Unit Testing for SPIN: runspin and parsepan

Theo C. Ruys
RUwise – The Netherlands
theo.ruys@gmail.com

ABSTRACT

This paper presents *runspin* and *parsepan*, two utilities to ease the verification process with the SPIN model checker. *runspin* allows the management of verification configurations within PROMELA models and takes care of the automatic verification. Moreover, *runspin* adds essential data to the verification report. *parsepan* is used to selectively retrieve information from verification reports. Used together the two tools can act as unit testing engine for PROMELA models.

Categories and Subject Descriptors

[**Software and its engineering**]: Software functional properties—*Formal methods, Model checking*

General Terms

Management, Verification Trajectory

1. INTRODUCTION

It is indisputable that the verification results obtained using a verification tool should always be reproducible. Without tool support, the verification engineer has to resort to general engineering practices and record all verification activities into a (digital) log-book. Consequently, the quality of the verification process depends on the accuracy of the validation engineer. The careful recording of information on the different models during the verification phase becomes even more indispensable when errors are found in one of the verification models. Apart from the fact that the erroneous models have to be corrected and reverified, all models that have been verified previously and which are affected by the error should be reverified as well [7].

State-of-the-art verification tools like SPIN [2, 12] provide the user with an extensive set of options and directives to optimize and tune the functionality and performance of the verification run. To reproduce a verification run of a model checker *all* these options should be recorded. Preferably together with the verification model and the verification results. Furthermore, the nature of verification tools is that they either need a lot of processing time or need a lot of memory or even both. Statistics on such properties of verification runs are valuable attributes of the verification trajectory [7], especially when benchmarking verification tools.

Over the last fifteen years we have been involved in several verification projects with SPIN (e.g., [5, 6, 8, 11]), where we had to deal with many different PROMELA models and extensive sets of properties to verify. This usually resulted in a huge collection of verification reports. Collecting the interesting data from these verification reports turned out to be tedious and error-prone.

In this paper we present two complementary tools to assist the verification engineer when using SPIN: *runspin* is a script to automate the complete verification of a PROMELA model, and *parsepan* can be used to retrieve specific information from SPIN's verification reports. Both tools have been developed along several verification projects and have evolved from in-house scripts to mature utilities, and should be useful for (advanced) users of SPIN. Both *runspin* and *parsepan* are open-source and available from [10] and [9], respectively.

In Sec. 2 we explain how *runspin* and *parsepan* are typically used. In Sec. 3 we dive deeper into the features of *runspin* and Sec. 4 discusses *parsepan* in more detail. Sec. 5 ends the paper with a short summary and some directions for future work.

2. USAGE

To control the verification process with SPIN, we would typically use the make utility [1] in combination with small write-once-use-once scripts . The *runspin/parsepan* toolset lifts much of the burden of administrating and controling the verification trajectory from the verification engineer, though.

The *runspin/parsepan* toolset has the following features:

- *grouping*: compilation- and verification options can be stored within the Promela file,
- *unit-testing*: after modifying the Promela model, all properties get (re)verified automatically,
- *enhanced reporting*: compilation- and verification options are included in the verification report, and

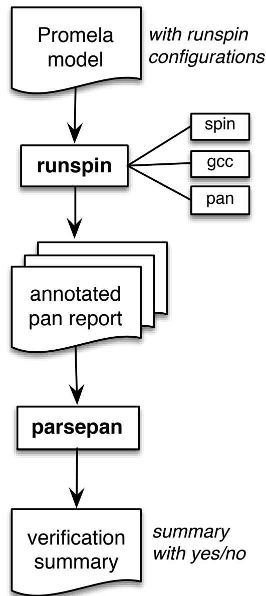

Figure 1: Unit-testing.
`runspin -a -p model.prom | parsepan -s -t`

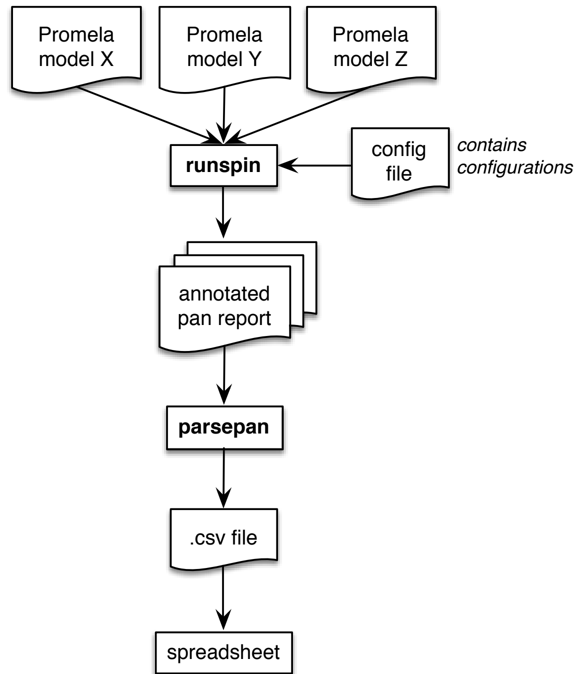

Figure 2: Analysing verification results.
`runspin -f ex.cfg -n cf *.prom | parsepan -d 'c' > cf.csv`

- *retrieval*: straightforward selection of individual data values of the verification report to allow further analysis.

