DOI 10.1007/s00165-004-0040-y
BCS © 2004
Formal Aspects of Computing (2004) 16: 166–193

Formal Aspects of Computing

A generalized semantics of PROMELA for abstract model checking

María del Mar Gallardo, Pedro Merino and Ernesto Pimentel
Dpto. de Lenguajes y Ciencias de la Computación, University of Malaga, 29071 Malaga, Spain

Abstract. Semantics of description languages for complex systems are a central issue for implementing verification methods such as *abstract model checking*. This technique is employed to verify systems by inspecting only a small state space that represents its potential behaviors. This paper presents a generalized operational semantics of the modelling language PROMELA that provides the theoretical basis to introduce this promising method in the model checker SPIN. The generalization consists of identifying language aspects affected by the abstraction. Using these aspects as parameters, it is possible to obtain and relate different interpretations of the language. The new semantics provides a framework to reason about how to construct the tool αSPIN as an extension of SPIN.

Keywords: Model checking; Abstraction; Structured operational semantics; PROMELA; SPIN

1. Introduction

The existence of a formal semantics for an engineering design language is convenient in order to have a non-ambiguous description of complex systems, and to construct tools for analysis (simulation or verification, testing) or for code generation. Nevertheless, considering as non-disputable the need for a semantics, it is reasonable to argue that its structure could be oriented to a specific application. One particularly interesting case is *abstract model checking* [CGL94, DGG97], which uses *abstractions* to increase power of *model checking* [CES86, CGP00] for automatic verification.

Model checking techniques allow us to decide whether one or all executions produced by a model of a system satisfy a temporal logic formula. They work by generating and inspecting the states produced by the model, so they are only applicable to small or medium size systems. Abstraction techniques consists of replacing the model of the system with an abstract version, reducing the state-space to be explored. Many works discuss how the tools supporting abstract model checking demand semantics based theoretical frameworks to ensure that the results obtained with the abstract models can be applied to the initial model [CGL94, LGS95, DGG97, GrS97, BLO98, GaM99, RuS99]. Most of these works base the construction of the abstract model on the *abstract interpretation theory* [CoC77], which itself is a theory based on semantics: it is necessary to relate the standard meaning of a program or a model with the meaning of the abstract version. This paper is devoted to a semantics making the study of this relation easier, and enabling us to include correct abstractions in the model checker SPIN [Hol91, Hol97, Hol03, Spin].

The model checker SPIN is probably the one attracting most users in academic and industrial environments, for several reasons. The tool implements the most recent optimization mechanisms, and it is free for non-commercial use. The input language, PROMELA, is relatively small and sufficiently expressive to represent realistic concurrent

Correspondence and offprint requests to: Pedro Merino, Email: pedro@lcc.uma.es, Tel.: +34-9-52132752, fax: +34-9-52131397
Work supported by projects TIC2002-04309-C02-02 and TIC2001-2705-C03-02

and distributed systems. Its syntax is a mixture of well known languages such as C and CSP, and its execution model is an extension of the basic communicating finite state machine model.

There are a number of recently proposed formal semantics for PROMELA [NaH97, Wei97, Bev97, Spin], but they are not suitable to support reasoning about abstractions. They were mainly defined to document the language for final users or for simulating the language. Our semantics is oriented to abstraction and it also gives a description of PROMELA in a clear and elegant way, thus fulfilling the same aim as the other proposals.

Our approach is based on two main ideas. The first one is to separate the elements of the language employed for simulation from the ones employed for verification. The second one is to identify the aspects (parameters) of PROMELA that can be affected by the abstraction and define a generalized semantics that could be suitably instantiated (parameterized) to obtain the standard semantics of the language or abstract versions. Using these aspects as parameters, it is possible to obtain and relate different interpretations of the language. This relation among semantics is the basis for defining correctness conditions to ensure that the abstract model of a system (its execution under the abstract semantics) preserves interesting properties regarding the standard model.

Using this semantics and the parameters behind it, in recent works we have presented a number of conditions to guarantee several preservation results:

1. In [GaM99], we studied the conditions to ensure that the abstract model simulates the concrete one.
2. In [GMP02a] we presented a new method (called *over-approximation*) to check whether the refutation of temporal logic formulas [MaP92] in the abstract model implies refutation in the concrete model. In in [GMP02b], this work is compared to the classic approach to check satisfaction of temporal logic [CGL94, DGG97] (called *under-approximation*).
3. In [GMP02c] we combined the over-approximation and the under-approximation methods to improve the verification by refinement of temporal logic formulae.
4. Most of the conditions and preservation results in previous works can be automatically checked and have made the implementation of abstraction by syntactic transformation of PROMELA possible, enabling us to employ SPIN as the basic tool inside αSPIN [GMM02, αSPIN].

This paper focuses on presenting most of the details of the generalized semantics that originated the results cited above. In particular, the contributions of the paper are the following. We provide a Structured Operational Semantics (SOS) (as defined by Plotkin [Plo81]), whereby the behavior of a PROMELA specification is given in terms of the behavior of its components via inference rules inductively defined taking into account its syntax. Then, the semantics is employed to define correct abstract models for simulation. Using these abstract models, we introduce the abstract verification of temporal properties represented with the PROMELA notation for Büchi automata, the *never claim*. Finally, we establish a relation between automata-based abstract model checking and the approach based on temporal logic (under-approximation and over-approximation).

The remainder of the paper is organized as follows. Sect. 2 presents the language PROMELA and the model checker SPIN. In Sect. 3, we present the generalized semantics. The abstraction of the model is introduced in Sect. 4, where we present the conditions for correct simulations depending on the parameters of the semantics. Sect. 5 extends the semantics with the verification-oriented elements of the language. Sect. 6 employs this extension to reason about correct abstract model checking with automata and relates these results to the two temporal logic based approaches. Related works are presented in Sect. 7, and Conclusions in Sect. 8. Finally, Sect. 9 is an Appendix with the complete definitions of some concepts used in the paper.

