Original scientific paper

Informacije MIDEM
Journal of Microelectronics,
Electronic Components and Materials
Vol. 43, No. 4 (2013), 235 – 250

Model Checking using Spin and SpinRCP

Zmago Brezočnik, Boštjan Vlaovič, Aleksander Vreže

Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia

Abstract: Spin is one of the leading verification tools for the model checking of distributed systems. It is used over a broad spectrum of applications where systems can be represented as asynchronously running processes. This paper provides an overview of the concepts of model checking, the Spin model checker together with its input language Promela, and of the available graphical user interfaces to Spin. In order to offer Spin users an integrated development environment for Spin, we have developed a SpinRCP. We introduce its structure and demonstrate some of its features by considering a standard algorithm for leader election in a unidirectional ring.

Keywords: formal verification, model checking, modelling, simulation, Promela, Spin, SpinRCP

Preverjanje modelov z uporabo orodij Spin in SpinRCP

Izvleček: Spin je eno izmed vodilnih verifikacijskih orodij za preverjanje modelov porazdeljenih sistemov. Uporablja se za širok spekter aplikacij, pri katerih lahko sisteme predstavimo kot asinhrono izvajajoče se procese. V članku podajamo kratek pregled osnovnih pojmov o preverjanju modelov, preverjalniku modelov Spin, njegovem vhodnem jeziku Promela in razpoložljivih grafičnih uporabniških vmesnikih za Spin. Da bi uporabnikom orodja Spin ponudili integrirano razvojno okolje za Spin, smo razvili SpinRCP. Predstavljamo njegovo strukturo in prikažemo nekatere izmed njegovih značilnosti na primeru standardnega algoritma za izbiro vodje v enosmernem obroču.

Ključne besede: formalna verifikacija, preverjanje modelov, modeliranje, simulacija, Promela, Spin, SpinRCP

* Corresponding Author's e-mail: brezocnik@uni-mb.si

1 Introduction

The constantly increasing sizes and complexities of contemporary ICT (Information and Communication Technology) systems as well as demands for reduction in costs and a shortening of time-to-market for a new product, confront designers with harder and harder tasks for ensuring the correct functioning of developed systems. Nowadays, ICT systems are becoming ubiquitous in our daily lives and we have to rely on their correctness. If any of them malfunction, it will be at least annoying for its user but in the case of safety-critical systems, such as for example systems in transport, medicine, industry, ecology, telecommunications, space exploration, and the military, each undiscovered error in the hardware or software may cause a lot of damage, threatening the health or even the lives of people. Let us recall some dramatic examples. Between 1985 and 1987 four cancer patients died and two were seriously injured following incorrect behaviour (100-times radiation overdosing) of the Therac-25 anti-tumour irradiating machine. The cause was a design error in the control software. On 4th June 1996, the Ariane 5 rocket changed trajectory and exploded 40 seconds after being launched on its first flight. The cause of failure was a variable overflow during the conversion of a 64-bit real number to a 16-bit integer. In 1997, a computer on-board the Mars Pathfinder had to be reset often due to an error in the algorithm for access of several processes to a common computer bus. A design error in the Intel Pentium floating point division algorithm in 1994 caused the loss of about 450 million dollars when replacing faulty processors. Therefore, the reliability of systems is a key issue in the system design process that takes up the greatest part of the design time and effort. A very promising approach towards ensuring the correctness of ICT systems is that of model checking. Model checking is a formal verification method that can automatically verify the desired behavioural properties of a given system through exhaustively exploring all states of a system's suitable model. One of the most successful model checkers is Spin [1, 2]. Although it was originally designed as a tool for protocol verification, it is capable of model checking and simulating almost any system consisting of asynchronously running processes. Spin is a command line tool that accepts user commands from a command line and also outputs the results there. In order to ease Spin model checking, some graphical user interfaces have appeared for Spin. Two of them were developed at the Faculty of Electrical Engineering and Computer Science at the University of Maribor. In this paper we introduce the latest one called SpinRCP.

Section 2 introduces the basic concepts of model checking including its strengths and weaknesses. Section 3 provides a short overview of Spin and its input language called Promela. Section 4 gives an overview of the currently available graphical user interfaces for Spin. Section 5 introduces our new integrated development environment (IDE) for Spin called SpinRCP. The use of SpinRCP is demonstrated on the model of an algorithm for leader election in a unidirectional ring in Section 6. Section 7 draws together the concluding remarks.

2 Model checking

Model checking is an automated technique that, given the finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model. A detailed explanation of the model checking technique can be found in [3]. A schematic view of model checking is shown in Fig. 1. We can distinguish between three different phases when applying model checking to a system: the modelling, running, and analysis phases.

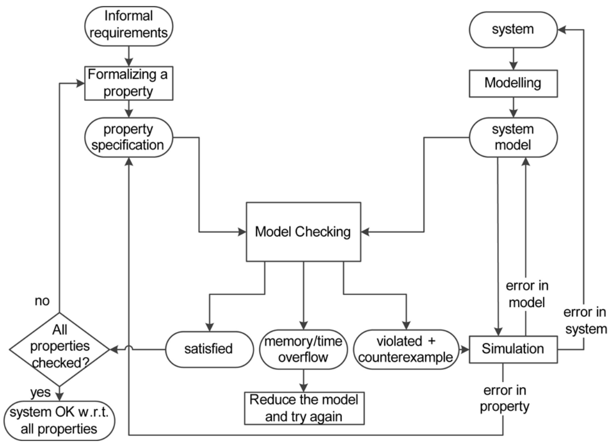


Figure 1: Schematic view of model checking.

In the *modelling phase*, a model of the system under consideration has to be obtained. Models describe the possible behaviours of systems in a precise and unambiguous manner. They are usually expressed as finite-state automata that consist of a finite set of states and a final set of transitions. Models can be created either manually or automatically using a model extraction tool. In both cases the model has to be described in a description language used by the model checker. Simple errors within the model can be found by simulation prior to the model checking. Such inexhaustive simulation can disclose errors in the model but it cannot guarantee that the model is error-free. In order to find any remaining subtle errors, we have to resort to rigorous verification using model checking. The informal requirements for the system's behaviour have to be formalised into precise and unambiguous properties using an adequate property specification language. A kind of temporal logic is most often used. The model checking process actually checks whether the system description *M* is a model of a temporal logic formula *P*. A temporal formula can express the behaviour of a system over time. It allows the specifying of relevant system properties regarding the following kinds: functional correctness (does the system do what it should?), safety ("something bad never happens"), liveness ("something good will eventually happen"), fairness (does an event occur infinitely often under certain conditions?), and real-time properties (does the system act in due time?).

In the *running phase* the user has to set various options, parameters, and directives for the model checker that should be used for verification. Then, the model checker automatically checks the validity of the property under examination in all states of the system model.

In the *analysis phase* the user has to evaluate the outcome of model checking performed within the previous phase. There are three possible outcomes: the specified property is either valid or violated in the given model, or the model turns out to be too large to fit in the available computer memory or the model checking run is unfinished within a reasonable amount of time. If the property is valid, the next property can be checked with a new run perhaps with different options, parameters, and/or directives. If the property is violated, a counterexample is generated that can be analysed by simulation. The counterexample guides the simulation run across a path where the property is violated. A thorough counterexample analysis can reveal the cause of the violation. It may be a *modelling error*, a *system design error*, or a *property error*. In the case of the modelling error, the model does not reflect the design of the system. The model has to be corrected and all the properties have to be checked again. In the case of the system design error, the system does not behave as required and has to be corrected. The last possible cause is that the property does not reflect the informal requirement for the system behaviour. This implies a revision of the property and a new verification for this property. The designed system is verified with respect to the given properties if and only if all of them have been proven valid. Whenever the model is too large to be handled within the available amount of computer memory or within a reasonable time, the model has to be reduced using different abstractions.

Model checking has many strengths including the following ones: it is a general technique that is applicable to a wide range of systems (e.g., protocols, hardware, software), it allows partial verification (only the more relevant properties are checked), it is nothing harder to find less likely errors than the more likely ones, it provides a counterexample in case a property is invalidated, it is an automatic technique and software tools exist, or it is more and more accepted in the industry.

On the other hand, it also has some weaknesses: it is mainly appropriate for control-intensive applications and much less suitable for data-intensive applications, it is applicable only to final-state systems, it verifies a system model, and not the system itself, its results are only as good as the system model, it checks only given properties and provides no guarantee about the completeness of the results, it is faced with state-space explosion, or it cannot handle parameterised systems.

Despite these weaknesses it is a fact that model checking is an effective technique for exposing potential design errors and thus increases the quality of a system design.

3 Spin and Promela

This section provides a short introduction to the input language of Spin called Promela and the main features of Spin.

3.1 Short Introduction to Promela

Promela (**Pro**cess or **Pro**tocol **Me**ta **La**nguage) is a language for describing finite state automata. It is not intended to be an implementation language but a system description language that is aimed at facilitating the searches for good abstractions of systems' designs. The language is oriented towards the modelling of process coordination and synchronisation, and not to computation. The reason is simple. Promela is not a programming language but a verification modelling language. Promela language reference can be found in [1].

