Building Graphical Promela Models using UPPAAL GUI

Master's Thesis Report

by

Vasu Hossaholal Lingegowda
Software Systems Engineering
Group: B2-201

under the guidance of

Dr. Alexandre David

Department of Computer Science
Aalborg University, Aalborg
Spring 2006

Abstract

The report proposes a new graphical input specification language g-Promela for SPIN model checker. Basic features of PROMELA language are defined using the formal syntax of graphical elements as an extension of labeled transition systems. The labels are specified with valid PROMELA statements. To facilitate the handling of g-Promela specifications, we have sketched the design of the Graphical Promela Interface(GPI) toolset, which is built upon the existing UPPAAL GUI. The graphical models are then translated to textual PROMELA to verify temporal properties using SPIN engine in the background. The translation is automated by building a parser which is part of the adapter tool built between the User Interface and SPIN engine. As a prime example, peterson's Mutual Exclusion Algorithm is modeled and verified.

Acknowledgements

I would like to express my gratitude to the people who supported me during the preparation of this thesis. Professor Alexandre David always had time for patiently supervising my work. I did benefit from his valuable support and guidance, and from many fruitful discussions. Finally, no part of my studies had been possible without the generous support from my parents. Thank you all!

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

The work presented in this report aims at providing a graphical specification language to model checker SPIN [1] and a tool editor for building and verifying the correctness of graphical models. SPIN is a tool for analyzing the logical consistency of concurrent systems, specially data communication protocols. The system is described in PROMELA [2], a textual based modeling language having c-like syntax. Inspired by easy-to-use graphical based tools such as UPPAAL [3, 4], we propose a graphical approach equivalent to textual PROMELA.

The newly defined graphical specification language is called g-Promela. The models are based on the formal models of LTS and is then mapped to an implementation (a textual PROMELA program). Formal rules are defined for translation from LTS model to generate a PROMELA specification that can be input to the SPIN model checker to verify a wide range of properties. To define the formal syntax and semantics for the graphical elements, we use the existing graphical notations of UPPAAL. Tools such as UPPAAL with their interactive, easy to use graphical interface and use of the extended version of Labeled Transition Systems (LTS) with clocks and invariants (called Timed Automata [5]) have gained wide acceptance in recent years. The reasons the selecting SPIN and UPPAAL out of various existing tools is given in the subsequent section after a brief introduction for model checking in general.

1.2 Model Checking

As defined by Clarke & Emerson in 1981, Model checking is "an automated technique that given a finite-state model of a system and a logical property, system-

atically checks whether this property holds for that model". Model Checking is mostly employed in verification and validation of concurrent processes, communication protocols and reactive systems. It has several advantages over traditional approaches to these kind of problems such as simulation, testing and deductive reasoning. Popular tools include: SPIN, Design/CPN, UPPAAL, NUSMV2 and Kronos.

Model checking limits itself to systems where decidability is guaranteed (e.g. systems with only a finite number of state bits). Given sufficient amount of time and memory, a model checking tool is guaranteed to terminate with a YES/NO answer. Instances of finite state systems handled with model checking include e.g. hardware controllers and communication protocols. In some cases bugs can be found from infinite state systems by restricting them to finite state ones. One can for example model message FIFOs of infinite size with bounded size FIFOs, and still find some of the bugs bugs which appear in the (harder) infinite-state case. Note that if no bugs are found in the finite-state version, that does not mean that the infinite-state version is correct!
In model checking process the following phase can be identified:

- Modeling - How to model the system in a way acceptable from a model checking perspective.
- Specification - What properties should the system satisfy? Most model checkers use temporal logic to specify the properties.
- Verification - Push the "model check" button. In practice life is not this easy, and analysis of the model checking results is needed. If for example a property does not hold, where does the bug exist? Model checkers produce counterexamples which help in locating the bug. The bug might also be in the specification or in the system model, so these must be analyzed carefully.

With main focus to validate and verify common design flaws in distributed systems such as deadlock, livelock, underspecification, overspecification, violations of constraints and real-time performances, there are wide variety of Model checking tools available. Selecting the right tool for the job is sometimes hard. In the next sub-sections, a survey of various popular tools has been discussed. SPIN and UPPAAL are discussed and compared in greater detail as both act as prime tools in the report.

1.3 Comparison of Model Checking tools

1.3.1 SPIN

SPIN is a widely distributed software package that supports the formal verification of distributed systems. The software was developed at Bell Labs in the formal methods and verification group starting in 1980 and since then targets only efficient software verification and not hardware verification.

SPIN uses a high level language to specify systems descriptions, called PROMELA (PROcess MEta LAnguage). SPIN has been used to trace logical design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols, etc. The tool checks the logical consistency of a specification. It reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes.

SPIN works on-the-fly, which means that it avoids the need to construct of a global state graph, or a Kripke structure, as a prerequisite for the verification of any system properties.

SPIN can be used as a full LTL model checking system, supporting all correctness requirements expressible in linear time temporal logic, but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Many of the latter properties can be expressed, and verified, without the use of LTL. Correctness properties can be specified as system or process invariants (using assertions), or as general linear temporal logic requirements (LTL), either directly in the syntax of LTL, or indirectly as Büchi Automata (called never claims).

SPIN supports both rendezvous and buffered message passing, and communication through shared memory. Mixed systems, using both synchronous and asynchronous communications, are also supported. Message channel identifiers for both rendezvous and buffered channels, can be passed from one process to another in messages.

SPIN supports random, interactive and guided simulation, and both exhaustive and partial proof techniques. The tool is meant to scale smoothly with problem size, and is specifically designed to handle even very large problem sizes. To optimize the verification runs, the tool exploits efficient partial order reduction techniques, and (optionally) BDD-like storage techniques.

SPIN can be used in three basic modes:

1. as a simulator, allowing for rapid prototyping with a random, guided, or interactive simulations

2. as an exhaustive state space analyzer, capable of rigorously proving the validity of user specified correctness requirements (using partial order reduction theory to optimize the search)

3. as a bit-state space analyzer that can validate even very large protocol systems with maximal coverage of the state space (a proof approximation technique).

