A Promela front-end for Spot

Guillaume Sadegh

revision 1880

SPOT is a C++ library for model-checking. For verification, SPOT uses an input-format that describes a Transition-based Generalized Büchi Automata (TGBA). However, this format does not seem accessible for users with its poor abstraction and with the size of automata which often have millions of states.

PROMELA (Process Meta-Language) is the verification modeling language used by the SPIN model checker. It lets users describe a parallel system for verification in a high level programming language.

We present a way to add a PROMELA front-end in SPOT, which will allow to explore the state-graph on-the-fly and thus avoid to store all the states of our automata.

Keywords
SPOT, model checking, PROMELA, virtual machine, PROMELA front-end, Transition-based Generalized Büchi automaton, modeling language, NIPS, SPIN.

Laboratoire de Recherche et Développement de l’Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
guillaume.sadegh@lrde.epita.fr – http://www.lrde.epita.fr
Copying this document

Copyright © 2008 LRDE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just “Copying this document”, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is provided in the file COPYING.DOC.
## Contents

1 Introduction 4

2 High level modeling language 5
   2.1 Prerequisite 5
   2.2 The Promela language 7
      2.2.1 Language specifications 7
      2.2.2 Example: Dining philosophers problem 8
      2.2.3 never claims 9

3 Promela front-end in Spot 11
   3.1 Initial “hack” 11
   3.2 The NIPS virtual machine 11

4 Benchmark 14
   4.1 Benchmark with Spin 14

5 Discussion 16
   5.1 Related work 16
      5.1.1 Promela front-end in other model checkers 16
      5.1.2 Spot with an external tool 16
   5.2 Future work 17

6 Conclusion 18

7 Bibliography 19
Chapter 1

Introduction

SPOT (Duret-Lutz and Poitrenaud, 2004) is an object oriented model checking library. It provides a set of algorithms and data types to build a custom model checker.

Model checking is an automatic technique to do formal verification of the model of a system. This verification is exhaustive on all the behaviors of the system. To do this verification, the model checker needs two inputs: a model of the system to verify and a set of properties that the system must satisfy. This properties are described with Linear-Time Temporal (LTL), a formalism that describes the ordering of events in time using special temporal operators.

SPOT relies on the automata-theoretic approach to model checking. The kind of automaton used by SPOT is called Transition-based Generalized Büchi Automaton (TGBA).

A Transition-based Generalized Büchi Automaton is an extension of a finite state machine to infinite inputs. It accepts an infinite input sequence if there exists a run of the automaton which visits a set of predefined transitions infinitely often.

SPOT currently uses an input format to recognize models which represents a TGBA, with its states and transitions. This format requires users to express their models as an automaton having a very low level of abstraction.

SPIN (Holzmann, 1990, 2003) is an open source model checker used by thousands of people worldwide. PROMELA (Process Meta Language) is the verification modeling language of SPIN. It allows to design distributed systems in a high level language, and to express properties to verify on this system.

This report presents the implementation of a PROMELA front-end for SPOT, which will let users describe models with a high level specification language. The input for this high level modeling language will let us use algorithms present in SPOT with existing PROMELA models, and then check more complex systems.

Organization: Chapter 2 contains prerequisites on verification modeling language and automata-theoretic approach for model checking, and presents the PROMELA language.

In Chapter 3, we will present the implementation done in SPOT. Then, in Chapter 4, we will process some benchmarks on our implementation. The Chapter 5 will present the related work and our future works on this front-end, and we will conclude in the Chapter 6.
Chapter 2

High level modeling language

2.1 Prerequisite

SPOT relies on the automata-theoretic approach to model-checking. This approach is illustrated by the Figure 2.1. As illustrated, the model checker takes as input a high-level model $M$ and a LTL formula $\varphi$, and outputs whether the formula verifies the model ($M \models \varphi$) or provides a counter-example where the formula is not verify.

In this Figure, algorithms are in the rounded boxes. These algorithms are:

- The synchronized product, which processes the synchronized product between the negated formula automaton, whose language, $L(A_{\neg \varphi})$, is the set of all executions that would invalidate $\varphi$, and the state-graph automaton, whose language, $L(A_M)$, is the set of all possible executions of the system. Then, the synchronized product produces an automaton, whose language is the set of all possible executions of the system that would invalidate $\varphi$.

- The emptiness check, which checks if the language which is the set of all possible executions of the system that would invalidate $\varphi$ is empty. If the language is empty, the formula verifies the model.