A Promela model consists of processes, variables, and channels. It corresponds to a finite state transition system, hence all objects in Promela are bounded. The processes are global objects, whilst the variables and channels may be declared either as global or local to a process. A new inline or block scope for variables has been introduced since Spin Version 6.

The processes are instantiations of proctype declarations and define parts of the system's behaviour. A proctype consists of a name, a list of formal parameters, local variable declarations, and a body. The body consists of a sequence of statements. A process executes concurrently with all other processes. It communicates with other processes using either global (shared) variables or channels. There may be several processes of the same type. Each process has its own local state that consists of a process counter (current location within the proctype) and the values of the local variables. Processes can be created within any process at any point of execution using the run statement. They can also be created by adding the keyword active in front of the proctype declaration. A so-called init process becomes active in the initial state of the system. It serves to initialise other processes and global variables.

Promela has nine basic (integer) variable types: bit or bool (1 bit), byte (8 bits), chan (8 bit), mtype (8 bit), pid (8 bit), short (16 bits), int (32 bits), unsigned ($1 \le n < 32$ bits). More complex types (records) can be composed from the basic types with the keyword typedef.

Message channels in Promela are used to model the exchange of data between processes. Channels can be either asynchronous or synchronous (rendezvous). Asynchronous channels are queues or buffers, and can store a finite number of messages. Synchronous channels have a length of zero. Channel declaration is introduced by the type name chan followed by the channel name, length of the channel queue (channel capacity) and the structure of the messages, which can be stored in the channel as a comma-separated list of type names.

The body of a process consists of statements that execute sequentially. There are two kinds of statements: statements that never block and statements that may block. An expression is also a statement and is executable if it evaluates to non-zero but is blocked otherwise. No other statement within a process may be executed until the statement evaluates to true, which may result from a variable assignment or a received message, for example. Statements if and do consist of one or more option sequences. An option sequence begins with a guard, and if the guard evaluates to true, the option sequence is selected for execution. If more than one guard is true, the selection of the statement to be executed is non-deterministic. If no choice is executable, the statement is blocked. The only difference between these two statements is that after the execution of the selected statement, a do statement repeats the choice selection but an if statement ends the execution. The always executable break statement exits a do loop.

There are four ways to express correctness properties in Promela: assertions, trace assertions, labels, and never claims. Assertions express invariants. If an invariant is false at the point where the assertion occurs, Spin reports an error. Trace assertions perform similarly to channels. Labels can be of type *accept*, *end*, and *progress*. They mark the states in a process and have a special meaning when Spin is run in verification mode. The accept state labels are normally used only in never claims that are mentioned below. The verifier can find all cycles that do pass through a state with an accept label. The end label marks a valid end state that is not at the end of a process' code (i.e., the closing curly brace in the corresponding proctype body). The progress label marks a statement in a Promela model that accomplishes something desirable and thus makes progress. The never claim expresses behaviour that should never occur. Usually, it is automatically generated from a linear temporal logic (LTL) formula.

3.2 Principles of Spin

Spin is one of the most often used and successful model checkers. It was written by Gerard J. Holzmann at Bell Labs [1, 2]. The software has been available freely since 1991, and continues to evolve. The latest version of Spin is Version 6.2.5 issued on 4th May 2013. Spin download distributions and a lot of additional tools and relevant information can be found at the Spin home page http://spinroot.com/. Spin beginners' tutorial and practically-oriented introduction to the principles of the Spin model checker are available in [4] and [5], respectively. The Spin model checker takes a Promela model as an input. Hence where its name is derived from. Spin is an acronym for **S**imple **P**romela **In**terpreter. In 2001, the Association for Computing Machinery (ACM) recognized Dr. Gerard Holzmann by a prestigious ACM Software System Award for Spin.

Let us suppose that a Promela model M has *n* processes, defined by proctype declarations. Spin first transforms them into finite state automata A1, A2, …, An. Then, it creates an asynchronous product of all automata. The states of this product automaton are called the state space or the reachability graph of the model. Next, consider that a property that has to be satisfied by a model, is expressed by an LTL formula *f*. Spin first generates a never claim for the negated formula ¬*f* and converts it into a corresponding Büchi automaton, B. Spin can check if M satisfies *f* by computing the global system automaton S

$$S = B \otimes \prod_{I=1}^{n} A_i$$

We use the operator ∏ to represent an asynchronous product of multiple component automata, and ⊗ to represent the synchronous product of two automata [1]. The automaton S is now analysed for its acceptance properties. If S has accepting ω-runs (i.e., a certain state in the set of its final states is visited infinitely often in the run), then formula *f* can be violated (and ¬*f* can be satisfied). If no accepting ω-runs are found, then the system is considered valid. In this way liveness properties are formulated. Informally, a liveness property says that "something good will eventually happen". Spin works the other way round. It tries to find a (infinite) loop in which the "good things do not happen". If there is no such loop, the property is satisfied. Another class of properties is safety properties (e.g., invariance, deadlock, livelock, unreachable states). Informally, a safety property says that "something bad never happens". Spin tries to find an execution path leading to the "bad" thing. If there is no such execution path, the property is satisfied.

For each model to be verified, Spin first generates a C verifier program, which can then be compiled to the executable verifier called pan. When it finds an error, a counterexample is generated, which can be used in a guided simulation that replays the execution path that violated the property. Random and interactive simulation is also available. They can be used for the sake of early detection of (simple) errors before verification.

Spin has several optimisation algorithms to make the verification process more effective (e.g., partial order reduction, statement merging, state compression, bitstate hashing, hash-compact) [1]. In addition, a slicing algorithm gives the user hints of what can be "thrown away" in a model description in order to reduce space and time consumption.

3.3 Spin Application Domains

Due to the nature of Promela, it can be used to model many different kinds of systems of asynchronously (and synchronously) communicating processes [6]. Thus, Spin that takes a Promela model as its input, has been used for a wide range of different applications: protocol design and verification, feature interaction, safety critical system verification, embedded and reactive system verification, hardware circuit modelling, hardware/software codesign, finding solutions for optimising problems, verifying contracts and guidelines, verifying business processes, modelling telephone switching systems, modelling multimedia presentations, modelling routers and network traffic, modelling operating system kernels, scheduling and plan execution, verifying bytecode, modelling genes, etc.

4 Graphical environments for Spin

The most basic way of using Spin is textual. A user enters the commands at the command line and Spin outputs are printed on standard output afterwards. Such an approach to model checking could be difficult and discouraging especially for newcomers who are not yet well acquainted with Spin commands. Of course, the availability of a user-friendly graphical interface to Spin is of a considerable benefit for skilled users as well. In order to accomplish these needs, several different graphical interfaces or environments for Spin have been developed. This section provides a brief overview of Xspin, jSpin, Eclipse Plug-in for Spin, iSpin, and EpiSpin. In the next section we present our SpinRCP.

4.1 Xspin

Xspin [1] was the first graphical interface to Spin. The interface runs independently from Spin itself. It generates the proper Spin commands in the background, based on user menu selections and button clicks, then obtains the Spin output and wherever possible attempts to generate a graphical representation of this output. Xspin interface was very suitable for dealing with small models, but coping with larger ones was much more difficult due to the lack of syntax colouring and code folding. In addition, the opening of a separate window for each task during investigation of a model was inconvenient.

Xspin was written in Tcl/Tk script language. The last version of Xspin was Xspin Version 5.2.5 from 17th April 2010. Since then it has no longer been supported. In Spin Version 6 it was superseded by iSpin.

4.2 jSpin

jSpin [7] is an alternative graphical user interface for the Spin model checker. It was developed by M. Ben-Ari primarily for pedagogical purposes. It is written in Java.

jSpin's interesting part is its SpinSpider component, which is very useful for demonstrating the properties of concurrent programming. As in the case of Xspin, syntax colouring and code folding are missing. In contrast to Xspin, message passing between communicating processes is displayed only in textual form.

4.3 Eclipse Plug-in for Spin

In [8, 9] we introduced a new approach for automatic model extraction and applied it in our tool called *sdl2pml*. It can generate a Promela model of a system specified in SDL. The tool was tested on the implementation of an ISDN User Adaptation (IUA) protocol, which is part of the SI3000 softswitch. The specification was developed by Iskratel d.o.o., which is the largest Slovenian telecommunication equipment developer. The generated Promela model is huge with its 79,281 lines of code. We didn't have any suitable Promela editor that would be suitable for such large models. Therefore, we decided to create an Eclipse Plug-in for Spin with a user-friendly Promela editor that includes syntax colouring and code folding and similar capabilities for running Spin syntax check, simulation and verification as Xspin [10, 11]. In order to ease analyses of extremely long Spin simulation trails (e.g., Spin trail of the simple call with the use of IUA protocol consists of 55,371 lines of text and contains 21 processes that communicate using 261 messages) we have developed a Spin Trail to Message Sequence Chart (MSC) tool called *st2msc* [12]. It converts a Spin simulation trail into a standard MSC textual representation according to standard Z.120 [13]. Professional telecommunication design tools such as ObjectGEODE can read this MSC representation and display message sequence charts in graphical form.