1.3.2 UPPAAL

UPPAAL is a tool for modeling, simulation and verification of real-time systems. UPPAAL is developed jointly by Uppsala University and Aalborg University. UPPAAL usage is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real valued clocks, communicating through channels or shared variables. The verification is done by automatic model-checker engine. Using UPPAAL we can construct abstract models of any system, simulate its dynamic behavior, specify and verify its *safety* and *bounded liveness* properties which can be useful to analyse and design embedded systems and real time systems. It consists of two main parts: a graphical user interface and a model-checker engine. It requires that Java 1.5 is installed on the computer. The engine part is by default executed on the same computer as the user interface, but can also run on a more powerful server.

UPPAAL has three main utilities: a description language, a simulator, and a model-checker. The description language is useful to represent the system as collection of inter related Timed Automata. Description language also has several data variables which can be applied on these Timed Automata designed with its help. The simulator and verifier are useful to graphically analyse the system behavior by considering various constraints on state space generated by the system. The simulator graphically represents the system behavior, it represents each and every transition the system took and the corresponding values for each transition. Verifier on the other hand is useful to check the properties of the system. In the query box we can verify any property whether its satisfiable or not for the whole state space generated by the system.

The following list summaries the selected features for comparison of SPIN and UPPAAL tool with related verification systems.

SPIN

- Modeling formalism: Promela language, systems can be seen as a set of synchronized extended finite state machines
- Model Checkers: on-the-fly LTL, safety through assertions, LTL to Büchi translation
- Other features: partial order reductions and bit-state hashing can be combined with LTL model checking, model slicing
- Comments: Very fast state space generation, fast partial order reduction algorithm,hash table stored in physical memory (instead of on a file on hard-disk). Has a primitive user interface XSPIN to display the random traces.
- Suggested uses: Modeling of communication protocols, LTL model checker

UPPAAL

- Modeling formalism: Networks of Timed Automata
- Model Checkers: subset of CTL(Computation tree logic)
- Other features: clocks, integer variables, structured data types & channel synchronization
- Comments: ported to different platforms and in constant development of various flavors such as cost-UPPAAL, distributed-UPPAAL,T-UPPAAL.
- Suggested uses: communication protocols, multimedia applications

NUSMV2

- Modeling formalism: SMV input language, a simple circuit description language
- Model Checkers: BDD based CTL and LTL model checkers (under fairness), bounded model checking with LTL (including past operators).
- Other features: Deadlock checking, computing the number of reachable states, simulation
- Comments: A strong BDD based CTL model checker (a SMV rewrite), a reasonable bounded LTL model checker
- Suggested uses: verifying digital circuits (or systems easily modeled as circuits), CTL under fairness model checking, bounded model checking

MARIA

- Modeling formalism: Algebraic Petri nets (including P/T-nets)
- Model Checkers: on-the-fly LTL model checking under (strong and weak) fairness, safety
- Other features: extensive support for structured data types, parallel safety model checker
- Comments: useful for systems with complex data manipulations, uses disk to manage larger state spaces
- Suggested uses: systems with lots of fairness constraints, as a back-end for programming languages (data type support eases this tremendously)

SPIN vs UPPAAL

Both the tools are quite popular and have been used to detect design errors in applications from many different domains. Both the tools has a different input language. SPIN's input language, PROMELA, is a state-based, imperative language. Whereas, UPPAAL's input language is a specific class of timed automata, combining both action-based and state-based features. Both the specification languages used for systems verification are mainly focused on the specification of process interaction at the system level. Each tool handles time differently but, has a similar temporal logic for expressing properties of a model: SPIN uses LTL while UPPAAL uses TCTL respectively. Each of these tools use a different strategy for verification: SPIN does model checking on-the-fly. UPPAAL checks invariant and liveness properties by on-the-fly exploration of the state space of a system in terms of symbolic states represented by constraints.

1.4 Related Work

Recent technical developments have made concurrent behavior to be animated, mechanically analyzed and then verified automatically by making design models visual. v-Promela, a visual object-oriented language interface for SPIN also reconciles Promela with graphical notations for design of concurrent systems using UML-RT and ROOM. It introduces a graphical notation to describe both the behavior and structure of a system in a hierarchical fashion [6]. Attempts have been made to translate UML activity diagrams and state charts to PROMELA code. [7]
SPIN also serves as a background engine in many other tools, e.g.:PEP, Bandera [8], Java Pathfinder-1, Approve, VIP, JSPIN.

1.5 Outline

The remainder of the report is structured as follows. Chapter 2 defines the syntax and semantics of graphical elements introduced in g-Promela language. Chapter 3 details rules that effective translation of g-Promela models into PROMELA and presents some examples of property verifications. In Chapter 4, describes the implementation of the adapter tool which communicates between the Graphical Promela Interface(GPI) and the verification engine - SPIN. The last chapter outlines conclusion and hints future work.

Chapter 2

g-Promela language description

g-Promela is a the graphical description language. It is a non-deterministic guarded command language with data types and is defined as a extended labelled transition system. g-Promela language features are discussed by defining the syntax of the graphical elements introduced and also of the basic textual constructs inherited from PROMELA. Since g-Promela language inherits the basic structure of the model, constructs, syntax and semantics of traditional textual Promela language, we start with a brief summary of the structure of Promela language constructs.

2.1 PROMELA language summary

The Spin Model Checker have been developed for the analysis and verification of communication protocols. Its input language syntax is derived from C and control statements based on Dijkstra's guarded commands. Also it uses the denotations for communications from Hoare's CSP language: the send operator is represented by an exclamation mark and the receiver operator is represented by an question mark [9, 10].

In Promela, system components are modeled as processes that can communicate via channels either by buffered message exchanges or rendez-vous operations, and also through shared memory represented as global variables. The execution of statements is asynchronous and interleaved, which means that in every step only one enabled action is performed, without any assumptions of the relative speed of process executions.

Given as input a Promela model, Spin generates a C program that performs a verification of the system by scanning the state space using a depth-first search (DFS) algorithm. This way, both safety properties such as absence of deadlock, unspecified message receptions, invalid end states and assertions can be checked,

as well as liveness properties such as non-progress cycles and eventual reception of messages. The so-called never claims, which are best seen as monitoring processes that run in lock step with the rest of the system, are the most general way to express properties in Spin. Being Buchi Automata, they can express arbitrary omega-regular properties. Spin provides an automatic translator from formulae in linear-time temporal logic (LTL) to never claims. When errors are reported, the trace of actions leading to an invalid state or cycle is saved, so that the erroneous sequence can be replayed as a guided simulation. For large state spaces methods, Spin features state-vector compressing, partial-order reduction and bit-state hashing. The following sub-sections of this report, introduces the primitive language features of textual PROMELA.

