements starting from the beginning of
the array.

The following code shows the usage of these constant functions. Readers can
compare it with the code related to arrayview to get familiar with the new syntax
for changing view. (I also write down the new view for the proof in comments after
fun foo (pf: !arraybufview (3, 0, 1)): void = let
  val x0 = arraybufview_get (pf | 0)
  // pf: arraybufview (3, 1, 1)

  val () = arraybufview_set (pf | 1, 100)
  // pf: arraybufview (3, 1, 2)

  val () = arraybufview_set (pf | 2, 100)
  // pf: arraybufview (3, 1, 3)

  val x1 = arraybufview_get (pf | 1)
  // pf: arraybufview (3, 2, 3)

  val x2 = arraybufview_get (pf | 2)
  // pf: arraybufview (3, 3, 3)

  prval () = arraybufview_rewind (pf)
  // pf: arraybufview (3, 0, 0)

  val () = arraybufview_set (pf | 0, 100)
  // pf: arraybufview (3, 0, 1)
in () end

Note that I use prval () for the pattern match of the invocation of arraybufview_rewind, which is a syntactic feature helping ATS compiler locating proof-related code for erasure after type checking.

### 2.7 Concurrent Programming with ATS

The semantics of ATS can be extended to support concurrent programming, which is shown in (Shi, 2008). To be more specific, the semantics of primitives for thread creation should ensure the transfer of linear resources from the creator thread to the
one being created without any loss or duplication. In this way, programmers can rely on linear types to do certain reasoning about global states of programs. Note that the reasoning on local objects involving dependent types remains intact due to the fact that such objects cannot be altered by other threads.

2.8 Extending ATS with Abstract Types and Constant Functions

It is a common practice, when doing ATS programming, to define new types as well as related functions according to designs of programs to be built. Such types and functions can be implemented based on existing types and functions inherent to ATS. Moreover, we can introduce them as abstract types and constant functions, as we do in the examples from previous sections. The real implementation is left external to the ATS language, e.g. in C programming language if programmers choose to compile ATS programs into C before creating final binaries. Of course, this practice amounts to unrestrained casting (from one type to another) in programming and its use requires great caution. Theoretically, each time we extend ATS by new types and related constant functions, we need to formally state their semantics and prove the properties of the type system of the extended language (e.g. type soundness). This can usually be done following standard procedures as shown in (Shi, 2008).

Some intuition for extending ATS while maintaining properties of the type system goes as follows. For dependent types, programmers should establish the relation between types (indices) and properties of data objects during the stage of design, and then implement those constant functions following the design. For instance, a function, whose type states that the return value is a negative integer, should indeed returns such a value. Some examples about extending ATS involving linear types are shown in section 2.9.
In the chapters to come in this dissertation, I add various abstract types as well as constant functions (primitives) to extend ATS into a modeling language following the aforementioned intuition. Detailed proofs for properties of the extended type system are omitted in this thesis. However, successful applications of the new modeling language circumstantially support the claim that type soundness is preserved after the extension.

2.9 Verification of ATS Programs

As we have seen in many examples in this chapter, ATS is well suited for the verification of functional properties, i.e. a function meets its specification, which is encoded in ATS via types. However such methodology for program verification has its limitation.

First, types in ATS can be viewed as formulae in first-order predicate logic. Therefore the definition of some properties becomes heavy whereas it would be immediate in a higher order language. Though we can introduce uninterpreted functions into statics to ease the specification of properties, programmers may have to manually insert proof code to help the type checker of ATS discharge those constraints generated during type checking, which can be a very heavy task. To alleviate the burden of programmers, we favour a style of Programmer-Centric Theorem-Proving (Ren and Xi, 2013). Following such style, programmers declare and use functions for manipulating proofs without actually implementing them. This is similar to constructing informal paper-and-pencil proofs (in mathematics and elsewhere). In essence, we trade the formality of verification for the efficiency of development.

Second, properties expressed via types are generally local to the function considered. But it is often necessary to prove global properties. Linear types in ATS can of help to certain extent, e.g. proving that there is no memory leak in a well-typed program. However, it is extremely heavy if not infeasible to express or verify global
properties over combination of several functions or high level temporal properties in the setting of concurrent programming, such as the absence of deadlock. I will use the following example to demonstrate such limitation.

```plaintext
abstype lock (int)
absview lock_v (int)
fun lock_acquire {x:int} (lock (x)): (lock_v (x) | void)
fun lock_release {x:int} (lock_v (x) | lock (x)): void
fun access_data {x:nat} (!lock_v (x) | int x): void
```

I introduce an abstract type constructor `lock` for locks that can be shared as global values. Accordingly, `lock_v` is for constructing views proving the acquisition of locks, which can be generated and consumed by the functions `lock_acquire` and `lock_release`. (Note that the implementation of these two functions should follow the common semantics of lock operations, i.e. if multiple threads invoke `lock_acquire` simultaneously, only one can successfully return, and the others shall be blocked till the lock is released.) The type of the function `access_data` guarantees that corresponding lock has to be acquired first before accessing certain data, thus preventing race condition in the program. And if we only use `lock_acquire` and `lock_release` to manipulating locks, we cannot release a lock before acquiring it. And the chance that we forget to release the lock after acquiring it is greatly diminished. However, the aforementioned types do not protect a thread from invoking `lock_acquire` twice consecutively, which leads to a deadlock. The modality in ATS for resource sharing (Shi et al., 2010) can help prevent this error. But it would bring severe restriction for constructing programs, e.g. a thread cannot hold two different locks simultaneously, which is a common practice in concurrent programming. To ensure that a well-typed program possesses certain global properties, it is a common practice in ATS to require that programmers only use a set of well-designed constant functions with sophisticated types to manipulate data objects. (For example, we can
set up constant functions to ensure that locks can only be acquired in certain order in order to guarantee deadlock freeness in a well-typed program.) Such method is too rigid in the sense that lots of useful programs cannot be well-typed at all. Such limitation of types-based verification is well-presented facing the problem of deadlock freeness, let alone the verification of more sophisticated temporal properties. Therefore, I choose to combine another verification technique — model checking with ATS to ease the verification of global properties.
Chapter 3

ATS/PML for Modeling Concurrent Systems

The goal of my research is to provide theories and practical tools to allow programmers to take advantage of advanced type systems when manually constructing models for practical applications as model construction is just a special form of program construction. Under such goal, I decided to start from adding advanced types to a state-of-the-art modeling language in order to attract its users to adopt type checking for detecting modeling errors at compile-time while aiming for constructing highly efficient models targeting the modeling language.

The rest of the chapter is organized as follows. First, I introduce the modeling language PROMELA, which I choose as the base language for my research, including its features, semantics, usages, and insufficiency of its type system. Then I present a new modeling language ATS/PML, which has direct root in PROMELA from the perspective of semantics, but is equipped with advanced types including dependent types and linear types grafted from the language ATS. Going further, rules for mapping between PROMELA and ATS/PML are given to illustrate the idea of augmenting the core of ATS into ATS/PML. Finally, I demonstrate via examples usages of ATS/PML and how its equipped advanced type system can facilitate the construction of faithful models.
3.1 Overview of SPIN and PROMELA

This document does not purport to be a tutorial on the SPIN model checker and the corresponding modeling language PROMELA. However, it is useful to describe certain key concepts and features of them so that the relation between PROMELA and ATS/PML can be fully explained. The SPIN model checker is a well-known verification tool for concurrent software systems. The specification language that it accepts is called PROMELA. PROMELA is an acronym for Process Meta-Language, indicating its emphasis on the modeling of process synchronization and coordination. We call a model written in PROMELA a PROMELA model, or simply a model when there is no ambiguity. PROMELA supports the specification of correctness properties inside the models via assertions and never-claims. Also, correctness properties can be specified via Linear Temporal Logic (LTL) externally. The SPIN model checker is used to verify PROMELA models against various correctness properties.

The following sections give a brief review of the PROMELA modeling language with focus on the features covered or exploited by ATS/PML. Comprehensive description of features of PROMELA as well as usage of the SPIN model checker can be found in books (Holzmann, 2003) and (Ben-Ari, 2008).

3.1.1 Modeling in PROMELA

The PROMELA modeling language is targeted to model systems consisting of processes executing asynchronously as well as interacting with each other via shared variables and message channels. A PROMELA model is constructed from three basic types of objects:

1. Processes

2. Data objects
3. Message channels

Processes

Processes are instantiations of process types (proctypes), which serves to define behaviors of a type of processes. The PROMELA language is quite similar to mainstream imperative programming languages such as C when being use to define a process type. There must be at least one proctype definition in a model. A process type is defined with the syntax:

```
proctype procname (arguments) { body }
```

The argument contains declaration of data objects local to the process. The body is a sequence of statements defining and manipulating objects such as creating a new instance of a process type, accessing a data object, sending or receiving messages via a channel, and etc.

The following code shows a definition of a process type whose behavior is to print out its input argument.

```
proctype foo (int x) {
    printf("x is %d\n", x);
}
```

There are several ways to instantiate processes in PROMELA. We can create multiple instantiations of a given proctype by adding the desired number in square brackets to the active prefix, for instance as follows:

```
active [2] proctype show_pid() {
    printf("my pid is: %d\n", _pid);
}
```

Each running process has a unique process id. These id’s are always non-negative, and are assigned in order of creation, starting at zero for the first created process.
Each process can refer to its own process id via the predefined local variable _pid. And the type of _pid is pid, which shall be discussed in more details in next sections.

Another way to instantiate new PROMELA processes is to use the predefined operator run. For instance, the last example can be rewritten as follows:

```prome
proctype show_pid() {
    printf("my pid is: %d\n", _pid);
}

init {
    run show_pid();
    run show_pid();
}
```

The init in this solution is a special process type. It has only one instantiation, which is called init process. Intuitively, this process gets executed first whenever presented in models. (I will explain more about the semantics of the model execution in 3.1.2.

### Data Objects

There are two levels of scope in PROMELA models: global and process local. Like common programming languages, an object has to be declared before it can be referenced. A data object local to a process cannot be referenced outside the process. (To be more precise, it is possible to refer to a local object of a process when specifying correctness properties with special annotation. ATS/PML does not support such feature currently.) Global objects can be referenced anywhere in models as well as correctness properties specified externally to the models.

The basic types for data objects in PROMELA are summarized in Table 3.1. The second column in Table 3.1 lists the typical range of values corresponding to each type on most machines. The precise values for the sizes as well as ranges of these
Table 3.1: Basic Types for Data Object in PROMELA

types are determined by the version of SPIN model checker. Such values are not addressed here since the basic types in ATS/PML are only mapped to those basic types in PROMELA. Programmers of ATS/PML should refer to the manuals of the SPIN model checkers they are using for such information.

In a PROMELA model, we can specify an initial value when defining a variable. Variables of basic types are initialized to zero if no initial values are given. Some examples go as follows:

```plaintext
bit x; // initially 0
bit y = 1;
int a = 24;
bool b; // initially false
mtype m; // uninitialized mtype variable
chan ch; // uninitialized message channel
```

The type `mtype` can be used to give mnemonic names to values as demonstrated by the code below:

```plaintext
mtype = {Red, Green, Blue};
mtype = {Up, Down, Left, Right}
mtype m = Left; // mtype variable, initially Left
```

Internally, each name is represented by a positive number ranging from 1 to 255.
Though we can introduce sets of names multiple times in a model as shown in the code above, all these names share the same space. Thus we can have at most 255 names in a model.

**Message Channels**

A variable of the type `chan` is used to hold a message channel. To actually create a message channel, we need to specify more details about the channel including the size of the buffer for the channel as well as the types for all fields in a message. The following code defines two variables of type `chan`.

```plaintext
chan cha = [0] of {int};
chan chb = [2] of {int, char}
```

Variable `cha` holds a channel, which can transfer messages consisting of one integer. And the buffer size is 0, which means that the `send` and `receive` operations to the channel occurs in a synchronous manner. Variable `chb` holds a channel, whose message consists of one integer and one character. The buffer size is 2, indicating two messages can be stored in the channel buffer before being received.

PROMELA supports one-dimensional array of the aforementioned basic types. We can specify one initial value for all the members of an array when defining an array of variables as shown in the code below:

```plaintext
int arrint[2] = 1; // All elements of array arrint are initialized to 1.
char arrchar[3];  // All elements of array arrchar are initialized to 0.
char arrchan[4];  // All elements of array arrchan are uninitialized.
```

**Basic Statements**

The body of a process type is comprised of a sequence of statements. There are 5 basic types of statements: assignments, assertions, print statements, send / receive statements and expression statements. Each kind of statements in PROMELA has
its semantics of executability, which provides the basic means in the language for modeling synchronization among processes. Depending on the system state, any statement in a PROMELA model is either executable or blocked.

An assignment in PROMELA is of the following form:

\[ \text{variable} = \text{expression} \]

Expressions in PROMELA are similar to those in C programming language, so as the evaluation rules including precedence, type conversion, and etc. Some of the operators in PROMELA are listed in Table 3.2, most of which are now supported in ATS/PML.

The semantics rules of PROMELA state that assignment statements are unconditionally executable.

Expressions alone can be used as statements, and are called expression statements. An expression statement is executable as long as the expression evaluates to true. (Non-zero values are converted to true automatically.) One thing worth mentioning is that it is guaranteed by PROMELA’s syntax that if an expression evaluates to false, the evaluation process does not cause any side effect.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ) [ ]</td>
<td>parentheses, array brackets</td>
</tr>
<tr>
<td>! ~ ++ --</td>
<td>negation, complement, increment, decrement</td>
</tr>
<tr>
<td>* / %</td>
<td>multiplication, division, modulo</td>
</tr>
<tr>
<td>+ -</td>
<td>addition, subtraction</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>rational operators</td>
</tr>
<tr>
<td>== !=</td>
<td>equal, unequal</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise and</td>
</tr>
<tr>
<td>^</td>
<td>bitwise exclusive or</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 3.2: Operator Precedence, High to Low**
Assertions are used to state the safety properties of models inside the models themselves and are of the following form:

\[ \text{assert}(\text{expression}) \]

An assertion is unconditionally executable. When the expression evaluates to false, the simulation or verification algorithms of the SPIN model checker would terminate and report the error trace leading to the state.

The print statements in PROMELA are similar to those in the C programming language and are unconditionally executable.

Given a variable holding a message channel, we use send/receive statements to send or receive messages from the channel. A send statement starts with a variable name, followed by the "!" symbol and a sequence of expressions matching the type of the channel. For example, in the following code we define a channel and send one message over it:

\begin{verbatim}
chan ch = [1] of {mtype, int, bool};
ch!Red, 1, false;
\end{verbatim}

By default, a send statement is only executable if the target channel is not full, and otherwise it blocks. A receive statement starts with a variable name, followed by the "?" symbol and a sequence of variables or constants. The following code shows the retrieval of a message from a message channel. The fields of a message are stored in three variables of appropriate types.

\begin{verbatim}
mtype m;
int i;
bool b;
ch?m, x, b;
\end{verbatim}

The receive statement is executable only if the source channel is non-empty. Going further, Some or all of the parameters (e.g. variables \( m, i, b \) in the code above) can be
given as constants instead of variables. For example, the following code demonstrates the retrieval of a message whose first part must be of the value `Red` and last part must be `false`.

```
ch?Red, x, false
```

In this case, there is an extra condition on the executability of the receive statement, i.e. the value of all message fields given as constants must match the value of the corresponding fields in the message that is to be received. Such feature is very useful for modeling communication protocols between processes. ATS/PML is designed to partly cover this feature and improve its usage via session types.

**Compound Statements**

As a modeling language, PROMELA contains many features that are not found in mainstream programming languages in order to facilitate the construction of high level abstract models of distributed systems. It supports many types of compound statements for the specification of the execution of multiple processes, such as atomic sequences, deterministic steps, non-deterministic selection, non-deterministic repetition, and etc.

Atomic sequence of statements in a process is executed in an uninterruptable manner. Another way to put it, if one process starts executing its atomic sequence, no other processes are allowed to execute until the running process reaches the end of the atomic sequence. For example,

```
atomic {
    x = 1;
    y = 2;
}
```

is a atomic sequence consisting of two assignment statements. One exception to such atomicity is that if one statement in the sequence blocks, other processes can take
control and start executing. When the previously blocked statement in the atomic sequence becomes executable, the control flow may go back to the original process non-deterministically, and the remaining statements in the sequence will be executed atomically as if the execution has not been interrupted at all.

Deterministic steps are similar to atomic sequence of statements except that no steps may be blocked during the execution. It is an error if a statement other than the first one becomes blocked. An example of deterministic steps is shown below:

```plaintext
d_step {
    x = 1;
    y = 2;
}
```

Appropriate usage of atomic sequence as well as deterministic steps can help reduce the size of models to be verified. I will take advantage these features of PROMELA when designing algorithms for converting models in ATS/PML into PROMELA.

Non-deterministic selection (a.k.a. if statement) allows the specification of non-deterministic branching control flow. It contains several branches, each of which has an expression (a.k.a. guard) and a sequence of statements. A branch is executable only if its guard evaluates to true. Only one branch will be selected for execution non-deterministically if many branches are executable. One special branch is the else branch, which does not have a guard. If presented, an else branch gets chosen to execute only if the guards of all other branches evaluate to false. Syntactically a non-deterministic selection is similar to an if statement in C programming language as shown in the following code:

```plaintext
if
  :: x > 0 -> printf("branch 1");
  :: x == 1 -> printf("branch 2");
  :: x < 0 -> printf("branch 3")
```
If $x$ is equal to 1, then either branch 1 or branch 2 may execute. Branch else will execute only if $x$ is equal to 0.

Non-deterministic repetition (a.k.a. do statement) is similar to the if statement. The difference is that after executing the statements in one branch, the repetition structure is repeated until the special break statement is encountered. For example,

do
:: $x > 0$ -> printf("branch 1");
:: $x == 1$ -> printf("branch 2");
:: $x < 0$ -> printf("branch 3")
:: else -> printf("branch else"); break;
od

If $x$ is always equal to 1, then branch 1 and branch 2 may be chosen non-deterministically and repeatedly for execution. When $x$ is equal to 0, branch else will execute and break out of the repetition.

**Inline Procedures**

Since PROMELA is a language for building models for verification, it lacks some features common in many programming languages, e.g. functions and pointers to data or functions. Instead, PROMELA provides the feature of inline procedure to allow programmers to encapsulate a sequence of statements in a way similar to traditional procedure call in common programming languages. For example, the following code defines an inline procedure for exchanging the values of two variables:

```c
inline exchange(x, y, temp) {
    temp = x;
    x = y;
    y = temp;
}
```
The header of an inline definition lists its parameters. During compilation, PROMELA compiler replaces each invocation of inline procedure by the corresponding inline body and replaces those parameters by the arguments of the invocation textually. There is no concept of value passing with inline procedures. As such, inline procedures are very similar to the usage of macros except one major difference. That is, errors caused by statements inside an inline procedure have line numbers corresponding to the definition of the inline procedure instead of its invocation.

3.1.2 Semantics of PROMELA Models

A PROMELA model consists of several processes, each of which consists of a sequence of statements. The operational semantics of a PROMELA model is a labeled transition system (LTS) generated by executing all the processes concurrently using certain semantics engine described in Chapter 7 of (Holzmann, 2003). Intuitively, we can view the semantics engine as a machine equipped with shared memory and message channels, which can execute those processes in a stepwise manner: non-deterministically selecting and then executing one basic statement from a single process at a time. The execution would cause effects on the state of the machine as well as the status of those processes, which correspond to the state transition in the labeled transition system corresponding to the model.

3.2 PROMELA Meets Advanced Types

3.2.1 From AT S to ATS/PML

The PROMELA modeling language requires programmers to supply type information during model construction as shown in section 3.1. However these information is not fully used during type checking and certain type errors are only detected at runtime, that is, when model checking is performed. To mitigate such problem,
ETCH (Donaldson and Gay, 2005) was introduced as an enhanced type checking tool supporting constraints-based type inference for the PROMELA language, but it is still too weak to help detect errors such as out-of-bounds array subscripting and communication protocol violation due to the limited type information provided by PROMELA models.

Type inference for advanced types such as dependent types and linear types is in general undecidable. The syntax of PROMELA (especially for the type part) is far from sufficient to allow programmers to supply sophisticated type information by annotation. Moreover, when using advanced types during programming, it is preferred that programmers can document their reasoning inline with the actual functional code via types. However it is infeasible to support such programming paradigm by reusing PROMELA’s syntax, whose main focus is to express state transition in a concise manner.

As a result, I decided not to add extra syntax into PROMELA to support advanced types. Instead, I choose to create a new modeling language, which is equipped with an advanced type system, and has semantics tightly coupled with that of PROMELA. The syntax of ATS language was designed at the first place as well as improved by feedback from practical usage to support the deployment of dependent types and linear types in practical programming. After setting constraints on ATS’ syntax, I come up with a modeling language ATS/PML, which is a subset of ATS from the perspective of syntax and types. That is, a model in ATS/PML is a valid program in ATS, but the reverse may not be true.

The basic idea of the design of ATS/PML goes as follows: for each feature of PROMELA, I choose one or a combination of several syntactic features of ATS to represent it; I also introduce several primitives (entities with special names) into ATS/PML to help cover PROMELA features difficult to represent solely relying on
ATS’s syntax. In this way, a model in ATS/PML can be mapped to a model in
PROMELA, and the later defines the semantics of the former. Note that with such
setting, the semantics of ATS/PML bears strong resemblance to the semantics of ATS
in the multi-core setting (Shi and Xi, 2009). The mapping rules are designed to ensure
that the syntactic typing rules of ATS/PML, inherited intactly from ATS is compatible
with the semantics of ATS/PML, which in turn roots in the semantics of PROMELA.
Finally, it is worth noting that the primitives I introduce into ATS/PML are assigned
well-designed types as well as special semantics concerning model checking so that
model checking and type checking can be combined synergically for system modeling
and verification.