The Spin Plug-in is written in Java and uses numerous other plug-ins available within the Eclipse environment. It has proved to be very useful when working with large models but also has some shortcomings: an interactive simulation is not implemented, better filtering of the Spin simulation output trail is missing, and graphical display of MSCs is only possible with the use of external tools.

4.4 iSpin

iSpin [14] is the graphical user interface that has replaced Xspin since Spin Version 6.0.0. It was provided by the Spin author, G. J. Holzman, and is included in each new Spin release. Just like Xspin, iSpin is implemented using the Tcl programming language and the Tk graphical user interface toolkit.

iSpin is a good upgrade of Xspin and is much more user-friendly, because all operations display their results in different areas of a single window and don't open new windows as in the case of Xspin. iSpin is the only graphical user interface to Spin that supports a swarm verification run [15] that distributes verification loads amongst more computing cores or platforms. Syntax colouring and code folding would be welcome for the editing of large Promela models.

4.5 EpiSpin

EpiSpin [16], introduced by de Vos et. al., is an Eclipse plug-in for editing Promela models and starting Spin verification and simulation runs. It includes its own error checker that instantaneously feedsback on syntax and semantic errors, syntax highlighting, code folding, reference resolving, and a graphical tool for displaying the static relationships between channels, and processes and global variables. EpiSpin has been built using the Spoofax language workbench.

This tool is primarily oriented towards providing various editor services for editing Promela models and a dot graph that shows how model processes can communicate using channels and global variables.

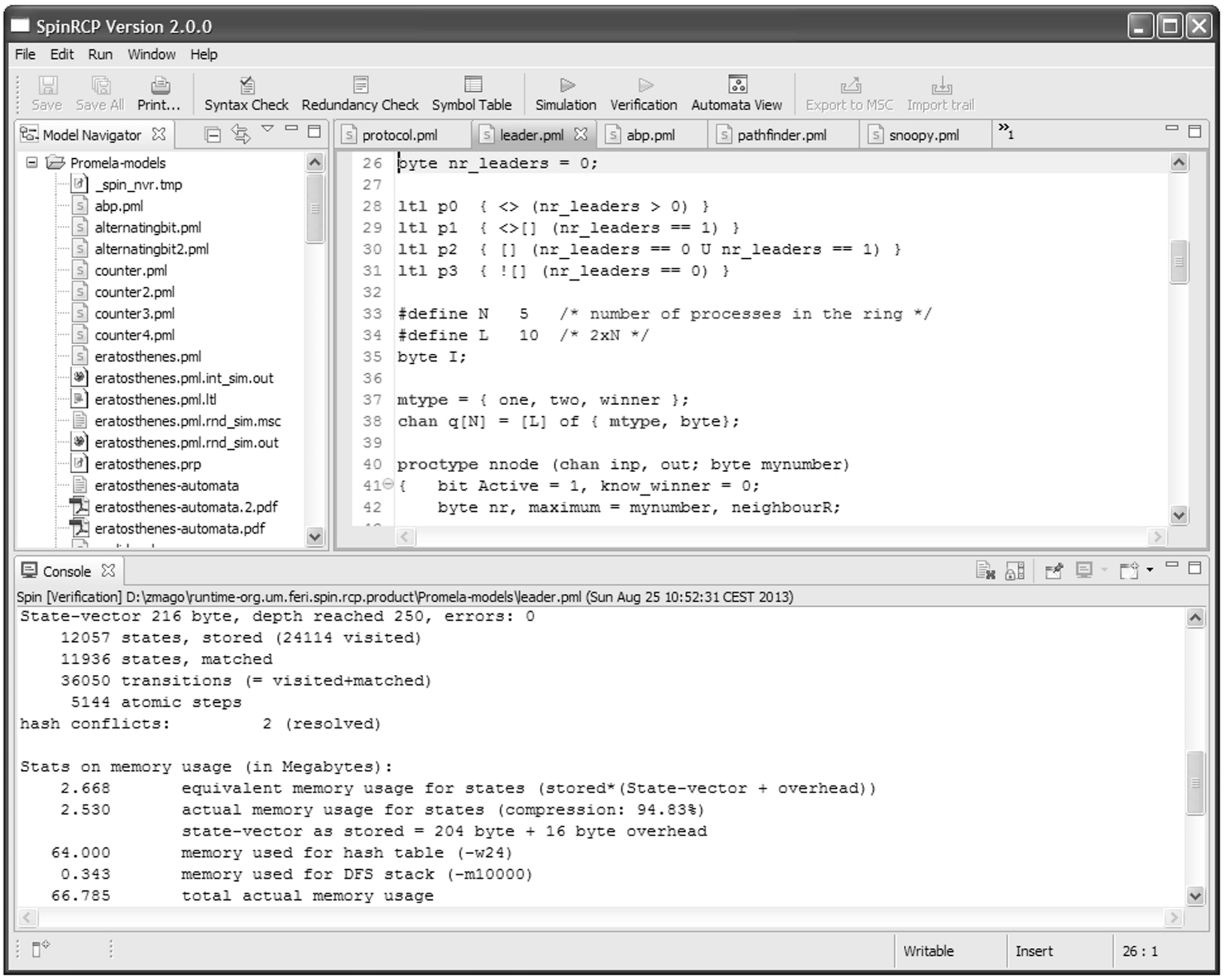
5 SpinRCP IDE

Motivated with good responses to our Eclipse Plug-in for Spin and suggestions for the integration of the full functionality of the then graphical interface Xspin into Eclipse, we decided to develop an integrated development environment that will facilitate entering new or reviewing extracted Promela models from an existing software code, simple parameters choosing for individual operations on the model, running Spin verification and simulation, and keeping records of file versions [17]. This new environment will remove some disadvantages of Eclipse Plug-in for Spin, particularly the lack of interactive simulation implementation and the need for external tools to display MSCs graphically.

For the implementation of this environment we selected the Eclipse Rich Client Platform (RCP) technology. RCP is the minimum set of plug-ins needed to build a rich client application. It allows us to quickly build a professional-looking application, with native look-and-feel, on multiple platforms.

5.1 SpinRCP architecture

After launching the application, one single window entitled SpinRCP with its version number opens. Ac-


SpinRCP Version 2.0.0
File Edit Run Window Help
Save Save All Print... Syntax Check Redundancy Check Symbol Table Simulation Verification Automata View Export to MSC Import trail
Model Navigator
Promela-models
_spin_nvr.tmp
abp.pml
alternatingbit.pml
alternatingbit2.pml
counter.pml
counter2.pml
counter3.pml
counter4.pml
eratosthenes.pml
eratosthenes.pml.int_sim.out
eratosthenes.pml.ltl
eratosthenes.pml.rnd_sim.msc
eratosthenes.pml.rnd_sim.out
eratosthenes.prp
eratosthenes-automata
eratosthenes-automata.2.pdf
eratosthenes-automata.pdf
protocol.pml leader.pml abp.pml pathfinder.pml snoopy.pml
Console
Writable Insert 26 : 1

```
byte nr_leaders = 0;

ltl p0  { <> (nr_leaders > 0) }
ltl p1  { <>[] (nr_leaders == 1) }
ltl p2  { [] (nr_leaders == 0 U nr_leaders == 1) }
ltl p3  { ![] (nr_leaders == 0) }

#define N   5   /* number of processes in the ring */
#define L   10  /* 2xN */
byte I;

mtype = { one, two, winner };
chan q[N] = [L] of { mtype, byte};

proctype nnode (chan inp, out; byte mynumber)
{   bit Active = 1, know_winner = 0;
    byte nr, maximum = mynumber, neighbourR;
```

```
Spin [Verification] D:\zmago\runtime-org.um.feri.spin.rcp.product\Promela-models\leader.pml (Sun Aug 25 10:52:31 CEST 2013)
State-vector 216 byte, depth reached 250, errors: 0
    12057 states, stored (24114 visited)
    11936 states, matched
    36050 transitions (= visited+matched)
     5144 atomic steps
hash conflicts:          2 (resolved)

Stats on memory usage (in Megabytes):
    2.668       equivalent memory usage for states (stored*(State-vector + overhead))
    2.530       actual memory usage for states (compression: 94.83%)
                state-vector as stored = 204 byte + 16 byte overhead
   64.000       memory used for hash table (-w24)
    0.343       memory used for DFS stack (-m10000)
   66.785       total actual memory usage
```

Figure 2: SpinRCP Workbench during verification.

cording to the terminology of Eclipse it is called a Workbench. A Workbench consists of perspectives, views and editors. A perspective is a group of views and editors in the Workbench window. A view and an editor are visual components within the Workbench. In SpinRCP, there are many different views (e.g., Model Navigator, Console, Simulation, Spin Trail To MSC, CVS Repositories, ...) and only two editors (Promela Editor as a special case of Text Editor and MSC Viewer as a special case of Graphical Editor). Some features are common to both views and editors. We use the term "part" to mean either a view or an editor. Parts can be active or inactive, but only one part can be active at any one time. The active part is the one where the title bar is highlighted. Using a simple drag-and-drop operation you can relocate and/or resize any part and thus reform the SpinRCP perspective at your will. The outlook of Workbench with SpinRCP perspective during verification is shown in Fig. 2. It consists of the Menu bar, the Tool bar, the Model Navigator View, the Promela Editor, and the Console View. Below we present in short these parts of the Workbench.