2.1.1 PROMELA Model

Promela is a language developed mainly to be used for specifying and validating protocol models. On a given layer of a protocol there is a need for specifying procedure rules in a formal way in order to describe the service provided at the layer. Promela is able to specify and verify the procedure rules. Note that the Promela language is an abstraction of the protocol so only the model is validated. The model describes the interaction of processes in a distributed system but does not deal with implementation details such as how a message is transmitted, encoded or stored.

By concentrating on the design of the interactions between processes, the foundation of the protocol can be validated without worrying about minor details. The model should be as simple as possible but also sufficiently detailed to represent all types of coordination problems that might occur in distributed systems.

The Promela language uses three types of objects when defining validation models:

- processes
- message channels
- state variables

All processes are global objects, whereas variables and channels can be either global or local to a process.

2.1.2 Executability of statements

Concerning executability in Promela there is no difference between conditions and statements. All statements have a truth value of the statement's executability.

This means that all statements are either executable or blocked, depending on the current values of variables or the contents of message channels. Some statements are always executable such as assignments and declarations. The following examples should explain this very important feature of Promela.

$a == b$; printf("Promela execution blocked")

If the value of variable a is equal to the value of variable b the printf command is executed. If it is false, the process will in this case block and wait for an event to happen in another process until the condition (a == b) becomes executable (true).

a=b; printf("Promela rules"')

In this example the first statement is evaluated for its executability. But since it is an assignment, which is always executable the printf command can be executed.

2.1.3 Processes

In order to execute a process it must be named, its type defined and instantiated.

proctype A() { byte state; state = 3 }

The process above declares a process named A with a local variable state that is assigned to 3. A is of type proctype, which defines all types of processes that can be instantiated. A proctype definition only declares process behavior. A process execution can be started from the init process or from another process. All processes are however children of the init process, which will be described next.

2.1.4 The init process

Initially just one process is executed, the init process. The init process can start processes. These processes can then again start new processes, which will run concurrently with the other processes, but the init process is the root of all the processes. The init process is comparable to the main() function in a C program.

init { run A(); run B() }

The init process above starts two processes A and B, which will run concurrently.

2.1.5 Message channels

Message channels are used to model the transfer of data from one process to another.

chan qname = [3] of { int }

The statement above initializes a channel qname that can store up to 3 messages of the type int in a queue structure (first-in first-out). The statement

qname!expr

sends the value of the expression expr to the channel queue. The statement

qname?msg

retrieves a message from the head of the queue and stores it in the variable msg. This form of communication is asynchronous communication. Promela can also handle synchronous communication in cases where a channel doesn't contain a queue for storing messages. In such a case transmission can occur only when a channel is used for sending and receiving simultaneously.

2.1.6 Variables and datatypes

A variable can be one of the following nine predefined data types:

Type	Range of values
bit	0,1
bool	false,true
byte	0..255
chan	1..255
mtype	1..255
pid	0..255
short	–215 .. 215 – 1
int	–231 .. 231 – 1
unsigned	0 .. 2n – 1

2.1.7 Expressions and Declarations

The expression syntax of PROMELA and UPPAAL coincides with that of C language. Hence, g-Promela also uses C-like expression language. The lexical elements, or tokens of these two languages are identifiers, keywords, constants,

operator symbols, and punctuations. As in PROMELA language, processes, channels, variables, etc must be declared before they can be used. Variables and channels can be declared either locally, within a process, or globally. A process can only be declared globally in a proctype declaration. Local declarations may appear anywhere in a process body. A local variable can be referenced from its point of declaration to the end of the proctype body in which it appears, even when it appears in a nested block (i.e., a piece of code enclosed in curly braces).

2.1.8 Atomic statement

In Promela atomic sequences can be used. A sequence of statements enclosed in parentheses prefixed with the keyword atomic indicates that the sequence is to be executed as one indivisible unit, non-interleaved with other processes. Atomic sequences is described with the following example.

byte state = 1;
proctype A() { atomic { $(state == 1) -> state = state + 1$ } }
proctype B() { atomic { $(state == 1) -> state = state - 1$ } }
init { run A(); run B() }

Both processes uses the global variable state. The final value would in this case be either 0 or 2, because the sequences in both processes is executed atomic. Had the sequences of A and B not been executed atomic both processes can pass the condition simultaneously and the variable state could be incremented and decremented and the final value of state would therefore be unpredictable since it could be both 0,1 or 2.

2.1.9 IF..FI selection

The Promela language also contains a CASE selection structure. The following is an example of such a structure.

if
:: (a != b) -> option1
:: (a == b) -> option2
fi

Only one sequence from the list is executed but the first statement, called a guard must be executable. As described in section A.1 a guard can also be as statement like for instance an assignment. More than one guard can be executable. If this is the case one of the sequences is selected at random. If none of

the guards are executable, the process blocks until at least one of them can be selected.

2.1.10 DO..OD repetition

Repetition is supported using a do..od construction. This construction is basically an IF..FI selection that loops until a BREAK command is met. The following example randomly increments or decrements the variable count.

```
byte count;
proctype counter()
{
do
:: count = count + 1
:: count = count - 1
:: (count == 0) − > break
od
}
```

After one of the options is completed the structure is repeated until the break statement is reached. When the count variable has the value 0 the break statement can be executed but it is not necessarily executed since the first two options still can be executed.

2.1.11 Other commands

Promela also includes the goto statement which causes a jump unconditional to a named label.

The timeout statement allows a process to abort when waiting for a condition that can no longer become true, for example an input from an empty channel. It can be considered an artificial, predefined condition that becomes true only when no other statements in the distributed system are executable.

2.1.12 PROMELA Example

```
active proctype gcd(int x, y) {
 L:      if
        ::  (x > y) -> x = x-y; goto L
        ::  (x < y) -> y = y-x; goto L
        ::  (x == y) -> assert(x == y);
        fi;
      printf("gcd = %d\n", x)
 }
```

2.2 Informal Description of g-Promela

This section presents an informal description of g-Promela model based on a simple example. The newly defined graphical modeling language based on transition system is formally introduced and then equipped it with trace-based formal semantics. This semantics describes the specification language that allows for safety and reachability. We have used the existing UPPAAL GUI as the basis for Graphical Promela Interface(GPI) development for various reasons discussed in chapter 1. We then elaborate this and give the formal syntax and semantics for g-Promela models.