2. Overview of PROMELA and SPIN

In the last few years, SPIN has become one of the most employed model checkers in both academic and industrial areas (see the annual workshops devoted to this tool on the official SPIN web page [Spin]). SPIN supports the simulation of system prototypes written in the modelling language PROMELA, and it can perform a very efficient verification of usual safety properties (like absence of deadlock) as well as complex liveness requirements expressed with linear temporal logic (LTL) [MaP92]. The aim of this section is to give an overview of the language and the tool. The official web page should be consulted for details.

PROMELA is a non-deterministic modelling language designed for describing systems composed of concurrent processes that communicate asynchronously (such as the software for distributed systems). Syntax elements are borrowed from Dijkstra's guarded command language, Hoare's CSP language and C programming language.

PROMELA constructs[1] can be divided into two categories: behavioral and verification constructs. The behavioral part is used for representing the actual behavior of the systems. These representations can be directly used in simulation mode to (randomly) obtain several potential execution sequences of the system. The systems are modelled as instances of communicating process types (`proctype`). A *process* is defined as a sequence of possibly labelled instructions preceded by the declarative part as follows:

$$Process ::= [\texttt{active}[``[\text{''}\texttt{NumberOfInstances}``]\text{''}]]\,\texttt{proctype}\ ProcessTypeID\ \{Decl; InstSeq\}$$

where *Decl* is the declarative part of the model. The syntax of main instructions is defined using BNF notation in Fig. 1. We omit details of the declarative part of the language. Although declarations appear before statements in the figure, PROMELA allows them to appear mixed between statements.

Next, we describe Fig. 1. Note that in the paper we will use non-terminal symbols like *Basic* to denote both the corresponding rule in the grammar and the set of instructions generated by the rule itself. The rule *Basic* defines the subset of the PROMELA instructions that may modify the model state. *Assign* represents PROMELA assignments. The instructions for sending (receiving) messages to (from) channels whose size is greater or equal to zero are represented, respectively, by *Input* and *Output*. We assume that the instructions for the communication via *Rendezvous*, when the size of the channel is zero, are defined by these two rules. However, in order to simplify the semantic rules defining *rendezvous*, we use the set $Rendez \subset Input \cup Output$ to denote the set of instructions for sending or receiving data through zero sized channels.

BExp represents the Boolean expressions including tests over variables and contents of channels. Boolean expressions in PROMELA are side-effect free. In particular, *BExp* includes tests such as *nfull(c)* (*nempty(c)*), $c?[v_1, \ldots, v_s]$ and $c??[v_1, \ldots, v_s]$, which, respectively, check if channel c is not full (not empty), if the first channel message matches the tuple $v_1, \ldots, v_s$ and if some message in c matches the tuple. *BExp* also contains the predefined Boolean expressions `skip` (which is a syntactic sugar for `true`) and `timeout` (whose meaning will be explained later). Finally, *Print* represents the set of `printf` instructions.

If and *Do* represent the control flow instructions usual in imperative languages. However, note that branches defined by *Branch* and used in *If*, *Do* are non deterministic. *Atomic* and *D_Step* are used to implement atomic sequences of instructions. *D_Step* can only be utilized when no nested sentence may suspend during execution, that is, only deterministic instructions can be placed inside a `d_step` sentence. In contrast, it is possible for a process to suspend during the execution of an *Atomic* instruction. In this case, the control is transferred to another process. *Unless* defines the set of `unless` instructions. Instructions before `unless` are executed while the first instruction following `unless` (the *escape* instruction) is suspended. Therefore, *escape* defines the conditions to abort the execution of sentences before `unless`. Finally, instruction `run pname(ap)` creates a process of *proctype pname* and passes the actual parameters *ap* to it.

Example 1. The model in Fig. 2 contains three PROMELA constructs, namely `atomic`, *rendezvous*, and `unless` whose interaction is hard to asses as mentioned on the SPIN web page [Spin]. Process `p1` sends data to be analyzed by process `p2`. The analysis consists in filtering even and odd data, and it is modelled with the `unless` constructs using the test for even values as the escape expression. Constant `N` is only employed to prevent an uncontrolled overflow in the variables to produce new data. Note that all instructions are labelled.

The verification constructs of PROMELA are used to represent desirable (or undesirable) properties about the system behavior. The set of properties comprises *assertion violations*, *invalid end states*, *acceptance cycles*, *non-progress cycles*, and *never claims* (that represent Büchi automata to reason about infinite execution [VaW86]). The most powerful one is *never claim*, which is just defined as a special process like

$$neverclaim ::= \texttt{never}\ \{\ TestInstSeq\ \}$$

where *TestInstSeq* represents a restricted sequence of instructions (*Inst*) that can only evaluate Boolean expressions (*BExp*), and define the flow-control (*If*, *Do*, *Jump*). Boolean expressions in *never claim* are extended with two special constructions *pc_value(pid)* and *enabled(pid)*.

The simulation of the model with SPIN only comprises the checking of a restricted set of properties over the visited execution sequences. The verification consists of automatically checking all the properties against the behavioral part of the system by an exhaustive search. These two tasks are done in a very efficient way with techniques such as *partial order reduction*, *bit state hashing* and *state compression*.