3.2.2 Leading Example

In this section, I use Peterson’s algorithm as a concrete example to show what AT-
S/PML is like and illustrate how to construct models in it.

Figure 3.1 contains a model for the Peterson’s algorithm written in RPOMELA.
This model has two processes with identical bodies manipulating three global vari-
ables. One thing worth mentioning is the statement with an attached (*), which may
block the control flow until it evaluates to true. This little example already touches
several common issues in writing models for real applications. First of all, the be-
inning of the model contains definitions of several global variables. Some of them
(turn and flag) are part of the algorithm while others (cnt) are only for verification
purpose. Accesses to these variables are scattered across the model, making it an
error-prone style of model construction in practice. Secondly, it is assumed implicitly
that there are only two processes involved in the model which have process ids 0 and
1. An assumption as such may be broken when a model evolves. Thirdly, the code
for updating cnt is for verification purpose. The correctness of the verification may
be tempered if such code is not written correctly (e.g., an increment of cnt is not
byte cnt;
bool turn, flag[2];

active [2] proctype proc() {
    pid self, other;
    //
    // _pid: the current process id
    //
    self = _pid; other = 1 - self;
    again:
        flag[self] = true;
        turn = other;
        // Blocked till evaluating to true.
        (flag[other] == false || self == turn); // (*)
        cnt = cnt + 1;
        assert(cnt == 1);
        cnt = cnt - 1;
        flag[self] = false;
        goto again;
    }

Figure 3.1: Peterson’s algorithm modeled in RPOMELA
matched by a decrement of \( cnt \).

\[
\text{fun proctype$proc() = let}
\begin{align*}
\text{val self} &= \text{pml$mypid()} \\
\text{val other} &= 1 - \text{self} \\
\text{prval ()} &= \text{lemma_pid_scope()} // (**)
\end{align*}
\]

\[
\text{fun loop(): void = let}
\begin{align*}
\text{val ()} &= \text{flag_set(self, true)} \\
\text{val ()} &= \text{turn_set(other)} \\
\text{val ()} &= \text{pml$wait_until(}
\text{not(flag_get(other)) \text{ || (self = turn_get()))}
\text{) // pml$wait until}
\text{prval (v0 | ()) = pml$vlock_assert()}
\text{// This is a critical section}
\text{prval () = pml$vlock_release(v0)} \\
\text{val ()} &= \text{flag_set(self, false)}
\end{align*}
\]

\[
in \text{loop() end}
\]

\[
in \text{loop() end}
\]

**Figure 3.2:** Peterson’s algorithm modeled in ATS/PML

To tackle these issues, I rewrite in Figure 3.2 a corresponding model in ATS/PML that involves both dependent types and linear types. The semantics of the code should be readily accessible when one compares it directly to the previous model in PROMELA. I briefly explain some of the involved functions and types as follows.

Accessing global variables is encapsulated into the following user-defined functions:

\[
\begin{align*}
\text{fun flag_set(pid: int(mypid), v: bool): void} \\
\text{fun flag_get\{i:int|i=-0||i=1\} (pid: int(i)): bool} \\
\text{fun turn_set(pid: int(1-mypid)): void} \\
\text{fun turn_get(): [i:int|i=-0||i=1] (int(i))}
\end{align*}
\]

where each type of the form \( \text{int}(t) \) is a singleton type for the only integer value equal to \( t \).
Note that PROMELA specifies that \texttt{pid} is an implicit global variable representing the current process id. Accordingly, I introduce into ATS/PML a static (integer) term \texttt{mypid} to represent its value and a function \texttt{pml$mypid} of the following interface to obtain the current process id:

\begin{verbatim}
fun pml$mypid(): int(mypid)
\end{verbatim}

The type of \texttt{flag$_\texttt{set}} can be interpreted as requiring its first argument to be the current process id (and its second argument a boolean value). As such, the design principle is formally enforced that a process can only modify its own flag. Similarly, the function \texttt{turn$_\texttt{set}} is assigned a type which requires that the calling process should only set the turn for the other process.

The type of the argument \texttt{pid} of the function \texttt{flag$_\texttt{get}} is \texttt{int(i)} while this \texttt{i} satisfies the pre-condition stating \texttt{i = 0} or \texttt{i = 1}. In other words, the type of \texttt{flag$_\texttt{get}} implies that the function can only be applied to an integer value equaling \texttt{0} or \texttt{1}. In ATS \texttt{i} is referred to as a static term and \texttt{ind} as a dynamic expression. We can impose constraints on static terms (e.g. the code \{\texttt{i: int | i==0 || i==1}\} states that \texttt{i} is an integer equaling either \texttt{0} or \texttt{1}), which in turn restrict the dynamic expressions related to the static terms through typing. With \texttt{flag$_\texttt{get}}, a programmer can no longer write well-typed code that may potentially incur out-of-bounds array subscripting involving the array \texttt{flag}.

In ATS, we use curly braces \{\ldots\} for universal quantification and square brackets \[\ldots\] for existential quantification. The type assigned to \texttt{turn$_\texttt{get}} indicates that any value it returns must be an integer equaling either \texttt{0} or \texttt{1}.

The code for each process in the model for Peterson’s algorithm in PROMELA can essentially be translated into the function \texttt{proctype$proc} in ATS/PML (where the symbol \$ is allowed to appear in identifiers in ATS). The dynamic expressions \texttt{self} and \texttt{other} have the types \texttt{int(mypid)} and \texttt{int(1 – mypid)}, respectively.
If we removed the line with an attached (**), type checking would fail since the type checker could no longer verify the constraint that the value of other equals either 0 or 1 (when checking the invocation of function flag_get). The return type of lemma_pid_scope contains precisely what is needed to establish this constraint:

```
prfun lemma_pid_scope(): [mypid == 0 || mypid == 1] void
```

where the keyword prfun in ATS specifically indicates that lemma_pid_scope is a proof function, whose sole purpose is to help type checking. In particular, its invocation has no effect during model checking. Note that the proof function lemma_pid_scope is not implemented; its presence is primarily for making certain forms of implicit and informal reasoning more explicit and more formal. It is possible that the claim by this proof function does not hold (and we will show a way to address this issue in the next section.). Nevertheless, the use of this proof function can at least serve as a reminder to the programmer that certain care must be taken with regard to the assumption on the current process id.

The primitives pml$vlock_assert and pml$vlock_release are given the following interface:

```
fun pml$vlock_assert(): (lock_v | void)
fun pml$vlock_release(lock_v): void
```

These two functions are implemented in PROMELA (discussed in Section 3.2.3) to detect violation of mutual exclusion during model checking. Conceptually, their behavior are equivalent to acquiring and releasing a (mutex) lock given that simultaneous acquisition of the same lock (by two processes) indeed does not happen during model checking, which gives names to these two primitives.

What is really interesting here is that the return type lock_v of the primitive pml$vlock_assert is a special kind of linear type (or view as is called in ATS). This means that each value returned by pml$vlock_assert is a linear proof, which must be
consumed eventually. The primitive \textit{pml$\text{nlock\_release}$} is introduced into ATS/PML to consume a linear proof of the view \texttt{lock\_v}. If the call to \textit{pml$\text{nlock\_release}$} was omitted, then type checking (of \textit{prototype$\text{proc}$}) would fail (due to the presence of an un-consumed linear proof). Various common programming patterns involving mutual exclusion can be readily captured in ATS/PML with the use of linear types. Note that the keyword \texttt{prval} (instead of \texttt{val}) is used here, which, in ATS, indicates that the pattern match is for proof purpose and shall be erased after type checking. However, the compiler for ATS/PML recognizes that \texttt{prval} is used together with model checking related primitives (\texttt{pml$\text{nlock\_assert}$} here) prefixed by \texttt{pml$\text{n}$}, and thus treats the pattern match as if it is prefixed with \texttt{val}. Such usage of \texttt{prval} is for marking, inside a ATS/PML model, code solely for the purpose of model checking.

As the last part of this section, I would like to present a very simple (but typical) case of type-based refinement. If we change the argument \texttt{other} to \texttt{self} in the call \texttt{flag\_get(other)} in the body of the function \texttt{prototype$\text{proc}$}, the function can still pass type checking (but not model checking). As each process (in Peterson’s algorithm) always knows the value of its own flag, the sole purpose of calling \texttt{flag\_get} is to obtain the flag of the other process. This means that we can choose a more restricted type for \texttt{flag\_get}:

\begin{verbatim}
fun flag_get(pid: int(1 - mypid)): bool
\end{verbatim}

In this way, the call \texttt{flag\_get(self)} can no longer pass type checking. We often perform this kind of type-based refinement on a constructed model as a static form of debugging (in the hope to flush out potential bugs in modeling).

\section*{3.2.3 Combining Type Checking with Model Checking}

In Section 3.2.2, certain typical uses of dependent types and linear types in model construction are explained through a simple example (of modeling Peterson’s algorithm). It is expected that programmers of ATS/Veri should have no difficulty in
relating these uses of advanced types to a realistic situation where such types can help detect potential modeling errors.

When type checking a program, the type checker of ATS first collects all of the constraints that need to be verified in order to assure the well-typedness of the program and then passes these constraints to an SMT-solver to check for their satisfiability. A programmer can help the type checker discharge constraints by manually inserting proof code (e.g. invoking a call to the function lemma_pid_scope). Some proofs can be established based on (local) reasoning inside a single process, while others may have to rely on (global) assumptions across multiple processes. The latter kind of proofs are those that are often difficult to handle through type checking alone and can benefit greatly from model checking. In particular, we see that using advanced types at the stage of model construction offers an approach to guiding the use of model checking so that it can be more effectively employed in practice (e.g., targeting the verification of properties that need global reasoning).

In ATS/PML, there is a primitive pml$assert declared as follows:

\[
\text{prfun pml$assert\{b:bool\}(bool(b)):\ [b==true] \ void}
\]

This type indicates that the argument of pml$assert is a boolean expression whose value equals some static term \( b \) and this \( b \) must equal \( \text{true} \) in order for the primitive to actually return. Let us revisit the call to the proof function lemma_pid_scope in Figure 3·2. We can replace it with the following code so that the assumption obtained (from calling the proof function) can be verified through model checking:

\[
\text{prval () = pml$assert((self = 0) || (self = 1))}
\]

Note that the primitive pml$assert is indeed translated to assertions in Promela, which are frequently used by programmers. On one hand, the type of pml$assert makes it possible to take advantage of a valid assertion during the stage of type checking. On
the other, the type checker of ATS serves as a guide to locate places where assertions are needed.

So far the concept of combining type checking with model checking is demonstrated by the type and the PROMELA implementation of \texttt{pml\$assert}. The key idea is to assign meaningful types to certain PROMELA code with compatible semantics, i.e. the PROMELA code should check the validity of the properties specified via types. The implementation of \texttt{pml\$vlock\_get} and \texttt{pml\$vlock\_release} is given below to demonstrate such concept when linear types are involved.

```c
int g_lock = 0;

#define pml$lock_assert() \
 d_step { \
 assert(g_lock == 0); \
 g_lock = 1; \
 }

#define pml$lock_release () \
 g_lock = 0
```

The syntactic details about declaring functions (primitives) in ATS/PML and implementing them in PROMELA are discussed in Section 3.3.15. However, it is easy to see from the PROMELA code above that if the assertion does not fail during model checking, then there is at most one process holding the “virtual” lock represented by \texttt{g\_lock} at any time. This reflects the property of exclusion encoded via the usage of linear types.

### 3.2.4 Constructing Correct Models in ATS/PML

ATS/PML allows a programmer to naturally rely on advanced types to express ideas (on the design of the model), while staying semantically in PROMELA. On one hand, ATS/PML supports various features of PROMELA (discussed in 3.3) such as
guarded blocking, non-determinism, loops, channel operations (but certain features such as local jump are dropped). On the other hand, a program in ATS/PML is just a program in ATS as far as type checking is of the concern. Checking Figure 3.3 gives

![System Overview Diagram](image)

**Figure 3.3: System Overview**

an overview of the procedure for constructing and verifying models in ATS/PML. First, a programmer implements, in PROMELA, an application-specific library for basic operations (e.g. manipulating global objects) which are used as building blocks for the model, as well as assign meaningful types to the interfaces of these basic operations. Note that I already introduced a set of common operations into ATS/PML (e.g. `pml$assert`), which are called primitives. Some primitives are assigned advanced types (e.g. `pml$vlock_assert`) so that programmers in ATS/PML can use them directly. (Such types are known internally by the compiler from ATS/PML to PROMELA.) Some primitives (e.g. channel related primitives discussed in Section 3.3.16) have parameterized implementation in PROMELA. And ATS/PML provides the mechanism for programmers to specify different types for such primitives in order to achieve different precision. Second, based on these interfaces (primitives) with advanced types, the programmer constructs models in ATS/PML, relying on type checking to identify flaws in his or her logic reasoning. Third, after all the type
errors are eliminated, the compiler I built is used to translated the model in ATS/PML to a model in PROMELA automatically. Last, the programmer can use the SPIN model checker to verify the translated model against various properties specified internally to the model or externally via LTL formulae.

3.3 Translating ATS/PML to PROMELA

The focus of this section is to discuss about the mapping rules from ATS/PML to PROMELA. Such rules exploit various syntactic features of ATS as well as primitives I introduced. I also built a compiler according to these rules to translate models in ATS/PML into PROMELA models. Note that there are three languages involved in the sequel: PROMELA, ATS, and ATS/PML. Though ATS/PML is a sub-language of ATS from the perspective of syntax and type system, its semantics is based on the translated code in PROMELA. Certain syntax features of ATS/PML may have different terminology from their translated counterparts in PROMELA, while others may use the same terminology.

3.3.1 Processes

Process types in PROMELA define the behavior of processes. The syntax for the definition of the correspondence in ATS/PML is similar to a function definition in ATS with two constraints: 1. the function name has the prefix proctype$; 2. the return type of the function is void. The following ATS/PML code defines a process type:

\[
\text{fun proctype$foo (x: int, y: int): void = let}
\]
\[
\text{val v1 = x + y}
\]
\[
\text{val v2 = x - y}
\]
\[
in \text{end}
\]
Similar to the definition of process types in PROMELA language, a process type in ATS/PML may have several parameters annotated with types (e.g. type \texttt{int} for \texttt{x} and \texttt{y}. To accommodate the type checking rules of ATS and the semantics of PROMELA for creating new processes with copied arguments, it is required that the types for parameters in a process type in ATS/PML cannot be reference type.

Following the syntax of ATS, the body of a process type is an expression. In this example, the body is a \texttt{let} expression, which contains two sequential steps for variable definition.

### 3.3.2 \texttt{let} Expressions

I choose to use \texttt{let} expressions to represent sequential steps in PROMELA. More specifically, the code between the keyword \texttt{let} and \texttt{in} of an \texttt{let} expression is mapped to a sequence of statements in PROMELA as shown in the following example

```
1   let
2     val v1 = 1 + 2
3     var v2 = v1 - 3
4     var v3: int
5     val () = v3 := v1 + v2
6   in end
```

Both name bindings and variable definitions in ATS (line 2 to 4 in the example above) are called \textit{variable definitions} in ATS/PML. And they are mapped to variable definitions in PROMELA. Line 5 is called an \textit{assignment statement} in ATS/PML, and it is mapped to an assignment statement in PROMELA. Note that each line between the keyword \texttt{let} and \texttt{in} is called a \textit{step} in ATS/PML, and is mapped to one line in PROMELA in order to facilitate programmers to understand the semantics of a model in ATS/PML, which is rooted in the corresponding PROMELA model. To achieve such one to one mapping, the expressions to be assigned in ATS/PML cannot contain further control structures and are thus called \textit{simple expressions}.
### 3.3.3 Simple Expression

Simple expressions are mapped to expressions in PROMELA. They cannot contain `let` expressions, `case` expressions, or `if` expressions, which will be discussed soon. Similar to expressions in PROMELA, simple expressions in ATS/PML are built upon variable names, constants, operators, and function invocations. Operators listed in table 3.2 are supported in ATS/PML. Moreover simple expressions can contain primitive invocations and external function invocations, which shall be explained soon.

### 3.3.4 Print Statement

The following code demonstrates the usage of print statement in ATS/PML.

```plaintext
fun proctype$foo (): void = let
    val () = $extfcall (void, "printf", "%d + %d = %d\n", 1, 2, 1 + 2)
in end

Though print statements are usually indispensable when debugging a model, they have no effect at the stage of model checking. Therefore, I choose to represent a print statement in ATS/PML with little use of types in ATS. The keyword $extfcall is a special feature in ATS, which is used to make an external function call. Its first argument is the return type of the call, and its second argument is the name of the called function (represented as a string), and its rest of arguments are the arguments of the called function. The type checker of ATS checks neither the type of printf, nor the types of its arguments. The print statement in the example above is mapped to the following code in PROMELA.

```plaintext
printf("%d + %d = %d\n", 1, 2, 1 + 2);
```

### 3.3.5 Primitives

In this section, I use the term "primitive" to refer to those entities in ATS/PML which have special meaning for the compilation of ATS/PML. Primitives are usually used
along with syntactic features of ATS to represent features of PROMELA. Some primitives discussed here include `pml$wait_until`, `pml$run`, `pml$mypid`, `pml$assert`, `pml$randome`, and those related to channel and array operations.

### 3.3.6 Expression Statements

An expression statement in PROMELA serves as a blocking guard at the line it resides. It is represented in ATS/PML as the following:

```
val () = pml$wait_until(exp)
```

The whole line serves an *expression statement* in ATS/PML, which can be used inside a `let` expression. And the primitive `pml$wait_until` here takes the form of a function in ATS, which is of type `(bool) -> void`. The input argument must be a simple expression of type `bool`. Noting that, `pml$wait_until` has to be used with pattern match to occupy one line in ATS/PML as shown in the example. It cannot be used as an expression of type `void` (e.g. inside a simple expression). In this section, there are many primitives like `pml$wait_until` shown here, whose usages are under extra syntactic constraints besides those from their syntax and types in ATS. In the next iteration of development, I will improve the compiler for ATS/PML to provide better error reporting in order to prevent the misuses of these primitives at early stage.

### 3.3.7 Initiating Processes

We can initiate a new process in ATS/PML by invoking the primitive `pml$run` as shown in the following code example:

```
pml$run (proctype$foo (1, 2))
```

The primitive `pml$run` takes the form of a function whose type is `void -> pid`. Also it is required that the its argument has to be a valid process type with appropriate
arguments. Unlike \texttt{pml\$wait\_until}, the invocation of \texttt{pml\$run} can be used as a simple expression in ATS/PML.

### 3.3.8 Process Id

As a counterpart to the local variable \_pid in PROMELA, the primitive \texttt{pml\$mypid}, which takes the form of a function in ATS, can be invoked to get the id of the current process. Such invocation can be used as a simple expression in ATS/PML. The following example shows several usages of \texttt{pml\$mypid}.

```plaintext
fun proctype$foo (): void = let
    val id = pml$mypid ()
    val () = $extfcall (void, "printf", "id is %d\n", pml$mypid ())
    val () = pml$wait\_until (pml$mypid () > 5)
in end
```

### 3.3.9 Initialization of Models

A complete model in ATS/PML contains the definition of a special process type \texttt{pml\$init}, which corresponds to the \texttt{init} process in PROMELA. Normally, \texttt{pml\$init} contains code for the initialization of processes in a model. For example, the following model has three processes in total during model checking, one of type \texttt{pml\$init} and two of type \texttt{proctype$foo}.