2.2.1 Modeling

Modeling of g-Promela models, is done in Graphical Promela Interface, which uses the existing graphical components of UPPAAL. As an example, the textual representation of the model to compute greatest common divisor(GCD) specified in section 2.1.11 and its equivalent graphical system description is given in [fig 2.1]. The textual format (i.e., .pml) is the actual PROMELA specification language. The g-Promela description language proposed supports only modeling of simple PROMELA models. As described in chapter 2, there are various kinds of language features and different ways to model the same problem description in PROMELA language like in C and other high-level programming languages. The init process can be complemented with active processes. The case selection structure of if..fi and repetition do..od constructions can be represented by labeled goto statements. The current implementation supports only active processes and uses labeled goto statement when a valid transition occurs from one location to the next location.

Locations

Consider the g-Promela model of [fig 2.1]. The model consists of a single process gcd with control nodes and/or locations S0, S1, S2, S3, S4, S5. In addition to

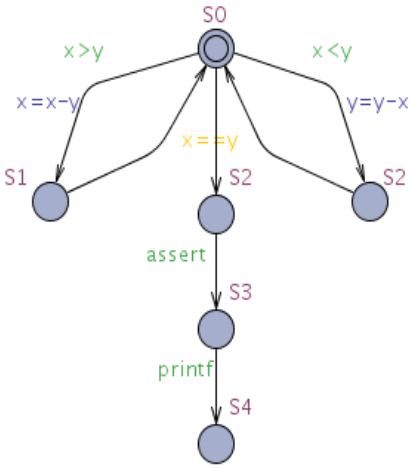

Figure 2.1: Example of g-Promela model to compute GCD

these discrete control structures, the model uses two integer variables x and y.

Guards

A guard is a boolean expression enabling a transition. In [fig 2.1] the edge between S0 and S1 can only be taken, when the value of the integer variable x is greater than y. Similarly the edge between S0 and S3 can only be taken when the value of the integer variable x equals y. Formally, guards are data constraint and is of a similar form i $\backsim$ j or i - j $\backsim$ k but with k being an arbitrary integer and $\backsim \in \{<, <=, ==, >=, >\}$. In general, the language used to define the expressions in the guards are defined using the PROMELA language. The syntax of PROMELA for simple expressions and assignments is similar to the syntax of C and C++. The default guard of an edge is true.

Edges

The edges of the automata are decorated with three types of labels: a guard, expressing a condition on the values of integer and boolean variables that must be satisfied in order for the edge to be taken; a synchronization action which is performed when the edge is taken and assignments to integer variables. All three types of labels are optional.

UPPAAL GUI also has it's control node decorated with invariants, which are conditions expressing constraints on the clock values in order for control to re-

main in a particular node. This label is not considered and simply needs to be ignored while building g-Promela models as by default, SPIN doesn't have the notion of time in terms of clock constraints. Though clocks can be simulated in normal PROMELA models as shown in [] and attempts like in RT-SPIN [] to include features of clock, the models considered in this report doesn't support clocks and it's declarations.

Templates

Each process in a system is an instance of a template. A template consists of an automaton with locations, edges, guards and actions. The template also has formal parameters that can be replaced with actual channels, variables, and constants when the template is instantiated as a process. The motivation for the templates is that systems often have several processes that are very alike.

2.3 Basic Definitions

A common framework for the representation of distributed and concurrent systems is provided by the concept of transition systems. A transition system specifies the allowed evolutions of the system: starting from some initial state, the system evolves by performing actions that take the system to a new state. It's also called Kripke structure, in honor of the logician Saul A. Kripke who used transition systems to define the semantics of modal logics.

Definition 2.3.1 (Transition Systems) *A transition system is a tuple* $\langle S, Act, \rightarrow, I \rangle$ *where*

- S *is a set of states,*
- Act *is a set of actions,*
- $\rightarrow \subseteq S \times Act \times S$ *is a transition relation,*
- $I \subseteq S$ *is a set of initial states.*

where S and Act are finite or countably infinite.

Definition 2.3.2 (Actions) *Let Chan be a finite set of channels, ranged over by c. We define Act to be a finite set of actions ranged over by a. For each channel in Chan we define two actions such that* $Act = \{c! | \ c \in Chan\} \bigcup \{c? | \ c \in Chan\}$. *We define a complement operator* $\bar{}$*:* $Act \rightarrow Act$ *as* $\bar{c!} = c?$ *and* $\bar{c?} = c!$.

Definition 2.3.3 (Labeled Transition Systems) *A labeled transition system relates the triple $\langle S, \ell, \rightarrow \rangle$ in the following way.*

- *S is a set of states,*
- *ℓ is a set of labels, and*
- *$\rightarrow$ is a set of transitions $\rightarrow \subseteq S \times \ell \times S$.*
- *If $(S_1, \alpha, S_2) \in \rightarrow$ we write $S_1 \rightarrow S_2$*

2.4 Formal Syntax

2.4.1 Syntax of PROMELA Model

The basis of the Promela model is the notion of classical finite state automata(FSA). Every PROMELA *proctype* defines a FSA, (S,s_0,L,T,F) as defined below. The set of states of this automaton S corresponds to the possible points of control within the *proctype*. Transition relation T defines the flow of control. The transition label set L links each transition in T with a specific basic statement that defines the executability and the effect of that transition. The set of final states F, finally is defined with the help of PROMELA end-state, accept-state, and progress state labels. To provide a more expressive model and to ease the modeling task, FSA is further extended with more general types of data variables such as boolean and integer variables.

Definition 2.4.1 (Process Automaton) *Every Promela proctype defines a finite state automaton, $\langle S, s0, L, T, F \rangle$, where*

- *S is a set of states*
- *s0 is the initial state, $s0 \in S$*
- *L is a finite set of labels*
- *T is a set of transitions, $T \subseteq S \times L \times S$*
- *F is a set of final states, $F \subseteq S$*

A label $l \in L$ is one of the six basic statements:

- *expression // executable if not zero*
- *assignment // always executable*
- *assert(expr) // always executable*

- *printf // output statement*
- *send (ch!) // executable if channel c is not full*
- *receive (ch?) // executable if channel c is not empty*

Other Promela statements serve to specify possible flow of control, i.e. the transition relation T.