Configuration. A complete set of commands and options to conduct a single verification run is called a configuration. For Spin a configuration consist of three parts: the options for `spin`, the options for the C compiler (`gcc`) and the options for the `pan` verifier. A configuration usually corresponds to a correctness property that has to be verified.

To illustrate how *runspin* and *parsepan* are typically used, we discuss two examples. In Fig. 1, *runspin* and *parsepan* are used in a unit test setting. The Promela model contains several configurations that have to be verified. After modifying the Promela model, *runspin* ensures that all properties are (re)verified. The verification output is passed to *parsepan* which shows a summary of the all verification reports.

In Fig. 2, the verification results of several Promela (benchmark) models are compared. The configurations are read from a configuration file and *runspin* takes care of verifying all Promela models against all configurations. The verification results are passed to *parsepan* which transforms the output to a comma separated values (.csv) file. This file is then imported into a spreadsheet, where further analysis can take place.

To illustrate the use of *runspin* and *parsepan* we introduce a small example. Fig. 3 shows a Promela model of Peterson's algorithm [4], an algorithm for mutual exclusion. The critical section is modelled by the variable `mutex`. The mutual exclusion algorithm should satisfy at least three correctness requirements: freedom from deadlock, mutual exclusion: only one process can be in the critical section at the same time, and freedom from starvation: if any process tries to enter its critical section, then that process must eventually succeed.

3. RUNSPIN

runspin is a bash script (400 lines of code) that automates the complete verification of a Promela model with Spin and adds extra information to `pan`'s verification report. First, *runspin* will retrieve the configuration to verify the Promela model. Then *runspin* invokes `spin` to generate the verification program `pan.c`. Subsequently, *runspin* invokes `gcc` to compile `pan.c`. Finally *runspin* executes the compiled `pan` verifier. Apart from automating the verification process, *runspin* adds valuable extra information to `pan`'s verification report, e.g., the Promela file name, the name of the configuration, the exact commands used, date of verification, unix time, etc. Fig. 4 shows (the top part of) the verification report after verifying the configuration named `deadlock` with *runspin*:

```
runspin -n deadlock -p peterson.prom
```

The lines starting with `>` have been added by *runspin*. We have omitted `pan`'s standard part of the verification report.

Configurations are typically stored within the Promela file as a single line comment. See the first three lines of Fig. 3. The string `runspin` identifies a configuration. The string after the underscore is the name of the configuration. Each command of the configuration is prefixed by `%` (to mimick a shell prompt) and the last command should also be terminated by `%`. Configurations can also be specified in a separate file. The name of the configuration is then enclosed in

```
/* runspin_deadlock: %spin -a        %gcc -o pan -DSAFETY pan.c       %./pan         % */
/* runspin_mutex:    %spin -DLTL -a  %gcc -o pan pan.c                %./pan -a      % */
/* runspin_progress: %spin -a        %gcc -o pan -DNP -DNOCLAIM pan.c %./pan -l -f   % */

bit  q1, q2;
byte turn;
byte mutex;

active proctype P1() {
    do
    ::  q1=true; turn=1;
        !q2 || (turn == 2);
progress:
        mutex++; mutex--;   /* critical section */
        q1=false;
    od;
}

active proctype P2() {
    do
    ::  q2=true; turn=2;
        !q1 || (turn == 1);
        mutex++; mutex--;   /* critical section */
        q2=false;
    od;
}

#ifdef LTL
ltl mutex { [] (mutex!=2) }
#endif
```

Figure 3: Peterson's algorithm [4] as PROMELA model.

```
> promela file    : peterson.prom
> date            : 16-Apr-2014 16:15:10
> spin version    : Spin Version 6.2.7 -- 2 March 2014
> gcc version     : 4.2.1
> runspin command : runspin -n deadlock -p peterson.prom
> config source   : commands retrieved from promela file (option -p)
> config name     : deadlock
> spin command    : spin -a
> gcc command     : gcc -o pan -DSAFETY pan.c
> pan command     : ./pan

(Spin Version 6.2.7 -- 2 March 2014)
    + Partial Order Reduction
...
```