5.2 Menu bar

At the top of the Workbench there is a Menu bar with five menus already known from Eclipse: File (to open, save, or print a file, and exit the application), Edit (to undo or redo an editing action, to cut, copy, paste, or delete a selected text, and to search for and replace a specific text in the Promela Editor or the Console), Run (to run any of the SpinRCP tools), Window (to show any of the available views, to open, save, or reset a perspective, to set general or Spin preferences), Help (to obtain installation details about SpinRCP, to browse help contents including the possibility of printing the selected topics without or with all subtopics, allows locating local topics, remote documents, and other documents given a search query, gives context-sensitive help, and displays a list of key bindings).

5.3 Tool bar

Below the Menu bar is the Tool bar. The first three Tool bar icons (Save, Save All, and Print…) are shortcuts of the equally named operations from the File menu. They are followed by eight shortcut buttons for launching a specific SpinRCP action. Syntax Check uses Spin –a option for performing a thorough model syntax check and generates the source C program for a model-specific verifier. Redundancy Check uses Spin –A option to apply a property-based slicing algorithm for the model, which can detect eventual redundancies in the model and generate suggestions on how the model could be revised in order to use less memory. Symbol Table uses the Spin –d option to produce symbol table information for the Promela model. The information for each Promela object depends on its type. Simulation opens the Simulation preference page, where the user can select the type of simulation (random, guided, interactive) and some simulation parameters. Then the simulation view opens and the user can start the selected type of simulation. Verification opens the Verification preference page, where the user can select and/or enter many basic and advanced verification options and then verification starts. Automata View opens the Automata View preference page, where the user can select in which graphical format the automata should be displayed. Ten different file formats are currently available. SpinRCP uses options –o3 and –a to generate the verifier source C code, then compiles it to pan and runs pan using run-time option –D. This option generates state tables for each proctype and each never claim in the format accepted by the dot tool from Graphviz [18]. These state tables are redirected to a text file. Next, the dot tool transforms this text file to a set of files (one file for each proctype or never claim) with the previously selected graphical format. Then, a dialogue appears, where the proctype or the never claim to be displayed can be selected. Finally, a system program, assigned to a given file type is opened and the selected automaton is displayed. In the Simulation View, where the MSC Viewer is active, two other shortcut buttons are enabled – Export to MSC and Import trail. By clicking the first one the currently displayed MSC in the MSC Viewer is exported to a selected file with standard MSC textual representation according to standard Z.120 [13] using the st2msc tool introduced already in [12]. By clicking the second button the Spin simulation trail in a selected file is read-in and the MSC is displayed in the active MSC Viewer.

5.4 Model Navigator

The Model Navigator View is designed to create a new file, folder, or project resource, an untitled text file, or import resources from the local file system into an existing project. The resources on local disk are represented in a tree structure. If a user double-clicks a file, an appropriate internal editor or external program opens this file. For Promela files of type pml and simulation trail files of type out the Promela Editor and MSC Viewer Editor are default editors, respectively. On the SpinRCP General preference page either an internal editor or an external program can be associated for each file type. Using the Project wizard within the Model Navigator we have to generate a project, which is actually a place, where our models and other files, produced by the SpinRCP IDE, will be stored. Within a wizard for the creation of a new Promela model, a user has to select a project (that is a container where the model is to be stored) and enter the file name of type pml. The Promela Editor opens a new file with a given name and automatically inserts an empty init process so that the user doesn't need to do it manually.

5.5 Promela Editor

We have developed the Promela Editor already for our Eclipse Plug-in for Spin [10, 11]. In SpinRCP it is almost unchanged. For ease of viewing and editing models the following features are available: syntax highlighting, code folding, content assist, and marking a place of a syntax error.

Syntax highlighting is a feature of Promela Editor that displays the text of the source file in different colours according to the category of terms. Highlighting does not affect the meaning of the text itself; it is only intended for human readers. The colours for different groups of reserved words, comments, and default text can be selected on the Promela Editor preference page. The default colours for text and comments are black and blue, respectively. According to the Promela language reference in [1], Promela reserved words can be grouped into seven sections: Meta Terms, Declarators, Control Flow Constructors, Basic Statements, Predefined Functions and Operators, Embedded C code, and Omissions. All the reserved words in a section will be displayed in the same colour. In each section a different colour can be set. By default, reserved words from all seven sections are displayed in the same (violet) colour. By default, the colours are enabled. They can also be disabled by deselecting the *Enable colours* checkbox.

Code folding allows the user to selectively hide or display sections of a currently-edited file. This allows the user to manage large amounts of text whilst viewing only those subsections of the text that are specifically relevant at any given time. Code folding is possible between an opening curly bracket "{" and closing curly bracket "}". This feature is commonly used to hide/display the bodies of large proctype declarations and is essential for studying the specifications in real systems.

Content assist or autocomplete is a functionality provided by SpinRCP that helps the user to write code faster. A user can just type in the first letter(s) of the reserved word and then press Ctrl+Space to be offered all the choices that match the entered letters that are valid for the current context. He/she simply selects the wanted word. This help is especially useful for a beginner who is not yet well acquainted with Promela syntax.

The Promela Editor uses a mechanism for marking syntax errors. If the Spin syntax checker detects a syntax error within Promela source code, SpinRCP parses its error message and finds out the line number where the error occurred. The Promela Editor marks this line with an error icon. A detail reason for the displayed syntax error is shown in the Console and in the Problems View.

Many instances of the Promela Editor containing Promela (or other) files can be opened simultaneously but only one editor can be active at any one time. The active editor is the one, the title of which is highlighted.

5.6 Console

Console is a view that is used for displaying the synthesised Spin commands and Spin textual outputs for all actions on the model. Below the console title bar a header line is displayed that consists of four strings: "Spin" – the parent application that is executing, the name of the action that is being performed (e.g., Verification, Random Simulation, etc.) in square brackets, the full path to the file containing the Promela model, over which the action is being performed, the exact date and time at which the action started.

All outputs for the same type of action (e.g., verification) and the same model are written to the same console. Thus, the number of created consoles at any time is equal to the sum of the number of different actions that have been performed on each model so far. Filtering of Spin outputs is helpful for a user who can now gather the results that are important to him/her much more easily. The contents of any console can also be cleared or copy-pasted in the whole or just within a selected area anywhere else for later use.

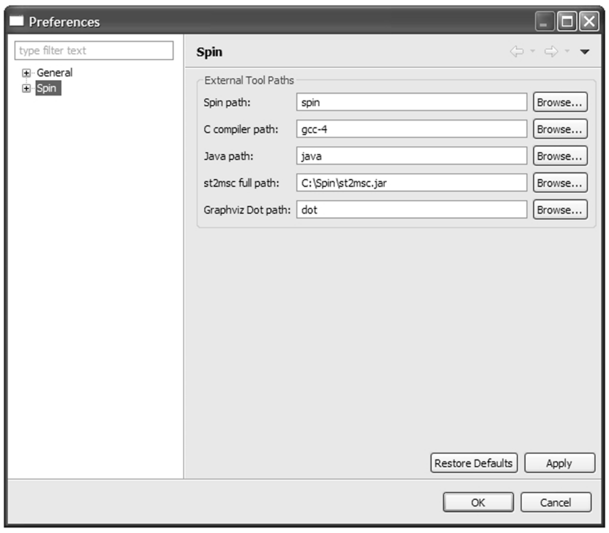
5.7 Preference pages

SpinRCP has several preference pages on which the user can set values to different kinds of variables. The following preference pages are available:

- General,
- Spin,
- Automata View,
- MSC Viewer,
- Promela Editor,
- Simulation,
- Verification.

In the General preference page a subset of the more common general preferences from Eclipse can be set (appearances of colours and fonts, editors file associations, different text editors' options, keys' bindings, perspectives, and workspace options, etc.).

Spin preference page (Fig. 3) is used to set paths to external tools. These paths have to be set before the first usage of SpinRCP. Spin is the main external tool for which the SpinRCP was developed at all. Therefore, it is mandatory to set the path to Spin. Since Spin generates a verifier for each model being checked as C source code, which has to be compiled to an executable code, we need an installed C compiler. We use gcc-4 from Cygwin environment and that is why we have set the path to gcc-4 C compiler. The conversion of a Spin textual simulation output trail to MSCs is done by the st2msc tool [12]. Therefore, the absolute path to the file st2msc.jar has to be set. Of course, the path to the Java Runtime Environment that runs st2msc.jar has to be set, too. A path to the Graphviz dot tool is necessary if we want to generate graphs of the processes and never claims. If the PATH environment variable of the computer system includes paths to Spin, C compiler, Java, and dot, we don't need to enter their absolute paths but just their names as shown in Fig. 3. Paths with spaces are also allowed. SpinRCP recognises such paths and encloses them in double quotes.