```plaintext
fun proctype$foo (): void = let
    val () = $extfcall (void, "printf", "id is %d\n", pml$mypid ())
in end

fun pml$init (): void = let
    val id = pml$run (proctype$foo ())
    val _ = pml$run (proctype$foo ())
in end
```
Note that the body of the process type \texttt{pml$init} is mapped to an atomic sequence of statements. For example, the \texttt{pml$init} in the previous example is mapped to PROMELA as follows:

```plaintext
init {
  atomic {
    pid id;
    id = run foo();
    _ = run foo()
  }
}
```

It is deemed a good practice in ATS/PML to use the process \texttt{pml$init} only for the initialization of other processes. The usage of \texttt{atomic} helps remove unnecessary states caused by the interleaving of the process \texttt{pml$init} and other processes.

### 3.3.10 Assertions

Assertions in PROMELA are represented in ATS/PML by the following

```plaintext
val () = pml$assert (exp)
```

Though the primitive \texttt{pml$assert} takes the form of a function in ATS, its invocation has to be used with the corresponding pattern match. The whole line constitutes a step in ATS/PML, and the mapping in PROMELA is simply \texttt{assert(exp0)}, where \texttt{exp0} is translated expression in PROMELA for the simple expression \texttt{exp} in ATS/PML.

### 3.3.11 Non-deterministic selection

In order to accommodate both the type checking rules of ATS and semantics of PROMELA, I choose to represent non-deterministic selection in PROMELA by \texttt{case} expression in ATS with additional syntactic constraints as shown below:
case+ pml$random of
  | 0 => let
    val () = pml$wait_until (exp0) // guard
    val () = $extfcall (void, "printf", "this is branch 0\n")
in end
  | 1 => let
    val x = 3 // An assignment is also a guard.
    val () = $extfcall (void, "printf", "this is branch 1\n")
in end
  ...
  | n => let
    val () = pml$wait_until (expn) // guard
    val () = $extfcall (void, "printf", "this is branch n\n")
in end
  | _ => let
    val () = pml$wait_until (
      ~ (exp0 || ... || expn)) // guard
    val () = $extfcall (void, "printf", "this is branch else\n")
in end

The primitive pml$random is a macro in ATS defined by the following:

#define pml$random 0

Each branch of the case expression uses an integer (any value is fine) for the pattern match to meet ATS' requirement for syntax checking and type checking. It is required in ATS/PML that the body of a branch is 1. a let expression, whose first step is mapped to a guard in PROMELA, which is discussed here; or 2. a case expression tailored for send and receive operations in PROMELA, which shall be discussed later in this section. Thus the code above is mapped to PROMELA as the following:

if
  :: exp0 -> printf("this is branch 0\n")
  :: x = 3 -> printf("this is branch 1\n")
  ...

The semantics of ATS/PML indicates that the default branch (the one with _ for pattern match) is chosen only if all other branches are blocked. The type checking rules of ATS does not support such semantics for the case expression. To help type checking for the default branch, it is required that the first step of a default branch should be an expression statement containing the negation of the disjunction of guards of all other branches. This step is solely for type checking and is omitted when mapping to PROMELA. Currently, the compiler of ATS/PML does not check whether this expression statement actually encodes the assumption that all the other guards are false. Programmers may put any tautology here just as a place holder if type checking the default branch does not require the assumption. Putting a negation of tautology here would cause the type checker to accept any code in the default branch and hence shall be forbidden. In the next iteration of development, it is expected to add a new syntax to ATS to accommodate the non-determinism directly in order to eliminate the manual insertion of such extra step.

3.3.12 Deterministic Branch

An if-expression in ATS/PML is translated to an if-statement in PROMELA. For example, an if-expression in ATS/PML of the following form

\[
\text{if (exp0) then exp1 else exp2}
\]

is translated into an if-statement in PROMELA as follows.

\[
\begin{align*}
\text{if} \\
:: \text{exp0} & \rightarrow \text{statements1} \\
:: \text{else} & \rightarrow \text{statements2} \\
\text{fi}
\end{align*}
\]
where statements1 and statements2 are PROMELA code translated from exp1 and exp2.

3.3.13 Inline Procedures

*Inline procedures* in ATS/PML takes the form of functions in ATS, whose name must have the prefix `inline$`. Also such functions can only return proofs. A very simple inline procedure is shown in the following code:

```plaintext
fun inline$foo (x: int): void = let
  val y = x + 1
  val () = $extfcall (void, "printf", y is %d\n", y)
in end
```

Inline procedures in ATS/PML are mapped to those in PROMELA. The mapping rules are similar to those for mapping process types. In the example above, the input argument `x` cannot be modified inside the function body according to the type checking rules of ATS. However, it is a common practice in PROMELA to modify input arguments via assignments. To achieve this, we can use reference types in ATS for those input parameters. For example, the previous example can be modified to use reference types as the following:

```plaintext
fun inline$foo (x: &int): void = let
  val () = x := x + 1
  val () = $extfcall (void, "printf", x is %d\n", x)
in end
```

Considering that inline procedures cannot return concrete values, we can use parameters with reference type to hold the return value.

3.3.14 Repetition

Iterations in PROMELA can be represented by ATS/PML code which takes the form of (mutually) tail-recursive functions in ATS. It is required in ATS/PML that...
the names of such tail-recursive functions must have prefix `inline` to indicate that they shall be translated into PROMELA in the form of inline procedures containing repetitions. And the algorithm for such translation is similar to that for tail-call optimization.

Before placing rules about what kind of tail-recursive functions valid in ATS can be used as repetition in ATS/PML, I would start illustrating how to transform iteration in PROMELA into ATS/PML. The basic idea goes as follows: collect all the variables which are modified in the iteration; turn these variables into parameters of the recursive functions; turn the loop back into recursive-calls. The following code shows a process type contains a repetition in PROMELA.\(^1\)

```plaintext
proctype foo (int x) {
    int counter = 0;
    int sum = 0;
    loop:
        sum = sum + counter;
        counter = counter + 1;
        if
            :: counter > x -> skip
            :: else -> goto loop
        fi

        printf("sum is %d\n", sum)
}
```

And we can rewrite it in ATS/PML as the following with iteration replaced by an invocation of inline procedure:

```plaintext
fun proctype$foo (x: int): void = let
    var count = 0
    var sum = 0

\(^1\)For the sake of illustration, the code actually uses label and `goto` statement instead of `while` statement shown in Section 3.1.1 considering they are semantically equivalent in PROMELA.
fun inline$loop (c: &int, s: &int): void = let
  val () = s := s + c
  val () = c := c + 1
in
  if c > x then ()
  else inline$loop (c, s)
end
val () = inline$loop (count, sum)

val () = $extfcall (void, "printf", "sum is %d\n", sum)
in end

Note that all the parameters in ATS/PML corresponding to updated variables in
PROMELA are of reference types. Also the function inline$loop is defined as an
inner function inside proctype$foo so that the former can access the value x without
declaring it as a parameter.

Going further, PROMELA code involving multiple labels and goto statements
can be represented by mutually tail-recursive functions in ATS/PML with each label
mapped to one recursive function and each goto statement mapped to one recursive-
call. For example, the PROMELA code in Figure 3·4 can be rewritten in ATS/PML
as shown in Figure 3·5.

It is fair to say that though ATS/PML does not support goto statements, we can
achieve similar functionality via the usage of mutually tail-recursive functions. The
algorithm for translating such functions in ATS/PML into PROMELA is illustrated
as follows. In ATS/PML, mutually tail-recursive functions are defined as a group as
shown in Figure 3·6. Each function in the group may be mapped to an inline pro-
cedure in PROMELA depending on whether such function is actually called outside
the definition group. Without losing generality, we assume that inline$foo_1 is
mapped into PROMELA procedure foo_1. The body of foo_1 contains n blocks
of PROMELA code, each of which corresponds to a function in the group, as well as
proctype foo (int n) {
    int x, y = 0;
    incx:
    if
      :: x > n -> goto end
      :: else -> x = x + 1; goto incy
    fi

    incy:
    y = y + 1;
    goto incx

    end:
}

**Figure 3.4:** PROMELA code with labels and goto statements

fun proctype$foo (n: int): void = let
    var x: int = 0
    var y: int = 0

    fun inline$incx (x1: &int, y1: &int): void =
        if x1 > n then ()
        else let
            val () = x1 := x1 + 1
        in
            inline$incy (x1, y1)
        end
    and inline$incy (x2: &int, y2: &int): void = let
        val () = y2 := y2 + 1
    in
        inline$incx (x2, y2)
    end

    val () = inline$incx (x, y)
in end

**Figure 3.5:** ATS/PML code with mutually tail-recursive functions
fun inline$foo_1 (parameters): void = 
  // body_1 
and 
inline$foo_2 (parameters): void = 
  // body_2 
... 
and 
inline$foo_n (parameters): void = 
  // body_n 

Figure 3-6: ATS/PML syntax for mutually recursive function group

a special label indicating the end of the procedure. The structure of the procedure
foo_1 can be roughly described as follows:

// body of procedure foo_1 
block_1  // sequence of code 
block_2 
... 
block_n 
foo_1_end:  // label indicating end of procedure 

The beginning of each block is a label uniquely generated from the correspond-
ing function. The second part of the block is a sequence of PROMELA statements 
translated from the body of the corresponding function as if it is a normal procedure 
in ATS/PML except 1. those tail-recursive calls are replaced by goto statements to 
those labels previously generated from corresponding functions; 2. all parameters of 
reference types are renamed, which shall be explain in detail later in this section. The 
rest of the block is simply a goto statement to the end of the procedure.

Without losing generality, the following code roughly shows the content of block_1.

// content of block_1 
label_foo_1:  // label generated from corresponding function 

sequence of PROMELA statements containing "goto"
To order to be able to rename all the parameters of reference types in the group, it is required that all the functions in a group must have the same amount of parameters which are of reference types. Also all tail-recursive function calls have to use these parameters in the same ordering as they are declared in function headers. As such we can rename those parameters of reference types in different functions by the same set of names as mentioned previously. Otherwise, the algorithm simply does not work and reports error. For example, we cannot write $\texttt{inline\$incy (y1, x1)}$ in function $\texttt{inline\$incx}$ in Figure 3·5 since $x1$ is declared before $y1$.

Using such algorithm, the code in Figure 3·5 can be translated in PROMELA as follows:

```plaintext
inline incx(x, y) {
    skip;
    incx_: 
    if 
        :: (x) > (n) ->
        :: else ->
            x = (x) + (1);
            goto incy_
    fi;
    goto incx_end;
incy_: 
    y = (y) + (1);
    goto incx_; 
incx_end: }
```

```plaintext
proctype foo(int n) {
    int x;
    int y;
}
```
\begin{verbatim}
x = 0;
y = 0;
incx(x, y);
\end{verbatim}

It is fair to say that the generated code above is close to the PROMELA model in Figure 3.4 with similar number of variables as well as steps in the process type. This is the result I want to achieve i.e. 1. ATS/PML should be as expressive as PROMELA so that models in PROMELA can be rewritten in ATS/PML with equivalent semantics; 2. Model checking models generated from ATS/PML should be as efficient as model checking the original models written in PROMELA directly. Superficially, it may seem that writing models in ATS/PML over PROMELA is simply a choice of syntax style. However, such syntax change can greatly facilitate programmers to reason about models they are writing with the help of advanced type system provided by ATS. I will illustrate more about this idea later in this chapter.

### 3.3.15 Interaction with PROMELA

It is required that a model in ATS/PML is written in a single file. Such file can also contain PROMELA code enclosed by the symbols `% (opening) and % (closing). Such PROMELA code are called embedded code and is pasted into the generated PROMELA code at an unspecified position.

PROMELA macros and inline procedures defined in embedded code can be accessed in ATS/PML code in the same file if appropriate interfaces are provided for them in ATS/PML. Such interfaces take the form of function declarations in ATS. The following code demonstrates how such feature is used in ATS/PML.

\begin{verbatim}
{%
int g = 1;

#define get_g() g
\end{verbatim}
inline get_g2(x) {
    x = g
}

inline set_g(x) {
    g = x
%
}

// External function declarations
extern fun get_g (): int
extern fun get_g2 (x: &int): void
extern fun set_g (x: int): void

fun proctype$foo (): void = let
    val lg1 = get_g ()
    val () = $extfcall (void, "printf", "g is %d\n", lg1)
    var lg2: int = 0
    val () = get_g2 (lg2)
    val () = pml$assert0 (lg1 = lg2)
    val () = set_g (lg1 + lg2)
in end

The embedded PROMELA code in the example above contains the definition of a
global variable g as well as a macro and two inline procedures for accessing g. Interfaces in ATS/PML for these macro and inline procedures are in the form of external
function declarations as shown in the example. These function declarations must have
the same names as their counterparts in PROMELA. The invocation of these external
functions in ATS/PML is translated into invocation of macro or inline procedures (with same names) in PROMELA. For example, the process type proctype$foo in the example above is translated into the PROMELA code shown in Figure 3-7.

It is preferred in ATS/PML to define global variables as well as related access
proctype foo_0() {
    int lg1;
    int lg2;
    lg1 = get_g(); // macro invocation
    printf("g is %d\n", lg1);
    lg2 = 0;
    get_g2(lg2); // inline procedure invocation
    assert((lg1) == (lg2));
    set_g((lg1) + (lg2)) // inline procedure invocation
}

Figure 3.7: Sample code for interaction with PROMELA

functions (or macros) in PROMELA and declare interfaces for them in ATS/PML. Such interfaces can be given informative types in ATS to help programmers reason about models. However, such feature should also be used with caution since there is no check against whether the type of an interface actually matches its implementation in PROMELA.

3.3.16 Channel Operations

PROMELA has limited support for types for communication channels in the way that programmers cannot fully specify types for channels at will. This is the problem I want to address when designing features for ATS/PML to cover channel related features in PROMELA.

To begin with, Figure 3.8 lists basic primitives for channel operations in ATS/PML. Corresponding to the channel type chan in PROMELA, I introduce into ATS/PML the type pml$chan as a channel type in the form of an abstract type in ATS. The rest of the primitives are for creating channels, sending message over channels, receiving message from channels, and checking the emptiness of channels, as indicated by their names. Note that the primitives pml$chan_create$, pml$chan_recv$, and pml$chan_send$ take the form of function template in ATS with the type of mes-
abstype pml$chan

fun {a:vt@ype} pml$chan_create$ (  
    int (*buffer size, must be constant when invoked*)): pml$chan
fun {a:vt@ype} pml$chan_send$ (ch: pml$chan, ele: a): void
fun {a:vt@ype} pml$chan_recv$ (ch: pml$chan): a
fun pml$chan_isempty$ {a:vt@ype} (ch: !a): bool
fun pml$chan_isnotempty$ {a:vt@ype} (ch: !a): bool

Figure 3·8: Basic Channel Operations

sages over channels specified as the template parameter. The compiler must know the  
real values (types) for these template parameters at invocations of these primitives in  
order to generate PROMELA code. Usually, such type information can be inferred by  
the compiler, Also programmers can help compiler by manually providing template  
arguments. Currently, only basic types (e.g. int), channel types (explained in this  
section), and datatypes in ATS/PML can be used for messages over channels.

Besides the aforementioned type related constraints, the usage of these primitives has to satisfy certain syntactic constraints. First, the first argument of the  
primitive pml$chan_create$ must be a constant due to the fact that the size of  
the channel buffer in PROMELA has to be declared statically. Second, the primitive pml$chan_recv has to be used with case expression in ATS. For example, the  
following code in PROMELA

```
proctype foo() {
    chan ch = [1] of {int};
    int x;
    ch!5;
    ch?x;
    printf("x is %d\n", x);
    ...... // more statements
}
```

shall be rewritten in ATS/PML as the following:
fun proctype$foo () = let
val ch = pml$chan_create$<int>(1)
val () = pml$chan_send$<int>(ch, 5)
in
  case pml$chan_recv$<int> (ch) of
    | x => let
      val () = $extfcall (void, "printf", "x is %d\n", x)
... // more statements
  in end
end

The case expression containing pml$chan_recv$ can have only one pattern match
(x in the previous example). And the body of matching clause can be any valid ex-
pression in ATS/PML (let expression in this example).

In PROMELA, we can specify that specific messages are expected. Such feature
can be achieved in ATS/PML by using constants in a pattern or appending when
condition to a pattern match. For example, the following ATS/PML code

fun proctype$foo () = let
val ch = pml$chan_create$<int>(1)
val y = 1 + 2
val () = pml$chan_send$<int>(ch, 1)
in
  case- pml$chan_recv$<int> (ch) of
    | 1 (* constant as a pattern *) => let
      val () = pml$chan_send$<int>(ch, 6)
in
    case- pml$chan_recv$<int> (ch) of
      | x when x = y + 3 (* "when" condition *) => let
        val () = $extfcall (void, "printf", "end\n")
in end
  end
end

is translated into PROMELA as the following:
active proctype foo1() {
    chan ch = [1] of {int};
    int y = 1 + 2;
    ch!1;
    ch?1;
    ch!6;
    ch?eval(y + 3);
    printf("end\n")
}

In this example, the first invocation of pml$chan_recv uses a constant 1 as the pattern, while the second uses a when expression after the pattern. Note that we have to use the keyword case- here to inform the type checker of ATS not to check that pattern match is exhaustive since we only intend to receive special messages.

So far, the primitives presented above in ATS/PML do not provide more type safety than PROMELA does considering that programmers can specify different types for messages when creating a channel and sending (or receiving) messages over the channel. The purpose of providing these primitives is to allow programmers to come up a rough model at first place. Then they can refine the types used in the model and rely on the type checker of ATS to detect potential errors. As such, types are used as a tool for statically debugging models. Such methodology is greatly favoured in ATS programming, and can also be adopted in ATS/PML. Following such methodology, a programmers may declare his or her own set of primitives in ATS/PML for channel operations as follows:

abstype chanref(a:vt@ype) = pml$chan

extern fun {a:vt@ype} pml$chan_create$chanref (n: int): chanref(a)
extern fun {a:vt@ype} pml$chan_send$chanref (chanref(a), a): void
extern fun {a:vt@ype} pml$chan_recv$chanref (chanref(a)): a

Using the keyword abstype, a new type constructor chanref is declared. With
such type constructor, a channel, the messages over which are of type \( a \), can be given the type \( \text{chanref}(a) \). (Note that the right hand side of the equation (= \( \text{pml}\$\text{chan} \)) is to inform the compiler of ATS/PML that the constructor \( \text{chanref} \) is actually for creating channel types. And the type checker of ATS does not use such information at all.) The types for the primitives \( \text{pml}\$\text{chan}_\text{send}$chanref and \( \text{pml}\$\text{chan}_\text{recv}$chanref\) enforce that only messages of type \( a \) can be transferred over channels of type \( \text{chanref}(a) \).

When introducing such primitives into ATS/PML, programmers have to follow the following rules in order to help the compiler of ATS/PML to figure out their semantics and generate appropriate code for their invocations.

First, the newly introduced primitives should have names with certain prefix, e.g. \( \text{pml}\$\text{chan}_\text{create}$\) for channel creation, \( \text{pml}\$\text{chan}_\text{send}$\) for send operation, and \( \text{pml}\$\text{chan}_\text{recv}$\) for receive operation.

Second, the ordering of parameters as well as their types have to be arranged appropriately. Rules for such arrangement are listed below.

- Creating Channels (\( \text{pml}\$\text{chan}_\text{create}$\))

  The input parameter of the primitive is for the size of the channel buffer, thus its type should be compatible with numerical values (e.g. \( \text{int} \)). And the input argument has to be a constant due to the fact that the size of the channel buffer in PROMELA has to be declared statically. Also the type for the return value has to be a channel type, which is either \( \text{pml}\$\text{chan} \) or equal to it.

- Sending Messages (\( \text{pml}\$\text{chan}_\text{send}$\)) The type for the first parameter must be of channel type. The type for the second parameter is used as the type for the messages being sent. As mentioned before, only basic types, channel types, and datatypes are supported. In primitive \( \text{pml}\$\text{chan}_\text{send}$chanref, the type
is represented by the template parameter. Thus programmers may have to manually specify it when invoking the primitive.

- Receiving Messages (\texttt{pml$chan\_recv$}) The type for the first parameter must be of channel type. The type for the return value is used as the type for the messages being received. And the same rules mentioned in Sending Messages apply to it as well.

Last, the usage of these user-introduced primitives should satisfy the same constraints as those on the usage of basic primitives discussed previously.

The channel type constructor \texttt{chanref} is just one of the many possible ways to assign types to channels. The point is that there is no single set of primitives with types applicable to all scenarios. For instance, with \texttt{chanref}, there is no way to define a channel, whose messages contain channels of the same type (e.g. the channel itself). Such usage is legitimate in PROMELA, and has been employed in realistic PROMELA models, e.g. a model of a telephone system (Calder and Miller, 2001). This problem can be solved in ATS/PML by declaring the following primitives:

```
abstype chanrecur = pml$chan

extern fun pml$chan\_create$chanrecur (n: int): chanrecur
extern fun pml$chan\_send$chanrecur (chanrecur, chanrecur): void
extern fun pml$chan\_recv$chanrecur (chanrecur): chanrecur
```

It is suggested that a programmer sets up different sets of primitives with appropriate types for different kinds of channels. And primitives with well-designed types can be saved for reuse. I will illustrate more about this via examples in the Section 3.4.
3.3.17 Array Operations

The design of array operations in ATS/PML to cover features in PROMELA is very similar to that of channel operations. The basic primitives for array operations are listed below.

```
abstype pml$array

fun {a: vt@ype} pml$array_create$ (sz: int (* array size, must be constant when invoked *), ele: a // initial value): pml$array

fun pml$array_get$ {a: vt@ype} (arr: pml$array, n: int): a

fun pml$array_set$ {a: vt@ype} (arr: pml$array, n: int, ele: a): void
```

Corresponding to the array type in PROMELA, I introduce into ATS/PML the type `pml$array` as an array type in the form of an abstract type in ATS. The rest of the primitives are for creation, subscription, and update of arrays, as indicated by their names. Note that the primitive `pml$array_create` take the form of function template in ATS with the type of array elements specified as the template parameter. Similar to the usage of those basic primitives for channel operations, programmers may need to specify explicitly the template argument when invoking the primitive. Currently, only basic types (e.g. int) and channel types in ATS/PML can be used for elements of arrays. Especially, array of arrays is not supported currently.

Going further, the usage of these primitives has to satisfy the following syntactic constraints. First, the primitive `pml$array_create$` has to be used with pattern match of a single name. For example, the following code

```
val x = pml$array_create$<int> (3, 0)
```
is a correct usage, which is mapped into a PROMELA statement
\[
\text{int } x[3] = 0
\]
Also, the first argument of the primitive must be a constant. Second, the primitive \texttt{pml$array$set$} has to be used with empty pattern match, such as the following:
\[
\text{val } () = \texttt{pml$array$set$}(x, 2, 1)
\]
Similar to primitives for channel operations, programmers can declare own set of primitives for array operations following naming conventions as well as certain syntactic rules. Then the compiler of ATS/PML can generate PROMELA code for the invocation of these primitives. For example, a programmer may declare his or her own set of primitives in ATS/PML for array operations as follows:
\[
\text{abstype arrayref (a:vt@ype) = pml$array$
\]
\[
\text{extern fun } \{a: \text{vt@ype}\} \texttt{pml$array$create$arrayref$} (\texttt{int, ele: a}): \text{arrayref } (a)
\]
\[
\text{extern fun } \texttt{pml$array$get$arrayref$} \{a: \text{vt@ype}\} (\texttt{arr: arrayref a, n: int}): a
\]
\[
\text{extern fun } \texttt{pml$array=set$arrayref$} \{a: \text{vt@ype}\} (\texttt{arr: arrayref a, n: int, ele: a}): \text{void}
\]
Using the keyword \texttt{abstype}, a new type constructor \texttt{arrayref} is declared. With such type constructor, an array containing elements of type \texttt{a}, can be given the type \texttt{arrayref(a)}. (Note that the right hand side of the equation (\texttt{= pml$array$}) is to inform the compiler of ATS/PML that the constructor \texttt{arrayref} is actually for creating array types. And the type checker of ATS does not use such information at all.) The types for the primitives \texttt{pml$array$get$arrayref$} and \texttt{pml$array=set$arrayref$} enforce that only elements of type \texttt{a} can be stored in arrays of type \texttt{arrayref(a)}.