Figure 4: Verification report: information added by *runspin*.

```
$runspin -a -p peterson.prom | parsepan --summary --header
rs_promela_file     rs_config_name  errors      states_stored   state_vector
peterson.prom       deadlock        0           36              20
peterson.prom       mutex           0           36              36
peterson.prom       progress        0           69              36
```

Figure 5: Output of a unit-test run.

```
[deadlock]       // invalid-end states
spin -a
gcc -o pan -DSAFETY pan.c
./pan

[mutex]          // use LTL to check mutex
spin -DLTL -a
gcc -o pan pan.c
./pan -a

[progress]       // checking for starvation
spin -a
gcc -o pan -DNP -DNOCLAIM pan.c
./pan -l -f
```

Figure 6: Example of a configuration file.

brackets and the three series of commands are written on three separate lines. Java-style one-line comments (//) can be used to document the configuration file. Fig. 6 shows the same configurations as specified in Fig. 3, but now as a configuration file. In practice, however, a configuration file is typically used when several PROMELA models are verified using the same configuration. This is useful for benchmark studies, when several different PROMELA model the same system. We used this extensively for [8].

4. PARSEPAN

parsepan is a Python script (400 lines of code of which 200 lines are pattern definitions) that parses a SPIN verification report and exports the (key,value) pairs of the verification report as character delimited values. The format of *parsepan*'s ouptut can be customized through several options. We deliberately let *parsepan* export delimited output only: it is straightforward to transform the output of *parsepan* to other formats (e.g., LaTeX, XML).

Apart from `pan`'s standard verification report, *parsepan* recognizes the output added by the *runspin* script. *parsepan* recognizes more than 120 different data items in a verification report. By default, *parsepan* outputs all data items that it recognizes. Naturally, *parsepan* has several options to control what data values are reported and in what order they are presented.

Fig. 5 shows an example of running *runspin* and *parsepan* as unit test. The *runspin* command ensures that all configurations in `peterson.prom` are verified. The verification reports are passed to *parsepan* which presents a summary (i.e., the most important data values, including the number of errors) of the verifications together with a header row of the key names of the data values. Data values are separated by tabs. Note that the key names of data items added by *runspin* have a `rs_` prefix.

5. CONCLUSIONS

In this paper we have presented *runspin* and *parsepan*, two utilities to ease the verification process with SPIN. *runspin* allows the management of verification configurations within PROMELA models and takes care of the automatic verification. Furthermore, *runspin* adds essential data to the verification report. The *parsepan* utility can be used to selectively retrieve information from verification reports. Used together the two tools can act as unit testing engine for PROMELA models.

With new verification projects ahead, the development of both tools will further evolve. Currently *runspin* and *parsepan* only support the SPIN model checker. In the near future we will add support for the swarm tool [3]. For a new major release of *runspin* we plan to generalize *runspin* to a tool which can automate the validation process of other validation tools as well.

6. REFERENCES

[1] S. I. Feldman. Make – A Program for Maintaining Computer Programs. *Software – Practice and Experience*, 9(3):255–265, Mar. 1979.

[2] G. J. Holzman. *The* SPIN *Model Checker – Primer and Reference Manual*. Addison-Wesley, Boston, USA, 2003.

[3] G. J. Holzmann, R. Joshi, and A. Groce. Tackling Large Verification Problems with the Swarm Tool. In K. Havelund, R. Majumdar, and J. Palsberg, editors, *Proc. of SPIN 2008*, LNCS 5156, pages 134–143. Springer, 2008.

[4] G. L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, 1981.

[5] T. C. Ruys. *Towards Effective Model Checking*. PhD thesis, Formal Methods and Tools group, Department of Computer Science, University of Twente, Enschede, The Netherlands, March 2001.

[6] T. C. Ruys. Optimal Scheduling Using Branch and Bound with SPIN 4.0. In T. Ball and S. K. Rajamani, editors, *Model Checking Software - Proc. of the 10th Int. SPIN Workshop (SPIN 2003), Portland, OR, USA, May 9-10, 2003*, LNCS 2648, pages 1–17. Springer, 2003.

[7] T. C. Ruys and E. Brinksma. Managing the Verification Trajectory. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 4(2):246–259, 2003.

[8] T. C. Ruys and P. Kars. Gossiping Girls Are All Alike. In A. F. Donaldson and D. Parker, editors, *Proc. of SPIN 2012*, LNCS 7385, pages 117–136. Springer, 2012.

[9] GitHub: repository for parsepan. https://github.com/tcruys/parsepan.

[10] GitHub: repository for runspin. https://github.com/tcruys/runspin.

[11] Rigorous Examination of Reactive Systems (RERS) Challenge. http://www.rers-challenge.org.

[12] The SPIN Model Checker. http://spinroot.com/.