SPOT does not provide the branch related to the High-level model. It consists in the interpretation of a high-level model as a TGBA. To do this interpretation, we need to generate a state-graph from a model.

State-graph

From a model, we want to generate a graph which represents all the behaviors that the system may meet. A state will be a snapshot of a behavior of the system (it will contain the value of variables and the status of each process) and transitions will represent reachable behaviors of the system from previous behavior.

This graph is generated by simulating the execution of the PROMELA model. It is called state-graph and represents the state space.

We will build the TGBA of the model by the exploration of the state space of the model. An example of state-graph generation will be presented in section 2.2.2 on page 9.
2.1 Prerequisite

Represented by PROMELA in SPIN

- High-level model $M$
- State-graph generation
- LTL formula $\varphi$
- LTL-to-Büchi translation
- Negated formula automaton $A_{\neg \varphi}$
- Product automaton $A_M \otimes A_{\neg \varphi}$
- Synch. product $\mathcal{L}(A_M \otimes A_{\neg \varphi}) = \mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg \varphi})$
- Emptiness check $\mathcal{L}(A_M \otimes A_{\neg \varphi}) \not= \emptyset$

$M \models \varphi$ or counter-example

Figure 2.1: Automata-theoretic approach to model-checking.
Combinatorial explosion

One of the biggest issues in model checking is the state-space explosion. Systems can produce state-graphs with billions of states and transitions. These graphs are too large to be stored in memory. However, some solutions exist to handle models which cannot be stored.

A common way to deal with the state graph explosion is to use state graph partial-order reduction techniques that will reduce the automaton.

To avoid storing all the states, another technique is to explore the graph on the fly. States are computed only when they are visited. With this technique, we need to generate the whole state space only in the worst case. SPOT is built to work with algorithms working on the fly. The tgba abstract class in SPOT provides two main methods: get_init_state(), which returns the initial state of the automaton, and succ_iter(), which returns an iterator on successors of a state. All emptiness check and synchronized product algorithms are working with this class.

2.2 The Promela language

Common approaches in state-based model checking use high level modeling languages. We can cite CSP (Hoare, 1978), LOTOS (Bolognesi and Brinksma, 1987), Murϕ (Dill et al., 1992), DVE (Barnat et al., 2006), or PROMELA (Holzmann, 1990, 2003) which are used to describe state spaces with abstraction.

We have decided to use PROMELA as verification modeling language. PROMELA is the most used modeling language and provides the larger set of system descriptions.

2.2.1 Language specifications

PROMELA is the language used in SPIN to represent concurrent systems with abstraction.

PROMELA programs consist of processes, message channels, and variables. Processes are global objects that represent the concurrent entities of the distributed system. Message channels and variables can be declared either globally or locally in a process. PROMELA supports rendezvous and asynchronous communication between processes via channels. Processes specify behavior, while channels and global variables define the environment in which the processes run.

SPIN does not use the LTL syntax to express properties. Instead, it uses the never claims described in PROMELA. These never claims are interpreted in the model as a monitor process, which runs in parallel.

In SPOT, we only want to use PROMELA to express models but not their properties. We want to continue to use our synchronized product algorithms between a space-state automaton and the negated formula automaton.

We cannot find complete operational-semantics definition of the PROMELA language which will describe how to pass from a model express in PROMELA to a state-space. The original publication on operational-semantics (Natarajan and Holzmann, 1996) is now outdated, as SPIN evolves. It was improved by Weise (1997), but some semantics stay unsound like nested do loops in combination with goto statements. This issues are coming from Weber (2007), which describes the state of the art of PROMELA semantics in the “4.1 Promela Semantics” section. The only reference on operational-semantics of PROMELA is the source code of SPIN.
2.2 The Promela language

2.2.2 Example: Dining philosophers problem

To give a better overview of the language, here follows a small example. This example is a model in PROMELA of the famous dining philosophers problem.

The problem

\( N \) philosophers sit at a circle table with a bowl of rice. A stick is placed in between each philosopher. Philosophers have two states: thinking or eating with two sticks. Then, we can notice that there is a possibility of deadlock when every philosopher takes his left stick and waits for the right sticks.

The Promela model

The following code is a wrong solution to the problem with 2 philosophers.