Alternatively, the definition of PROMELA model can be given using abstract objects such as variables, message channels and asynchronous processes. The semantics engine operates on these abstract objects to determine how a given PROMELA model defines system executions, including the rules that apply to the interleaved execution of process actions.

Definition 2.4.2 (PROCESS) *A **process** is a tuple* $\langle pid, lvars, lstates, initial, curstate, trans \rangle$ *where*

- *pid is a positive integer that uniquely identifies the process,*
- *lvars is a finite set of local variables, each with a scope that is restricted to the process with instantiation number pid,*
- *lstates is a finite set of integers,*
- *initial and curstate are elements of set lstates, and*
- *trans is a finite set of transitions on lstates.*

Definition 2.4.3 (SYSTEM STATE) *A global**system state** is a tuple of the form* $\langle gvars, procs, chan, exclusive, handshake, timeout, else, stutter \rangle$ *where*

- *gvars is a finite set of variables with global scope,*
- *procs is a finite set of processes,*
- *chans is a finite set of message channels,*
- *exclusive, and handshake are integers,*
- *timeout, else and stuter are booleans.*

Definition 2.4.4 (TRANSITION) *A **transition** in process P is defined by a tuple* $\langle tr\_id, source, target, cond, effect, prty, rv \rangle$ *where*

- *tr_id is a non-negative integer,*
- *source and target are elements from set P.lststes(i.e., integers),*
- *cond is a boolean condition on a global system state*
- *effect is a function that modifies the global system state*
- *prty and rv are integers used inside cond and effect definitions to enforce the semantics of unless constructs and rendezous operations.*

2.4.2 Syntax of g-Promela Model

A g-Promela model defines a network of communicating processes with transitions labeled by the communications and other executable Promela statements. We define formal syntax of these g-Promela models as a parallel composition of processes. For simplicity, we assume a set of labels Labels that ranges over syntactically correct assignments, guards and synchronization labels. As a well-formedness condition, labels are constrained to occur only in appropriate places, contain only declared variables, and have to respect the variable types.

Definition 2.4.5 (g-Promela Process) *An g-Promela process A is a tuple* $\langle L, T, Type, l^0 \rangle$*, where*

- *L is a set of locations,*
- *T is a set of transitions* $l \xrightarrow{g,s,a} l'$*, where* l*,* $l' \in L$*, g is a guard, s is a synchronization label optional , and a is an assignment possibly empty, and*
- $l^0 \in L$ *is the initial location.*

We use the following access functions to refer to guards, synchronizations, and assignments.

- *Guard :* $T \rightarrow$ *Labels maps to the guard of a transition (possibly constant true),*
- *Sync :* $T \rightarrow Labels \cup \{\phi\}$ *maps to the synchronization label of a transition (if any), and*
- *Assign :* $T \rightarrow Labels \cup \{\phi\}$ *maps to the assignment associated with a transition (possibly the empty assignment).*

Definition 2.4.6 (g-Promela Model) *A g-Promela model is a tuple* $\left\langle \vec{A}, Vars, Channels \right\rangle$*, where*

- *$\vec{A}$ is a vector of processes $A_1, ..., A_n$;*
  We use the index i to refer to A_i-specific parts $L_i, T_i, Type_i, and l_i^0$,
- *Vars is a set of variables, i.e., integers and arrays,*
- *Channels is a set of synchronization and buffered channels, $Channels \cap Vars = \phi$,*

Definition 2.4.7 (CONFIGURATION) *A configuration of a g-Promela model $\left\langle \vec{A}, Vars, Channels, Type \right\rangle$, is a duplate $(\vec{l}, e)$, where $\vec{l}$ is a vector of locations and e is the environment for discrete variables i.e.:*

- *$\vec{l} = (l_1, ..., l_n)$, where $l_i \in L_i$ is a location of process A_i (called control situation) and*
- *e: $Vars \rightarrow (Z)^*$ maps every variable v to a value (if int(v)) or a tuple of values(in case of array(v))*

2.5 Trace Semantics

The operational semantics of concurrent processes can be given in terms of automata commonly termed as Labeled Transition System. Informally, an automaton is constructed using locations connected with edges. A state of the system is determined by the current location of each process and the values of the data variables. A transition from one state to the next follows an edge in one or two processes and updates the variables according to the assignment label(s) of the edge(s). To control when to fire a transition, it is possible to use guards and synchronizations. A guard is a condition on the variables saying when a transition is enabled. The synchronization mechanism in g-Promela allows two processes to perform a hand-shaking synchronisation. If two processes have enabled transitions with complementary synchronisation labels, e.g. a! and a?, they may take a compound transition. In a synchronisation both processes will change location simultaneously. When taking a transition assignments of variables and other basic executable statements are possible as actions. LTS used for modeling the behavior of a program also gives us the relation with linear temporal logic, as consequence of the fact that every LTL formula can be translated into a specific class of labeled transition systems, known as Buchi automata. The formal semantics of the g-Promela models in terms of an operational model with processes defined as transition systems (i.e., automata) is given below.

g-Promela models evolve according to legal action steps and the compendium of all legal steps defines the behavior of the model and is formulated as simple and synchronized action steps. To modify the control situation $\vec{l}$, we use the

notation $\vec{l}[l'/l_i]$ to indicate that at position i, l_i was replaced by l'_i and other positions did not change. We use the notation $e \models_{loc} \Phi$ to indicate that a boolean expression Φ holds true under the evaluation e for the contained variables, and $(\vec{l}, e) \models_{loc} \Phi$ analogously in the case that Φ contains expressions of the form $A_i.l_i$ denoting that process A_i is in location l_i.

Definition 2.5.1 (Simple Action Step) *For a configuration $(\vec{l}, e)$, a simple action step is enabled, if there is a transition $l_i \xrightarrow{g,a} l'_i \in T_i, l_i$ in $\vec{l}$, such that $e \models_{loc} g$.*

Definition 2.5.2 (Synchronized Action Step) *For a configuration $(\vec{l}, e)$, a synchronized action step is enabled, iff for a channel b there exist two transitions $l_i \xrightarrow{g_i,b!,a_i} l'_i \in T$, and $l_j \xrightarrow{g_j,b?,a_j} l'_j \in T, l_i, L_j$ in $\vec{l}$, $i \neq j$ such that $e \models_{loc} g_i \wedge g_j$.*

Chapter 3

Translation from g-Promela to PROMELA model

3.1 Modeling

3.1.1 Mutual exclusion algorithm

To build a g-Promela model as an automaton, we study Petterson's mutual exclusion algorithm as an example. In the mutual exclusion problem, there is a collection of asynchronous processes. Each process contains a distinct part of the code called a critical section (or region). The process's remaining code is referred to as a noncritical section (or region). Each process alternately executes its noncritical and critical sections. Processes can proceed in parallel outside of the critical section but only one process at a time can execute the critical section.