When introducing such primitives into ATS/PML, programmers have to follow the following rules in order to help the compiler of ATS/PML to figure out their semantics and generate appropriate code for their invocations.
First, the newly introduced primitives should have names with certain prefix, i.e., \texttt{pml\_array\_create} for array creation, \texttt{pml\_array\_get} for subscription, and \texttt{pml\_array\_set} for array update.

Second, the ordering of parameters as well as their types have to be arranged appropriately. Rules for such arrangement are listed below.

- **Creating Arrays (pml\_array\_create)**

  The first parameter of the primitive is for the size of the array, thus its type should be compatible with numerical values (e.g. \texttt{int}). And the corresponding argument has to be a constant. The second parameter is used to initialize the content of the array. Thus its type must be compatible with the type of the elements of the array. Also the type for the return value has to be a array type, which is either \texttt{pml\_array} or equal to it.

- **Subscripting Array (pml\_array\_get)** The type for the first parameter must be of array type. The type for the second parameter is used as the index to access an element inside the array, thus its type should be compatible with numerical values (e.g. \texttt{int}). The type of the return value is used as the type for the elements inside the array. As mentioned before, only basic types, and channel types are supported for array element. In primitive \texttt{pml\_array\_get\_arrayref}, the type is represented by the template parameter. Thus programmers may have to manually specify it when invoking the primitive.

- **Updating Array (pml\_array\_set)** The type for the first parameter must be of array type. The type for the second parameter is used as the index to access an element inside the array, thus its type should be compatible with numerical values (e.g. \texttt{int}). The type for the third parameter is used as the type for the array elements. And the same rules mentioned in the paragraph of Subscripting
Array apply to it as well.

Last, the usage of these user introduced primitives should satisfy the same constraints as those on the usage of basic primitives discussed previously.

### 3.3.18 Mapping Types in ATS to PROMELA

#### Basic Types

ATS/PML supports the following basic types `int`, `int n`, `bool`, `bool b`, `uchar`, and `uchar n` given that `n`, `b` is of sort `int` and `bool` respectively. During translation, both `int` and `int n` are mapped to `int` in PROMELA, both `bool` and `bool b` are mapped to `bool` in PROMELA, and both `uchar` and `uchar n` are mapped to byte in PROMELA. The type for process id in ATS/PML is `int n` given that `n` is of sort `int`, and such type is mapped to `int` in PROMELA, and the PROMELA compiler shall do necessary conversion during compilation of PROMELA models before verification considering the type for process id in PROMELA is `pid` instead of `int`.

#### Channel Types

Any channel type in ATS is mapped to the type `chan` in PROMELA.

#### Array Types

Appropriate array types in PROMELA are generated when translating invocations of the function `pml$array_create$` in ATS/PML to PROMELA, which are the only places where array types in PROMELA are needed. (This is due to the current implementation that in ATS/PML, it is not allowed to use an array as an argument for a process, or the content of a message, or an element in another array.)
Datatypes

`datatype` cannot be used directly as types for variables in ATS/PML. However, a datatype can be used to describe the payload of a channel. I will discuss such usage of datatype in ATS/PML in Section 3.4.2.

### 3.4 Using Advanced Types for Modeling: Case Study

#### 3.4.1 Dependent Types and Linear Types

My work is partly motivated by a previous research on verifying an interrupt-driven Slats and Flaps Control Unit Software programmed in C via model checking (Chen et al., 2015). The authors of the paper took part of the C code of the control unit, which had passed the unit testing stage and rewrote it in PROMELA so that model checking techniques can be applied to find slipped faults. The objective of the research is to identify errors rather than to prove correctness. Abstraction was made to reduce the size of the model which may cause some errors to be ignored or missed. However, it is ensured that the discovered errors were real errors in the C code. In the paper, their main objective is to check the correctness of the algorithms used in the buffer operations, which are widely used by various components in aircraft software, and are among the most complex and error-prone behaviors of aircraft software. In the verification, a total of four flawed code fragments was identified, including a minor efficiency issue, which is regarded as a very successful result according to the programming team.

In their PROMELA, an array-based buffer is manipulated by two interrupt-driven modules. When each module is working, it has exclusive control of all the global variables in the model. Thus we can reason about the states of the buffer and many other global variables locally via types without being affected by other processes. This is where we can resort to types for the verification of functional behaviors of the
code. Therefore, I took some of the PROMELA code and rewrote it in ATS/PML to see whether same errors can be identified using those advanced types offered by ATS/PML.

The buffer involved is implemented in the form of a segment in a global array of fixed size. Two global variables are used to indicate the beginning and ending (exclusive) indices for the buffer inside the array. (Note that the segment cannot loop around, which means that the beginning index cannot be greater than the ending index.) On one hand, an environment input module frequently adds data to the end of the buffer and increases the ending index accordingly. If the ending index reaches the end of the array, then appropriate operations have to be taken to safely discard the incoming data. On the other hand, a periodic control module reads and processes the data inside the buffer starting from the beginning of the array. Data is processed in frame size (8 bytes). If the remaining data is less than one frame, it will be moved to the beginning of the array. The periodic control module updates the ending index accordingly and then exits from the interrupt. Figure 3-9 shows the change of states of the buffer as well as other global variables involved caused by the invocation of the periodic control module.

Figure 3-9: Change of States by Periodic Control Module

In this section, I will illustrate the process of modeling the periodic control unit in
ATS/PML. To beginning with, the types for the basic operations for manipulating global variables are shown in Figure 3-10. I introduce two constants of sort $\text{gid}$ in the statics of ATS/PML to represent the identifiers for two global variables $\text{read}$ and $\text{len}$. The view $\text{gv (id, n)}$ states that a global variable whose identifier is $\text{id}$ stores an integer of value $n$. A value of such view serves as a linear proof for the statement. The view $\text{arraybufview (len, beg, tail)}$ is used to describe the state of the array-based buffer given that $\text{len}$, $\text{beg}$, and $\text{tail}$ are of sort $\text{int}$. The two functions $\text{inline$process\_frame$}$ and $\text{inline$move\_data$}$ are for processing one frame of data from the buffer and moving an incomplete frame of data to the beginning of the array respectively. These two functions can be implemented in PROMELA directly, or based on some more basic operations (e.g. $\text{arraybufview\_get}$) with advanced types, as shown in Section 2.6. The type of the function $\text{inline$ReceiveData$}$ represents the changes of states of the buffer as well as related global variables storing indices, which are caused by the operation of the periodic control module.

Based on the interfaces for basic operations including $\text{inline$process\_frame$}$ and $\text{inline$move\_data$}$, the function $\text{inline$ReceiveData$}$ can be implemented as shown in Figure 3-11. If we follow the PROMELA code modeling the periodic control module faithfully, the line with (*) should not be commented out. However, the type checker of ATS would issue a type error if we do so. Clearly, such extra increase of the buffer index is a bug in the code. The usage of dependent types and linear types of ATS helps us detect such bug and identify its cause. Readers may argue that it is easy to notice such repetition of code in consecutive lines in a code review. This is not true here for the following reasons. First, there do exist some extra operations between the two consecutive lines which I have omitted here since they are unrelated to the buffer operation. Second, in the original PROMELA code, which is in imperative style, the commented line is far away outside an if statement. All these make it very
# define DATA_FRAME_LENGTH 8

sortdef gid = int
stacst g_read: int
stacst g_len: int

absview gv (gid, int)

extern fun gv_get {id: gid} {x0: int} (!gv (id, x0)): int x0
extern fun gv_set {id: gid} {x0, x: int} (  
  !gv (id, x0) >> gv (id, x) | int x): void

absview arraybufview (len: int, beg:int, tail: int)

extern fun inline$process_frame
{arrlen, beg, tail: nat
 | tail <= arrlen; tail - beg >= DATA_FRAME_LENGTH
} (pf_buf: !arraybufview (arrlen, beg, tail)  
  >> arraybufview (arrlen, beg + 8, tail)  
  | beg: int beg  
): bool

extern fun inline$move_data
{arrlen, beg, tail: nat
 | tail <= arrlen; beg <= tail; tail < beg + 8
} (pf_buf: !arraybufview (arrlen, beg, tail)  
  >> arraybufview (arrlen, 0, tail - beg)  
  | beg: int beg, tail: int tail
): void

extern fun inline$ReceiveData
{tail1:nat} {arrlen:nat | tail1 <= arrlen} (  
  pf_read: !gv (g_read, 0) >> gv (g_read, beg2)  
, pf_len: !gv (g_len, tail1) >> gv (g_len, tail2)  
, pf_buf: !arraybufview (arrlen, 0, tail1)  
  >> arraybufview (arrlen, 0, tail2)):  
# [beg2, tail2: nat  
| tail2 < DATA_FRAME_LENGTH;  
(tail1 - tail2) % DATA_FRAME_LENGTH == 0] void

Figure 3.10: Advanced Types for Basic Operations
implement inline$ReceiveData {tail1} {arrlen}(pf_read,pf_len,pf_buf) = let
val tail = gv_get (pf_len) in
if tail < DATA_FRAME_LENGTH then ()
else let
  fun loop {beg1, tail1: nat
  | tail1 - beg1 >= DATA_FRAME_LENGTH; tail1 <= arrlen} (pf_read: !gv (g_read, beg1) >> gv (g_read, beg2),
    pf_len: !gv (g_len, tail1) >> gv (g_len, tail2),
    pf_buf: !arraybufview (arrlen, beg1, tail1)
      >> arraybufview (arrlen, 0, tail2)) : #[beg2, tail2: nat
  | tail2 < DATA_FRAME_LENGTH;
    (tail1 - beg1 - tail2) % DATA_FRAME_LENGTH == 0] void =
  if inline$process_frame (pf_buf | gv_get (pf_read)) then let
    val () = gv_set (pf_read | gv_get (pf_read) + 8)
    val tail = gv_get (pf_len)
    val tail1 = tail - gv_get (pf_read) in
    if tail1 >= DATA_FRAME_LENGTH then loop (pf_read, pf_len, pf_buf)
    else let
      val () = inline$move_data (pf_buf | gv_get (pf_read), tail)
      val () = gv_set (pf_len | tail1)
    in end
  else let
    val () = gv_set (pf_read | 0)
    // val () = gv_set (pf_read | 0) // (*)
    val tail = gv_get (pf_len)
    val tail1 = tail - gv_get (pf_read) in
    if tail1 >= DATA_FRAME_LENGTH then loop (pf_read, pf_len, pf_buf)
    else let
      val () = inline$move_data (pf_buf | gv_get (pf_read), tail)
      val () = gv_set (pf_len | tail1)
    in end
  end
val () = gv_set (pf_read | 0)
in loop (pf_read, pf_len, pf_buf) end
end

Figure 3.11: Modeling Operation of Periodic Control Module
difficult for a programmer to identify such flaw in the logic reasoning in the original PROMELA code.

While it is easy to find the cause of an error via type checking, it is very difficult to diagnose a bug in model checking. In this example, the same error was also found in the previous research (Chen et al., 2015) via model checking. However, the original problem turns into an integer overflow, and is then captured by the model checker as an array out-of-bound subscription error. It usually takes a great effort to find the cause of an error solely based on the trace provided by a model checker.

Going further, another bug identified by the previous research can also be signified by the application of types in ATS/PML. Initially, I assigned certain linear type to the elements in the buffer to help prevent the loss of data. However, I could not implement the model in ATS/PML without using certain unsafe type casting. This usually serves as a warning to programmers for introducing potential bugs. In this example, this reminds programmers to take care of the data, which is simply thrown away in the original PROMELA code when the buffer is full. Such careless management of data in the PROMELA code leads to an error that a fake frame composed by two incomplete frames may pass the sanity check and be processed by the system.

### 3.4.2 Session Types

Despite the wide usage of channels in various models, channel types in PROMELA are not fully specified. Simply put, the type `chan` in PROMELA does not contain information about messages being transferred via the channel. In ATS/PML, we can define advanced channel types as well as related primitives with meaningful types as shown in Section 3.3.16. Going further, datatype in ATS is well-suited for describing the content of a message, which usually consists of multiple parts of different types. For example, a server may receive a message of type `{mtype, chan}` in PROMELA. In ATS/PML, we can define a datatype for such message as follows.
datatype
ServerOpt =
  | RETURN of (channel0)
  | REQUEST of (channeg1(ss_client))

channel0 and channeg1(ss_client) are two user-defined channel types, which
will be explained later in the sequel, and which are more informative than the channel
type in PROMELA. Assume server is a channel of type chanref (ServerOpt) (dis-
cussed in Section 3.3.16). The following code demonstrates how to receive a message
from server.

case pml$random of
  | 0 => (case- pml$chan_recv$chanref (server)
                         of ~RETURN (agent) => ...... // operations after receiving
                     )
  | 1 => (case- pml$chan_recv$chanref (server)
                         of ~REQUEST (client) => ...... // operations after receiving
                     )

PROMELA models are usually used to verify properties of communication proto-
cols. It is difficult to tell whether the PROMELA models actually implement the
protocol. chanref mentioned above is far from sufficient to detect errors in the
implementation of models. Let us use a generic client-server model adapted from
Chapter 15 of (Holzmann, 2003) to demonstrate this problem. Figure 3·12 shows
the PROMELA code for this model, which is annotated with asterisks which are for
discussion purposes, and should otherwise be ignored.

We can rewrite the model in ATS/PML using chanref for channel types. Some
errors, e.g. exchanging the usage of agent and client in the process Agent, can be cap-
tured by type checking. However, some errors related to the sequence of interactions
via channels cannot be captured. For instance, if we append a statement of break to
the statement client!hold(agent) at (*) in Agent process, the Client process will
mtype = { request, deny, hold, grant, return0 };  
chan server = [0] of { mtype, chan };  
proctype Agent(chan agent; chan client) {  
do  
:: client!hold(agent)  
:: client!deny(agent) -> break (**)
:: client!grant(agent) ->  
    wait: agent?return0; break  
od;  
server!return0(agent)  
}  
proctype Server() {  
    chan client, agent;  
do  
:: server?request(client) ->  
    if  
    :: agent_not_available() -> client!deny(0)  
    :: agent_available() ->  
        agent = get_agent();  
        run Agent(agent,client)  
    fi  
:: server?return0(agent) ->  
    put_agent(agent)  
od  
}  
proctype Client()  
{  
    chan me = [0] of { mtype, chan };  
    chan agent;  
do  
:: timeout ->  
    server!request(me);  
do  
:: me?hold(agent)  
:: me?deny(agent) -> break  
:: me?grant(agent) -> agent!return0; break  
od  
}  

Figure 3.12: Agent, Client, and Server Processes
be listening on the channel *me* forever. On the contrary, if we remove the `break` from the statement `client!deny(agent) \rightarrow break` at (***) in *Agent* process, the *Agent* process will be waiting on the channel *client* for sending in the next iteration, while another agent may start using the *client* channel, which is not a correct behavior according to the original design.

Session types (Dezani-Ciancaglini and de'Liguoro, 2010) are one of the formalisms that have been proposed to structure interaction and reason over communicating processes and their behaviour. The usage of channels in the example follows the principle of session types, though not very strictly. Now I will illustrate the process of exploiting session types in ATS/PML to implement this model, relying on type checking to detect those aforementioned errors and many other similar ones. The same methodology can be applied to construct many other models for communication among processes.

The basic idea for session types goes as follows. A session is a sequence of interactions between two concurrently running processes, and a session type is a form of type for specifying (or classifying) such interactions. In the client-server example here, we define a type `channel0` in the ATS/PML code below for an unidirectional channel, connecting two processes.

```plaintext
absvtype channel0 = pml$chan // channel type
// endpoint for receiving
absvtype chanpos1(ss:vt@ype) = pml$chan // channel type
// endpoint for sending
absvtype channeg1(ss:vt@ype) = pml$chan // channel type

// channel creation
extern fun {a:vt@ype} pml$chan_create$channel0 (n: int): channel0
// channel splitting
extern fun channel0_split {init:vt@ype}
  (chan: !channel0 >> chanpos1(init)): channeg1(init)
```
One channel can be split into two endpoints via the function \texttt{channel0\_split}: a positive end and a negative end; the end held by the receiver is positive and the end held by the sender is negative. I introduce two abstract type constructors \texttt{chanpos1} and \texttt{channeg1} for constructing positive channels and negative channels, respectively, where a positive (negative) channel refers to the positive (negative) end of a channel. The corresponding type indices, which are of sort \texttt{view@ype} are used to represent the state of the channels. For example, in the code below, an abstract type \texttt{chnil} is introduced solely to represent the final state of a channel. And the types of the two functions \texttt{channeg1\_nil\_close} and \texttt{chanpos1\_nil\_close} ensure that only a positive channel in the final state can be turned back into a \texttt{channel0} and only a negative channel in the final state can be destroyed. (Note that these two functions are declared as proof functions, whose invocation is erased before translating from ATS/PML to PROMELA, since channels cannot be really released once created in PROMELA.)

```plaintext
// final state of channels
abstype chnil
//
extern prfun channeg1\_nil\_close (channeg1(chnil)): void
extern prfun chanpos1\_nil\_close (!chanpos1(chnil) >> channel0): void
```

With such setting of channel types, we can define types for messages transferred via channels using guarded recursive datatype in ATS (Xi et al., 2003). In the client-server example, the \textit{Client} process always receives from a specific channel, we call it the \textit{client} channel. Similarly, the \textit{Agent} process always receives from a specific \textit{agent} channel. The messages transferred in these two channels can be assigned meaningful types as shown in the following code.

```plaintext
// states of protocols
absvtype ss\_client
absvtype ss\_grant
```
// messages received by Client process
data
type
ClientOpt(start:vt@ype, next:vt@ype) =
  | DENY(ss_client, chnil) of ()
  | HOLD(ss_client, ss_client) of ()
  | GRANT(ss_client, chnil) of (channeg1(ss_grant))

// messages received by Agent process
data
type
AgentOpt(start:vt@ype, next:vt@ype) =
  | RETURN(ss_grant, chnil) of ()

Two types ss_client and ss_grant are introduced to represent the states of channels. Both ClientOpt and AgentOpt take two indices of sort viewt@ype, which represent the states before and after the transfer of a message. The sending and receiving functions corresponding to these two message types are given below.

// send to client channel
extern fun pml$chan_send$channeg1_client{beg,next:vt@ype} (    
  chan: !channeg1(beg) >> channeg1(next)
  , x: ClientOpt(beg, next)): void

// receive from client channel
extern fun pml$chan_recv$chanpos1_client {beg:vt@ype} (    
  chan: !chanpos1(beg) >> chanpos1(next))
  : #[next:vt@ype] ClientOpt(beg,next)

// send to agent channel
extern fun pml$chan_send$channeg1_agent{beg,next:vt@ype} (    
  chan: !channeg1(beg) >> channeg1(next)
  , x: AgentOpt(beg, next)): void

// receive from agent channel
extern fun pml$chan_recv$chanpos1_agent {beg:vt@ype} (    
  chan: !chanpos1(beg) >> chanpos1(next))
The type of $pml$chan$\_send$channeg1$\_client$ can be interpreted as follows. Given a negative channel at state $beg$, we can send a message of type $ClientOpt(beg, next)$ into the channel, and the state of the channel changes to $next$ after the transfer of the message, which is indicated by the change of the type of $chan$ from $channeg1(beg)$ to $channeg1(next)$. Similarly, the type of $pml$chanrecv$\_chanpos1\_client$ indicates that the state of the channel after receiving a message corresponds to the type of the message being received.

In a model, we can mix the usage of session types and normal channel types. In the client-server example, the Server process always receives from a global channel. And we call it server channel. Due to its simplicity, I choose not to encode channel states via types. The message type for the server channel is given in the code below. Also we can reuse the channel type ($chanref$) as well as related primitives for sending and receiving defined in Section 3.3.16 for the server channel. For readability, I include all those definitions in the code below as well.

\begin{verbatim}
abstype chanref(a:vt@ype) = pml$chan

extern fun {a:vt@ype} pml$chan_create$chanref (n: int): chanref(a)
extern fun {a:vt@ype} pml$chan_send$chanref (chanref(a), a): void
extern fun {a:vt@ype} pml$chan_recv$chanref (chanref(a)): a

datavtype
ServerOpt
  | RETURN of (channel0)
  | REQUEST of (channeg1(ss_client))
\end{verbatim}

With all the definition of types as well as primitives related to channels, the whole model can be rewritten in ATS/PML in Figure 3.13, Figure 3.14, and Figure 3.15, which contains definitions for Agent process, Server process, and Client process re-
respectively.

In the code for Agent process, if we omit the recursive call of inline$loop after sending a DENY () message, which corresponds to appending a break statement in the original PROMELA model as I discuss previously, a type error would be issued indicating that the client channel is not handled properly. Similar errors that can only be found by model checking can now be identified by type checking at the stage of model construction.

