LTL Model Checking under Fairness in ProB

Ivaylo Dobrikov, Michael Leuschel, and Daniel Plagge

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{dobrikov,leuschel,plagge}@cs.uni-duesseldorf.de

Abstract. Model checking of liveness properties often results in unrealistic, unfair infinite behaviors as counterexamples. Fairness is a notion where the search is constrained to infinite paths that do not ignore infinitely the execution of a set of enabled actions. In this work we present an implementation for efficient checking of LTL formulas under strong and weak fairness in ProB, available for model checking B, Event-B, Z, CSP and CSP‖B models. The fairness checking algorithm can cope with both weak and strong fairness conditions, where the respective fairness conditions can be joined by means of the logical operators for conjunction and disjunction, which makes setting up and checking fairness to a property more flexible. We evaluate the implementation on various CSP models and compare it to the fairness implementation of the PAT tool.

1 Introduction and Motivation

Many system requirements can be readily specified in temporal logic such as the linear-time temporal logic (LTL). Subsequently, using an LTL model checker one can check automatically the property specified in LTL on the respective finite state model. There are two general approaches for developing an LTL model checker: the tableau approach [10] and the automata-theoretic approach [14].

The ProB LTL model checker, introduced in [11], follows the tableau approach from [10] and can check properties specified in LTL$^{[e]}$[11], an extended version of LTL providing also support for transition propositions. The algorithm presented in [11] can cope with deadlock states and partially explored state spaces. The LTL search algorithm of ProB is implemented in C using a callback mechanism for exploring the states and evaluating the atomic propositions in SICStus Prolog.

Adding fairness constraints to liveness properties is sometimes necessary in order to exclude unreasonable behaviors of the model and to direct the search for counterexamples on "fair" paths only. The fact that the ProB LTL model checker can deal with transition propositions using LTL$^{[e]}$ enables the user to easily express the fairness conditions as an LTL$^{[e]}$ formula [15]. That is, fairness constraints *fair* can be added as a premise to a liveness property f by means of implication. Then, one can check "$fair \Rightarrow f$" in order to restrict the search for fault system behaviors on paths fair in regard to the imposed fairness constraints *fair*. However, setting fairness constraints to an LTL$^{[e]}$ formula via re-formulating the formula causes an exponential growth of the search graph and on that account is considered to be in most cases a very inefficient approach.

In this work we briefly describe the implementation of the fairness algorithm in ProB's LTL model checker [11] and explain how one can flexibly impose fairness conditions. Additionally, we discuss the enhancements of the LTL model checking process in ProB and evaluate the fairness implementation by comparing ProB and PAT on various CSP specifications.

2 Preliminaries

Linear time properties that require some progress in the system are called *liveness* properties. Intuitively, liveness properties state that "something good" will happen in the future [8]. Liveness properties are violated by infinite computations comprising a bad cycle for the property.

The LTL model checker of ProB uses a tableau approach for checking whether an LTL$^{[e]}$ formula is satisfied or violated by a model. In general, the model checker algorithm searches for a strongly connected component (SCC) with certain properties, referred also as *self-fulfilling* SCC [10]. Such an SCC contains a (bad) cycle that represents a violation of the liveness property.

Fairness is used to rule out bad behaviors that may be considered as unrealistic by the developer of the formal model. There are different variants of fairness in terms of at which granularity level of the system are imposed: *action-based* [7], *state-based* fairness [4], *process* fairness [5], etc. In this work we concentrate on *action-based* fairness and more particularly, on *weak* and *strong* fairness, notions often used in verification of many applied systems [2], [15].

An infinite computation is weakly fair with respect to an action a when: if a is continuously enabled from some point, then a is executed infinitely often. Further, an infinite computation is said to be strongly fair with respect to an action a when: if a is enabled infinitely many times, then it is executed infinitely often. In LTL$^{[e]}$ the fairness conditions can be imposed by means of the execution operator $[\cdot]$ and the derived LTL operators G (globally) and F (eventually). If, for example, a search for a counterexample for some LTL formula f should be constrained on infinite paths that are weakly fair with respect to some action a, then one can re-formulate f as follows:

$$(FG\ e(a) \Rightarrow GF\ [a]) \Rightarrow f.$$

Similarly, one can constrain the search to computations violating the property f which are strongly fair in regard to some event a by re-formulating f as follows:

$$(GF\ e(a) \Rightarrow GF[a]) \Rightarrow f.$$

In both formulae $e(a)$ is an atomic proposition stating that a is enabled at the currently processed state.