[1] In the rest of the paper, we will use the terms instructions, sentences, constructs and statements indifferently.

```
InstSeq ::= [L :] Inst{; [L :] Inst}*
Inst ::= Basic | Jump | If | Do | Unless | Run | Atomic | D_Step
Basic ::= BExp | Assign | Input | Output | Print
Jump ::= goto l | break
If ::= if BranchSeq [ ElseBranch ] fi
Do ::= do BranchSeq [ ElseBranch ] od
Unless ::= "{"InstSeq"}" unless "{"InstSeq"}"
Run ::= run ProcessTypeId ActualParameters
Atomic ::= atomic "{" InstSeq "}"
D_Step ::= d_step "{" DetSeq "}"
Input ::= ChannelId (? | ??) ExpSeq | ChannelId (? | ??) < ExpSeq >
Output ::= ChannelId (! | !!) ExpSeq
Branch ::= :: InstSeq
BranchSeq ::= Branch{Branch}*
ElseBranch ::= :: else[- > InstSeq]
Det ::= DetBasic | DetIf | DetDo | Jump
DetSeq ::= Det{; Det}*
DetBranch ::= :: DetSeq
DetBranchSeq ::= DetBranch{DetBranch}*
DetElseBranch ::= :: else[- > DetSeq]
DetBasic ::= BExp | Assign | Print
DetIf ::= if {DetBranchSeq}[DetElseBranch] fi
DetDo ::= do {DetBranchSeq}[DetElseBranch] od
```

Fig. 1. A fragment of the PROMELA syntax

```
#define N 10
chan c=[0] of {int};

active proctype p1() {
  int i = 1;
  L1: do
      :: L2: i < N -> L3: atomic{L4: c!i; L5: i = i+1}
      :: L6: i >= N -> L7: i = 0;
      od;
}

active proctype p2() {
  int j = 1;
  start: {L8: c ? j; L9: printf(j); L10: goto start} unless {L11: (j\%2) == 0};

  even: atomic{L12: printf("Even");L13: j = 1; L14: goto start}
}
```

Fig. 2. Model even/odd

Furthermore, SPIN implements the translation of LTL formulas to *never claims* [GPV95], thus allowing the verification of temporal properties specified in this way. Well-formed formulas of linear temporal logic (LTL) are inductively constructed from a set of atomic propositions (in PROMELA, propositions are tests over data, channels or labels), the standard Boolean operators, and the *temporal operators*: *always* "$\Box$", *eventually* "$\Diamond$", *next* "$\bigcirc$", and *until* "U". Formulas are interpreted with respect to model state sequences $t = s_0 \rightarrow s_1 \rightarrow \ldots$. Each sequence expresses a possible model execution from state s_0. The use of *temporal operators* permits construction of formulas that depend on the current and future states of a configuration sequence. The semantics of LTL is shown in Fig. 3 where p is a proposition, f and g are temporal formulas, and t_i denotes the suffix trace of t starting at state s_i. For the sake of convenience, we assume that all formulas are in negation normal form, that is, negation only appears in propositions. Note that we have not defined the satisfaction of negated formulas. Instead, we treat the evaluation of negated propositions independently of their corresponding non-negated ones (as will be explained in the following sections, this is a common practice in abstract model checking). Note that $\forall$ and $\exists$ are not LTL operators. They are employed to simplify the notation in the discussion about abstract model checking (Sect. 6). In the last two definitions M represents the set of execution traces produced by the model.

3. The generalized semantics for simulation

In this section, we define the operational behavior of a PROMELA model by means of a trace-based semantics. This SOS semantics is built from stratified semantic rules defining the behavior of a PROMELA model at different levels.

$t \models p$	*iff*	$s_0 \models p$
$t \models \Box f$	*iff*	*for all* $j \geqslant 0, t_j \models f$
$t \models \Diamond f$	*iff*	*there exists* $j \geqslant 0$ *such that* $t_j \models f$
$t \models \bigcirc f$	*iff*	$t_1 \models f$
$t \models fUg$	*iff*	*there exists* $k \geqslant 0$ *such that* $t_k \models g$ *and* *for all* $0 \leqslant j < k, t_j \models f$
$M \models \forall f$	*iff*	*for all trace* $t \in M, t \models f$
$M \models \exists f$	*iff*	*there exists a trace* $t \in M$ *such that* $t \models f$

Fig. 3. LTL semantics

The semantics is generalized in the sense that it describes the operational behavior of a model, making implicit the characteristics which are modified when abstracting the model, such as data and messages. These aspects are given by functions τ (*test*) and φ (*effect*) presented in Sect. 3.1.4. At this point, we do not include the semantics for verification, but it will be introduced later to deal with abstract model checking.

We use a bottom-up approach to present the semantic rules to simulate PROMELA models. We start with the rules for the isolated execution of instructions in a process (transition relation $\longmapsto_{proc}$), then we employ these rules to define the interaction among processes (transition relations $\longmapsto_{int}$ and $\longmapsto_{mod}$) and finally we define the observable steps for simulation ($\longmapsto_{sim}$). Since the bottom-up approach defines each transition relation depending on the previous one, and therefore each rule used in a given transition relation has been previously defined, this approach seems to be easier to read. However, the top-down approach, developed in [GMP01], which goes from the high level language constructors, may also be useful to hide the low level details. Anyway, both SOS-based approaches have turned out to be very valuable to include extensions of the language. For instance, in Sect. 5, we append rules to include the *never claim* construction and the verification semantics.

As commented above, in this paper, we first present the semantic rules at process level $\longmapsto_{proc}$, which define the behavior of each process independently of the context in which it executes. The following two intermediate levels $\longmapsto_{int}$ and $\longmapsto_{mod}$ deal with different aspects of the interaction of processes. Relation $\longmapsto_{int}$ mainly describes rendezvous and `timeout`, while relation $\longmapsto_{mod}$ describes the *Atomic*, *D_Step* and *Unless* instructions. Finally, the highest level of the PROMELA semantics for simulation $\longmapsto_{sim}$ gives us the process interaction through co-routines. We will relate each semantic rule in a level with its use in upper levels in order to maintain, in some way, the top-down point of view.