It is often claimed that the usage of session types can guarantee that there is no deadlock in a model. However, for such benefit, we have to use session types in a very strict way, e.g. that one process cannot hold both endpoints for a channel at the same time, which sometimes makes it impossible to implement some models. In the example here, such restriction is not followed since both Client process and Agent process may have both endpoints of their own channel respectively at hand before sending one endpoint to another process. Hence, we use type checking only to ensure that each party involved in the communication follows the protocol. Then we can use model checking to verify various properties of the model including deadlock freeness.
fun proctype$agent (agent: channel0, client: channeg1 (ss_client)): void = let
    fun inline$loop (agent: !channel0, client: channeg1 (ss_client)): void =
        case+ pml$random of
            0 => let
                val () = pml$chan_send$channeg1_client (client, HOLD ())
            in
                // cannot omit the recursive call
                inline$loop (agent, client)
            end
        | 1 => let
            val () = pml$chan_send$channeg1_client (client, DENY ())
            prval () = channeg1_nil_close (client)
        in
            // cannot call inline$loop (agent, client) again
            // the status of client has changed
        end
        | 2 => let
            val () = pml$chan_send$channeg1_client (client, GRANT (channel0_split {ss_grant} (agent)))
        in
            case- pml$chan_recv$chanpos1_agent (agent) of
                ~RETURN () => let
                    prval () = chanpos1_nil_close (agent)
                    prval () = channeg1_nil_close (client)
                in end
        end
    in
        val () = inline$loop (agent, client)
        val () = pml$chan_send$chanref (server, RETURN (agent))
in end

Figure 3.13: ATS/PML model for Agent process
val server = pml$chan_create$chanref<ServerOpt> (0)

fun proctype$server (): void = let
  extern fun agent_not_available (): bool
  extern fun agent_available (): bool
  extern fun get_agent (): channel0
  extern fun put_agent (channel0): void

  fun inline$loop (): void =
    case pml$random of
      | 0 => (case- pml$chan_recv$chanref (server)
        of ~RETURN (agent) => let
          val () = put_agent (agent)
        in inline$loop () end
      )
      | 1 => (case- pml$chan_recv$chanref (server)
        of ~REQUEST (client) =>
          case+ pml$random of
            | 0 => let
              val () = pml$wait_until0$ (agent_not_available ())
              val () = pml$chan_send$channeg1_client (client, DENY ())
              prval () = channeg1_nil_close (client)
            in inline$loop () end
            | 1 => let
              val () = pml$wait_until0$ (agent_available ())
              val agent = get_agent ()
              val _ = pml$run (proctype$agent (agent, client))
            in inline$loop () end
          )
        in inline$loop () end
      )
end

Figure 3.14: ATS/PML model for Server process
fun proctype$client () : void = let
  val me = pml$chan_create$channel0 (0)

fun inline$loop1 (me: channel0): void = let
  val () = pml$wait_until0$ (pml$timeout ())
  val () = pml$chan_send$chanref (
    server, REQUEST (channel0_split (me)))

fun inline$loop2 (me: !chanpos1 (ss_client) >> chanpos1 (chnil)): void =
  case+ pml$random of
  | 0 => (case- pml$chan_recv$chanpos1_client{ss_client} (me) of
     | ~HOLD () => inline$loop2 (me) // loop
  )
  | 1 => (case- pml$chan_recv$chanpos1_client (me) of
     | ~DENY () => () // break
  )
  | 2 => (case- pml$chan_recv$chanpos1_client (me) of
     | ~GRANT (agent) => let
       val () = pml$chan_send$channeg1_agent(agent,RETURN ())
       prval () = channeg1_nil_close (agent)
     in end // break
  )

  val () = inline$loop2 (me)

  prval () = chanpos1_nil_close (me)
in
  inline$loop1 (me)
end
in
inline$loop1 (me)
end

Figure 3.15: ATS/PML model for Client process
Chapter 4

ATS/Veri: A Modeling Language with Advanced Types

4.1 Introduction

The advanced type system of ATS/PML allows programmers to assign meaningful types to primitives with sophisticated semantics. As such, type checking can help prevent bugs at the stage of model construction, thus alleviating burden on model checking. Also model checking can help solve type constraints derived from the verification of properties encoded via types, thus simplifying the type checking procedure and improving the usability of the type system. In short, programmers are now able to choose between type checking and model checking to verify different kind of properties. In summary, the creation of ATS/PML helps tackle a problem engineers are facing in industry, that is how to write high quality models using a popular modeling language PROMELA. However, PROMELA focuses on expressing highly abstract models amenable to efficient model checking and sacrifices lots of features common in software systems, such as stack, normal function invocation, scheduling, memory model, and etc. Strongly coupled with PROMELA, ATS/PML suffers the same problem.

To tackle this issue, I present in this chapter a new modeling language ATS/Veri, which focuses on enabling programmers to model real-world multi-threaded software applications in a more natural and efficient manner. I also provide a platform to verify
models in ATS/Veri using model checking techniques. Instead of proving correctness of models, the emphasis of ATS/Veri is to help programmers construct high quality models via type checking and rely on model checking to identify remaining errors as much as possible. Though it is possible that sophisticated models in ATS/Veri may not be model checked completely due to the problem of state explosion, such models are of great value as formal specifications with relatively sophisticated details. First, the clarity ATS/Veri can provide as a formal modeling language is a benefit in itself. Second, models in ATS/Veri can be used as the guide or design for real implementations in a manual or automatic manner. Moreover they can serve as the base for more abstract specifications, which are amenable for complete verification. The name of ATS/Veri indicates its root in ATS for both type system and semantics as well as its capability of combining type checking and model checking synergically for program verification.

In this chapter, I will first illustrate the design of ATS/Veri in Section 4.2, which is an extension of the core of ATS with primitives for modeling concurrent software systems. Section 4.3 illustrates the types and usages of various primitives in ATS/Veri. Section 4.4 introduces an intermediate representation of ATS/Veri, based on which the formal definition of the semantics of ATS/Veri is given in Section 4.5.

4.2 Design of ATS/Veri

ATS/Veri is an extension of the core of ATS, which is a ML-like functional programming language. Both syntax and semantics of ATS is inherited by ATS/Veri. The goal is to allow programmers to write models in ATS/Veri just like writing functional programs in ATS. The extension includes various primitives supporting synchronization between threads as well as interaction between threads and the underlining scheduler. More detailed description about different aspects of ATS/Veri goes as follows.
4.2.1 Syntax

Unlike ATS/PML, ATS/Veri does not rely on pre-defined composition of syntactic features to represent semantics not belonging to ATS. Ideally, all the syntax shown in Section 2.3 is allowed in ATS/Veri. However due to the diversity of ATS’ features, the current implementation of the model checker of ATS/Veri (discussed later in Section 5.2) does not support all of them. Features not supported currently include patterns involving `datatype` in ATS, case-expression, and usage of variables.

Patterns currently supported in ATS/Veri cannot have constants inside. Also tuple is the only constructor supported for pattern construction. Moreover, the usage of functions is limited in the way that besides being called for function invocation, they can only be used as arguments for the invocation of the primitive for creating new threads (explained later in this chapter). In general, they cannot be used as arguments or return values.

4.2.2 Types

On one hand, ATS/Veri inherits both dependent types and linear types from ATS, so as the support for the programming paradigm of Programming with Theorem Proving. On the other hand, the feasibility of model checking is taken into consideration when designing ATS/Veri. Thus some primary types in ATS are not supported in ATS/Veri such as string and float. Currently ATS/Veri supports the following primary types: `int`, `char`, `bool`, `void`, `int(i)`, `char(c)`, and `bool(b)` given that `i`, `c`, and `b` are of sorts `int`, `char`, `bool` respectively.

To better support modeling concurrent software systems, I introduce various primitives, which are equipped with well-designed types to improve their correct usage. To construct such types for primitives, several abstract types are introduced into ATS/Veri, which are meaningful to the model checker of ATS/Veri. These types
are discussed in Section 4.3 along with their related interfaces. Roughly speaking, the type constructors \texttt{shared\_t}, \texttt{mutex\_t}, and \texttt{mutex\_v} are for synchronization among threads, \texttt{atomref} and \texttt{atomarrayref} are for atomic data access, \texttt{vlock\_vt} is for assertion about mutual exclusion, \texttt{atomic\_view} is for atomic execution, and \texttt{list\_t} is for the functional list data structure.

Tuples are supported in ATS/Veri and can contain any non-linear types as its elements. Also the type for list is supported (\texttt{list\_t} in the code above) as well as related primitives for manipulating lists. Going further, the support of user defined types via datatypes are left to the next iteration of development since they only improve the usability of ATS/Veri and have no influence on its semantics and the underlining model checking. Lastly, reference types, as supported in ATS/PML, is currently not supported by ATS/Veri considering that they do not enhance the expressiveness of ATS/Veri, but help improve its usability as well as the efficiency of model checking.

4.3 Primitives

4.3.1 Summary of Primitives

ATS/Veri supports various primitives including common operators in ATS, auxiliary primitives for manipulating functional data structure, and primitives affecting system states.

Some common operators are listed below.

- Arithmetic Operator: $+, -, *, /, \%$
- Comparison Operator: $<, \leq, >, \geq, ==, !$
- Logic Operator: $\&\&$, $||$

Type and primitives for manipulating lists are listed below.
Primitives which have effects on system states (Def 4.5.7) play an important role when setting up the semantics of ATS/Veri. Formal definitions of their effects are to be illustrated in Section 4.5.5. In the section, I mainly illustrate their usage in modeling concurrent software systems with focus on their types.

4.3.2 Synchronization Primitives

Practical concurrent systems consist of multiple components cooperating together towards certain goals. In such setting, it is important for a modeling language to enable designers to specify the constituents of a system and the synchronization among them. Therefore I introduce into ATS/Veri a set of pthread-like primitives to support the modeling of various synchronization mechanisms including mutexes, joins, and condition variables. POSIX thread (pthread) libraries have been widely used in concurrent programming, thus making the corresponding primitives in ATS/Veri programmer friendly for modeling concurrent systems. Moreover, well-designed types are assigned to these primitives so that the chances of misusing them, which are quite common in practice, are greatly reduced. For example, the primitives for mutex operations, as shown in Figure 4.1, are assigned types involving views `mutex_v` in-
indicating the ownership of a mutex. After calling \texttt{conats_mutex_acquire} to acquire the mutex, we get a linear proof that we are holding its ownership. We have to call \texttt{conats_mutex_release} exactly once to give up this ownership. Otherwise type checking would issue an error, thus prevents us from not releasing a mutex or releasing it more than once.

Going beyond elementary support for concurrent primitives, I formalize the concept of shared objects, which are accessible by all threads, as a special set of types (\texttt{shared\_t} in Figure 4-1). \texttt{shared\_t} is introduced as an abstract type constructor. Given a static variable \texttt{obj} of sort \texttt{viewt@ype} and a static variable \texttt{n} of sort \texttt{int}, \texttt{shared(obj, n)} is an abstract type for shared objects that contain objects of the type \texttt{obj} \footnote{\texttt{viewt@ype} to be more precise.} as well as \texttt{n} condition variables for threads to wait on. The first two functions associated with \texttt{shared\_t} are for creating a sharing object with certain initial content. The \texttt{conats_shared_acquire} and \texttt{conats_shared_release} are for acquiring and releasing the lock of a shared object, respectively. \texttt{conats_sharedn_signal} signals a thread waiting on certain condition variable of a shared object, while \texttt{conats_sharedn_broadcast} signals all the threads waiting on certain condition variable. And \texttt{conats_sharedn_condwait} conditionally waits on condition variable of a shared object.

Now, I will use the modeling of producer-consumer problem as a user case to demonstrate the usage of these primitives.

In Figure 4-2, \texttt{buffer} is introduced as an abstract viewtype. Given a \texttt{viewt@ype} \texttt{obj}, an integer \textit{M} and another integer \textit{N}, \texttt{buffer(obj, M, N)} is for a buffer of maximal capacity \textit{M} that contains \textit{N} items of type \texttt{obj}. Note that the term \texttt{buffer(obj)} is a type alias for a buffer with contents within its capacity. The functions operating on buffers are given explanation in the comment. Note that dependent types are used to impose certain pre-conditions and post-conditions on them. For instance, the type
// Types and primitives for mutexes.
abstype mutex_t
typedef mutex = mutex_t
absview mutex_v

fun conats_mutex_create (): mutex
fun conats_mutex_acquire (m: mutex): (mutex_v | void)
fun conats_mutex_release (v: mutex_v | m: mutex): void

// Types and primitives for shared objects.
abstype shared_t (viewtype, int)
typedef shared (a:viewtype) = shared_t (a, 1)

fun conats_shared_create {a: viewtype} (ele: a): shared (a)
fun conats_sharedn_create {a: viewtype} {n:pos} (ele: a, n: int n): shared_t (a, n)

fun conats_shared_acquire {a: viewtype} {n:pos} (s: shared_t (a, n)): a
fun conats_shared_release {a: viewtype} {n:pos} (s: shared_t (a, n), ele: a): void
fun conats_shared_signal {a: viewtype} (s: shared (a), ele: a): a
fun conats_sharedn_signal {a: viewtype} {i,n:nat | i < n} (s: shared_t (a, n), i: int i, ele: a): a
fun conats_shared_broadcast {a: viewtype}(s:shared (a), ele:a): a
fun conats_sharedn_broadcast {a: viewtype} {i,n:nat | i < n} (s: shared_t (a, n), i: int i, ele: a): a
fun conats_shared_condwait {a: viewtype}(s:shared (a), ele:a): a
fun conats_sharedn_condwait {a: viewtype} {i,n:nat | i < n} (s: shared_t (a, n), i: int i, ele: a): a

// Primitives for thread creation.
fun conats_tid_allocate (): [tid:pos] int tid
typedef thread_fun_t (a: viewtype) = (a -<fun1> void)
fun conats_thread_create {a:viewtype} {tid:pos} (tfun: thread_fun_t a, arg: a, tid: int tid): void

Figure 4.1: Synchronization Primitives
absvtype buffer (obj:viewtype, m:int, n:int) // linear abstract type
vtypedef buffer (obj:viewtype) = [m,n:nat|m >= n]buffer (obj, m, n)

fun buffer_isemp // Is buffer empty?
   {a:viewtype}{m,n:nat} (buf: !buffer (a, m, n)): bool (n == 0)
fun buffer_isful // Is buffer full?
   {a:viewtype}{m,n:nat} (buf: !buffer (a, m, n)): bool (m == n)
//
fun buffer_insert // Insert into a non-full buffer.
   {a:viewtype}{m,n:nat | m > n} (buf: !buffer(a, m, n) >>
                              buffer(a, m, n+1), x: a): void
fun buffer_remove // Remove from a non-empty buffer.
   {a:viewtype}{m,n:nat | n > 0} (buf: !buffer(a, m, n) >>
                              buffer(a, m, n-1)): a
//
vtypedef sbuffer (obj:viewtype) = shared_t (buffer (obj), 2)

// producing an item
fun produce_item {obj: viewtype} (): obj
//
// consuming a given item
fun consume_item {obj: viewtype} (x: obj): void
//
fun consumer {obj:viewtype} (sbuf: sbuffer (obj)): void
//
fun producer {obj:viewtype} (sbuf: sbuffer (obj)): void

Figure 4.2: Shared objects
assigned to `buffer_remove` means that the function can only be called on a buffer
that is not empty and the buffer contains one less element after the call returns.

With `buffer` and `shared_t`, we define type `sbuffer(obj)`, which represents the
shared buffer containing elements of type `obj` with two condition variables. And we
declare a function `consumer`, by which a thread can keep getting items from the shared
buffer and consume them by the function `consume_item`. We also declare a function
`producer` by which a thread can keep producing items by the function `produce_item`
and putting them into the shared buffer.

Figure 4.3 and Figure 4.4 show the implementation of the functions `producer` and
`consumer` based on some auxiliary functions, which model a solution for the producer-
consumer problem.

In summary the design of shared object is similar to the concept of monitor
(Hansen, 1999). To access the content of a shared object (monitor), a thread has
to call `shared_acquire` to acquire the lock (enter the monitor). And the type system
of ATS enforces that a thread must call `shared_release` to relinquish the access to
the content of a shared object and release the lock (leave the monitor). Thus the
deadlocking due to a lock holder forgetting to release the lock is eliminated. And it
is formally verified in the type system of ATS that the implementation in Figure 4.3
and Figure 4.4 cannot cause buffer overflow or underflow as stated by the types of
`buffer_insert` and `buffer_remove`. Such types also enforce programmers to recheck the
waiting condition after the thread is waken up, a programming style well accepted
but not enforced in practical programming languages such as C or Java, let alone
modeling languages.

However, if we remove the if-expression containing a call to `conats_sharedn_signal`
in `fconsumer`, then type checking `fconsumer` issues no errors. This is very unfor-
tunate as it is clear that this change can cause a deadlock as a blocked producer
may never be awakened. The incompetence of type checking to detect such errors motivates me to equip ATS/Veri with capability of model checking so that properties involving global reasoning can be verified.

4.3.3 Global Variable Access Primitives

I introduce into ATS/Veri the types \texttt{atomref} and \texttt{atomarrayref} to represent global variables and global arrays. Primitives manipulating them are listed in Figure 4.5. The \texttt{atom} in names for these primitives indicates that their execution is atomic. The formal definition of atomicity is given in Section 4.5.5.

These types and primitives are usually used to implement models of abstract data structures, which are required to form a complete a model for the whole system so that model checking can be readily applied. For instance, in our model for the solution of producer-consumer problem, the type for the underlining buffer is abstract, also the type for the element inside the buffer is unspecified. Since most model checking techniques can only be applied on all-inclusive models, we have to not only implement the type for buffer and its related operations, but also specify the type of the element. Considering that the main goal is to verify the implementation for producer and consumer, and that an abstract model for the buffer can help reduce the state space encountered during the stage of model checking, I choose to implement a buffer in the model based on the assumption that it is a buffer containing at most one integer. Arguably, if the implementation for producer and consumer is correct for a shared buffer of small capacity, it has better chances to be correct as well for large capacity. An \texttt{atomref} is used to hold the number of the current elements in the buffer (either 0 or 1). Dummy implementation for \texttt{buffer_insert} and \texttt{buffer_remove} are given, which do not really use the elements in a buffer. The code is shown in Figure 4.6. While certain ATS syntax may look unfamiliar and shall be explained below, the code should intuitively not be difficult to follow.
#define NOTEMP 0
#define NOTFUL 1

// Keep adding elements into buffer.
fun fproducer {a:vt@ype} (s: sbuffer (a), data: a): void = let
  val buf = conats_shared_acquire (s)

  fun insert (buf: buffer (a), data: a): buffer (a) = let
    val isful = buffer_isful (buf)
    in
      if isful then let
        val buf = conats_sharedn_condwait (s, NOTEMP, buf)
        in
          insert (buf, data)
      end else let
        val isnil = buffer_isemp (buf)
        val () = buffer_insert (buf, data)
        in
          if isnil then conats_sharedn_signal (s, NOTFUL, buf)
          else buf
      end
    end
  end

  val buf = insert (buf, data)
  val () = conats_shared_release (s, buf);
in
() end
end

implement producer {obj} (s) = let
  val () = fproducer {obj} (s, produce_item ())
in
  producer {obj} (s)
end

Figure 4.3: Model of a Producer Solution
// Keep removing elements from buffer.
fun fconsumer {a: vtype} (s: sbuffer (a)):<fun1> a = let
  val buf = conats_shared_acquire (s)

  fun takeout (buf: buffer (a)):<cloref1> (buffer (a), a) = let
    val isnil = buffer_isemp (buf)
    in
    if isnil then let
      val buf = conats_sharedn_condwait (s, NOTFUL, buf)
      in
        takeout (buf)
    end else let
      val isful = buffer_isful (buf)
      val data = buffer_remove (buf)
      in
      if isful then let
        // Omitting the following would cause deadlock
        val buf = conats_sharedn_signal (s, NOTEMP, buf)
        in (buf, data) end
      else (buf, data) end
    end
  end

  val (buf, data) = takeout (buf)
  val () = conats_shared_release (s, buf);
  in
  data
  end

implement consumer {obj} (s) = let
  val () = consume_item (fconsumer {obj} (s))
  in
  consumer {obj} (s)
  end

Figure 4-4: Model of a Consumer Solution
// Type and primitives for global variables.
abstype atomref (a: t@ype)

fun conats_atomref_create {a:t@ype} (data: a): atomref a
fun conats_atomref_update {a:t@ype} (gv: atomref a, data: a): void
fun conats_atomref_get {a:t@ype} (gv: atomref a): a

// Type and primitives for global arrays.
abstype atomarrayref (a: t@ype)

fun conats_atomarrayref_create {a:t@ype} (len: int, data: a): atomarrayref a
fun conats_atomarrayref_update {a:t@ype} (gv: atomarrayref a, pos: int, data: a): void
fun conats_atomarrayref_get {a:t@ype} (gv: atomarrayref a, pos: int): a

Figure 4-5: Global Variable Access Primitives of ATS/Veri

The local ... in ... end syntax is a special feature of ATS, which allows a programmer to implement abstract types between local and in. In the example, I implement the type buffer (a, m, n) by atomref (int) using the assume keyword. Then for the implementation of all the functions between in and end, these two types are converted to each other implicitly during type checking. (Note that such type equality cannot be seen out of the local ... in ... end syntax, thus the type checking for previous code for producer and consumer is not influenced at all.) The keyword castvtp0 is for explicit conversion of types, which is ignored after type checking. It is worth mentioning the usage of prval in the function buffer_insert. A pattern match with the keyword prval will be completely erased after type checking so that the conversion of x to an integer has no effect during model checking. The only purpose of conversion here is to accommodate the type checking for linear types in ATS. After the conversion, the type checker of ATS will not track the life of x any more since its view has been consumed. The concept of implementing abstract types
and interfaces using type conversion here is similar to the interaction with PROMELA in ATS/PML (discussed in Section 3.3.15), but in a controllable manner since all the code is in ATS/Veri and all conversions are stated explicitly. This leads to the benefit that programmers for ATS/Veri only need to know its semantics and will not be confined to only one model checking technique.

Note that implementation for produce_item, consume_item, and the definition for a new function buffer_create (used in Figure 4·8) are also given in Figure 4·6 so that no function used in the model is left undefined.

4.3.4 Thread Creation Primitives

Intuitively, the semantics of a ATS/Veri model is similar to the execution of a multi-threaded program. The main thread is created implicitly for the execution of code in global scope (not in any function). New threads can be created by invoking the primitive conats_thread_create shown in Figure 4·7. The first argument of the primitive is of a function type, which takes one argument whose type is of sort viewt@ype, and has no return value. Note that in the current implementation, the argument has to be a function name or a lambda expression and cannot be any other value of function type. This function serves as the body of the new thread. The second argument of the primitive will be used by the new thread to invoke the aforementioned function. The third argument is the id of the new thread, which can be accessed within the thread via conats_get_thread_id. conats_tid_allocate is for generating thread id’s, which start from 1 and get incremented by 1 for each invocation. (The main thread has id 0.) Note that currently a thread id is not recycled when a thread ends.