3 Fairness Algorithm and Implementation

Given a model M and an LTL$^{[e]}$ formula f, the ProB LTL model checker checks $M \vDash f$ by searching for self-fulfilling strongly connected components (SCCs) that

can be reached from some initial state of M. In case that such a self-fulfilling SCC is found the model checker will return a counterexample for f. Otherwise, if no self-fulfilling SCC is discovered, we have proven that $M \vDash f$. The search for SCCs in the ProB LTL$^{[e]}$ model checker is based on the Tarjan's algorithm [13].

We extended the search algorithm of the LTL model checker [11] for supporting fairness checking separately, i.e. not adding the fairness constraints by encoding them as a premise to the original LTL$^{[e]}$ formula. In general, the idea of our fairness implementation is to check if each found self-fulfilling SCC C satisfies the imposed fairness conditions. If C is unfair with respect to the fairness constraints, then the model checker declines C as a possible counterexample for f and continues the search for fair self-fulfilling SCCs until a fair self-fulfilling SCC is found or all possible states are visited. Otherwise, if the discovered SCC C is fair, then the search finishes with generating a counterexample satisfying the imposed fairness constraints and violating the formula being checked. The process of model checking under fairness in ProB can be illustrated as in Figure 1.

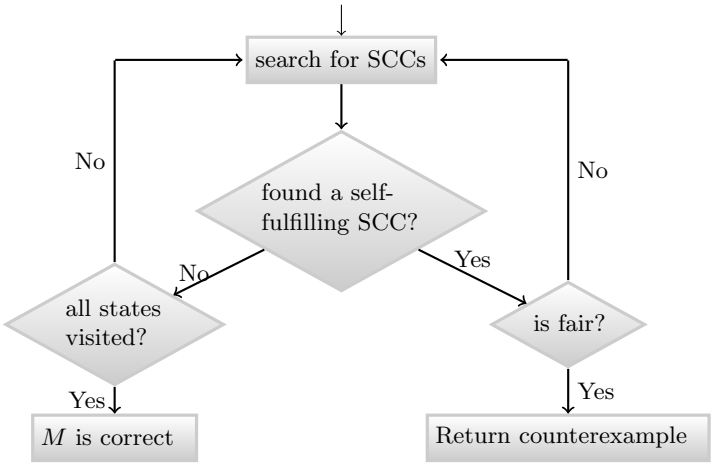

Fig. 1. LTL model checking under fairness

Since the fairness checks are performed on the discovered self-fulfilling SCCs, it was not necessary to modify the main search algorithm. Basically, we added a new procedure testing additionally the respective SCC in case the user has set some fairness constraints. We implemented support for action-based weak and strong fairness. The implementation allows setting fairness constraints on all possible actions of the system or on a subset thereof. Furthermore, both the weak and strong fairness assumptions can be imposed simultaneously for a given formula. That is, a property that should be satisfied under certain weak and strong fairness conditions can be checked in one run of the model checker.

We extended the LTL$^{[e]}$ grammar with four new operators: `sef` and `wef` for imposing strong fairness and weak fairness in regard to all actions of the system, respectively, and `sf`$(\cdot)$ (strong fairness) and `wf`$(\cdot)$ (weak fairness) for

setting fairness conditions on single actions of the system. Both operators $\texttt{sf}(\cdot)$ and $\texttt{wf}(\cdot)$ expect a transition proposition as an argument and can be used in combination with conjunction and disjunction in order to allow to impose more sophisticated fairness assumptions. The syntactic extensions enable the user to set the fairness constraints *fair* to a formula f in the well-known way: $\mathit{fair} \Rightarrow f$. Both the strong and weak fairness requirements, can be given simultaneously as a premise to the LTL$^{[e]}$ property by joining them with the conjunction operator. The fairness constraints are recognised on the syntactical level by the LTL$^{[e]}$ parser and are not included in the original property. The syntax for imposing fairness conditions in ProB can be outlined by the following grammar:

$$
\begin{array}{lllllll}
\mathit{fair} & ::= & \mathit{wfair} \mid \mathit{sfair} & \mid \mathit{wfair} \wedge \mathit{sfair} & \mid \texttt{wef} & \mid \texttt{sef} \\
\mathit{wfair} & ::= & \texttt{wf}(\mathit{tp}) & \mid \mathit{wfair} \vee \mathit{wfair} & \mid \mathit{wfair} \wedge \mathit{wfair} & \mid \texttt{(}\mathit{wfair}\texttt{)} \\
\mathit{sfair} & ::= & \texttt{sf}(\mathit{tp}) & \mid \mathit{sfair} \vee \mathit{sfair} & \mid \mathit{sfair} \wedge \mathit{sfair} & \mid \texttt{(}\mathit{sfair}\texttt{)}
\end{array}
$$

where *tp* is a transition proposition, and `wef` and `sef` are the tokens for setting weak and strong fairness conditions on all possible transitions, respectively.

To give an example of how one can set up fairness constraints to an LTL$^{[e]}$ formula in ProB consider a semaphored-based mutual exclusion algorithm for two processes.[1] Assume that each process is simplified to perform three types of actions: *req* (sending a request for entering the critical section), *enter* (entering the critical section), and *rel* (leaving the critical section). Further, consider the following property P: "*each process gets access to its critical section infinitely often*". To prove P on the model one needs to assume that all executions are weakly fair in regard to the *req* actions of both processes and that every execution is strongly fair in regard to the *enter* actions of both processes. Suppose that *req*.1 and *req*.2 denote the request actions of process 1 and process 2, respectively; and *enter*.1 and *enter*.2 the enter actions of process 1 and process 2, respectively. The corresponding fairness conditions can then be expressed by the grammar above as follows:

$$\big(\texttt{wf}(\mathit{req}.1) \wedge \texttt{wf}(\mathit{req}.2)\big) \wedge \big(\texttt{sf}(\mathit{enter}.1) \wedge \texttt{sf}(\mathit{enter}.2)\big)$$

Evaluation. First of all, encoding the fairness conditions by means of an LTL$^{[e]}$ formula and then adding it as a premise to the property is very inefficient since the state space of the search graph grows exponentially in the length of the formula [10]. For instance, if we encode the fairness conditions of the semaphored-based mutual exclusion model for two processes in LTL$^{[e]}$ and then check the re-formulated formula on the model, then the ProB LTL model checker would need to explore overall 1,048,576 atoms (the number of nodes in the search graph) to check P. On the other hand, checking P on the model using the fairness checking capabilities of ProB will need to explore only 44 atoms.

For the evaluation of the algorithm we have tested various CSP specifications where imposing fairness constraints is necessary to prove certain liveness properties. Further, we have evaluated model checking under fairness on ProB

[1] A detailed description of the algorithm could be viewed, for example, in [1] Chapter 2.

and PAT. PAT provides among others support for LTL model checking with fairness assumptions [12] for CSP#. A part of the results[2] of the evaluation is given in Table 1. It has been acquired by executing each test case 10 times with ProB 1.5.1 and PAT 3.5.1 on a Virtual Machine Version of Windows 7 (64 Bit) installed on a MacBook Pro Intel Core i5 Dual 2.90 GHz with 16 GB RAM.