Promela example: Dining philosophers problem

```promela
chan stick_1 = [1] of {bool};
chan stick_2 = [1] of {bool};
byte p1, p2; /* Id of processes */
proctype philo(chan left_stick, right_stick)
{
    do
        :: left_stick?_; /* Wait for data from the left stick */
        take_r: right_stick?_; /* Wait for data from the right stick */
        release_l: left_stick!1; /* Fill the left stick */
        release_r: right_stick!1; /* Fill the right stick */
    od
}
init
[
    atomic {
        stick_1!1;
        stick_2!1;
        p1 = run philo(stick_1, stick_2); /* Philosopher 1 */
        p2 = run philo(stick_2, stick_1); /* Philosopher 2 */
    }
]
```

Each stick is represented by a channel with a capacity of one. Sticks have then two states: empty or full. The communication is asynchronous, so receiving a message is a blocking action when the stick channel is empty.

Philosophers are represented by processes running in parallel. Each process has the same behavior:

1. Wait to receive a value from his left stick.
2. Wait to receive a value from his right stick.
3. Send a value to the left stick.
4. Send a value to the right stick.
The initialization is surrounded by the atomic keyword. This means no process can run before the achievement of the initialization (channels filled and processes started).

State-graph representation

From this model, we need to generate the state-space graph of this model before doing the synchronized product. The state-space graph is produced by the simulation of all the behaviors of the system.

The Figure 2.2 is the state-space representation of the model. The sticks’ states are: “e” when the stick is empty, and “f” when it is filled by a value.

The first philosopher steps of the process are represented horizontally in the picture, and the second philosopher steps vertically. The notation \texttt{p[p1]@take_r} means the label \texttt{take_r} of the process \texttt{philosopher 1}. This state space represents the model after its initialization.

![State-space graph of the dining philosophers problem with 2 philosophers.](image)

In our example, the model is designed without \texttt{never} claims. When some are present, \texttt{never} claims also use the state-space.

2.2.3 \texttt{never} claims

In this example, it could be interesting to show how to write a \texttt{never} claim, and how to link it to LTL.

With our example, a condition that must never happen could be: “The two philosophers must never been blocked”. In other terms, we could say “The philosopher 1 must always pass through a non-blocking instruction”. With LTL semantics, this statement is \texttt{GF(p[p1]@release_r)}, with \texttt{G} for \textit{always}, \texttt{F} for \textit{eventually in the future} and \texttt{p[p1]@release_r} refers to the instruction “fill the right stick of the philosopher 1”.

---

**Figure 2.2**: State space of the dining philosophers problem with 2 philosophers.

In our example, the model is designed without \texttt{never} claims. When some are present, \texttt{never} claims also use the state-space.

2.2.3 \texttt{never} claims

In this example, it could be interesting to show how to write a \texttt{never} claim, and how to link it to LTL.

With our example, a condition that must never happen could be: “The two philosophers must never been blocked”. In other terms, we could say “The philosopher 1 must always pass through a non-blocking instruction”. With LTL semantics, this statement is \texttt{GF(p[p1]@release_r)}, with \texttt{G} for \textit{always}, \texttt{F} for \textit{eventually in the future} and \texttt{p[p1]@release_r} refers to the instruction “fill the right stick of the philosopher 1”.

---

**Figure 2.2**: State space of the dining philosophers problem with 2 philosophers.

In our example, the model is designed without \texttt{never} claims. When some are present, \texttt{never} claims also use the state-space.

2.2.3 \texttt{never} claims

In this example, it could be interesting to show how to write a \texttt{never} claim, and how to link it to LTL.

With our example, a condition that must never happen could be: “The two philosophers must never been blocked”. In other terms, we could say “The philosopher 1 must always pass through a non-blocking instruction”. With LTL semantics, this statement is \texttt{GF(p[p1]@release_r)}, with \texttt{G} for \textit{always}, \texttt{F} for \textit{eventually in the future} and \texttt{p[p1]@release_r} refers to the instruction “fill the right stick of the philosopher 1”.

---

**Figure 2.2**: State space of the dining philosophers problem with 2 philosophers.

In our example, the model is designed without \texttt{never} claims. When some are present, \texttt{never} claims also use the state-space.

2.2.3 \texttt{never} claims

In this example, it could be interesting to show how to write a \texttt{never} claim, and how to link it to LTL.