A complete model for the producer-consumer problem is given in Figure 4·8 to illustrate the usage of these primitives.
local
  // treat buffer (a, m, n) as buffer (int, 1, 1)
  assume buffer (a:viewt@ype, m:int, n:int) = atomref (int)
in
  implement buffer_isemp {a} {m,n} (buf) = let
  val n = castvwtp0{int n} (conats_atomref_get (buf))
in
  n = 0
end

implement buffer_isful {a} {m,n} (buf) = let
  val n = castvwtp0{int n} (conats_atomref_get (buf))
  val m = castvwtp0{int m} 1
in
  m = n
end

implement buffer_insert {a} {m,n} (buf, x): void = let
  val n = conats_atomref_get (buf)
  val () = conats_atomref_update (buf, n + 1)
  prval _ = castvwtp0 {int} (x)
in end

implement buffer_remove {a} {m,n} (buf) = let
  val n = conats_atomref_get (buf)
  val () = conats_atomref_update (buf, n - 1)
in
  castvwt0 {a} (0)
end

implement produce_item {obj} () = castvwt0 {obj} (0)
implement consume_item {obj} (x) = let
  prval _ = castvwt0 {int} (x)
in end

// Construct the model of whole system.
// Create a buffer of length 1.
fun buffer_create (): buffer (int, 1, 1) =
  conats_atomref_create (0)
end

Figure 4-6: Implementation for Buffer
fun conats_tid_allocate (): [tid:pos] int tid

typedef thread_fun_t (a: viewt@ype) = (a -<fun1> void)

fun conats_thread_create {a:viewt@ype} {tid:pos} (tfun: thread_fun_t a, arg: a, tid: int tid): void

stacst curtid: int
fun conats_get_thread_id (): int curtid

**Figure 4.7:** Thread Creation Primitives of ATS/Veri

// Create a buffer for model construction.
val buf = buffer_create ()

// Turn a linear buffer into a shared buffer.
val s = conats_sharedn_create {buffer (int)} (buf, 2)

val tid1 = conats_tid_allocate ()
val tid2 = conats_tidAllocate ()

val () = conats_thread_create {int} (lam x => producer {int} (s), 0, tid1)
val () = conats_thread_create {int} (lam x => consumer {int} (s), 0, tid2)

**Figure 4.8:** Complete Model for Producer-Consumer Problem
// Primitive for atomic execution (non-preemption)
absview atomic_view
prfun mc$atomic_start (): (atomic_view | void)
prfun mc$atomic_end (atomic_view): void

// Primitive for logic assertion.
prfun mc$assert {b: bool} (x: bool b):<fun> [b == true] void

// Type and primitive for linear assertion.
absvtype vlock_vt (int, int, int, int)

prfun mc$vlock_get {x,y: nat} {xi,yi: pos}
  (x: int x
   , y: int y
   , xi: int xi
   , yi: int yi
  ): mc_vlock_vt (x, y, xi, yi)

prfun mc$vlock_put {x,y: nat} {xi,yi: pos}
  (v: mc_vlock_vt (x, y, xi, yi)): void

Figure 4.9: Model Checking Primitives of ATS/Veri

4.3.5 Model Checking Primitives

Primitives related to model checking capability are given in Figure 4.9. They are introduced as proof functions in ATS (with the keyword prfun) with names prefixed by mc$, which distinguishes them from other primitives for their special semantics related to model checking. Unlike other proof code, which is erased after the type checking, the usage of such primitives is kept till the stage of model checking. Formal definition of their effect on model checking is discussed in Section 4.5.5.

Intuitively, a thread will not be preempted by the scheduler after the invocation of mc$atomic_start till the invocation of mc$atomic_end. The concept is similar to the the atomic or d_step in PROMELA. However unlike PROMELA, the usage of these two primitives is not restricted by any syntactic boundary. The entering and
leaving of non-preemptable zone can take place in different functions. The types of \texttt{mc$atomic\_start} and \texttt{mc$atomic\_end} are similar to those for primitives related to mutex. Such types help prevent a programmer from misusing these primitives. Going further, a function can be assigned a type, which takes a proof of \texttt{atomic\_view}, to indicate that the body of the function shall be executed atomically. As such, types can be exploited for constructing sophisticated models in a modular manner.

Intuitively, we can view \texttt{mc\_assert} as a assertion normally used by a programmer for runtime checking, but this time the validity of such assertion is checked during the process of model checking. The type of \texttt{mc\_assert} indicates that its argument is a boolean expression whose value equals some static term \textit{b} and this \textit{b} must equal \textit{true} in order for the primitive to actually return. On one hand, such type makes it possible to take advantage of a valid assertion (verified at model checking) during the stage of type checking. On the other hand, the type checker of ATS can serve as a guide to locate places where assertions are needed.

I will illustrate the usage of \texttt{mc$vlock\_get} and \texttt{mc$vlock\_put} in Section 4.3.6 via a concrete example of modeling an asynchronous communication mechanism.

4.3.6 Virtual Lock

Improper handling of resources (e.g. memory) in a program may lead to various bugs (e.g. memory leak) in sequential programs. The problem gets even worse when entering the concurrent domain, in which simultaneous access to shared resource by multiple threads is feasible. One example is that we may lose the integrity of data if two threads are using a shared memory to transfer data. Techniques for solving this problem generally rely on mutual exclusion principles to control access to shared resources. Mutual exclusion introduces a measure of synchronization, but with the cost of losing efficiency. With a deliberate design, sometimes we can remove the need for synchronization while maintaining the desired property of mutual exclusion.
Simpsons four-slot fully asynchronous communication mechanism (Simpson, 1990) demonstrates such idea. However, its very difficult to verify that the deemed mutual exclusion property actually holds in the design. To tackle this problem, ATS/Veri is equipped with two primitives supporting the concept of “virtual lock” to allow programmers to specify assumptions of mutual exclusion to various granularities according to their design. And such assumption can then be verified by the model checker for ATS/Veri.

Let’s illustrate the usage of “virtual lock” using the following example of four-slot mechanism. Consider the scenario in which one writer and one reader try to communicate via a shared resource consisting of multiple memory slots. Due to hardware constraint, access to each memory slot is not atomic. Therefore, reader may get inconsistent data if writer is writing the same slot at the same time. The following code shows the proposed types for the shared resource ($\text{dataslots\_t}$) as well as the interfaces for accessing it.

```plaintext
// Define types for data slots.
abstype dataslots_t (t@ype, int, int)

absviewtype own_slot_vt (int, int)

// Create a two dimensional array
fun dataslots_create {a:t@ype} {x,y:int| x>1 && y>1} (x: int x, y: int y, v: a): dataslots_t (a, x, y)

fun dataslots_update {a:t@ype} {x,y,i,j:nat | i < x && j < y} (vpf: own_slot_vt (i, j)
| slots: dataslots_t (a, x, y), i: int i, j: int j, v: a
): (own_slot_vt (i, j) | void)

fun dataslots_get {a:t@ype} {x,y,i,j:nat | i < x && j < y} (vpf: own_slot_vt (i, j)
```
| slots: dataslots_t (a, x, y), i: int i, j: int j |
| : (own_slot_vt (i, j) | a) |

Intuitively, the usage of linear type own_slot_vt as well as the type indices involved indicates that dataslots_update and dataslots_get require mutual exclusion on the memory slot to be accessed. Normally, a programmer can meet such requirement by the usage of synchronization primitives (e.g. mutex). However, in the following code, such mutual exclusion is gained by the usage of a few global variables the access for which is atomic. The code is shown below.

```
// Define type for data slots used in the example.
// We use four data slots in two dimensions.
typedef data_t = dataslots_t (int, 2, 2)
val data: data_t = dataslots_create (2, 2, 0)

typedef int2 = [i: int | i >= 0 && i <= 1] int i

// control variables
val slot = conats_atomarrayref_create int2 (2, 0)
val latest = conats_atomref_create int2 (0)
val reading = conats_atomref_create int2 (0)

fun write (item: int): void = let
    val pair = 1 - conats_atomref_get (reading)
    val index = 1 - conats_atomarrayref_get (slot, pair)

    prval vpf = mclacquire_ownership (pair, index)
    val (vpf | _) = dataslots_update (vpf | data, pair, index, item)
    prval () = mcrelease_ownership (vpf)

    val () = conats_atomarrayref_update (slot, pair, index)
    val () = conats_atomref_update (latest, pair)
in end
```
fun read (): int = let
  val pair = conats_atomref_get (latest)
  // Switch the following two steps would cause inconsistence.
  val () = conats_atomref_update (reading, pair)
  val index = conats_atomarrayref_get (slot, pair)

  prval vpf = mclacquire_ownership (pair, index)
  val (vpf | item) = dataslots_get (vpf | data, pair, index)
  prval () = mcorelease_ownership (vpf)
  in item end

In the example, the shared resource (data) contains four slots in two dimensions. slot, lastest and reading are global array and global variables, which are created by the primitives conats_atomarrayref_create and conats_atomref_create. To accommodate the type checking of ATS, two functions mc$cquire_ownership and mc$crelease_ownership are used to generate and destroy the linear value (vpf) of type own_slot_vt (x,y). Such value appears on the left side of the symbol | in a pattern, and is called a model checking value or virtual value since its sole purpose is to incorporate the support of model checking. The implementation for mc$cquire_ownership and mc$crelease_ownership are given below based on the primitives mc$vlock_put and mc$vlock_get.

local
  assume own_slot_vt (i: int, j: int) = mc$vlock_vt (i, j, 1, 1)
in
  prfun mc$squire_ownership .<>. {i, j: nat}
  (i: int i, j: int j): own_slot_vt (i, j) =
  mc$vlock_get (i, j, 1, 1)

  prfun mc$srelease_ownership .<>. {i, j: nat}
  (vpf: own_slot_vt (i, j)): void = mc$vlock_put (vpf)
end
Intuitively, the primitive \texttt{mc$vlock\_get (x, y, a, b)} indicates the acquisition of a virtual lock covering a rectangle with \((x, y)\) as the upper left corner, \(a\) as the width (x-axis), and \(b\) as the height (y-axis), and \texttt{mc$vlock\_put} indicates the release of the lock. And the model checker for ATS/Veri would check that whether two threads try to acquire two virtual locks covering overlapping areas simultaneously. And this serves as the verification of mutual exclusion. To model checking the example, we would need to add the following code to implement those interfaces for accessing shared resource as well as to set up the complete model ready for model checking.

\begin{verbatim}
local
  assume dataslots_t (a:type, x: int, y: int) =
    atomarrayref (atomarrayref (a))
in
  implement dataslots_create {a} {x,y} (x, y, v) = let
    val ele = conats_atomarrayref_create {a} (y, v)
    val array =
      conats_atomarrayref_create {atomarrayref a} (x, ele)
  fun loop (x: int, y: int,
          array: atomarrayref (atomarrayref a),
          v: a): void =
    if x >= 0 then let
      val ele = conats_atomarrayref_create {a} (y, v)
      val () = conats_atomarrayref_update (array, x, ele)
    end else ()
  in
    loop (x - 1, y, array, v)
  end else ()

  val () = loop (x - 2, y, array, v)
in array end

  implement dataslots_update {a} {x,y,i,j} (vpf | slots, i, j, v) = let
\end{verbatim}
val ele = conats_atomarrayref_get (slots, i)
val () = conats_atomarrayref_update (ele, j, v)
in (vpf | ()) end

implement dataslots_get {a} {x,y,i,j} (
    vpf | slots, i, j) = let
    val ele = conats_atomarrayref_get (slots, i)
    val v = conats_atomarrayref_get (ele, j)
in (vpf | v) end
end

fun loop_writer (x: int): void = let
    val () = write (x)
in
    loop_writer (x)
end

fun loop_reader (x: int): void = let
    val _ = read ()
in
    loop_reader (x)
end

// Construct the model of whole system.
val tid1 = conats_tid_allocate ()
val tid2 = conats_tid_allocate ()
val () = conats_thread_create(loop_reader, 0, tid1)
val () = conats_thread_create(loop_writer, 0, tid2)

4.4 From Expression to Statement

4.4.1 Intermediate Representation (IR)

ATS is an expression-based language, i.e. the body of a function is an expression, whose evaluation result is the return value of the function. And the semantics of
ATS sets up rules for such evaluation, which is shown in Section 2.3. As an extension of ATS, it is possible to set up the operational semantics of ATS/Veri in a similar way to ATS. However, it is difficult to exploit specialized and high optimized tools for model checking based on such formation of semantics. Therefore, I choose to set up the semantics of ATS/Veri in a similar way to Promela. Under such goal, I introduce an intermediate representation (IR) of ATS/Veri as a base for setting up the semantics of ATS/Veri, which is illustrated in Section 4.5. In this section, I mainly focus on the translation process from ATS/Veri to IR, which is crucial for readers to understand the relation between the semantics of ATS and ATS/Veri, which in essence are equivalent.

The syntax of IR is shown in Figure 4.10. Note that similar to ATS, the code for proof in ATS/Veri has no influence on its operational semantics and is erased after type checking. Thus IR does not contain any syntax for proofs. Also, considering that types in a model in ATS/Veri has no influence on its operational semantics. Thus details about types in the IR are omitted in the syntax as well as the following discussion.

4.4.2 Translation from ATS/Veri to IR

The essence of the translation from ATS/Veri to IR is to translate an expression into a sequence of statements (stat-list in syntax). Such translation follows the semantics (evaluation rules) of ATS. Intuitively, each evaluation step in ATS is translated into a statement in IR. For instance, the following function in ATS/Veri

```plaintext
fun foo (x: int, y: int): int = let
  val n = (x + y) + goo (x, y)
in
  n + 3
end
```

is translated into IR as follows.
\[
\langle \text{prog} \rangle \quad ::= \quad \langle \text{func} \rangle^+ \\
\langle \text{func} \rangle \quad ::= \quad \langle \text{ident} \rangle \ ('\langle \text{para-list} \rangle \ ' \ '\{\langle \text{stat-list} \rangle \ ' \ '}') \\
\langle \text{para-list} \rangle \quad ::= \quad \langle \text{empty} \rangle \mid \langle \text{ident} \rangle \ ('', \langle \text{ident} \rangle)^* \\
\langle \text{statement} \rangle \quad ::= \quad \langle \text{name-binding} \rangle \\
\langle \text{if-stat} \rangle \quad ::= \quad 'if' \ ('\langle \text{value} \rangle \ ')' \ '\{\langle \text{stat-list} \rangle \ ' \ '} 'else' \ '\{\langle \text{stat-list} \rangle \ ' \ '}' \\
\langle \text{return-stat} \rangle \quad ::= \quad 'return' \langle \text{value} \rangle \\
\langle \text{load-ret} \rangle \quad ::= \quad 'loadret' \langle \text{ident} \rangle \\
\langle \text{tail-call} \rangle \quad ::= \quad 'tailcall' \langle \text{func-call} \rangle \\
\langle \text{sexpr} \rangle \quad ::= \quad \langle \text{value} \rangle \\
\langle \text{func-call} \rangle \quad ::= \quad \langle \text{ident} \rangle \ ('\langle \text{value} \rangle \ '*') \\
\langle \text{stat-list} \rangle \quad ::= \quad (\langle \text{statement} \rangle \ ',')^* \\
\langle \text{value-list} \rangle \quad ::= \quad \langle \text{empty} \rangle \mid \langle \text{value} \rangle \ ('', \langle \text{value} \rangle)^* \\
\langle \text{value} \rangle \quad ::= \quad \langle \text{constant} \rangle \mid \langle \text{ident} \rangle \mid '\langle \text{value} \rangle \ '*' \\
\]

Figure 4.10: Syntax for ATS/Veri IR
foo (x, y) {
    id1 = x + y;
    goo (x, y);
    loadret id2;
    n = id1 + id2;
    id3 = n + 3
    return id3;
}

The design of the translation process is intuitive and similar to the compilation process from functional programming languages to imperative languages. In the sequel, I mainly focus on several subtle yet important issues the translation process must tackle and omit the full details of the translation, which can be found in my actual implementation for model checking ATS/Veri.

**Identifiers**

To break down the evaluation of a sophisticated expression, identifiers (*ident* in the syntax) are generated for those intermediate results. The generation guarantees that such identifiers are unique. Also user supplied identifiers may need to be replaced with new ones to guarantee the uniqueness of identifiers. For example, the following code in ATS/Veri

```ats
val x = 1 + 2 + 3
```

is translated into

```ats
id1 = 1 + 2;
x = id1 + 3;
```

Note that a special identifier **null** is used for binding with any expression whose type is **void**.
Pattern Match

Currently only the pattern of tuple is supported in ATS/Veri, and pattern match in ATS/Veri is translated into multiple statements in IR. For example, the following code

```plaintext
val (x, y) = id
```

is translated into the following IR code

```plaintext
x = tuple_get_element (id, 0);
y = tuple_get_element (id, 1);
```

Note that `tuple_get_element` is a primitive supported in IR, which serves to access the elements inside a tuple.

Return Statement

If the value of an expression is used as the return value of a function, then a return statement (`return-stat` in syntax) is used. For example, the following code in ATS/Veri

```plaintext
fun foo ():int = let
val x = 1 + 2
in
  x + 3
end
```

is translated into

```plaintext
foo () {
  x = 1 + 2;
  id1 = x + 3;
  return id1;
}
```
If-Expression

An if-expression in ATS/Veri is translated to a sequence of statements in IR, the last of which is an if-statement (*if-stat* in syntax). For example, the following code in ATS/Veri

``` ATS/Veri
val x = 1 + if (1 > 2) then 1 else 2
```

is translated into

``` ATS/Veri
id2 = 1 > 2;
if (id2) {
    id1 = 1;
} else {
    id1 = 2;
}
x = 1 + id1;
```

Note that the translation ensures that the identifier (*id1* in the example), which is bound to the final value of the if-expression is used in both branches for the binding, while all the other identifiers are unique.

Note that if the evaluation result of an if-expression is the return value of a function, then the last statements in the two branches of the translated if-statement are return statements. For example, the following code in ATS/Veri

``` ATS/Veri
fun foo () = if (b) then 1 else 2
```

is translated into

``` ATS/Veri
foo () {
    if (b) {
        return 1;
    } else {
        return 2;
    }
}
```
**Function Invocation**

If the simple expression ($sexpr$) in a binding statement is an invocation of user defined functions or primitives with effects, then the binding statement is further translated into a function call ($func\_call$) and a load-ret statement ($load\_ret$). For example the following code

\[
x = \text{foo} (1);
\]

is further translated into

\[
\text{foo} (1);
\text{loadret } x;
\]

Note that the load statement is omitted if the type of expression is **void**.

**Tail-Call**

If after translation, the last three statements of a function in IR are a function call, a load-ret statement, and a return statement for the identifier in the load-ret statement, then these three statements need to be combined into one tail-call statement (**tail-call** in synatx). For example, the following code in IR

\[
\text{foo} ();
\text{loadret idn};
\text{return idn};
\]

is further translated into the following

\[
\text{tailcall foo ()}
\]

One common situation when such merge of statements is needed is that the model in ATS/Veri contains code for manipulating proofs after making a function call which is actually a tail-call if such proof code is erased.
4.5 Operational Semantics

A model in ATS/Veri is used to specify the behavior of a collection of threads that execute concurrently on a shared memory machine and communicate with each other through shared variables. Intuitively, by simulating the execution of a ATS/Veri model, we can generate a state transition system, which can be verified against various properties. Such state transition systems are the semantics of models in ATS/Veri. In this section, I give operational semantics rules of ATS/Veri, which defines how the state transition system for any given ATS/Veri model is to be generated. And such semantics rules are called the operational semantics of ATS/Veri.

4.5.1 Formal Definition

To help formally state the semantics rules, I give out as follows the definition of abstract objects that correspond to asynchronous threads, variables, and various synchronization objects. Then I define the concept of global system states and state transitions, corresponding, respectively, to nodes and edges in a state transition system.

**Definition 4.5.1. Binding**

A binding is a tuple (name, type, curval) where

- *name* is an identifier for the binding that is unique within the whole system,
- *type* is the type for the value bound to the name,
- *curval* is a value from the domain indicated by the type.

**Definition 4.5.2. Global Variable**

A global variable is a tuple (gv_id, type, curval) where

- *gv_id* is a natural number that uniquely identifies the variable,
- *type* is the type for the value contained in the variable,
- *curval* is a value from the domain indicated by the type.
Definition 4.5.3. Frame
A frame is a tuple \((\text{ins}_\text{addr}, \text{bindings})\) where

- \(\text{ins}_\text{addr}\) is a positive integer from a set of instruction addresses (Def 4.5.5), and we call it the **instruction pointer** of the frame,
- \(\text{bindings}\) is a finite set of bindings.

Definition 4.5.4. Thread
A thread is a tuple \((\text{tid}, \text{ins}_\text{addr}, \text{stack})\) where

- \(\text{tid}\) is a natural number that uniquely identifies the thread,
- \(\text{ins}_\text{addr}\) is a positive integer for the address of the instruction to be executed (Def 4.5.5,
- \(\text{stack}\) is a sequence of **frames** (Def 4.5.3).

Note that for readability, in the following sections, we call \(\text{tid}\) the **id** of the thread, \(\text{ins}_\text{addr}\) **current instruction pointer** of the thread, \(\text{stack}\) the **stack** of the thread. Also the **bindings** in the first frame of the stack is called the **current bindings** of the thread.

Definition 4.5.5. Instruction
An instruction is a tuple \((\text{ins}_\text{addr}, \text{scope}, \text{effect})\) where

- \(\text{ins}_\text{addr}\) is a positive integer that uniquely identifies the instruction, and we call it the **address** of the instruction,
- \(\text{scope}\) is either global or local,
- \(\text{effect}\) is a function that modifies the global **system state** (Def 4.5.7).