Table 1. Part of the Experimental Results (times in seconds)

Model &				**State Space**		
# Procs	**LTL$^{[e]}$ Formula**	**Fairness**	**Tool**	States	States×Aut	**Time ($^{+}$)**
Peterson*	$GF[cs.0]$	Weak	ProB	514	2,113	1.415 (1.387)
# Procs: 3			PAT	513	2,099	0.092 (0.052)
Peterson*	$GF[cs.0]$	Weak	ProB	10,369	42,001	38.869 (37.035)
# Procs: 4			PAT	10,368	53,122	1.504 (0.207)
DP	$GF[eat.0]$	Strong	ProB	1,763	7,604	4.131 (3.898)
# Procs: 6			PAT	1,762	4,273	0.333 (0.121)
DP	$GF[eat.0]$	Strong	ProB	22,363	96,500	188.800 (149.220)
# Procs: 8			PAT	22,362	240,620	24.098 (1.291)
Scheduler	$G([enter.1]$	Strong	ProB	9,478	46,656	16.644 (11.976)
# Procs: 7	$\Rightarrow F[leave.1])$		PAT	7,290	61,238	4.785 (0.384)
Scheduler	$G([enter.1]$	Strong	ProB	30,619	148,716	97.913 (87.074)
# Procs: 8	$\Rightarrow F[leave.1])$		PAT	24,057	227,450	18.233 (0.808)

$^{(*)}$ In ProB the model is specified and verified using the CSP ‖ B methodology [9].
$^{(+)}$ The time needed for exploring the state space of the model.

All models in Table 1 are provided as examples with the PAT tool and have been translated into CSP-M and CSP ‖ B to be tested also within ProB. In all test cases the respective fairness constraints (Weak and Strong) were imposed on all transitions. The first sub-column of the `State Space` column reports the number of states of the model, whereas the second one the number of states of the product of the system with the respective automaton of the formula. In the `Time` column we have listed the time needed for checking the LTL$^{[e]}$ formula on the model. In the parentheses of the `Time` column we have also given the times needed for the exploration of the complete state space of the respective model.[3]

In all test cases in Table 1 the PAT tool has outperformed ProB. Observing the times just for the state space exploration (the times in parentheses), we can see that the discrepancy in the performance of both tools remains. On the other hand, comparing the overhead in PAT caused for checking the property on the model and for performing all fairness checks with this of ProB one can observe that the differences are very small. For instance, for `Scheduler8` ProB needed 87.074 seconds to explore the state space of the model and 97.913 to explore the state space of the specification and check the LTL$^{[e]}$formula, i.e. ProB needed about ten seconds to check the property on the already explored state space and perform all necessary fairness checks. In the same time, the overhead

[2] The models and the results of the experiments can be obtained from the following web page `http://nightly.cobra.cs.uni-duesseldorf.de/fairness/`.
[3] Generally, we have performed deadlock checking on the model for both tools in order to measure the times for state space exploration.

for testing the LTL property under fairness in PAT is about 17 seconds. This suggests that the main reason for ProB being outperformed by PAT is due to the poor performance of ProB's CSP interpreter responsible for the state space exploration of CSP specifications.

Table 2. Experimental Results on MINT Linux (64 Bit) - (times in seconds)

Model	**LTL$^{[e]}$ Formula**	**Fairness**	**Tool**	**States × Aut**	**Time**
DP	$GF[eat.0]$	Strong	ProB	96,500	362.452
# Procs: 8			PAT (Mono)	240,620	380.755
Scheduler	$G([enter.1] \Rightarrow F[leave.1])$	Strong	ProB	148,716	110.454
# Procs: 8			PAT (Mono)	227,450	777.441

To reproduce the results from Table 1 one needs to run the experiments on Windows as PAT is mainly developed for Windows. On other operating systems such as Linux one can run PAT with the mono platform. However, experiments have shown that PAT 3.5.1 with mono performs poorly on other systems such as Linux and in most cases will be outperformed by ProB as can be seen in Table 2. The PAT experiments in Table 2 were performed with mono 3.2.8.

Table 3. Fairness Checking Statistics in ProB (times in seconds)

Model & # Procs	**LTL$^{[e]}$ Formula**	**# Atoms**	**# Rejected SCCs**	**Fairness Checking Time**	**Total Time**
DP 7	$sef \Rightarrow GF[eat.0]$	27,093	291	0.824	29.188
DP 8	$sef \Rightarrow GF[eat.0]$	96,501	824	4.311	205.559
ME_Sem 10	$sef \Rightarrow GF[enter.1]$	26,628	2,305	0.664	36.044
ME_Sem 11	$sef \Rightarrow GF[enter.1]$	57,349	5,121	1.898	162.136

In Table 3 we have listed several experiments run with ProB to reveal the overhead caused by the fairness check. We have measured the time needed for checking and rejecting of all non-fair SCCs violating the checked property. Although the number of non-fair SCCs is considerably high, the fairness checking times in all cases are very small in comparison to the overall checking times.