3.1. Definitions

The following definitions are employed in the semantic rules of the semantics. For the sake of simplifying the presentation, we do not give the precise definition of some concepts here. They will be given in the Appendix of the paper.

3.1.1. Instructions

Let $Inst$ be the set of instructions defined in the previous section. We assume that all *Basic/If/Do/Run* instructions of *processes are labelled*, i.e., each one of these instructions has the form $L : ins$ where $L \in Label$ is a unique label of the instruction $ins \in Basic \cup If \cup Do \cup Run$. Labels may be defined by the user or automatically assigned. End denotes the set of user-defined labels starting with end. The code of each process is finished with a label $L \in End$. Note that labels represent process program counters.

Function $I : Label \to Inst$ returns the instruction following a label. For instance, considering the code of Fig. 2, $I(L3) =$ `atomic{L4 : c!i; L5 : i = i + 1}`

Function $next : Label \to Label$ updates the process labels (program counters) during execution. It returns the label of the $next$ $Basic/If/Do/Run$ instruction to be executed. In the semantic rules of $\longmapsto_{proc}$, $next$ is only applied after executing a $Basic/Run$ instruction. Function $next$ follows the control flow in the process until it finds the label $next(L)$ pointing to the next instruction to be executed, which must be a $Basic/If/Do/Run$ instruction. In particular, the execution of $goto$ and $break$ instructions is hidden in $next$. Following the example in Fig. 2, once instruction L2 has been executed, we use $next$ to update the program counter to $next(L2) = L4$. Observe that $next$ models the sequential composition of instructions inside processes.

Function $g : Label \rightarrow \wp(Label)$ associates each label L with the set of labels pointing to the $Basic/Run$ instructions in their guards. When $I(L) \in Basic \cup Run$, $g(L) = \{L\}$, but when $I(L) \in If \cup Do$, since any instruction may be a guard, function g inspects each branch until it finds a $Basic/Run$ instruction. Similarly, function $g_{else} : Label \rightarrow Label_{\perp}$ returns the label of the first $Basic/If/Do/Run$ instruction in the $ElseBranch$ of If/Do instructions (it returns $\perp$ when the instruction has no *else* branch). For example, in Fig. 2, $g(L1) = \{L2, L6\}$.

Function $mode : Label \rightarrow \{ilv, atm, dst\}$ is used to detect the *execution mode* of a process, i.e., to know if the process is actually executing a *D_Step* or an *Atomic* instruction. This is used by relation $\longmapsto_{mod}$ to correctly interleave processes. Function $mode$ is defined as follows:

- $mode(L) = dst$, if L is within the scope of a `d_step` instruction; otherwise,
- $mode(L) = atm$, if L is within the scope of an `atomic` instruction; otherwise,
- $mode(L) = ilv$.

For instance, in the code of Fig. 2 $mode(L5) = atm$ and $mode(L10) = ilv$.

Function $rdv : Label \times Label \rightarrow \{false, true\}$ detects whether two instructions correspond to a $Rendevous$. The first parameter always corresponds to the instruction in the sender process. Continuing with the example, $rdv(L4, L8) = true$.

Finally, to correctly define the behavior of *Unless* instructions, we make use of function esc which gives us the label(s) of the *escape* sentence(s). Function esc returns a label sequence of labels because $unless$ instructions may be nested. Assuming that $Label^*$ is the set of all label sequences, we define function $esc : Label \rightarrow Label^*$ as $esc(L) = L_1 \cdot \cdots \cdot L_k \cdot L$, if L is in the scope of a nested `unless` instruction and the labels of its successive escape instructions are $L_1, \ldots, L_k$, following the nesting order. Note that we have added label L at the end of the sequence, therefore $esc(L) = L$ indicates that L is not inside an `unless` instruction.

3.1.2. Data

Let Var be the sets of names of variables and channels declared in the model. For each $v \in Var$, let $Const_v$ denote the $type$ of v, i.e., the set of values that v may store during execution. If v represents a channel, then $Const_v$ is the set of all finite sequences of items that v may contain. We define $Const = \bigcup_v Const_v$.

3.1.3. Process state

Let $Env = Var \rightarrow Const$ be the set of process environments. An element $e \in Env$ is a partial function that associates variables and channels with their actual contents. The function is partial because each process only knows its local variables and the global variables and channels. Note that each process environment has unrestricted access to global variables and channels.

Let $State = Env \times Label$ be the set of process states. State $\sigma = \langle \sigma_e, \sigma_l \rangle$ represents the actual value of the process variables and channels given by σ_e, and the program counter σ_l.

3.1.4. Parameters τ and φ

Consider the set $Basic$ of basic instructions defined in the previous section. The execution of these instructions is defined with the following functions:

- the mapping $\tau : BExp \times Env \rightarrow \{false, true\}$ provides the evaluation of a Boolean expression in a certain environment.
- the function $\varphi : Basic \times Env \rightarrow Env \cup \{error\}$ gives meaning to each basic PROMELA instruction, in such a way that $\varphi(ins, \sigma_e) = \sigma_e'$ means that by executing the instruction $ins \in Basic$ of a process, the environment σ_e changes into σ_e'. If the execution of ins in the environment σ_e produces an error, then $\varphi(ins, \sigma_e) = error$. Observe that when φ is applied to Boolean expressions, the environment is not modified.

Functions φ and τ are the parameters of our semantics. They deal with data manipulation and thus the separation in the semantics is between data and control. The definition of these functions depends on the data interpretation considered. Usually, φ and τ will implement the standard meaning of the *Basic* instructions (the one implemented by SPIN). However, when abstracting models, we will modify the implementation of these functions according to the new data utilized in the abstract model. We will study this specific problem in Sect. 4.