Spin

External Tool Paths

Spin path:	spin
C compiler path:	gcc-4
Java path:	java
st2msc full path:	C:\Spin\st2msc.jar
Graphviz Dot path:	dot

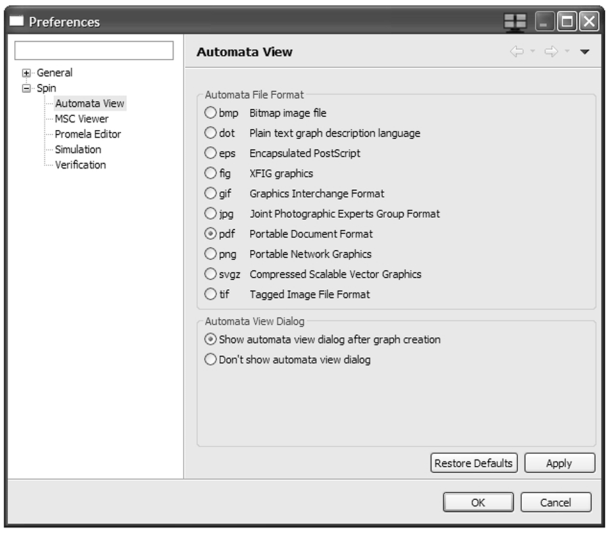
Figure 3: Spin preference page.

The Automata View preference page (Fig. 4) gives the user a choice to select a type of files that will contain the computed state transition system for all processes in the model and all never claims, and whether or not the automata view dialogue will be shown after graph creation. Currently the following file types are supported: bmp, dot, eps, fig, gif, jpg, pdf, png, svgz, and tif.

On the MSC Viewer preference page the MSC refresh interval can be set (default is 50 ms). In addition, it can also be set as to whether the message parameters will be shown in MSC diagrams or not.

The Promela Editor preference page is used to select colours for different categories of terms in Promela source file or disable colours.

The Simulation preference page (Fig. 5) offers the user the possibility of selecting the Spin simulation mode



Automata View

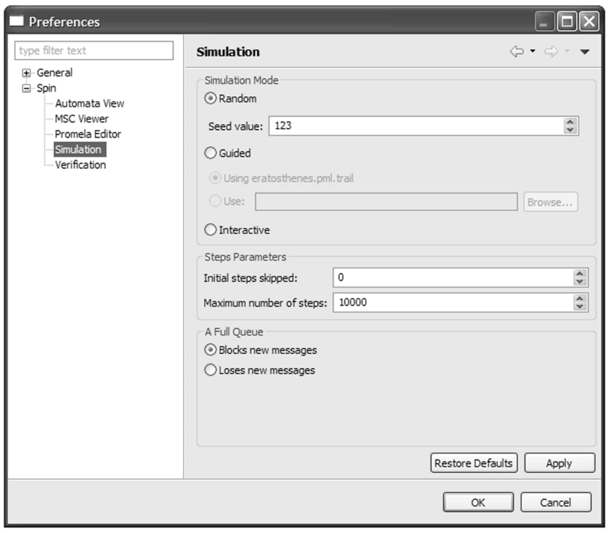
Automata File Format

bmp	Bitmap image file
dot	Plain text graph description language
eps	Encapsulated PostScript
fig	XFIG graphics
gif	Graphics Interchange Format
jpg	Joint Photographic Experts Group Format
[✓ pdf]	Portable Document Format
png	Portable Network Graphics
svgz	Compressed Scalable Vector Graphics
tif	Tagged Image File Format

Automata View Dialog

[✓ Show automata view dialog after graph creation]
Don't show automata view dialog

Figure 4: Automata View preference page



Simulation

Simulation Mode

[✓ Random]
Seed value: 123
Guided
Using eratosthenes.pml.trail
Use:
Interactive

Steps Parameters

Initial steps skipped: 0
Maximum number of steps: 10000

A Full Queue

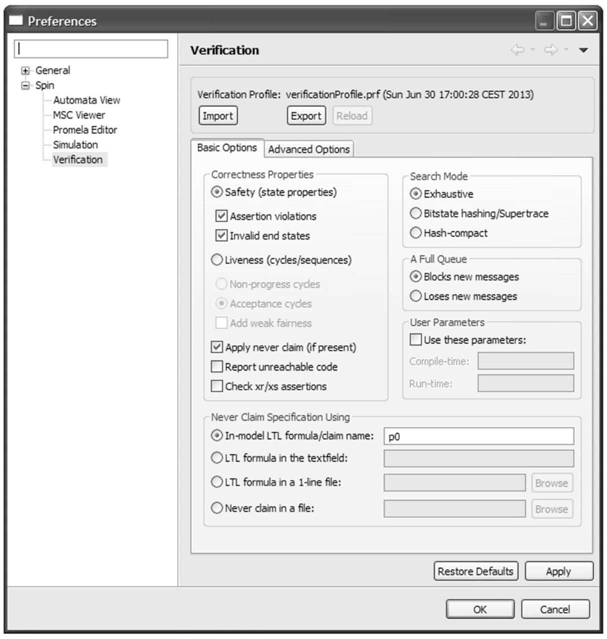
[✓ Blocks new messages]
Loses new messages

Figure 5: Simulation preference page.

(random, guided, or interactive). A seed value can be set for random simulation. Either the default Spin trail file with the extension .trail added to the original Promela source file or any other Spin trail file can be selected for guided simulation. The number of initial steps skipped (default 0), the maximum number of steps (default 10000), and how a full queue is simulated (either blocks or loses new messages), is also selected on this page.

The most complex preference page in SpinRCP is the Verification preference page (Fig. 6 and Fig. 7). In the upper part, the user can export current verification parameters (verification profile) to an xml file and import or reload a previously saved verification profile. Verification options are accessible below in two tabs: Basic Options and Advanced Options.

In the Basic Options tab (Fig. 6), a user can select a correctness property to be proved (either safety or liveness) with several additional options, the search mode (exhaustive, bitstate hashing or hash-compact), how a full queue behaves during verification (either blocks or loses new messages), the explicit use of user-entered compile-time and run-time parameters that supersede the clicked options and the elsewhere entered parameters, and how a never claim (if any) is specified. A never claim can be specified in four different ways. In the first one, which is a default, a never claim or an LTL formula is specified in the model itself. In the text field right to the label the name of the never claim/LTL formula to be checked against the model has to be written. Such an in-model never claim specification has been possible since Spin Version 6. The second method is to enter an LTL formula in the text field. The third one is to enter or select the file name, in which the single-line LTL formula is written. The last way to specify a never claim is to enter or select a file name with a contained never claim.



Verification

Verification Profile: verificationProfile.prf (Sun Jun 30 17:00:28 CEST 2013)

Correctness Properties
- [✓ Safety (state properties)]
  - [✓ Assertion violations]
  - [✓ Invalid end states]
- Liveness (cycles/sequences)
  - Non-progress cycles
  - [✓ Acceptance cycles]
  - Add weak fairness
- [✓ Apply never claim (if present)]
- Report unreachable code
- Check xr/xs assertions

Search Mode
- [✓ Exhaustive]
- Bitstate hashing/Supertrace
- Hash-compact

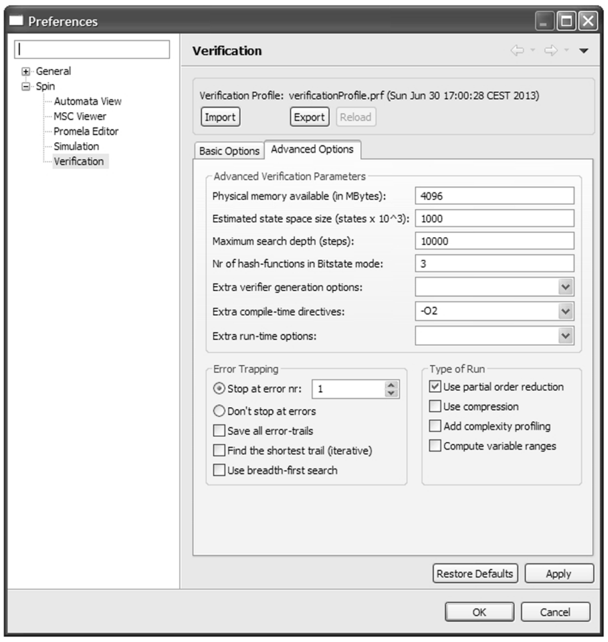
A Full Queue
- [✓ Blocks new messages]
- Loses new messages

User Parameters
- Use these parameters:
- Compile-time:
- Run-time:

Never Claim Specification Using
- [✓ In-model LTL formula/claim name:] p0
- LTL formula in the textfield:
- LTL formula in a 1-line file:
- Never claim in a file:

Figure 6: Basic options on the Verification preference page.

In the Advanced Options tab (Fig. 7) it is possible to enter several advanced verification parameters (the amount of the available physical memory in megabytes, the estimated state space size, the maximum search depth, number of hash-functions in bitstate mode, extra verifier generation options, extra compile-time directives, and extra run-time options) and select some error-trapping and verification run type options.