The algorithm for two processes in textual PROMELA is as follows :

```
proctype User1

assert(_pid==0 || _pid==1);
req[1]=1;
turn=_pid;
(req[1 - _pid] == 0 || turn == 1 - _pid);
//critical section
job1();
req[1]=0;
```

```
proctype User2

assert(_pid==0 || _pid==1);
req[2]=1;
turn=1-_pid;
(req[_pid] == 0 || turn == _pid);
//critical section
job2();
req[2]=0;
```

//_pid is a called process instantiation number which records the instantiation number of the current running process and its value can range from 0 to 255. In this example, it can take either 0 or 1 as we are considering only two processes.

The global array variable req[n] is associated with each process. When a process

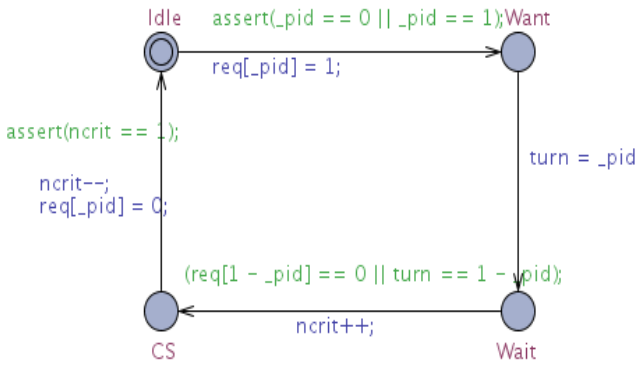

Figure 3.1: g-Promela Model: Template User

want to enter the critical section it sets its req variable. The protocol also uses the global variable turn that helps the processes to alternate their access to the critical section so they get a fair share. As per the rules for executability, depending on the system state, any statement in a SPIN model is either executable or blocked i.e., if a process reaches a point in its code where it has no executable statements left to execute, it simply blocks. So, the condition statement blocks until the other process has left the critical section (if it was there). Notice that the protocol is symmetric, so we may use as a template and create two instances of it prefixing active [2] before the proctype declaration. Notice here that the critical section part of the algorithm is abstracted as we are only interested in the control structure.
On our way towards a model of the algorithm, we can observe that the algorithm has four states. We mark them with a notation similar to goto labels as follows:

User 1

```
idle:   assert(_pid == 0  _pid == 1)
        req[1]=1;
want:   turn=_pid;
wait:   (req[1 - _pid] == 0  turn == 1 - _pid);
CS:     //critical section
        job1();
        //and return to idle
        req[1]=0;
```

We now create a new template called *User* and draw the automaton as depicted in [fig 3.1]. The location node *idle* is marked initial to state that it is the starting point for execution. The transition edge can be edited with condition statement in guard field, message channel statements for asynchronous communication in sync field and unconditional executable statements such as print statements and assignments in update field. Hence, the *assert* and *wait* conditions act as guards for transition from *idle* to *want* and *wait* to *cs* respectively.

3.2 Translation

Once the model is created, translation and mapping of graphical elements used in Graphical Promela Interface(GPI) to actual textual PROMELA specification can be done using the following algorithm:

```
read GPI xml-file
system := find all Transition Systems(TS) from templates, declarations and instantiation
for each TS in system:
        write system initializing information for TS
        write local declarations
        for each outgoing transition in node:
            for each guard in location:
                write guard
            for each synchronization in transition
                write synchronization
            for each assignment in transition
                write code that makes assignments iff transition is chosen
        write processes
write init file that initializes global declarations on top and TS as run processes
```

Firstly, xml file is read and an abstract syntax tree(AST) is returned. Objects from the information tree is then created and linked together in a hierarchical structure. When all the information has been read, we have to process the information. This step mainly converts template object to process objects, and lists global declarations by running through all information in xml file and the elements of a transition system. In the generated system there will be total of n+1 processes, where n is the number of transition systems in the system followed by an *init* process.