3.1.5. Configurations

A PROMELA model usually involves the execution of more than one process. Thus, the configuration of a model consists of the states of each model process. We identify each process with its pid, that is, an element of the set Pid. Let $\Gamma = \{\gamma | \gamma : Pid \to State \cup \{\bot\}\}$ be the set of model configurations. A configuration $\gamma \in \Gamma$ associates each process identifier $j \in Pid$ with its state $\gamma(j) \in State$. In order to simplify the notation, given $\gamma(j) \in State$, $\gamma(j).\sigma_e$ and $\gamma(j).\sigma_l$ denote the actual environment and the program counter of process j.

We assume that all process environments in a given configuration γ have consistent values for global variables and global channels, i.e., $\forall v \in Var$ if v is a global variable, then $\forall i, j \in Pid.\gamma(i).\sigma_e(v) = \gamma(j).\sigma_e(v)$. Similarly, if c is a channel shared by the processes $i, j \in Pid$, then $\gamma(i).\sigma_e(c) = \gamma(j).\sigma_e(c)$. Note that when a process changes its state, the global configuration remains consistent. If $j \in Pid$ does not represent a process instance, then $\gamma(j) = \bot$.

When executing a run instruction, function $newPid : \Gamma \to Pid$ assigns a pid to the new process. In addition, the state of this new process is created with the function $initP : State \to State$ using the state of the parent process.

We assume that given a PROMELA model M, $initial(M, \tau, \varphi) \in \Gamma$ gives us the initial configuration for M. That is, $initial(M, \tau, \varphi)$ initializes all global variables and channels, creates the initial processes declared in M, associates them with their corresponding pids, and initializes their local states. To initialize variables and channels, $initial$ makes use of functions τ and φ.

3.1.6. Executability

In PROMELA, only executable statements may be selected to be executed next. The *executability* of a sentence strongly depends on the sort of instruction. For example, assignments are always executables, but only true Boolean expressions are executable; *If* and *Do* statements are executable if some of their branches are executable; writing over (reading from) a channel is possible when the channel is not full (empty) and the data sent and expected by the receiver match.

Let $exec : Label \times Env \to \{false, true\}$ be the executable function. Thus, given $L \in Label$, the executability of the labelled instruction $I(L)$ in the environment σ_e is denoted by $exec(L, \sigma_e)$. The precise definition of $exec$ (given in the Appendix) makes use of function τ, previously defined. As function $exec$ is used in the semantic rules at process level, and at this level rendezvous cannot be described, we define $exec(L, \sigma_e) = true$, when $I(L) \in Rendez$.

As commented in Sect. 3.1.1, `goto` (a $Jump$ instruction) is not treated as a statement, but as a control-flow mechanism defining the instruction which follows the immediately preceding statement. However, no syntax restriction prevents a programmer from using a `goto` as a guard in a `do/if/unless/atomic/d_step` instruction. In this case, `goto` is used to define both the control-flow graph of the program and the executability of the branch/instruction. In order to distinguish these two roles, we assume that `goto` sentences are preceded by the `skip` instruction when used as guards. Thus, the first role is implemented by the `goto`, whereas the second one is implemented by `skip`. Therefore, `goto` sentences are never directly executed. Functions $next$ uses them to define the flow control graph of the program. The same discussion applies to other $Jump$ instructions, like `break`.

Function $nextexec : Pid \times \Gamma \to \wp(Label \cup (Label \times Pid \times Label))$ employs $exec$ to calculate the set of labels of the executable basic instructions $nextexec(j, \gamma)$, enabling process j to progress, considering that its program counter $\gamma(j).\sigma_l$ could be nested in a sequence of *Unless* instructions. In other words, this function manages the control flow when executing *Unless* instructions, although it is also applicable when the current instruction is not nested inside an *Unless*. Modelling rendezvous inside *Unless* is hard to resolve, as commented in the SPIN manual [Spin]. In order to manage this situation, function *nextexec* returns a set of single labels, corresponding to the instructions allowed which are not in *Rendez*. In addition, it may also return structures like (L, k, L') which indicate that instruction $I(L)$ of process j is a *Rendez* instruction allowed to be selected to continue the execution. The counterpart of $I(L)$ is the instruction $I(L')$ of process k. That is, $(L, k, L') \in nextexec(j, \gamma)$ means that the synchronous communication $I(L)||I(L')$ may be carried out where the sender process is j and the receiver process is k. The complete definition of *nextexec* is given in the Appendix in order to simplify the presentation at this point.

$$\texttt{Basic-proc}\quad \frac{I(\sigma_l)\in Basic,\ exec(\sigma_l,\sigma_e),\ \varphi(I(\sigma_l),\sigma_e)\neq error}{\langle\sigma_e,\sigma_l\rangle \overset{I(\sigma_l)}{\longmapsto}_{proc} \langle\varphi(I(\sigma_l),\sigma_e),next(\sigma_l)\rangle}$$

$$\texttt{Error-proc}\quad \frac{I(\sigma_l)\in Basic,\ exec(\sigma_l,\sigma),\ \varphi(I(\sigma_l),\sigma)=error}{\langle\sigma_e,\sigma_l\rangle \overset{error}{\longmapsto}_{proc}\ stop}$$

$$\texttt{Run-proc}\quad \frac{I(\sigma_l)\in Run}{\langle\sigma_e,\sigma_l\rangle \overset{run}{\longmapsto}_{proc} \langle\sigma_e,next(\sigma_l)\rangle}$$

$$\texttt{IfDo-proc}\quad \frac{I(\sigma_l)\in If\cup Do,\ \exists L_b\in g(\sigma_l).\langle\sigma_e,L_b\rangle\overset{inst}{\longmapsto}_{proc}\langle\sigma_e',\sigma_l'\rangle}{\langle\sigma_e,\sigma_l\rangle \overset{inst}{\longmapsto}_{proc} \langle\sigma_e',\sigma_l'\rangle}$$

$$\texttt{Else-proc}\quad \frac{I(\sigma_l)\in If\cup Do,\ \forall L_b\in g(\sigma_l).\neg exec(L_b,\sigma_e),\ g_{else}(\sigma_l)\neq\bot}{\langle\sigma_e,\sigma_l\rangle \overset{else}{\longmapsto}_{proc} \langle\sigma_e,g_{else}(\sigma_l)\rangle}$$

Fig. 4. Process-level rules

3.2. Semantic rules for simulation

In this section, we present the four levels of the SOS semantics in detail. We will comment each rule separately, and at the end of the Section, we will provide an example showing the functionality of some complex PROMELA constructors.