Definition 4.5.6. Mutex
A Mutex is a tuple \((\text{mid}, \text{taken}, \text{waitings})\) where

- \(\text{mid}\) is a natural number that uniquely identifies the mutex,
- \(\text{taken}\) is a boolean value,
- \(\text{waitings}\) is a sequence of natural numbers, each of which is a **tid** of a **thread** (Def 4.5.4) in **threads**.
Definition 4.5.7. System State

A system state is a tuple \((\text{cur}_\text{tid}, \text{runnables}, \text{gvals}, \text{mutexes}, \text{threads}, \text{code}, \text{preemptable})\) where

- \(\text{cur}_\text{tid}\) is a natural number for the \(\text{tid}\) of a thread (Def 4.5.4) in \(\text{threads}\),
- \(\text{runnables}\) is a sequence of natural numbers, each of which is a \(\text{tid}\) of a thread (Def 4.5.4) in \(\text{threads}\),
- \(\text{gvals}\) is a finite set of bindings (Def 4.5.1) and is called global bindings,
- \(\text{mutexes}\) is a finite set of mutexes (Def 4.5.6),
- \(\text{threads}\) is a finite set of threads (Def 4.5.4),
- \(\text{code}\) is a finite set of instructions (Def 4.5.5),
- \(\text{preemptable}\) is a boolean used to enforce the semantics of atomic (Section 4.5.5).

Note that for readability, in the following sections, we call \(\text{cur}_\text{tid}\) the current thread id of the system, \(\text{gvals}\) the global bindings, \(\text{threads}\) the thread pool. Also the thread whose id is equal to the current thread id is called the current thread, and the current bindings of the current thread is called the local bindings.

Given a model in ATS/Veri, we can generate the instructions accordingly, which is described below.

4.5.2 Create Instructions from a Model

Given a model in ATS/Veri, we first translate it into IR as described in section 4.4.1. For each statement, we create an instruction (Def 4.5.5) with a unique positive integer as its \(\text{ins}_\text{addr}\). The effect function depends on the type of an instruction (e.g. binding or return statement) as well as the function invocation it may contain.

Note that the effect of an instruction includes the change of the current instruction pointer of the thread to the address of certain next instruction. The choice of next instruction reflects the control flow of the IR code, e.g. a binding statement will choose the literally “next” statement in the IR code as the next instruction.
Going further, in ATS/Veri, it is allowed to write code in global scope (not in any function definition). Such code is translated into instructions whose scope is global, while the other instructions would have local as their scope. Certain (non-return) instructions in the global scope may not have next instructions at all. Such instructions lead to the end of the main thread.

4.5.3 Generation of State Transition Systems

With the definitions above, I now describe the operational semantics of ATS/Veri, i.e. the generation of the state transition system from a model in ATS/Veri. Intuitively, the generation process is similar to the execution of an ATS program on a virtual machine equipped with advanced features for scheduling and synchronization.

Given a model in ATS/Veri, the initial system state is a tuple \((\text{cur}_\text{tid} = 0, \text{runnable} = [], \text{gvals} = \{\}, \text{mutexes} = \{\}, \text{threads} = \{\text{thread}0\}, \text{code}, \text{preemptable} = \text{false})\) where \text{code} is the set of instructions translated from the model, and \text{thread}0 is a tuple \((\text{tid} = 0, \text{stack} = [(\text{ins}_\text{addr}, \text{bindings} = \{\})])\) where \text{ins}_\text{addr} is the address of the first global instruction.

Given a system state \((\text{cur}_\text{tid}, \text{runnable}, \text{gvals}, \text{mutexes}, \text{threads}, \text{code}, \text{preemptable})\), the next system state is generated as follows: If the current thread id is not \(-1\), then we use it as the thread id, otherwise we choose non-deterministically from the \text{runnable} a thread id. The chosen thread id is then stored into \text{cur}_\text{tid}. Based on the thread id, we can find a thread in the thread pool. Get the current instruction pointer of the thread, based on which we can find an instruction from \text{code}. We then apply the effect of the instruction to the system state and get a new system state. Considering that we may pick up the thread id non-deterministically, we may create multiple transitions to different new system states from one state.

If the \text{runnable} is empty, there is no outgoing transition from this state. Under such condition, if the thread pool is empty as well, then the current system state is a
normal end state, otherwise it is a state of deadlock.

Keep repeating the operation until no new states can be reached, then we have the whole state transition system corresponding to the model in ATS/Veri.

4.5.4 Effects of Instructions

As discussed in Section 4.5.3 the generation of the state transition system from an ATS/Veri model is heavily dependent on the effect of each instruction of the model. In this section, I illustrate the effects of these instruction as follows.

- name-binding: \( id = sexpr \)

  According to the translation process described in Section 4.4.2, the \( sexpr \) cannot be invocation of user defined functions or primitives with effects, but instead pure computations. All the identifiers in \( sexpr \) can be resolved in the global bindings and the local bindings for the computation of \( sexpr \). With the value for \( sexpr \), its type as well as the name \( id \), a binding (Def 4.5.1) is created and is inserted to the global or local bindings corresponding to the scope of the instruction.

  If the instruction does not have the next instruction, then the current thread is removed from the thread pool, the value of current thread id is removed from the runnables, and the current thread id is set to \(-1\). Otherwise, the current instruction pointer of the current thread is set to the address of the next instruction.

- if-stat: \( \text{if} \ (v) \ \{ \ldots \} \ \text{else} \ \{ \ldots \} \)

  The effect of an if-stat instruction is intuitive. It gets the value of \( v \) from global or local bindings. Then it would set the current instruction pointer of the current thread to the address of the appropriate next instruction according to the truth value of \( v \).
• **func-call**: `fname (v1, v2, ...)`

This instruction gets values for all the input arguments first, then creates a set of bindings based on the id’s of the parameters of the function as well as the arguments. After this, the instruction creates a new frame (`ins_addr`, `bindings`) where `ins_addr` is the address of the next instruction, and `binding` are those newly created bindings from function arguments. This new frame is then inserted into the stack of the current thread.

If the function is a user-defined function, then the current instruction pointer of the current thread is set to the address of the first instruction of the function. Otherwise the function must be a primitive with effect. In this case, the current instruction pointer of the current thread is set to the address of the next instruction. Also the effect of the primitive is applied on the current system state as described in Section 4.5.5.

• **return-stat**: `return v`

This instruction gets the value of `v` first, then creates a binding based on this value, its type, a special name `id_ret`, and inserts the binding to the local bindings. It then gets the value of the instruction pointer in the first frame of the stack of the current thread, and sets the instruction pointer of the current thread to this value.

• **load-ret**: `loadret id`

This instruction gets the value for the special name `id_ret` from the local bindings, and then removes the first frame in the stack of the current thread. With the return value, its type as well as the name `id`, a binding (Def 4.5.1) is created and is then inserted to the global or local bindings corresponding to the scope of the instruction.
If the instruction does not have the next instruction, then the current thread is removed from the thread pool, the value of current thread id is removed from the runnables, and the current thread id is set to $-1$. Otherwise, the current instruction pointer of the current thread is set to the address of the next instruction.

- **tail-call**: $\text{tailcall fname (v1, v2, ...)}$

  This instruction gets values for all the input arguments first, then creates a set of bindings based on the id’s of the parameters of the function as well as the arguments. After this, the instruction creates a new frame $(\text{ins\_addr, bindings})$ where $\text{ins\_addr}$ takes the value of the instruction point in the first frame in the stack of the current thread, and $\text{binding}$ are those newly created bindings from function arguments. Then the first frame in the stack of the current thread is removed. After this, the new frame is inserted into the stack of the current thread.

  If the function is a user-defined function, then the current instruction pointer of the current thread is set to the address of the first instruction of the function. Otherwise the function must be a primitive with effect. In this case, the current instruction pointer of the current thread is set to the value of the instruction point in the first frame in the stack of the current thread. Also the effect of the primitive is applied on the current system state as described in Section 4.5.5.

- **thread-exit**

  The is a special instruction which removes the current thread from the thread pool, and removes the value of current thread id from the runnables, and the current thread id is then set to $-1$. 
4.5.5 Effects of Primitives

Thread Operations

- **conats_thread_create (tfun, arg, tid)**

This primitive gets values for all the input arguments from the local bindings including a function id (*tfun*), from which we can get all the instructions comprising the body of the function, an argument *arg* to call the function *tfun*, and a *tid* as the thread id of the new thread.

A new frame (*ins_addr*, bindings) is created, where *ins_addr* takes the address of the special instruction *thead-exit* as discussed in Section 4.5.4, and *binding* includes only one binding for the argument of the function *tfun*. Then a new thread (*tid*, *ins_addr*, stack) is created, where *tid* is the value from the local bindings, *ins_addr* is the address of the first instruction of the function *tfun*, and *stack* contains the newly created frame.

Then the newly created thread is added into the thread pool, the value of its thread id is added into the runnables. If the preemptable in the system state is true, then the current thread id is set to −1 indicating a new round of scheduling.

Synchronization Operations

- **conats_mutex_acquire (m)**

This primitive gets *m* from the local bindings, which is the identifier of the mutex. From *m*, the corresponding mutex can be found in the *mutexes* of the system state, which is of the form (*mid*, taken, waitings). If taken is false, then it is set to true. Otherwise, the value of current thread id is removed from the runnables and then added to the waitings of the mutex. Finally, if
the preemptable in the system state is true, the current thread id is set to \(-1\) indicating a new round of scheduling.

- \texttt{conats_mutex_release (m)}

This primitive gets \(m\) from the local bindings, which is the identifier of the mutex. From \(m\), the corresponding mutex can be found in the \textit{mutexes} of the system state, which is of the from (mid, taken, waitings). If the sequence waitings is not empty, then one thread id is chosen non-deterministically from the waitings and is then added to the runnables. Otherwise the taken is set to false. Finally, if the preemptable in the system state is true, the current thread id is set to \(-1\) indicating a new round of scheduling.

\textbf{Model Checking Operations}

- \texttt{mc_atomic_start ()}

The primitive sets the preemptable in the system state to false.

- \texttt{mc_atomic_end ()}

The primitive sets the preemptable in the system state to true.
Chapter 5

Model Checking ATS/Veri

In the previous chapter, I illustrate the design of ATS/Veri as a modeling language with advanced types including both dependent types and linear types, and demonstrate its usage for modeling concurrent systems by examples. In this chapter, I focus on taking advantage of existing model checking techniques to realize the correctness verification of constructed models in ATS/Veri.

On one hand, state transition system (automata) is a well accepted formalism used by many prevalent model checking techniques (e.g. Automata-Theoretic Approach (Vardi and Wolper, 1986)). On the other hand, formal definition for the operational semantics of ATS/Veri, which simulates multi-threaded program evaluation on a multi-core machine with shared resources, is also given in the form of state transition systems in Section 4.5. Based on these, I choose to build tools to translate models in ATS/Veri to formalisms supported by corresponding tools. Note that, the translation should ensure a bisimulation relation between the two state transition systems behind ATS/Veri and the input formalisms of the targeting model checking tools.

Various model checkers provide distinct formalisms as front-end modeling languages to enable users to model systems with different characteristics as well as specify properties of different kinds more naturally and succinctly. (For example, the Alloy language (Milicevic et al., 2015) based on first-order relational logic is good for describing a set of structures but has no native support for specifying reactive sys-
tems.) Currently, I choose to translate models in ATS/Veri to the modeling language CSP# (Sun et al., 2009a), accepted by the model checker PAT, a Program Analysis Toolkit (Pat, 2014). I am motivated to choose CSP# by its flexibility and expressiveness as a modeling language and also by the strong model-checking support it receives from PAT. For modeling, CSP# combines features from both CSPM (Scattergood and Armstrong, 2011) and Promela (Holzmann, 2003), integrating high-level CSP-like process operators with low-level program constructs (e.g., assignments and while-loops). On one hand, it supports synchronized event with multiple data fields - the essence of CSP formalism (Roscoe, 1997). On the other hand, it provides shared variables, synchronous/asynchronous channels, block statements, and extension ability with C# code for functions and user-defined data types. For model-checking, PAT is designed and implemented as a self-contained framework to support reachability analysis, deadlock-freeness analysis, full LTL model checking under various granularity of fairness assumption, refinement checking, linearizability verification, and etc. In addition, PAT provides full-fledged supports of model simulation as well as counter-example presentation.

5.1 Crash course of CSP#

Before describing how to translate from models in ATS/Veri to CSP#, a brief primer on CSP# and corresponding model checker PAT is necessary. A much more thorough treatment of the language as well as the tool can be found in (Sun et al., 2009a), (Sun et al., 2009b), and (Sun et al., 2008).

5.1.1 Syntax of CSP#

CSP# is a modeling language which extends Hoare’s CSP (Hoare, 2004) with new language features. CSP# integrates high-level modeling operators (e.g., parallel composition, choice, interrupt, channel communication, etc) with shared variable and
low-level sequential codes, for the purpose of efficient mechanical system verification. Part of the syntax of CSP# is given in Figure 5-1 which will be used in the later content. $P$ and $Q$ are processes, $b$ is a boolean expression, $e$ is an event attached with a statement block of sequential program ($program$). It is worth mentioning that a process definition can have multiple parameters, which can be accessed within the body of the process. We can instantiate a process by invoking its name ($ident$) with corresponding arguments ($explst$). Besides event-based processes, a model in CSP# can contain definition of global variables, which can be updated by the sequential code in $program$. Such code may contain local variables, if-then-else, while, math function, and etc. Lastly, we can extend CSP# by implementing new data types as well as related operations in C# and exporting them into CSP#. Such feature is heavily exploited for building advanced data structures used in the translated models from ATS/Veri.

5.1.2 Operational Semantics

A model in CSP# consists of a process expression $P$ and a valuation function $V$ mapping a variable name to its value. The operational semantics of a CSP# model is a labelled transition system (LTS): $L^V_P = (S, init, \rightarrow)$ where $S$ is the set of reachable system configurations, $init$ is the initial configuration ($V, P$) and $\rightarrow$ is a labelled tran-
sition relationship conforming to the operation semantics of CSP#. The operational semantics of CSP# is presented as firing rules associated with each process construct in (Sun et al., 2009c).

5.2 Model Checking ATS/Veri with CSP#

The semantics of both ATS/Veri and CSP# is based on state transition systems. In this section, I discuss about the translation from ATS/Veri models to CSP# models while ensuring a bisimulation relation between the underlining state transition systems, and argue that such transformational approach for model checking is sound and complete.

In my design, a CSP# model translated from ATS/Veri consists of two parts. The first one is an application-specific part, describing the behavior of threads. The second a platform-specific part, which is a reusable and parameterized CSP# code base for the description of lower level software system concepts (e.g., thread identity, stack frame, synchronization, scheduling, and etc.), which are normally viewed as services provided by the operating system. Figure 5-2 rough describes the content of a translated model in CSP#.

5.2.1 The Application-Specific Part

The application-specific part of a generated CSP# model consists of a set of processes, each of which corresponds to a function in ATS/Veri, as well as a set of global variables in CSP# holding all the user-defined global bindings.