Verification

Verification Profile: verificationProfile.prf (Sun Jun 30 17:00:28 CEST 2013)

Advanced Verification Parameters

Physical memory available (in MBytes):	4096
Estimated state space size (states x 10^3):	1000
Maximum search depth (steps):	10000
Nr of hash-functions in Bitstate mode:	3
Extra verifier generation options:	
Extra compile-time directives:	-O2
Extra run-time options:	

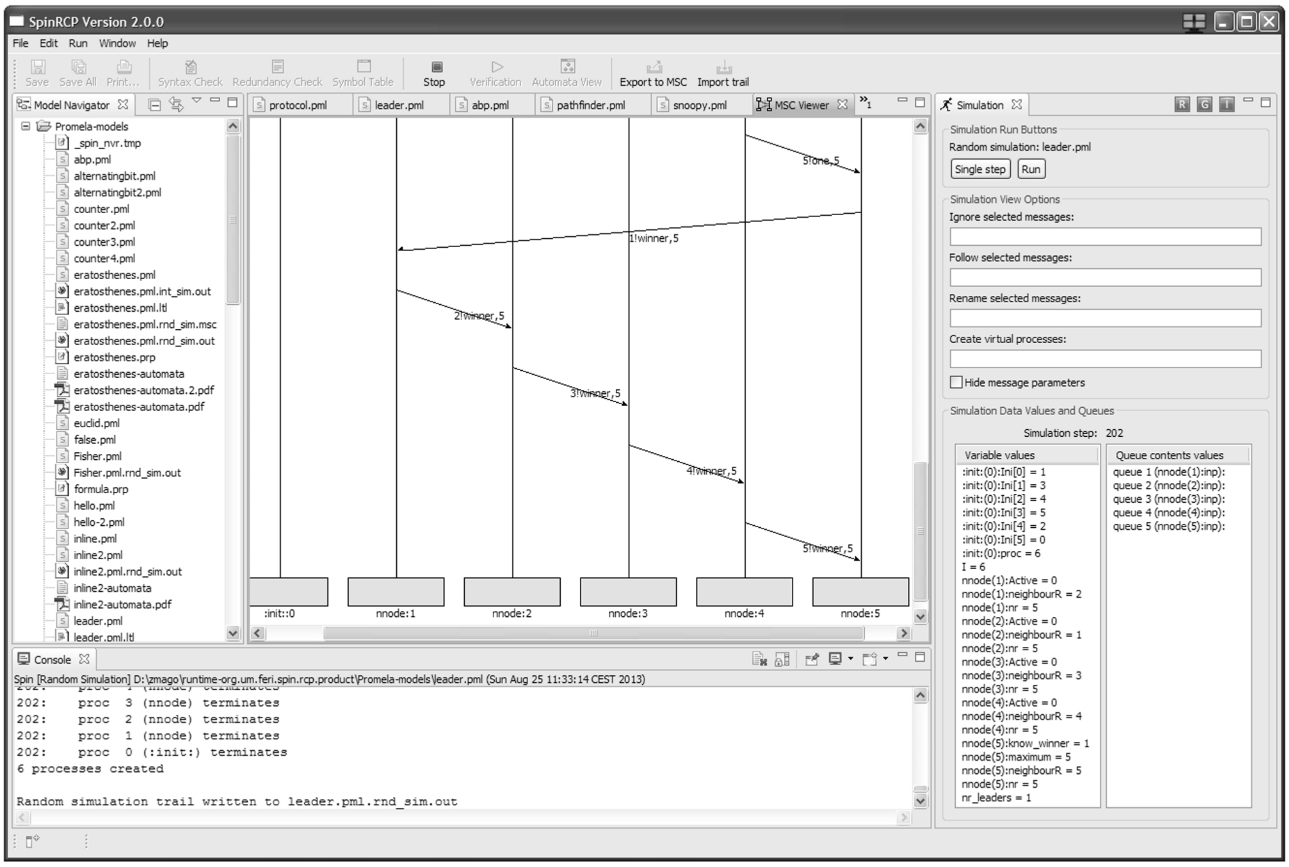
Error Trapping
- [✓ Stop at error nr:] 1
- Don't stop at errors
- Save all error-trails
- Find the shortest trail (iterative)
- Use breadth-first search

Type of Run
- [✓ Use partial order reduction]
- Use compression
- Add complexity profiling
- Compute variable ranges

Figure 7: Advanced options on the Verification preference page.

5.8 MSC Viewer

The MSC Viewer is used for a graphical display of the Spin simulation output trail. Otherwise, as in the case of [10, 11], external (often commercial) tools would be needed to accomplish this (e.g., ObjectGEODE). The MSC Viewer is shown in the central part of Fig. 8. It works in two different modes. In the first one it displays an already generated simulation trail from a file of type out. This kind of MSC display can be achieved either by a double-click on an out file in the Model Navigator view or by a click on the Import trail button in the Tool bar when the Simulation View is active. The second mode of displaying the MSC is "on-the-fly" when the simulation is running. In this mode, two Java threads run in parallel: a Spin simulation thread and an MSC refreshing thread. The simulation thread executes the Spin simulation (random, guided, or interactive). Simulation output is parsed line by line and for each new parsed line the list of created processes and messages that have been sent and received up to this time is updated. In parallel, the MSC refreshing thread is refreshing the MSC displayed in MSC Viewer. If any new process or any new message has been added to the corresponding list since the last screen refresh, the MSC is changed accordingly. The MSC Viewer is implemented using the Graphical Editing Framework (GEF) that provides technology for creating rich graphical editors and views for the Eclipse Workbench UI.


SpinRCP Version 2.0.0
File Edit Run Window Help
Save Save All Print... Syntax Check Redundancy Check Symbol Table Stop Verification Automata View Export to MSC Import trail
Model Navigator
protocol.pml leader.pml abp.pml pathfinder.pml snoopy.pml MSC Viewer
5!one,5
1!winner,5
2!winner,5
3!winner,5
4!winner,5
5!winner,5
:init::0
nnode:1
nnode:2
nnode:3
nnode:4
nnode:5
Simulation
Console

Spin [Random Simulation] D:\zmago\runtime-org.um.feri.spin.rcp.product\Promela-models\leader.pml (Sun Aug 25 11:33:14 CEST 2013)

```
202:    proc  3 (nnode) terminates
202:    proc  2 (nnode) terminates
202:    proc  1 (nnode) terminates
202:    proc  0 (:init:) terminates
6 processes created

Random simulation trail written to leader.pml.rnd_sim.out
```

Simulation Run Buttons
Random simulation: leader.pml

Simulation View Options
Ignore selected messages:
Follow selected messages:
Rename selected messages:
Create virtual processes:
☐ Hide message parameters

Simulation Data Values and Queues
Simulation step: 202

Variable values	Queue contents values
:init:(0):Ini[0] = 1	queue 1 (nnode(1):inp):
:init:(0):Ini[1] = 3	queue 2 (nnode(2):inp):
:init:(0):Ini[2] = 4	queue 3 (nnode(3):inp):
:init:(0):Ini[3] = 5	queue 4 (nnode(4):inp):
:init:(0):Ini[4] = 2	queue 5 (nnode(5):inp):
:init:(0):Ini[5] = 0	
:init:(0):proc = 6	
I = 6	
nnode(1):Active = 0	
nnode(1):neighbourR = 2	
nnode(1):nr = 5	
nnode(2):Active = 0	
nnode(2):neighbourR = 1	
nnode(2):nr = 5	
nnode(3):Active = 0	
nnode(3):neighbourR = 3	
nnode(3):nr = 5	
nnode(4):Active = 0	
nnode(4):neighbourR = 4	
nnode(4):nr = 5	
nnode(5):know_winner = 1	
nnode(5):maximum = 5	
nnode(5):neighbourR = 5	
nnode(5):nr = 5	
nr_leaders = 1	

Figure 8: SpinRCP Workbench during simulation.

5.9 Simulation View

The Simulation View is shown after the Simulation button in the Tool bar has been pressed and the Simulation preference page confirmed (right side of Fig. 8). At the top of the view there is a label showing the previously-selected simulation type including the Promela model filename and two simulation buttons, Single Step and Run. By clicking the Single Step button, MSC is being drawn message by message (if any message is sent and received in the model at all). Clicking on the Run button periodically updates the MSC each time the MSC refresh interval expires as given on the MSC Viewer preference page.

Several options to adapt the display of MSCs are available below the simulation buttons. The user can do the following:

- select a subset of messages that will be ignored and not displayed in the MSC diagram,
- select a subset of messages that will be displayed in the MSC diagram,
- rename selected messages,
- join two or more processes into a new virtual process, and
- select whether to show or hide message parameters.

The first two options are very useful if we have to explore simulation traces of large models with many processes and a large number of messages, and can thus concentrate only on those that we are interested in. Selected messages are entered using their space-separated IDs. The need for renaming of messages appears when our sdl2pml tool [8, 9] has extracted a very large Promela model from an SDL code of a real product. Since Promela allows a maximum of 255 different message types, the sdl2pml tool presents messages using integers. It is very difficult to track a simulation trail if messages are represented by numbers instead of having sensible names. The renaming of messages helps, in that the MSC is more understandable. For example, to rename a message with ID 1 to one, the following command has to be entered: 1>one. To rename more messages within a single command, a space has to be entered for separating individual renamings. When we want to make an abstraction of the model, we can use a powerful feature of joining a group of processes into a virtual process. For example, the 1,2>onetwo command has to be entered in order to join processes with IDs 1 and 2 into a virtual process called onetwo. More virtual processes can be created within a single command using a space separator. The abstraction by joining processes results in a smaller number of processes in the MSC as well as in a smaller number of messages displayed, as all messages within each group of joined processes are internal for the virtual process and therefore not shown. Parameter hiding is especially useful when reviewing simulation trails of real systems, where messages often contain many parameters that lead to less transparent diagrams.