Related Work. Besides the two notions of action-based fairness discussed in this paper, PAT [12] supports also verification under weak and strong process fairness. Furthermore, PAT provides also support for strong global fairness, fairness notion concerned with the infinite execution of both actions and states. One of the most prominent model checkers, SPIN [6], provides support for weak fairness. In [12], the performance of verification under weak fairness in SPIN is compared with that of PAT. In most of the test cases in [12], PAT performed better than SPIN. Another model checker that provides support for fairness is NuSMV [3], a symbolic model checker supporting two types of state-based fairness: justice and compassion. A justice constraint assumes that a given state formula is fulfilled infinitely often, whereas the compassion assumption requires that a formula must be true infinitely often if another state formula is true infinitely often.

Conclusion. We have presented a fairness implementation in ProB supporting verification under weak and strong action-based fairness for B, Event-B,

Z, CSP, and CSP‖B. Fairness assumptions in ProB can be easily imposed on all actions of the checked model, or on a subset thereof; it is even possible to specify action parameters. It appears that for LTL model checking of large-scale CSP specifications ProB performs poorly compared to other model checkers for CSP. However, the main motivation of ProB's CSP support was to provide an FDR/CSP-M compliant interpreter which can be used for CSP‖B, and which has not been tuned for model checking. On the positive side, experiments have shown that the overhead caused by the fairness checking procedure is considerably small and it can be applied to a wide range of specification formalisms.

Acknowledgements. We would like to thank David Williams for the ideas, very useful feedback and support on this work.

References

1. C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
2. S. Chouali, J. Julliand, P.-A. Masson, and F. Bellegarde. Pltl-partitioned model checking for reactive systems under fairness assumptions. *ACM Trans. Embed. Comput. Syst.*, 4(2):267–301, May 2005.
3. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, March 2000.
4. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.
5. N. Francez. *Fairness*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.
6. G. Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.
7. M. Kwiatkowska. Event fairness and non-interleaving concurrency. *Formal Aspects of Computing*, 1(1):213–228, 1989.
8. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, Mar. 1977.
9. M. Leuschel and M. Butler. Combining CSP and B for specification and property verification. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *FM'2005*, LNCS 3582, pages 221–236. Springer-Verlag, January 2005.
10. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. POPL '85, pages 97–107, New York, NY, USA, 1985. ACM.
11. D. Plagge and M. Leuschel. Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. *STTT*, 12(1):9–21, Feb 2010.
12. J. Sun, Y. Liu, J. S. Dong, and J. Pang. Pat: Towards flexible verification under fairness. In *CAV*, pages 709–714, 2009.
13. R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.
14. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.
15. D. M. Williams, J. de Ruiter, and W. Fokkink. Model Checking under Fairness in ProB and Its Application to Fair Exchange Protocols. In A. Roychoudhury and M. D'Souza, editors, *ICTAC*, LNCS 7521, pages 168–182. Springer, 2012.

A Experimental Setup (for referees)

The purpose of this appendix is to give an overview of how the experiments in Table 1 were carried out and to allow the reviewers to experiment with the model checkers of ProB and PAT. Furthermore, it should enable them to reproduce our experiments.

We ran our benchmarks with ProB version 1.5.1-beta6 and with the latest release of the PAT tool (version 3.5.1). A current nightly build of ProB is available from `http://nightly.cobra.cs.uni-duesseldorf.de/tcl/`. The latest release version of the PAT tool can be downloaded from the PAT tool homepage: `http://pat.comp.nus.edu.sg/`. All CSP and CSP# specifications can be downloaded from `http://nightly.cobra.cs.uni-duesseldorf.de/fairness/`, which are wrapped into the TestCasesExport.zip file. TestCasesExport.zip contains three folders:

- ProB: comprising all CSP and CSP‖B models for evaluating ProB and corresponding benchmark results,
- PAT: comprising all CSP# specifications for evaluating PAT and corresponding benchmark results,
- ProBFairnessStats: containing the results and CSP specifications of the benchmarks presented in Table 3,
- Linux: comprising the test cases and benchmarks results provided in Table 2.