3.2.1. Executing a process instruction

The transition relation for the process level is defined as $\overset{-}{\longmapsto}_{proc}\subseteq State\times L_{proc}\times(State\cup\{stop\})$, where the set of rule labels is $L_{proc}=Basic\cup\{run,else,error\}$. The rules for the relation $\overset{-}{\longmapsto}_{proc}$ are defined in Fig. 4.

* `Basic-proc`
  This rule models basic sentences. If the current instruction of the process is executable, and the execution transforms the local environment σ_e into σ_e', then the new local state stores this change, and explicitly moves the program counter to the next instruction. Recall that since $exec$ defines as executable the process interaction by rendezvous, this rule may always execute both parts of this communication. Upper transition relations adequately handle the correct execution of rendezvous.
* `Error-proc`
  If function φ returns $error$, the process ends and an execution error is reported.
* `Run-proc`
  At this level, the rule only updates the process counter. The intermediate-level rule `run-int` explicitly creates the new process.
* `IfDo-proc` and `Else-proc`
  Note that in PROMELA *If* and *Do* are non deterministic statements, i.e., when several branches are executable, the semantics selects one of them non-deterministically for execution. If no branch is executable and there is not an `else` option, the instruction blocks. The `else` branch is always executable but it is only selected when no other branch is executable.
  Rule `IfDo-proc` is recursive. To execute an `If/Do` instruction, we first select an executable branch. Then we execute it and obtain a new state, and finally we make the transition from the initial state to this new state. Rule `Else-proc` applies when no branch is executable but the instruction has an `else` branch. In this case, we select it by modifying the program counter with the label of the instruction following `else`. Functions g and g_{else} give the proper labels to go on with the execution. Once a branch has been executed, function $next$ (in rule `Basic-proc`) updates the program counter to the instruction following the word `fi`, when executing an *If* instruction, or back to the beginning of *Do*, when executing this instruction. *Do* finishes when function $next$ finds `break`.

$$\texttt{Single-int}\ \frac{\gamma(j)\overset{inst}{\longmapsto}_{proc}\sigma,\ inst{\in}Basic{-}(Rendez{\cup}\{\texttt{timeout}\})}{\gamma\overset{inst_j}{\longmapsto}_{int}\gamma[\sigma/j]}$$

$$\texttt{Run-int}\ \frac{\gamma(j)\overset{run}{\longmapsto}_{proc}\sigma,\ k{=}newPid(\gamma)}{\gamma\overset{run_j}{\longmapsto}_{int}\gamma[\sigma/j,\ initP(\gamma(j))/k]}$$

$$\texttt{Rend-int}\ \frac{\gamma(j)\overset{inst_1}{\longmapsto}_{proc}\sigma_1,\ \gamma[\sigma_1/j](k)\overset{inst_2}{\longmapsto}_{proc}\sigma_2, rdv(\gamma(j).\sigma_l,\gamma(k).\sigma_l)}{\gamma\overset{rend_k^j}{\longmapsto}_{int}\gamma[\sigma_1/j,\ \sigma_2/k]}$$

$$\texttt{Timeout-int}\ \frac{\gamma(j)\overset{timeout}{\longmapsto}_{proc}\sigma, \forall inst{\neq}\texttt{timeout}.\gamma(j)\overset{inst}{\not\longmapsto}_{proc}, \forall k{\neq}j.\gamma(k)\not\longmapsto_{proc}}{\gamma\overset{timeout_j}{\longmapsto}_{int}\gamma[\sigma/j]}$$

$$\texttt{Error-int}\ \frac{\gamma(j)\overset{error}{\longmapsto}_{proc}stop}{\gamma\overset{error_j}{\longmapsto}_{int}stop}$$

Fig. 5. Interaction-level rules

Since we have not included a rule for *Jump* statements, they do not produce observable steps. Its execution is hidden in function $next$.

3.2.2. Interacting processes

Interaction among processes is given using the semantics rules at process-level given by $\longmapsto_{proc}$. Two groups (levels) of rules define this interaction. The transition relation $\overset{-}{\longmapsto}_{int}\subseteq \Gamma \times L_{sim} \times (\Gamma \cup \{stop\})$ (Fig. 5) defines the behavior of the instructions whose execution involves more than one process, but without considering the execution mode of the processes. Basically, it defines the rendezvous, the dynamic creation of new processes and `timeout`. The transition relation $\overset{-}{\longmapsto}_{mod}\subseteq \Gamma \times L_{sim} \times (\Gamma \cup \{stop\})$ (Fig. 6) models the *atomic* execution defined by the *Atomic* and *D_step* instructions, and also the execution of the *Unless* statements. All rules share the labels in the set L_{sim}, which extend L_{proc}. Now we present the new labels utilized:

- $SimMode = \cup_{j\in Pid}\{dstep_j, atm_j\}$. The elements of $SimMode$ are used to indicate that the system has executed a $D_Step/Atomic$ instruction.
- $Rend^k = \cup_{j\in Pid}\{rend_j^k\}$, $ARend^k = \cup_{j\in Pid}\{arend_j^k\}$. The elements of $Rend^k$ and $ARend^k$ indicate that a rendezvous instruction has been executed, $k \in Pid$ being the pid of the sender process. The label $arend_j^k$ is used when the sender process k is inside an `atomic` instruction; otherwise $rend_j^k$ is used.
- $Rend = \cup_{k\in Pid} Rend^k$, $ARend = \cup_{k\in Pid} ARend^k$.
- $ProcError = \cup_{j\in Pid}\{error_j\}$. $ProcError$ contains the possible process execution errors.
- $SimEnd = \{ies, end\}$. $SimEnd$ indicates two possible ways of system termination: invalid end state (ies) and correct termination (end).
- $InstL = \cup_{j\in Pid}(\{inst_j : inst \in Basic\} \cup \{else_j, run_j\})$. The elements of $InstL$ are used to indicate the basic instruction $inst$ which has been executed.

Every set of labels defined above encodes possible actions enabling a system to progress, including a reference (by means of a subindex) to the process where they have been executed.

The set of labels in the simulation semantics is

$L_{sim} = SimMode \cup Rend \cup ARend \cup ProcError \cup SimEnd \cup InstL.$

In the labelled transition relations $\longmapsto_{int}$ and $\longmapsto_{mod}$, we use the following notation. Given a system configuration $\gamma \in \Gamma$, $\gamma[\sigma/j]$ denotes the configuration that is equal to γ for all processes, except for process j, whose state has been updated to σ. The notation $\gamma[\sigma_1/j_1, \ldots, \sigma_n/j_n]$ extends this definition to n processes. Taking into account that configurations must remain consistent, changing a process state may imply the update of the states of all system processes.

3.2.3. Transition relation $\longmapsto_{int}$

The following is an explanation of the semantic rules of $\longmapsto_{int}$ given in Fig. 5.

* `Single-int`
  This rule only adds the *pid* of the process executing the basic instruction to the label in the transition relation.
* `Run-int`
  When a process executes *Run*, the system configuration γ is augmented with the initial configuration of the new process instance. We use functions *newPid* and *initP* (defined in Sect. 3.1.5) to create a new process identifier and its initial state. The way identifiers are dynamically created is implementation-dependent.
* `Rend-int`
  This rule says that if the two communicating processes have reached the communication point, the configuration γ is updated with their new states. The key point is how to consider the executability of output and input instructions. This problem is solved by combining the use of the process-level rule `Basic-int` with the *exec* function. As commented above, we always define input/output with rendezvous channels as executable in the *exec* function. Thus, if both processes have reached the synchronization point, `Basic-int` can be directly employed. In addition, we model the fact that when a rendezvous is executed the receiver process takes the execution control by labelling the arrow with the identifier of the receiver process k. Thus, the receiver process never initiates the execution of rendezvous. Note that we also store the sender process identifier as a superindex. We will need this information to model the transfer of control when the rendezvous is inside an *Atomic* instruction in the next two levels. Rules `Rend1-mod`, `Rend2-mod` and `Rend3-mod` will fire this rule when a rendezvous can be carried out.
* `Timeout-int`
  A `timeout` instruction can be executed when no other instruction in the system is enabled.
* `Error-int`
  This rule is employed to pass the errors at process-level to the system level. At this point, only runtime errors, such as arithmetic ones, appear.

3.2.4. Transition relation $\longmapsto_{mod}$

As commented above, $\longmapsto_{mod}$ takes the execution mode of each process into account. To do it, we use function *mode* defined in Sect. 3.1.1. The first three rules represent the modes of execution: d_step, atomic, and interleaving. The following ones model the communication via rendezvous. Finally, an additional rule is employed to specify the correct end of the execution and three rules define the non-valid termination states.

Next, we discuss the rules appearing in Fig. 6.

* `Dst-mod`
  This rule defines the correct execution of a *D_Step* instruction by the process instance j. Note that the current basic instruction executed is hidden by the rule. The transition relation at simulation level will use this rule to execute the instructions inside *D_Step* without interruption. In addition, it is important to note that when the mode of a process is *dst*, we do not use the function *nextexec* which would give priority to escape instructions. The sequence of *D_Step* instructions is executed as a unique instruction even if they are nested in an *Unless* instruction.
* `Atm-mod`, `Ilv-mod`
  These rules define the correct execution of an instruction by the process instance j, considering the case when the instruction executed is nested in an *Unless* instruction. As occurred with *D_Step* when a process is inside an *Atomic* construction, the instruction executed is hidden by the rule. We have excluded the *Rendez* instructions which will be dealt with in the following rules.
* `Rend1-mod`, `Rend2-mod` and `Rend3-mod`
  These three rules are applied when there is an enabled *Rendez* instruction to be executed by process j as sender process. Each rule handles a different situation depending on whether the sender or the receiver process is in *atomic* mode. Rendezvous always transfers the control from the sender to the receiver. That is, when a *Rendez* instruction is started in the sender process, it has two effects: the communication of data and the transfer of control to the recipient. The use of *Rendez* instructions can produce special execution sequences when the sender is inside an *Atomic* instruction. In this case the atomic execution is suspended and the control is transferred to the receiver process. If this process is also inside an *Atomic* instruction, the situation corresponds to

Dst-mod $$\frac{mode(\gamma(j).\sigma_l)=dst, \gamma \overset{inst_j}{\longmapsto}_{int} \gamma'}{\gamma \overset{dstep_j}{\longmapsto}_{mod} \gamma'}$$

Atm-mod $$\frac{L \in nextexec(j,\gamma), mode(L)=atm, \gamma[L/\gamma(j).\sigma_l] \overset{inst_j}{\longmapsto}_{int} \gamma'}{\gamma \overset{atm_j}{\longmapsto}_{mod} \gamma'}$$