With our example, a condition that must never happen could be: “The two philosophers must never been blocked”. In other terms, we could say “The philosopher 1 must always pass through a non-blocking instruction”. With LTL semantics, this statement is \texttt{GF(p[p1]@release_r)}, with \texttt{G} for \textit{always}, \texttt{F} for \textit{eventually in the future} and \texttt{p[p1]@release_r} refers to the instruction “fill the right stick of the philosopher 1”.
never claims for the Dining philosophers problem

```plaintext
never {
    T0_init: /* Initial state. */
    if
        :: (philo[p1]@release_r) → goto accept_S9
        :: (1) → goto T0_init
    fi;
    accept_S9: /* Accepting state. */
    if
        :: (1) → goto T0_init
    fi;
}
```

As we can notice it, the never claims represent a Büchi automaton. It accepts the language if the run of the automaton visits the accepting state infinitely often. This automaton is represented in Figure 2.3.

![Figure 2.3: Never claim automaton.](image-url)
Chapter 3

Promela front-end in Spot

SPOT is built to work with TGBA. Many concrete implementation of TGBA exists in SPOT, in order to implement some optimization or space reduction according to the input. All these implementations are subclasses of the \texttt{tgba} class, already briefly presented.

Adding a PROMELA front-end in SPOT amounts to add a new concrete implementation of the \texttt{tgba} class.

3.1 Initial “hack”

The first solution to be compatible with models written in PROMELA, was to use SPIN to generate a TGBA. This solution was a perl script outside SPOT. SPIN provides a feature to explore the state graph of a PROMELA model. This process was a conversion of the PROMELA model to the SPOT’s input format.

This solution had some inconvenient. The biggest is of course the impossibility to do on-the-fly verification. All states must be loaded in memory before starting the verification, which causes issues when the automaton is huge (cf. state space explosion), and requires to load and store a large automaton file.

3.2 The NIPS virtual machine

The NIPS VM is described by Weber (2007) as a Virtual Machine for state space generation that is designed as a modular, efficient, reusable, embeddable explicit state model checker tool engine.

This virtual machine takes as input a NIPS bytecode generated from a PROMELA model with a PROMELA-to-NIPS bytecode compiler. The bytecode is an assembly language, where all atomics names originally written in the PROMELA model are lost.

From this bytecode, the virtual machine generates the state space of the PROMELA model, and provides functions to explore it.

Using this already existing virtual machine has several advantages. First of all, it saves us from implementing the PROMELA semantics, which is non-trivial and not entirely described in papers. We can notice that one of the acronym of NIPS is \textit{Never Implement PROMELA Semantics (again)}. 
Moreover, NIPS has been built to take care of the combinatorial explosion. NIPS provides functions to explore the state graph on the fly, which allows us not to store the entire state graph in memory.

However, this solution has some problems. The biggest is the lost of the name of atomics properties expressed in the PROMELA model. Without this information, we cannot define LTL formulæ which will control the system behavior.

In this case, we have to use never claims to express conditions on the system instead of LTL. Hence, with this front-end, the usual process presented in Figure Figure 2.1 has changed, and some bricks of SPOT are now useless. The Figure 3.1 illustrate the new behavior of SPOT. LTL-to-Büchi translation algorithms, and Synchronized product algorithms between two TGBA cannot by used anymore by SPOT. Since formulæ are included in the model, NIPS acts as a black-box, taking as input a bytecode the of PROMELA model and generating the state graph with the never claim. Then NIPS provides functions to explore the state graph on the fly, and gives information on the states, like successors or accepting states.

![Diagram](http://lrde.epita.fr/~adl/git/spot.git)

**Figure 3.1:** Model checking with SPOT and NIPS.

From this graph and accepting states, we have done a subclass of tgba with its methods calling NIPS VM functions. For SPOT, this interface with NIPS and then the PROMELA model is like other tgba. This implementation can be found in the SPOT repository\(^1\) in the directory iface/nips/.

\(^1\)http://lrde.epita.fr/~adl/git/spot.git
Some problems also occur with the NIPS virtual machine. The NIPS compiler does not support some syntactic expressions that SPIN supports, which may sometime require to partially rewrite PROMELA models which was working with SPIN. Also, the NIPS bytecode cannot express some concepts that SPIN can. For example, the _pid special variable of SPIN, which represents the PID of the current process is not handled by the NIPS bytecode. Moreover, the behavior of the NIPS VM does not always follow the same behavior as SPIN.
Chapter 4

Benchmark

4.1 Benchmark with Spin

With SPOT having a PROMELA input, we can do some benchmarks between SPIN and SPOT. Although we are expecting SPOT to be less efficient, the most important observation is our compatibility with Spin.