Chapter 4

Implementation

4.1 Architecture of GPI tool

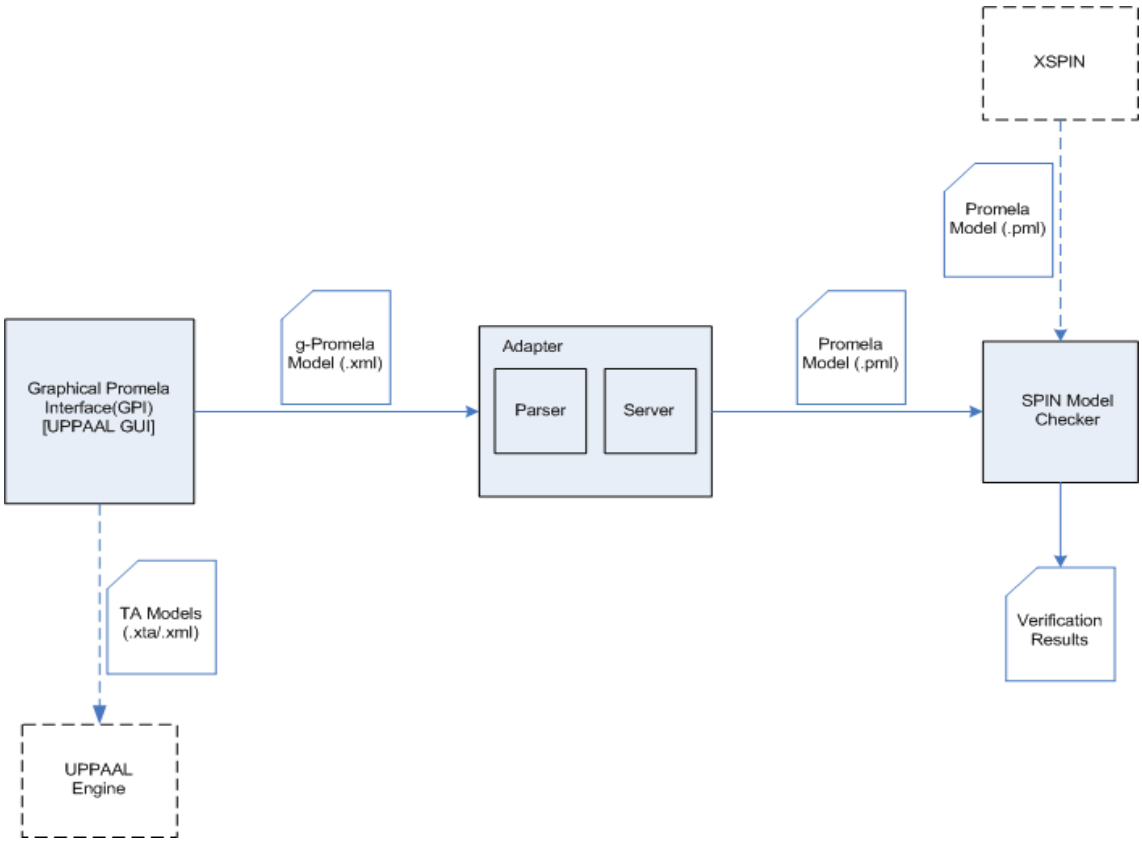

Figure 4.1: Architecture

g-Promela is a graphical notation for automated model analysis. This means that a comprehensive tool support for g-Promela is needed. As shown in [fig 4.1],

Graphical Promela Interface (GPI) is built upon the existing UPPAAL GUI. GPI, which acts as a client is used for graphical editing g-Promela models and storing them in an UPPAAL standard XML file. As mentioned in the section [1.3.2], the UPPAAL GUI is developed using Java 1.5. An adapter is built in between the GPI and SPIN Engine, which acts as a server. The adapter is also composed of a Parser. Parser is built using LibXML2 library functions which accepts the model built in GPI. Server acts as a communication channel and runs the PROMELA input file on the SPIN executable engine using a 'system' call. The verification result from SPIN is directed on to an output file for analysis.

4.2 Adapter

Once the graphical models are built using GPI, the control flows to adapter module as shown in the architecture. The adapter module is developed using the C++ programming language ans it constitutes two main modules. The parser module receives the system file from the GPI and translates it into equivalent textual PROMELA model. The current version of the tool outputs on to a *.pml* text file. The parser module uses libXML2 library functions available for C++ language. The pseudocode implementation of the parser is as shown in the next page:

```
proc xmlParse (xmlfile)
begin
string xmltags[]={'declaration','template','location', 'transition', 'instantiation', 'system' }
 string tokens, xmlLine;
  while not EOF
   begin while
    xmlLine = readline(xmlFile)
      tokens = xmlfile_getTokens(xmltags,xmlFile)
      switch (tokens)
      begin case
          case declaration:
            xmlParse_deceleration(xmlLine)
          case template:
            xmlParse_template(xmlLine)
          case location:
            xmlParse_location(xmlLine)
          case transition:
            xmlParse_transition(xmlLine)
          case instantiation:
            xmlParse_instantiation(xmlLine)
          case system:
            xmlParse_system(xmlLine)
      end case
  end while
 end xmlparse


 proc xmlParse_transition()
 begin
 string sourceid, targetid, label_kind[2], guard, assignment, synchronisation;
 if sourceid==targetid then
    appendstring(str,'do :: value(guard) ; value(assignment)od')
  else
    appendstring(str,'if :: value(guard) ; value(assignment)fi')
 writefile(str, run.pml)
 end xmlParse_transition
```

Server aspects to handle the communication protocol are partially inherited from the existing UPPAAL server. It handles transfer of system data by spooling to a stream buffer. The main purpose of model analysis is then achieved by translating g-Promela models to textual PROMELA by switching from edit mode to verification mode which uses SPIN model checker at the background. Properties that have been identified will be passed on to SPIN together with PROMELA

code. A suitable way of capturing and specifying properties is investigated in the next section. The analysis results can be fed back to interpret inside GPI in order to make sense in the context of the original g-promela model. An additional validation techniques for g-Promela models can be simulation to trace and observe individual process instances, data objects and inter-object messages. These advanced features are left as future work as it is not implemented in the current version of the GPI tool.

4.3 Verification of the system's correctness properties

Having introduced a modeling language for describing protocols this section focuses on the validation of a model. A complete verification model contains not just the specification of system behavior but also a formalization of the correctness requirements that apply to the system. Automata models offer a good formalism for the analysis of distributed system models. Most relevant to the verification of asynchronous process systems is a specific branch of temporal logic known as linear temporal logic, commonly abbreviated as LTL. Strictly speaking, the system description language PROMELA does not include syntax for the specification of temporal logic formulae, but SPIN does have a parser for such formulae and it can machanically translate them into PROMELA syntax, so that LTL can effectively become part of the language that is accepted by SPIN. LTL, however, can only be used for specifying correctness requirements on PROMELA verification models. The models themselves cannot be specified in LTL. SPIN's conversion algorithm translates LTL formulae into *never* claims, and it automatically places labels within the claim to capture the semantics of the $\omega - regular$ property that is expressed in LTL.

4.3.1 Types of properties

Assertion claims

Correctness claims for Promela models can be built up from simple propositions, where a proposition in a given state is either true or false. Promela uses the assert (condition) statement which is a boolean condition that must be satisfied whenever a process reaches a given state. The assert statement is always executable and can be placed anywhere in a Promela model. If the condition is true the statement has no effect. However if there is an execution sequence in which the condition is false when the assert statement becomes executable then the assertion claim fails. This example should show how the assertion claims is used.

bytestate = 1;
proctype A() {(state == 1) − > state = state + 1;
assert (state == 2)}
proctype B() {(state == 1) − > state = state - 1;
assert (state == 0)}
init { run A(); run B()}

In this example assertion claims is inserted after A or B has either incremented or decremented the variable state. As shown earlier the final value of state is not 2 or 0 for all execution sequences. The final value of state can also be 1, which means that the assertion claims in this example fails.

System invariant

A more general way of using the assert statement is to use system invariants. For instance if a boolean condition should be true in the start state of the system and remain true in all states of the system independently of the execution sequence that leads to each of the states. In Promela this can be done by creating a monitor process.

proctype monitor(){assert(invariant)}

The monitor process is independent of the rest of the system and can evaluate the assertion at any time. For every state of the system the assertion statement is executable. The process must however be initialized and be running in order to do the correctness check.