I will use the following example to demonstrate how a function in ATS/Veri is translated into a process in CSP#. The code below is a simple implementation of a factorial function in ATS/Veri.

```plaintext
// Code in ATS/Veri
fun fact (x: int):<fun1> int =
// Platform-Specific Part
1. Global variables for system identities (global variable in ATS/Veri, mutex, stack, and etc.).

2. Processes manipulating system identities.

3. Global variables and processes for scheduling operations.

// Application-Specific Part
1. Global variables for user defined global bindings.

2. Processes translated from user-defined functions.

// The initial configuration of the CSP# model
model = {initialize global variables}
-> (main (0) ||| random_scheduler)

Figure 5.2: Content of a Translated Model in CSP#

if x <= 1 then 1 else let
val y = fact (x - 1)
in
x * y
end

The corresponding code in IR is shown below, which covers name-binding, if-stat, return-stat, func-call, and load-ret instruction.

fact (x) {
id1 = (x <= 1);  // name-binding
if (id1) {
   return 1;  // return-stat
} else {
   id2 = x - 1;  // name-binding
   fact (id2);  // func-call
   loadret y;  // load-ret
   id3 = x * y;  // name-binding
}
return id3;  // return-stat

The translation from a function in IR to a process in CSP# is straight-forward. Each process has only one parameter \(tid\), which represents the id of the thread invoking the function. Each thread has a corresponding stack consisting of multiple frames, which is modelled in the platform-specific part discussed later. Note that the original arguments to a function are now passed via the stack, which matches the process of function invocation on a machine.

A name-binding instruction is translated to an event of sequential code for the computation involved \((e1, e3, \text{ and } e6 \text{ in Figure 5·3})\). Note that \text{stack-get} \text{ and } \text{stack-put} \text{ represent code in CSP#, which helps add a new binding to the current frame, and get the corresponding value for a name from the current frame, respectively.}

An if-stat instruction is translated to a conditional choice in CSP#, which is used to form a sequential composition with the process translated from the instructions after the if-stat instruction. (This case is not shown here since there is no instruction after the if-stat instruction.)

A return-stat instruction is translated into an event of sequential code pushing the return value to the current frame of the thread \((e2 \text{ and } e7 \text{ in Figure 5·3})\). Note that \text{stack-put-ret} represents code in CSP# for such purpose.

A func-call instruction is translated into an event of sequential code pushing a new frame onto the current stack and putting corresponding arguments into the frame \((e4 \text{ in Figure 5·3})\), as well as a process corresponding to the function being invoked. Note that \text{stack-add-frame-with} represents code in CSP# for such purpose.

A load-ret instruction is translated into an event of sequential code loading the return value from the stack and popping off the current frame of the stack \((e5 \text{ in Figure 5·3})\). Note that \text{stack-load-ret} represents code in CSP# for such purpose.
A *tail-call* instruction is translated into an event of sequential code replacing a new frame for the current one in the stack and putting corresponding arguments into it, as well as a process corresponding to the function being invoked.

### 5.2.2 The Platform-Specific Part

The platform-specific part consists of global variables, as well as related operations for manipulating them in the form of sequential code and processes in CSP#. Such code is for modeling ATS/Veri features including global variables, mutexes, condition variables, atomicity, and virtual locks.

Going further, stacks for threads are also modelled via global variables in CSP#. Related operations manipulating these global variables (e.g. stack-get, stack-put, stack-put-ret, stack-load-ret, and etc.) are given in the form of sequential code.
The most important part of the platform-specific part is for the modeling of thread id’s as well as a scheduler for the execution of threads in a model. Based on these, many aforementioned concepts (mutex, condition variable, atomicity, stack) are build.

**Thread Id**

Allocator for thread id’s is given in the platform-specific part in the form of a global variable. I implement the allocator in C# and export it as well as related interfaces into CSP# for allocating unique thread id’s. Programmers can use their own set of integers as thread id’s as long as they do not use the thread id of an existing thread to create a new thread. Note that a thread can locate its stack (in a global variable in CSP# managing the stacks of all threads) with its thread id as index. And there is a pre-configured fixed upper bound for such indices. And it is the programmers’ responsibility not to use an integer over such bound as thread id. The model checker would issue an error when encountering such situation during model checking.

**Scheduler**

Intuitively, a scheduler has two major tasks: 1. scheduling the execution of threads; 2. supporting the creation of new threads. The basic design idea of the scheduler goes as follows. A translated model consists of an interleaving of a scheduler and many threads. Only one thread can execute at a time, while the others and the scheduler are blocked. After the thread executes one step, it gives up its execution (blocking itself) and enables the scheduler to start execution. The scheduler picks up a thread randomly and enable the thread to resume execution while blocking itself. A thread can also issue a request to the scheduler to create a new thread executing certain function with arguments accordingly.

Figure 5.4 contains the pseudo-code in CSP# for a scheduler. The global variable `cur_tid` is used to coordinate both the scheduler and threads. The scheduler is
blocked until \textit{cur\_tid} becomes $-1$. Then it can start executing based on the value of \textit{sys\_new\_tid}.

If \textit{sys\_new\_tid} is negative, then the scheduler would randomly pick up a thread id from the set \textit{runnable}, and set it to \textit{cur\_tid}, and then turns into \textit{random\_scheduler}. As such the newly picked thread can start executing, while the scheduler is blocked again. For such purpose, the translation from user-defined functions in ATS/Veri to processes in CSP\# needs to be modified to add the checking and updating of \textit{cur\_tid} at each step. For example, Figure 5.5 shows a rough picture of how a process looks like, in which a process may actively give up execution after each event. The sequential code inside an event is still translated from instructions in the IR, but with one more extra statement \textbf{if (sys\_glb == 0) \{sys\_cur\_tid = -1;\}}, which enables the scheduler to start working. (Note that \textit{sys\_glb} is a global variable used for atomic execution. If it is set to 1, then no scheduling would happen.)

If \textit{sys\_new\_tid} has a non-negative value, then the scheduler would create a new thread with this value as the thread id. The function name (id) as well as corresponding arguments are passed to the scheduler via global variables. When translating a model in ATS/Veri, each function is assigned an id. Also the translation process records all the functions which are used to create new threads and then generates code accordingly for the scheduler to create new processes in CSP\# based on the function id. Note that after the creation of the new thread, the scheduler would schedule the next thread to execute, which may be the newly created thread.

\subsection{Correctness of the Transformational Approach}

The transformational approach for model checking ATS/Veri discussed in previous sections is designed to ensure that there is a special kind of bisimulation relation between a model in ATS/Veri and the translated model in CSP\#. The conformance of the two models can be established based on such bisimulation. Figure 5.6 shows
Global variables
// cur_tid holds the current thread id.
// runnable holds a set of runnable thread id’s.
// nthreads holds the size of runnable.
// sys_new_tid holds the id for the thread to be created
// sys_func_id holds the function id for the thread to be created
// sys_args holds the arguments for the thread to be created

random_scheduler =
[cur_tid == -1]
if (sys_new_tid < 0) {
  // scheduling existing threads
  if (nthreads == 0) {Skip;} // scheduler exits
  else {
    random_scheduling; // activate a runnable thread randomly
    random_scheduler;
  }
} else {
  // creating a new thread
  {Add new thread id to runnable, adjust nthreads,
   set up stack for the new thread
  } ->
  if (sys_func_id == idA) {
    (procA (sys_new_tid); thread_finalize (sys_new_tid))
    ||| random_scheduler
  }
  else if (sys_func_id == idB) {
    (procB (sys_new_tid); thread_finalize (sys_new_tid))
    ||| random_scheduler
  }
  else if ...
  ...
  else {STOP;}
}

Figure 5·4: Modeling of Scheduler

proc (tid) =
[cur_tid == tid] e1 -> [cur_tid == tid] e2 -> [cur_tid == tid] e3 -> ...

Figure 5·5: Process Capable of Scheduling
an example of such relation. The left side of the figure is a state transition system corresponding to a ATS/Veri model, while the right side is a labeled transition system for the corresponding CSP# model.

We use a tuple \((S, T)\) to denote a state transition system behind a ATS/Veri model, where \(S\) is the set of states and \(T \subseteq S \times S\) is the transition relation. Also we use a tuple \((S, T)\) to denote a labeled transition system over the alphabet \(A\) behind a CSP# model, where \(S\) is the set of states and \(T \subseteq S \times A \times S\) is the transition relation.

Given a state transition system \((S, T)\), a labeled transition system \((P, Q)\), and a relation \(R \subseteq S \times P\). We call a state \(p \in P\) an intermediate state if \(\forall s \in S. (s, p) \notin R\). All the other states in \(P\) are called non-intermediate states.

**Definition 5.2.1. Weak Bisimulation**

\(R\) is called a weak bisimulation over \(S\) and \(P\) if

- \(R\) is a one-to-one correspondence;

- For any transition \((s_1, s_2) \in T\), there is a sequence of transitions \((p_1, a_1, p_2), (p_2, a_2, p_3), \ldots, (p_{n-1}, a_{n-1}, p_n)\) each of which \(\in Q\), and \((s_1, p_1) \in R\) and \((s_2, p_n) \in R\), and \(p_i\) is an intermediate state for \(1 < i < n\).

- For any sequence of transitions \((p_1, a_1, p_2), (p_2, a_2, p_3), \ldots, (p_{n-1}, a_{n-1}, p_n)\) each of which \(\in Q\), where both \(p_1\) and \(p_2\) are non-intermediate states, and \(p_i\) is an intermediate state for \(1 < i < n\), there is a transition \((s_1, s_2) \in T\) such that \((s_1, p_1) \in R\) and \((s_2, p_n) \in R\).

Instead of mechanically proving that such bisimulation exists (which would involve a very complex proof and is not the focus of this thesis), I give out some intuition as follows. All the entities in the formal definition of the semantics of ATS/Veri are modelled accordingly in CSP# models. For instance, the user-defined global variables in ATS/Veri correspond to a global store in CSP# recording all the variables. So as the definition of stacks, mutex, and etc. Also the pool of threads are modelled by the
interleaving of corresponding processes as well as global variables including `cur_tid`, `Runnable`, and etc. in CSP#. Going further, the instruction pointer for each thread in ATS/Veri corresponds to the configuration of a process in CSP#. Based on these and the translation process, we can always translate a state in an ATS/Veri model to a state in a CSP# model (setting up configuration of processes as well as values for global variables). Such translation sets up the weak bisimulation relation between the two models.

Finally, it is worth noting that the translation process from ATS/Veri to CSP# guarantees the following. For any sequence of transitions $(p_1, a_1, p_2), (p_2, a_2, p_3), \ldots, (p_{n-1}, a_{n-1}, p_n)$ each of which $\in Q$, where both $p_1$ and $p_2$ are non-intermediate states, and $p_i$ is an intermediate state for $1 < i < n$, there is at most one transition (effectful transition in Figure 5.6) which modifies those global variables involved in the specification of properties to be checked. Based on this property and the weak bisimulation described above, the transformational approach in this thesis is both sound and complete for explicit state model checking.

Figure 5.6: Bisimulation Relation between ATS/Veri and CSP#
Chapter 6

Related Work

Verification of software systems has been a goal of many researchers and programmers over the decades. In this chapter, I will mention some closely related research work on the application of type checking and model checking to program verification.

6.1 Programming Languages with Advanced Types

The dependent types in Martin-Löf’s development of constructive type theory (Martin-Löf, 1985) provides a complementary approach for encoding invariants inside the type system, by refining the types with predicates that describe properties of values. In such manner, very rich invariants can be encoded in the types, which also leads to the undecidability of discharging the constraints resulting from type checking.

To address this, Liquid types (Rondon et al., 2008) exploits predicate abstraction, technique from software model checking, to tackle the problem of inferring dependent type annotations.

Hybrid type checking (Knowles et al., 2006) (Knowles and Flanagan, 2010) provides a pragmatic way for partial verification of sequential programs. A prototype functional programming language called Sage was built, which enables expressive program specifications via types (based a synthesis of dynamic types and refinement types). It performs hybrid type-checking, proving or refuting as much as possible statically, and then inserting run-time checks for unsolved constraints. In ATS/PML and ATS/Veri, a programmer can choose to insert dynamic checks equipped with
dependent types and linear types (e.g. \texttt{pml$assert} in ATS/PML, \texttt{mc$\text{v}lock\_get} in ATS/Veri) at certain granularity based on his or her own understanding and confidence of the underlining reasoning. And these checks are all required to be verified during model-checking.

Maude (Clavel et al., 2001) is a logic framework as well as an executable specification language based on rewriting logic. It supports a powerful form of generic programming, which may provide similar capabilities to parametric polymorphism and dependent types. The Maude LTL model-checker (Eker et al., 2002) supports on-the-fly explicit-state model-checking of concurrent systems expressed as rewrite theories. Thus any program in any programming language can theoretically be model-checked as long as the semantics of the underlying language can be formally encoded in Maude based on a form of rewriting semantics. On the contrary, the rules for the semantics of ATS/PML are defined externally and implemented in the compiler from ATS/PML to Promela. Also the type system in ATS is primarily designed to facilitate practical programming through program verification. We believe that a modeling language based on ATS can greatly help a programmer manually construct models for practical applications as model construction is really just a special form of program construction.

6.2 Modeling Languages and Model Checkers

To verify a system we need to describe two things: the set of facts (system properties) we want to verify, and the relevant aspects of the system that are needed to verify those facts. Languages used for modeling system have direct impact on which aspects of the system are observable, how complicated properties based on basic observable behaviors are described, and how automated verification can be carried out by machines. This section is devoted to a brief introduction to several prominent modeling
languages for concurrent systems and corresponding model checkers.

6.2.1 PROMELA and SPIN

PROMELA (Holzmann, 2003) is an acronym for Process Meta-Language, which is the specification language accepted by the model checker SPIN. The basic building blocks of PROMELA models are asynchronous processes, buffered and unbuffered message channels, synchronizing statements, shared variables, and structured data. Following the goal of describing abstractions of system, there is no innate support for complex data structure, and only macro is support to imitate function definition. But in recent versions (Holzmann and Joshi, 2004), it’s allowed to embed C code within PROMELA model, which makes the model more close to real system implementation. Assertions can be written along side the statements within the PROMELA model, which can be checked by the model checker SPIN. Besides freeness of deadlock, SPIN can also verify properties encoded by Linear-Temporal Logic (LTL) formulae with system states as basic propositions under fairness condition.

6.2.2 CSP\(_M\) and FDR

CSP\(_M\) (Roscoe, 1997), a machine-readable dialect of CSP, combines CSP with a functional programming language, enabling modeling systems with non-trivial data structures or functional aspects. In CSP\(_M\), system behaviors are described as process expressions combined with compositional operators. Besides, CSP\(_M\) enables rich data expressions such as sequences, sets, Boolean, tuples, lambda calculus. It also allows users to define data types using the reserved word “datatype” and functions can be declared following the functional paradigm similar to ML. One characteristic of CSP\(_M\) is that processes have no shared variables. Instead, inter-process communication is supported through synchronized message passing, which in turn is based on event synchronization. The only observable behavior of a process is the events
it communicates with the environment (other processes). Thus the properties to be
verified can be encoded in CSP\textsubscript{M} as processes as well. Then the checker FDR (201,
2010) is used to verify that one process is the refinement of another based on different
semantic models. This is called refinement checking. Model checking LTL properties
is not directly supported in FDR. Some works have been done to take advantage
of the refinement checking offered by FDR to do LTL checking by translating LTL
formulae into processes. Particularly, Leuschel et al. (Leuschel et al., 2001) applied
an emptiness test in a refinement between an unexpected specification and a process,
which is a synchronization of the to-be-verified process and the process for the LTL
formula. The translation between LTL formula and CSP process is arduous and no
counter-example can be given out if the checking fails. Lowe (Lowe, 2008) used a re-
fusal testing model to conduct the refusal refinement between the process representing
the system and the process traslated from LTL formula. However the supported LTL
formulae exclude operators eventually, until, and negation.

6.2.3 CSP# and PAT

CSP# (Communicating Sequential Programs) (Sun et al., 2009c) is a newly creately
modeling language, integrating high-level CSP-like process operators with low-level
program constructs such as assignment and while loops. On one hand, it supports
synchronized events with multiple data fields – the essence of CSP formalism. On
the other, it provides shared variables, synchronous/asynchronous channels, block
statements, and the extension ability with C# code for functions and user-defined
types. In short, CSP# combines features from both CSP\textsubscript{M} and PROMELA. Such
expressiveness of CSP# is supported by the model checker PAT (Process Analysis
Toolkit) (Sun et al., 2009b), which is a self-contained framework to support reach-
ability analysis, deadlock-freeness analysis, full LTL model checking under different
granularities of fairness assumption, refinement checking, verification of linearizabil-
ity, as well as a powerful simulator. Furthermore the layered architecture of PAT allows new modeling languages to be developed easily by providing the syntax rules and semantics. Currently, eleven modules, one of which supports CSP\#, have been developed to support composing, simulating and reasoning of concurrent, real-time systems and other possible domains.

6.2.4 CSP$_M$ and ProB

ProB was initially designed as an animator and model checker for B method (Abrial, 1996), and recently it supports refinement checking of CSP$_M$ (Leuschel and Fontaine, 2008). Furthermore, it provides the capability of LTL model checking for combined CSP and B specification (Butler and Leuschel, 2005). As a whole it presents a new animation and model checking tool for CSP with visual feedback in the source code. The empirical evidence provided in (Leuschel and Fontaine, 2008) shows that ProB outperforms FDR for certain systems while FDR is considerably faster dealing with system models well tuned for it. Therefore these are extreme difference between the two tools due to their design principles and they complement each other.

6.2.5 NuSMV

NuSMV is a model checker based on the SMV (Symbolic Model Verifier) software, which was the first implementation of the methodology called Symbolic Model Checking described in (McMillan, ). This class of model checkers verifies temporal logic properties in finite state systems with “implicit” techniques. NuSMV uses a symbolic representation of the specification in order to check a model against a property. It is able to deal with CTL properties as well as LTL formulae using different techniques including BDD-based symbolic model checking and bounded model checking as specified by users. NuSMV uses the SMV description language to specify finite state machines as input. However the description language is quite low level, which
makes it more difficult to model systems than other languages such as PROMELA.

6.2.6 SAL

SAL (Symbolic Analysis Laboratory) is a framework for combining different tools to calculate properties of concurrent systems. The heart of SAL is a language for specifying concurrent systems in a compositional way. Originally designed as an intermediate language, the language has involoved into a comprehensive specification language in its own right, with the support of a tool suite that includes state of the art symbolic (BDD-based) and bounded (SAT-based) model checkers, an experimental "Witness" model checker, and a unique "infinite" bounded model checker based on SMT solving. Auxiliary tools include a simulator, deadlock checker and an automated test generator. Researches in (Dutertre and Sorea, 2004) (Brown and Pike, 2007) focused on specifying and analysing timed systems by exploiting the infinite bounded model checker of SAL. It has also been used to analyze models in other specification languages such as Z (Smith and Wildman, 2005) and RSL (Perna and George, 2007).

6.2.7 SystemC and KRATOS

A SystemC design is a complex entity comprising a multi-threaded program where scheduling is cooperative, according to a specific set of rules, and the execution of threads is mutually exclusive. (Cimatti et al., 2011) presents KRATOS, a software model checker for verifying safety properties (in the form of program assertions) of SystemC designs. KRATOS uses SYSTEMC2C to translate the SystemC designs into threaded C programs. Combined with a concrete scheduler created based on the SystemC scheduler, such threaded C programs can then be analyzed based on the technique of lazy predicate abstraction. In essence, KRATOS can be viewed as a software model checker for certain forms of C programs, built on top of an extended version of NUSMV (Cimatti et al., 2002), which is tightly integrated with MATHSAT.
SMT solver (Bruttomesso et al., 2008).

6.2.8 IC3

IC3 (Incremental Construction of Inductive Clauses for Indubitable Correctness) (Bradley, 2011) is a new verification paradigm, original proposed for the analysis of sequential circuits. An investigation is presented in (Cimatti et al., 2012) on the application of IC3 to the case of software verification (C program in precise). The authors propose three variants of IC3: first, generalizing IC3 to the case of SMT, provides for the analysis of fully symbolically represented software; second, TREE-IC3, relies on an explicit treatment of the CFG (Control Flow Graph); third, a hybrid approach based on the use of interpolants to improve TREE-IC3. The experimental results show that IC3 is significantly improved by the aforementioned techniques, and the resulting IC3 based algorithms can compete with other existing tools based on predicate abstraction such as KRATOS and CAPCHECKER.

6.3 Model Checking Main Stream Programming Languages

6.3.1 Haskell

In (Brown, 2009), a technique is presented for generating formal models of programs written in Haskell using CHP library without source code analysis. The approach is to take the CHP library and provide a mirror implementation with near-identical Application Programming Interface (API). The mirror implementation does not properly execute the code as the original CHP would, but instead traces the structure of the program and produces a CSP model of the program when the program is executed. However, programs whose control flows are influenced by the value of the data shared between processes are not supported. And also, the model generation won’t halt when applied to buggy programs with non-terminating pure computations.
languages and their related been focused on extracting formal models from of intermediate level languages.

6.3.2 Java

Java2CSP (Shi, 2000) is a tool for translating Java programs (or more precisely, Java bytecode) into CSP models. It removes all the parts of a given program that do not affect the synchronization behavior of the program, introducing certain CSP *process patterns* to simulate concurrency concepts in Java such as shared variables, threads and monitors. The models generated by Java2CSP can then be checked by a tool like FDR for synchronization behavior (e.g., deadlock, livelock).

JPF (Java Pathfinder) (Visser et al., 2000) is a model checker for checking Java bytecode. Similar to SPIN, it checks all the reachable states of a given program while employing a variety of approaches to tackling the state explosion problem. Implemented in Java, JPF is more modular and readable than SPIN but much inferior in terms of run-time efficiency.

6.3.3 C

A tool is described in (Zaks and Joshi, 2008) for verifying multi-threaded C programs that uses the SPIN model-checker. It compiles a C program into (typed) bytecode in LLVM, and then employs a virtual machine to interpret the bytecode and computes program states under the guidance of SPIN. Most of the virtual machine consists of fragments of C code embedded within a PROMELA model, which can be executed atomically without creating explicit states. When interpreting bytecode, the virtual machine can utilize dynamic partial order reduction so as to alleviate the state explosion problem.

An approach to extracting CSP models from LLVM compiler intermediate representation (IR) of C++ programs is presented ((Kleine and Helke, 2009)). It divides
the low-level representation of a concurrent system into three parts: an application-specific one, which describes thread behavior, a domain-specific one, which encapsulates low-level software concepts such as scheduling and stack frame, and a platform-specific one, which is the hardware model; a CSP model can be extracted from the application-specific part and then combined with (parameterized) CSP models for the other two parts to form a complete CSP model, which can be model checked by existing checkers for CSP such as FDR2 and ProB. I am partly inspired by this approach in my design of model checking ATS/Veri.

6.3.4 Erlang

Erlang (Armstrong, 2007) is a functional programming language developed at Ericsson for the implementation of concurrent, distributed, fault-tolerant systems. McErlang (Fredlund and Svensson, 2007) is a model checker for Erlang programs and it is also implemented in Erlang. It supports a large subset of the Erlang programming language including the distributed and fault-tolerant parts of Erlang. The workflow of McErlang goes as follows. The program to be verified is first translated into a new Erlang program ready for model checking. The generated program is then compiled into Erlang byte code by normal Erlang compiler. Finally the program is run under the McErlang run time system, under the control of a verification algorithm, by the normal Erlang bytecode interpreter. The pure computation part of the code, i.e., code with no side effects, including garbage collection, is executed by the normal Erlang runtime system. However, the side effect part is executed under the McErlang run time system which is a complete rewrite in Erlang of the basic process creating, scheduling, communication and fault-handling machinery of Erlang (comprising a significant portion of the code of the model checker). Naturally, McErlang allows access to the program states and actions between states. A program to be verified can be instrumented by Probes (special Erlang function supported by McErlang runtime)
whose execution shall leave footprint in the program states. A user-provided monitor (written in Erlang as well) can then run in lock-step with the program to examine its states and actions (footprint left by probes especially), thus verifying temporal properties of the program. Such verification mechanism offers great flexibility and expressiveness, while makes the model checking process less time efficient.
Chapter 7

Conclusions and Future Work

7.1 Conclusions

Evolving from Dependent ML (Xi, 2007), the programming language ATS adapts a restricted form of dependent types, in which a dedicated domain (statics) is used to encode invariants about program behaviors. Type checking is decidable due to the decidability of the domain. In such way, programmers can specify and verify pre- and post-conditions that can be expressed over statics. My research starts from the programming language ATS equipped with both dependent types and linear types, and then augments it with primitives of appropriate types for modeling concurrent systems as well as asserting system invariants accordingly. My research results in two new modeling languages ATS/PML and ATS/Veri for concurrent systems.

ATS/PML allows a programmer to naturally rely on types to express ideas (on the design of a model) while staying semantically very close to Promela. On one hand, ATS/PML supports most of the features of Promela, such as guarded blocking, nondeterminism, loops, channel operations, while dropping certain features such as local jump. On the other hand, a model in ATS/PML is just a program in ATS as far as type checking is of the concern. ATS/PML can be viewed as a type-enhanced version of PROMELA in the sense that its semantics is strongly coupled with PROMELA. This design is expected to attract PROMELA users to start exploiting advanced types including dependent types and linear types to detect modeling errors at compile-time while aiming for constructing highly abstract models enabling efficient model checking.
with the SPIN model-checker.

ATS/Veri allows a programmer to construct models for real-world multi-threaded software applications in the same way as writing functional programs with support for synchronization, communication, and scheduling among threads. I formally defined the semantics of ATS/Veri for the development of corresponding model checkers. I also built a compiler to translate ATS/Veri into CSP# in order to exploit the state-of-the-art verification platform PAT for model checking ATS/Veri models. The correctness of such a transformational approach was illustrated based on the semantics of ATS/Veri and CSP#.

7.2 Limitation and Future Work

The two modeling languages ATS/PML and ATS/Veri demonstrate a practical way to combine type checking and model checking synergically for constructing high quality models. However, their usage are limited due to the design of their type systems and syntactic features.

First, both ATS/PML and ATS/Veri inherit the type system of ATS, thus their capability of exploiting type checking to identify flaws during model construction is limited by the expressiveness as well as the usability of the type system of ATS. To be more detailed, the type system of ATS supports a restricted form of dependent types, in which there is a complete separation between statics, in which types are formed and reasoned about, and dynamics, in which programs are constructed and evaluated. Also special constraints are set upon the statics in order to make the type checking procedure decidable. Due to such design of the type system, programmers of ATS/PML and ATS/Veri may need to encode the specification of program properties in the form of types in an appropriate way as well as provide proofs along side models to facilitate type checking. In this way, all constraints generated during type check-
ing can be solved without relying on model checking to check their validity. Though programmers in ATS/PML and ATS/Veri can use model checking to discharge constraints which cannot be solved during type checking, they should take advantage of such feature only in cases when type checking is impossible or too complicated.

Second, to facilitate the paradigm of programming with theorem proving, ATS/PML adopts the syntax of ATS with certain constraints. Mapping rules are set up carefully to bridge ATS/PML and PROMELA. However, not all PROMELA models can be syntactically rewritten in ATS/PML. For example, programmers may not be able to translate certain PROMELA models with heavy usage of `goto` statements to ATS/PML relying on mutually recursive tail-call. This limitation can be mitigated when programmers construct models directly in ATS/PML, which stimulate programmers to model systems in a way amenable to formal verification via type systems.

Third, for the modeling language ATS/Veri, programmers have to specify those temporal properties to be verified following the syntax of CSP#. Also refinement checking is not supported for ATS/Veri though the underline model checker used (PAT) supports it. I plan to improve the design of ATS/Veri to incorporate the specification of temporal properties inside models as well as support refinement model checking.

Fourth, in this thesis, my research focuses on verifying concurrent software systems, such as communication protocols and controllers, whose behaviors can be modelled by automata. A possible next step is to exploit the methodology of combining type checking with model checking synergically to the verification of systems with crucial timing or probabilistic behaviors. How types can help encode time-related properties, and be combined with model checking to verify the correctness of these properties remains an interesting topic.
Last, examples (Chen et al., 2015) have shown the urge as well as success of applying model checking techniques on complex software systems at the industrial level. It is worthwhile to make ATS/Veri to fully support most features of ATS as a practical programming language so that the advanced type system of ATS can help ensure that models in ATS/Veri actually encode the proposed designs in programmers’ mind. However the problem about how to efficiently apply model checking on highly sophisticated systems of practical value remains unsolved. To address this, the employment of efficient analysis and compression algorithms offered by the model checking community is deemed necessary and promising. Following this lead, I plan to incorporate LTSmin (Kant et al., 2015) (Laarman et al., 2011), a high-performance generic model checking toolset, to support model checking of the extended ATS/Veri.
References

(2010). Failures-Divergence Re


160