The same set of simulation view options is also available in the Spin Trail to MSC View, which is intended for converting a Spin simulation output trail to the standard MSC text format according to [13] in the same manner that the Export to MSC command does. In addition, Spin Trail to MSC View displays a list of created processes and messages transmitted between them during the simulation run.

During the simulation run the variable values and queue contents values are updated in two separate tables at the bottom of the Simulation View. The current simulation step number is shown at the top of the tables.

6 Leader election example

Let us demonstrate some features of SpinRCP by considering a standard algorithm for leader election in a unidirectional ring. An efficient algorithm for solving this problem was published in [19]. The Promela model of this algorithm is taken from Spin Version 6 distribution.

The leader election algorithm, when given a circular arrangement of *N* uniquely numbered processes in a unidirectional ring, determines the maximum number in a distributive manner. Communication occurs only between neighbours around the ring. All processes have the same program. They differ only by having distinct numbers (known only to the owners) in their local memory.

We suppose that *N* = 5. The Promela model contains two proctype definitions: init and nnode. The init process first assigns a unique number for each of the five processes using non-deterministic choices. Then it creates five instances of a nnode process and assigns them their numbers. Next, the five nnode processes start to send and receive messages around the ring and process them according to the algorithm. A process terminates when it recognizes whether it is a leader (has the greatest number in a ring) or not.

A global variable nr_leaders is defined and initialised to zero in line 26. In lines 28 through to 31, four required properties for the algorithm are specified with the following LTL formulas:

```
p0: <> (nr_leaders > 0)
p1: <>[] (nr_leaders == 1)
p2: [] (nr_leaders == 0 U nr_leaders == 1)
p3: ![] (nr_leaders == 0)
```

Such in-model specification of LTL properties has been supported since Spin Version 6. They state "positive" properties. Spin performs the negation automatically.

After a successful syntax check, eventual redundancy check and/or listing of a model symbol table, it is useful to become more acquainted with the model. For this purpose we can first generate and display a graphical representation of a state transition system (an automaton) for each proctype and never claim in the model. By clicking the Automata View button in the Tool bar, the Automata View preference page (Fig. 4) opens and gives us the choice of selecting the type of files that will contain the automata. Let us suppose that we select the pdf file type (as in Fig. 4). Then the following sequence of commands is executed:

```
spin –o3 –a leader.pml
gcc-4 –o pan pan.c
pan –D | dot>leader-automata
dot –O –Tpdf leader-automata
```

The first command generates the verifier source code for the model leader.pml without statement merging, the second one compiles it, the third one writes state tables in dot-format to leader-automata file, and the last one creates a pdf file with the automaton for each proctype and never claim. Now a new selection dialogue is open and we can select, which automaton we want so see. Each selected automaton is opened in a system application that is assigned for a given file type. Let us suppose that we want to see the automata for nnode, p0, p1, and p2. To save space, all of them are placed together in Fig. 9.

In order to deepen understanding of the model and perhaps to find some early (simple) errors before verification, it is wise to perform random and/or interactive simulation. In random simulation, Spin decides by itself, as to which one of the executable statements will be chosen at the points of non-deterministic selections. In interactive simulation, these decisions have to be made by a user. Fig. 8 shows the content of the console, the MSC Editor window, and the Simulation View on completion of the random simulation of the leader.pml model. At the bottom of the *Variable values* table in the Simulation View it is evident that the nr_leaders variable is given the final value 1.

We may want to prove several interesting properties about this algorithm but one of the most important

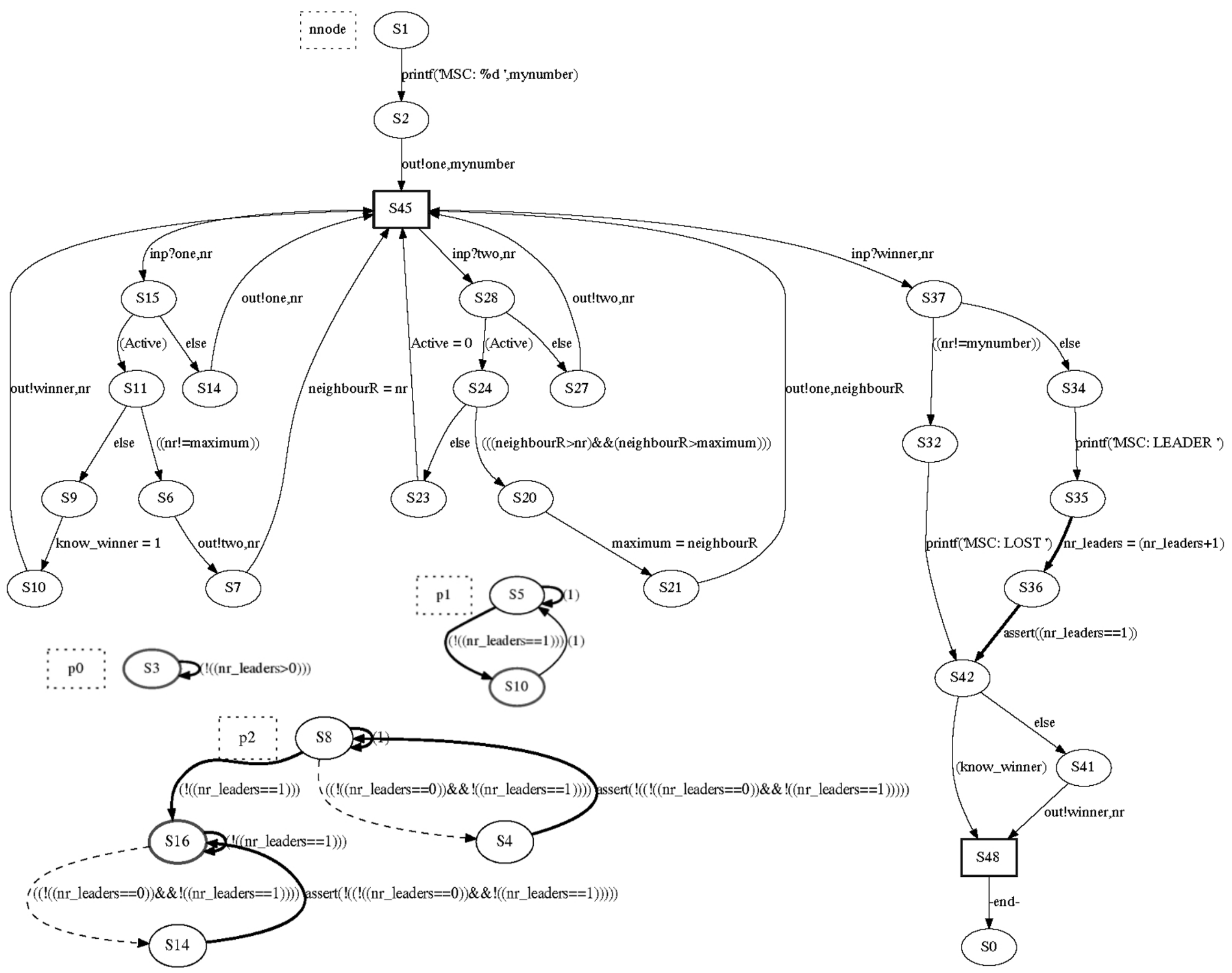


Figure 9: Automata for nnode, p0, p1, and p2.

seems to be the property that under no conditions should it be possible that more than one process declare to be the ring leader.

Firstly, let us check p0, i.e., that eventually the number of leaders is greater than 0. By clicking the Verification tool button, the Verification preference page (Fig. 6) is displayed. Since p0 specifies a liveness property, we must select the *Liveness* and *Acceptance cycles* radio buttons. Next, we select *Apply never claim* and as a manner of never claim specification select *In-model LTL formula/claim name*. Finally, we enter the name of the in-model LTL formula, p0, in the text field to the right of the label. As a result of verification, Spin returns that no error is found (i.e., the model fulfils the property p0) (see the Console in Fig. 2). But it is as yet unclear whether the algorithm will eventually always give one single leader. In order to verify this, we check if the model fulfils p1, i.e., if eventually the number of leaders will always be 1. Therefore, this time we enter p1 as the LTL formula name and run the verification again. Spin finds no errors, which means that the model fulfils p1. Now the only doubt about the correctness of the algorithm that still exists is whether the number of leaders goes from 0 to 1 directly with no intermediate numbers greater than 1. In order to find out the answer to this question, we check whether the system fulfils p2 as well, i.e., the number of leaders is always 0 until the number of leaders is 1. The new verification run succeeds and thus p2 is verified. p3 specifies that the number of leaders is not always zero. This means that eventually the number of leaders is not zero. Since this number cannot be negative, p3 means exactly the same as p0, and is thus already verified.