Experimenting with ProB. To run the experiments from Table 1 with ProB use the command line version `probcli` (`probcli.exe` for Windows) of ProB. Each test case in Table 1 was run with the `-assertions` option of `probcli`. To check, for example, the LTL formula "$G([enter.1] \Rightarrow F[leave.1])$" under strong fairness on scheduler6.csp specification in the ProB/Scheduler folder type the following command in the command line:

```
probcli -assertions scheduler6.csp
```

As a result, the assertion

```
SCHEDULER0 |= LTL: "SEF => G ([enter.1] => F [leave.1])"
```

will be successively checked. Besides the notification that the check was successively performed `probcli` will also print out other relevant informations such as number of callbacks, states, atoms (all possible valuations of the LTL formula $\times$ the number of states, see also table column `States` $\times$ Aut), number of rejected SCCs, fairness checking time, etc. For more details on how to interpret the results printed out on the console by the ProB LTL model checker consult the following tutorial: `https://www3.hhu.de/stups/prob/index.php/LTL_Model_Checking`. In the tutorial you will also find out how to use the LTL model checker in Tcl/Tk.

The model of the Peterson's algorithm in Table 1 is created using the CSP‖B methodology which is supported by ProB. To run the experiment for Peterson with 3 processes, for example, use the following commando

```
probcli Peterson3.mch -csp-guide Peterson3.csp -ltlassertions
```

where Peterson3.mch and Peterson3.csp are the specifications used for formalising the Peterson algorithm in CSP‖B, and `-ltlassertions` the option to start checking the LTL$^{[e]}$ formula $GF[cs.0]$ by the LTL model checker.

Experimenting with PAT. Note that to be able to reproduce the results from Table 1 for PAT you will need to run the experiments on Windows since PAT is mainly developed for Windows. PAT can be run also on other operating systems using the mono platform. However, initial experiments have shown that running PAT with mono is in almost all experiments outperformed by ProB. Similar to the CSP and CSP‖B specifications, all CSP# specifications are provided with one assertions that is checked for the respective specification. To run the experiments from Table 1 with PAT use the command line version `PAT3.Console.exe` of PAT. To check an LTL formula under weak and strong fairness use the options `-behavior 1` and `-behavior 2`, respectively. To check, for example, the LTL formula "$G([enter.1] \Rightarrow F[leave.1])$" under strong fairness on scheduler6.csp specification in the PAT/Scheduler folder type the following command in the command line:

```
PAT3.Console.exe -behavior 2 -v scheduler6.csp result.txt
```

In the command above the `result.txt` that will be written after the respective assertion check. In `result.txt` all statistics of the respective assertion check such as number of states, time to perform the assertion check, etc. will be recorded. The option `-v` is optional and is used to print out the text written in `result.txt` on the command line prompt.

A short description of the specifications. All specifications represent prominent examples from the parallel and distributed computing. As one can see from the tables in Section 3, we have tested each model for different number of processes. There is an identifier, named `N`, on the top of each specification that is used to set the number of the processes. The abbreviations in the first column `Mode & # Procs` of both tables denote the following specifications:

- `DP`: An example of the dining philosophers resolving the starvation problem by forcing the first philosopher to pick at first the right fork instead of the left one. The process which has been tested is `College` process.
- `ME_Sem`: A model of a mutual exclusion algorithm guaranteeing that maximum one process has an access to its critical section by a semaphore. The process that has been checked here is the `MAIN` process.
- `Peterson`: A specification of the Peterson's mutual exclusion algorithm. The process that has been tested here is the `MAIN` process. Note that in the case of ProB the algorithm was specified by means of the CSP‖B methodology.
- `Scheduler`: A model of a simple scheduler. The process being tested for this test cases was the `SCHEDULER0` process.