Deadlocks

In a finite state system all execution sequences either terminate or cycle back to a previously visited state. It is necessary to distinguish between expected and unexpected end-states when execution sequences cycles. The unexpected end-states includes deadlock states, where processes are waiting for each other in order to become executable, and error states. In promela end-state labels can be used for defining expected end-states. A proper system end-state would be a state where every process that was instantiated has either terminated or has reached a state marked as a proper (expected) end-state.

Non-progress cycles

In order to avoid cycles that display an infinite behavior Promela has a progress label. The progress label marks a state that must be executed for the protocol to make progress. In this way infinite delays can be avoided.

Livelocks

With the accept state label it is possible to express that something cannot happen infinitely often. In other words, an acceptance state label marks a state that may not be a part of a sequences of states that can be repeated infinitely often.

4.3.2 Verification steps in SPIN

This section describes the steps to verify the properties of the generated textual PROMELA models. First, a verifier is generated which is then compiled using standard C complier to generated an executable of the verifier. If model.pml is the verification model including an initial formalization of correctness properties, in its most basic mode verification in SPIN is done as follows:

```
$ spin -A model.pml          # perform syntax check

$ spin -a model.pml          # generate verifier

$ cc -o pan pan.c            # compile verifier

$ ./pan                      # perform verification
```

To specify an LTL formula and run the verifier, we need to define propositions in the formula by using global variables or #defines with all variables having global scope. In the example, for the proposition $\#define\ p\ ncrit <= 1$, a PROMELA code can be generated for LTL formulae such as [](p) or []¡¿(!q) using *-f* option. The LTL claim gets converted to NEVER claim which is stored in .ltl file. It is then appended to the generated textual PROMELA model file and then verified by generating the verifier. The verifier by default searches for safety properties i.e., assertion violations, deadlocks, etc. The other type of search is for liveness properties to show the absence of non-progress cycles or acceptance cycles. The default can be changed into a search for acceptance cycles if run-time option -a is used. To perform a search for non-progress cycles, we have to compile the pan.c source with the compile time directive -DNP and use run-time option -l, instead of -a.

We consider the mutual exclusion algorithm example to verify the properties. Two *assert* statements are considered in the example of [fig 3.1]. The $assert(\_pid == 0||\_pid == 1)$ statement in the beginning of the model, checks that there are only at most two instances with identifiers 0 and 1 where, _pid is the identifier of the process. The assert statement stated in the critical section(CS) $assert(ncrit == 1)$ checks that there is always at most one process in the critical section, where *ncrit* is a byte variable used to count the number of processes.

Chapter 5

Conclusion and Future Work

To summarise this thesis, we recall our initial intention: We aimed to define a graphical input specification language for SPIN model checker. This has been achieved for the basic features of PROMELA language by defining the syntax of graphical elements as an extension of labeled transition systems.

Because g-Promela models are compiled into textual PROMELA, the graphical models can be formal analysed using the SPIN model checker. To facilitate the handling of g-Promela specifications, we have sketched the design of the Graphical Promela Interface(GPI) toolset by building upon the existing UPPAAL GUI. The GPI tool architecture mainly includes the adapter program which constitutes parser and server modules. For validation purposes, the GPI uses various formats to specify properties specified in PROMELA. State properties can be attached to the models using assertions. Linear Time Temporal Logic (LTL) formulas can be used to capture temporal requirements which are translated automatically to never claims using the built-in commands of SPIN. However, the current version of GPI support only parsing of the g-Promela models and outputs the textual PROMELA to a textual file. Verification is then done using the system commands by generating the verifier and compiled using C complier to produce an executable verifier *pan* which is then used to verify the LTL query.

The current version of GPI can be extended as a generic framework for building graphical PROMELA models by implementing the following features:

Syntax checking of the PROMELA statements included as part of the labels in the g-Promela models can be done before uploading the system to the adapter. Syntax checking can also be done for the queries specified in the command statement. The work presented in this report supports modeling of simple graphical models with basic features of PROMELA language. This limitation can be overcome and made to represent all the features of the textual PROMELA language by exploiting the features of existing graphical elements and by specifying new

graphical elements. Like UPPAAL, the GPI tool can be made to support simulation of execution, feedback for verifier and generation of error trace.

Appendix A

Generated PROMELA model

A.1 Example: Mutual Exclusion Algorithm

```
#define p ncrit <= 1                          // Defining propositions
#define q ncrit = 0
bool turn, reqflag[2];                        // Global declarations
byte ncrit;
active [2] proctype user(){                   // Proctype body
 idle:   assert(_pid == 0 || _pid == 1);
         reqflag[_pid] = 1;
         goto want;
 want:   turn = _pid;
         goto wait;
 wait:   (reqflag[1 - _pid] == 0 || turn == 1 - _pid);
         printf("\n in wait");
         ncrit++;
         goto cs;
 cs:     assert(ncrit == 1);
         ncrit–;
         printf("\n in cs");
         reqflag[_pid] = 0;
         goto idle;
 }
never { /* [ ](p) */
accept_init:
T0_init:
if
:: ((p)) − > goto T0_init
fi;
}
```

Bibliography

[1] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual* (Addison-Wesley Professional, 2003).

[2] G. J. Holzmann, Basic spin manual, 1994.

[3] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, UPPAAL - a tool suite for automatic verification of real-time systems, in *Hybrid Systems*, pp. 232–243, 1995.

[4] A. David, G. Behrmann, and K. G. Larsen, A tutorial on uppaal, 2004.

[5] R. Alur and D. L. Dill, A theory of timed automata, 1994.

[6] S. Leue and G. Holzmann, v-promela: A visual, object-oriented language for spin.

[7] S. Tripakis and C. Courcoubetis, Extending promela and spin for real time, in *Tools and Algorithms for Construction and Analysis of Systems*, pp. 329–348, 1996.

[8] J. C. Corbett *et al.*, Bandera: extracting finite-state models from java source code, in *International Conference on Software Engineering*, pp. 439–448, 2000.

[9] R. Milner, *Communication and Concurrency* (Prentice-Hall, 1989).

[10] K. G. Luca Aceto and A. Ingólfsdóttir, An introduction to milner's ccs, BRICS,Department of Computer Science,Aalborg University,Denmark, 2005.