7 Conclusions

Using the Spin model checker on a huge model, automatically extracted from the SDL code of a real telecommunication industry product, encouraged us to develop an integrated development environment for Spin model checking called SpinRCP. SpinRCP is based on the Rich Client Platform technology. It is written in

Java as an Eclipse plug-in and then exported together with many other plug-ins as an Eclipse product. Therefore, it runs as a stand-alone RCP application on any platform without the need for an installed Eclipse but nevertheless has many of the useful Eclipse functionalities.

The whole application consists of 92 plug-ins. The Java source code for our plug-in called org.um.feri.spin.rcp is contained in 19 Java packages with a total of 84 files defining Java classes. The total amount of our source code is around 16,800 lines of Java code. The help contents for SpinRCP is implemented in a separate plug-in that contains more than 60 html files with descriptions of individual help topics and many xml configuration files. Currently, SpinRCP runs on 32- and 64-bit Windows operating systems. Platforms with other operating systems will be supported later. Once the website for SpinRCP is ready, it will be publicly announced.

Amongst the more important features of SpinRCP are the following ones: a user-friendly Promela editor with syntax colouring, code folding, keyword autocompletion, and syntax error marking, running Spin verification, random, guided, and interactive simulation, graphical MSC viewing, abstracting MSCs by joining some processes into an abstract process, conversion of Spin simulation output trail to a standard text file, which is readable by professional MSC viewers, displaying graphical automata representation of proctype definitions and never claims in a model.

There are still a lot of ideas for improvements and new features. Let us mention just some of them: better Spin simulation output filtering, stepping back in time during single step simulation, indication of the statement that is currently executed during a simulation run in the Promela source file, display of a process creation in the MSC Viewer, generation of state tables for proctype definitions and never claims, cleanup of temporary files, verification management, swarm support [15] for distributing a model checking task to more CPU cores or to a cloud of workstations, etc.

APPENDIX A

```
/*  Dolev, Klawe & Rodeh for leader election in unidirectional ring
 *  `An O(n log n) unidirectional distributed algorithm for extrema
 *  finding in a circle,'  J. of Algs, Vol 3. (1982), pp. 245-260

 *  Randomized initialization added -- Feb 17, 1997
 */

/* sample properties:
 *  <>elected
 *  <>[]oneLeader
 *  []  (noLeader U oneLeader)
 *  ![]  noLeader
 *
 *  ltl format specifies positive properties
 *  that should be satisfied -- spin will
 *  look for counter-examples to these properties
 *  verify as:
 *  spin -a leader.pml
 *  cc -o pan pan.c
 *  ./pan -N p0
 *  ./pan -N p1
 *  ./pan -N p2
 *  ./pan -N p3
 */

byte nr_leaders = 0;

ltl p0  { <> (nr_leaders > 0) }
ltl p1  { <>[] (nr_leaders == 1) }
ltl p2  { [] (nr_leaders == 0 U nr_leaders == 1) }
ltl p3  { ![] (nr_leaders == 0) }

#define N 5 /* number of processes in the ring */
#define L 10 /* 2xN */
byte I;

mtype = { one, two, winner };
chan q[N] = [L] of { mtype, byte};

proctype nnode (chan inp, out; byte mynumber)
{ bit Active = 1, know_winner = 0;
  byte nr, maximum = mynumber, neighbourR;

  xr inp;
  xs out;

  printf(„MSC: %d\n“, mynumber);
  out!one(mynumber);
end:  do
    :: inp?one(nr) ->
      if
      :: Active ->
        if
        :: nr != maximum ->
          out!two(nr);
          neighbourR = nr
        :: else ->
          know_winner = 1;
          out!winner,nr;
        fi
      :: else ->
        out!one(nr)
      fi

    :: inp?two(nr) ->
      if
      :: Active ->
        if
        :: neighbourR > nr && neighbourR > maximum ->
          maximum = neighbourR;
    out!one(neighbourR)
        :: else ->
          Active = 0
        fi
      :: else ->
        out!two(nr)
      fi
    :: inp?winner,nr ->
      if
      :: nr != mynumber ->
        printf(„MSC: LOST\n");
      :: else ->
        printf(„MSC: LEADER\n");
        nr_leaders++;
        assert(nr_leaders == 1)
      fi;
      if
      :: know_winner
      :: else -> out!winner,nr
      fi;
      break
  od
}

init {
  byte proc;
  byte Ini[6];/* N<=6 randomize the process numbers */
  atomic {

    I = 1; /* pick a number to be assigned 1..N */
    do
    :: I <= N ->
      if  /* non-deterministic choice */
      :: Ini[0] == 0 && N >= 1 -> Ini[0] = I
      :: Ini[1] == 0 && N >= 2 -> Ini[1] = I
      :: Ini[2] == 0 && N >= 3 -> Ini[2] = I
      :: Ini[3] == 0 && N >= 4 -> Ini[3] = I
      :: Ini[4] == 0 && N >= 5 -> Ini[4] = I
      :: Ini[5] == 0 && N >= 6 -> Ini[5] = I /* works for up to N=6 */
      fi;
      I++
    :: I > N -> /* assigned all numbers 1..N */
      break
    od;

    proc = 1;
    do
    :: proc <= N ->
      run nnode (q[proc-1], q[proc%N], Ini[proc-1]);
      proc++
    :: proc > N ->
      break
    od
  }
}
```

Acknowledgments

This work was partially supported by the Slovenian Research Agency within the research programme Advanced Methods of Interaction in Telecommunication.

References

1. G.J. Holzmann, "The Spin Model Checker: Primer and Reference Manual", Addison-Wesley, 2004.
2. G.J. Holzmann, "The Model Checker SPIN", IEEE Transaction on Software Engineering, vol. 23, no. 5, pp. 279-295, 1997.
3. C. Baier, J.-P. Katoen, "Principles of Model Checking", The MIT Press, 2008.
4. T.C. Ruys, "SPIN Beginners' Tutorial", SPIN 2002 Workshop, Grenoble, France, April 11, 2002, electronic file available at http://spinroot.com/spin/Man/.
5. M. Ben-Ari, "Principles of the Spin Model Checker", Springer, 2008.
6. V.J. Koskinen, J. Plosila, Applications for the Spin Model Checker – A Survey, Technical Report 782, Turku Centre for Computer Science, Finland, September 2006.
7. M. Ben-Ari, jSpin – "Java GUI for SPIN: User's Guide", Version 5.0, electronic file available at http://code.google.com/p/jspin/downloads/list, 2010.
8. T. Kovše, "Integration of Spin Tool into Development Environment Eclipse", Diploma Work (in Slovene), Faculty of Electrical Engineering, Slovenia, 2008.
9. T. Kovše, B. Vlaovič, A. Vreže, Z. Brezočnik, "Eclipse plug-in for spin and st2msc tools - tool presentation", Lecture Notes in Computer Science, vol. 5578, pp. 143-147, 2009.
10. B. Vlaovič, A. Vreže, Z. Brezočnik, T. Kapus, "Automated generation of Promela model from SDL specification", Computer Standards and Interfaces, vol. 29, no. 4, pp. 449-461, 2007.
11. A. Vreže, B. Vlaovič, Z. Brezočnik, "Sdl2pml - tool for automated generation of Promela model from

SDL specification", Computer Standards and Interfaces, vol. 31, no. 4, pp. 779-786, 2009.

12. T. Kovše, B. Vlaovič, A. Vreže, Z. Brezočnik, "Spin Trail to a Message Sequence Chart Conversion Tool", ConTEL 2009, I. Podnar Žarko, B. Vrdoljak (Eds.), Proceedings of the 10th International Conference on Telecommunications, Zagreb, Croatia, June 8-10, 2009.
13. "Message Sequence Chart (MSC): Recommendation ITU-T Z.120", International Telecommunication Union, 2011, electronic file available at http://www.itu.int/rec/T-REC-Z.120-201102-I/en.
14. M.J. Hornos, J.C. Augusto, "Installation Process and Main Functionalities of the Spin Model Checker", electronic file available at http://digibug.ugr.es/handle/10481/19601, 2012.
15. G.J. Holzmann, R. Joshi, A. Groce, "Swarm Verification Techniques", IEEE transactions on Software Engineering, vol. 37, no. 6, pp. 845-857, 2011.
16. B. de Vos, L.C.L Kats, C. Pronk, "EpiSpin: An Eclipse Plug-In for Promela/Spin Using Spoofax", In: A. Groce, M. Musuvathi (Eds.): SPIN 2011, LNCS 6823, Springer-Verlag, pp. 177-182, 2011.
17. T. Kovše, "Environment for formal verification of safety-critical systems", Master Thesis (in Slovene), Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia, 2011.
18. Graphviz – Graph Visualization Software, electronic file available at http://graphviz.org.
19. D. Dolev, M. Klave, M. Rodeh, "An O(n log n) Unidirectional Algorithm for Extrema Finding in a Circle", Journal of Algorithms, vol. 3, pp. 245-260, 1982.

Arrived: 01. 10. 2013
Accepted: 08. 11. 2013