For our benchmark, the model chosen is electro-mechanical relay circuits (van Eijk, 1997). In this model, the size of the state-graph depends on the number of circuits.

Our tests (Figure 4.1) will be with 5, 6 and 7 circuits. We will compare memory and speed. Our comparison is done without partial order reduction, in order to have almost the same state-graph than SPIN.

First of all, we can notice that the number of states and transitions is always five times larger when a circuit is added. The memory used by SPOT is constant according to the number of states. However, SPIN use compression for its states and transitions, which explains better memory results, and the slow growth of memory. With SPOT and NIPS, we cannot use an efficient compression for states and transitions because NIPS uses uncompressed states to provide successors and information.

We also notice that SPIN is five to more than ten times faster than SPOT. The memory is one of the responsible part of this situation, but many other reasons may also explain this gap. SPIN’s verifier is an optimized and generated monolithic C program, while SPOT is a more modular and readable C++ library.
### Figure 4.1: Memory and time used by SPOT and SPIN on the electro-mechanical relay circuits
Chapter 5

Discussion

5.1 Related work

5.1.1 Promela front-end in other model checkers

A few model-checker have added a PROMELA front-end to be SPIN compatible.

NIPS VM is the only virtual machine for space-state generation that we can find with a native support of PROMELA. This virtual machine has been built in order to be embeddable in other model checkers. Having a virtual machine for space-state generation that already implements PROMELA semantics is a precious help.

Many other model checking tools are also using NIPS. We can cite some applications:

DiVinE (Barnat et al., 2006) is model-checking tools which provides a distributed state space analysis. DiVinE use his own input format called DVE. However, in order to be compatible with the SPIN model checker, and have access to all PROMELA models already existing, and do tests with these models, they extended their model checker to an SPIN compatible model checker called DiVSPIN (Barnat et al., 2005), which uses the NIPS VM.

An External-Memory Model Checker (Behrmann et al., 2003) The virtual machine is used as state-space generation component in an adaptive external-memory model checking tool. The tool stores states not only in the main memory, but also on the hard drive, making it possible to check Promela models with larger state spaces.

VMSSG State space Converter. This tool generate a state space graph and display the graph in POVRay or DOT format, in outlining information like strongly connected components.

5.1.2 Spot with an external tool

Connecting SPOT with an external tool has already be done with GREATSPN and CHECKPN. GREATSPN is a software package for the modeling, validation, and performance evaluation of distributed systems using Petri Nets. It was done by adding a concrete class to the tgba class, with methods calling GREATSPN functions. A more precise description of this interface can be found in Duret-Lutz (2007), Annexes A.2.1.
5.2 Future work

Using NIPS is an easy solution for us to add a PROMELA front-end, however it has some limitations, like the impossibility to use LTL formulae from SPOT and to check our synchronized product algorithms.

As we have explained it, rewriting from scratch our own Promela front-end and its state-space generator would be a hard task, due partially to the lack of information on the operational-semantics of PROMELA.

However, the SPIN model checker has an option to generate a verifier written in C. The generated verifier has functions to explore the state-graph. We had the idea to use SPIN to provide a state-space exploration library using the generated code. With a modification of the SPIN generating component, we should be able to check the value of variables described in the PROMELA model, and then follow the expected behavior of SPOT with a LTL formula and a model. With this front-end, we will be able to check all our algorithms, synchronized product included.
Chapter 6

Conclusion

We presented an extension of SPOT with a PROMELA front-end. As we shown it, adding this high-level modeling language as input will let us make SPOT more usable, and test our algorithms with real complex models.

This front-end was done using a virtual machine for state-space generation called NIPS. Using this existing solution brings us some advantages, but also some drawbacks. The advantages are the already existing implementation of the PROMELA semantics, the generation of the state-space by the virtual machine, and the on-the-fly exploration. However, the virtual machine uses its own bytecode which loses the names of variables used in the model. Therefore we cannot express properties on our model in SPOT with LTL. We get around this constraint with never claims statements in PROMELA which let us express properties, but we cannot use and test all SPOT’s features, such as the synchronized product algorithms.

However, we can now use some features of SPOT with PROMELA, such as emptiness check algorithms.

A solution using SPIN itself as a space-state generator will allow us to check synchronized product algorithms and to use SPOT as expected. The future works on SPOT and the PROMELA front-end will be to modify SPIN to produce our generating-library, and to interface this library with SPOT.
Chapter 7

Bibliography