Ilv-mod $$\frac{L \in nextexec(j,\gamma), mode(L)=ilv, \gamma[L/\gamma(j).\sigma_l] \overset{inst_j}{\longmapsto}_{int} \gamma'}{\gamma \overset{inst_j}{\longmapsto}_{mod} \gamma'}$$

Rend1-mod $$\frac{(L,k,L') \in nextexec(j,\gamma), mode(L)=atm, \gamma[L/\gamma(j).\sigma_l, L'/\gamma(k).\sigma_l] \overset{rend_k^j}{\longmapsto}_{int} \gamma'}{\gamma \overset{arend_k^j}{\longmapsto}_{mod} \gamma'}$$

Rend2-mod $$\frac{(L,k,L') \in nextexec(j,\gamma), mode(L)=ilv, mode(L')=atm, \gamma[L/\gamma(j).\sigma_l, L'/\gamma(k).\sigma_l] \overset{rend_k^j}{\longmapsto}_{int} \gamma'}{\gamma \overset{atm_k}{\longmapsto}_{mod} \gamma'}$$

Rend3-mod $$\frac{(L,k,L') \in nextexec(j,\gamma), mode(L)=ilv, mode(L')=ilv, \gamma[L/\gamma(j).\sigma_l, L'/\gamma(k).\sigma_l] \overset{rend_k^j}{\longmapsto}_{int} \gamma'}{\gamma \overset{rend_k^j}{\longmapsto}_{mod} \gamma'}$$

End-mod $$\frac{\forall\ k.\gamma(k).\sigma_l \in End, \gamma \not\longmapsto_{int}}{\gamma \overset{end}{\longmapsto}_{mod} stop}$$

Deadlock-mod $$\frac{\exists\ k.\gamma(k).\sigma_l \notin End, \gamma \not\longmapsto_{int}}{\gamma \overset{ies}{\longmapsto}_{mod} stop}$$

Error1-mod $$\frac{\gamma \overset{error_j}{\longmapsto}_{int} stop}{\gamma \overset{error_j}{\longmapsto}_{mod} stop}$$

Error2-mod $$\frac{mode(\gamma(j).\sigma_l)=dst, \gamma \overset{inst_j}{\not\longmapsto}_{int}}{\gamma \overset{error_j}{\longmapsto}_{mod} stop}$$

Fig. 6. Sequence of instructions based on the mode

a typical co-routine, as modelled by the rule Co-rou1-sim at the simulation level. It is also possible that the receiver process is not in atomic mode. In this case, a side effect of the rendezvous is that the sender process losses execution control. Therefore, Rend1-mod models the case (using the label $arend_k^j$) when the sender process is in *atomic* mode. Rend2-mod handles the case (using the label atm_k) when the sender process is not in *atomic* mode, but the receiver process is. Finally, Rend3-mod is applied when both processes are in mode ilv.

* End-mod
  The rule End-mod applies when all the processes have reached instructions labelled as legal end points and none of them can continue the execution. In this case, the system has reached a legal (correct) termination.

* Deadlock-mod, Error1-mod and Error2-mod
  These rules capture all execution errors. The rule Deadlock-mod models the deadlock behavior: no process system can continue the execution, and one of them, at least, is not at a legal *End* label. The rule Error1-mod raises the errors in the lower levels to the high level. Rule Error2-mod applies when a process suspends while executing a *D_Step* instruction.

3.2.5. Observable steps for simulation

The top-level relation $\overset{-}{\longmapsto}_{sim} \subseteq \Gamma \times L_{sim} \times (\Gamma \cup \{stop\})$ defines the granularity of the execution, grouping sentences when using the D_Step and $Atomic$. The relation $\overset{-}{\longmapsto}_{sim}$ hides all process transitions corresponding to these instructions. The configuration $stop$ represents both the legal and illegal termination of the model. The label of the transition relation informs about possible executions errors. Next, we explain the rules in Fig. 7.

$$\texttt{Dst-sim}\ \frac{\gamma \not\longmapsto_{mod} stop, \gamma\ (\overset{dstep_j}{\longmapsto}_{mod})^+\ \gamma'\ \overset{dstep_j}{\not\longmapsto}_{mod}}{\gamma \overset{dstep_j}{\longmapsto}_{sim} \gamma'}$$

$$\texttt{Atm-sim}\ \frac{\gamma \not\longmapsto_{mod} stop, \gamma\ (\overset{atm_j}{\longmapsto}_{mod})^+\ \gamma',\ \gamma' \overset{atm_j}{\not\longmapsto}_{mod}, \forall k.\ \gamma'\ \overset{arend_k^j}{\not\longmapsto}_{mod}}{\gamma \overset{atm_j}{\longmapsto}_{sim} \gamma'}$$

$$\texttt{Co-rou1-sim}\ \frac{\gamma \not\longmapsto_{mod} stop, \exists k.(\ \gamma(\overset{atm_j}{\longmapsto}_{mod})^*\ \gamma'\ \overset{arend_k^j}{\longmapsto}_{mod}\ \gamma'' \overset{atm_k}{\longmapsto}_{sim}\ \gamma''')}{\gamma \overset{atm_j}{\longmapsto}_{sim} \gamma'''}$$

$$\texttt{Co-rou2-sim}\ \frac{\gamma \not\longmapsto_{mod} stop, \exists k.(\ \gamma(\overset{atm_j}{\longmapsto}_{mod})^*\ \gamma'\ \overset{arend_k^j}{\longmapsto}_{mod}\ \gamma'' \overset{atm_k}{\not\longmapsto}_{sim})}{\gamma \overset{atm_j}{\longmapsto}_{sim} \gamma''}$$

$$\texttt{Ilv-sim} \frac{\gamma \not\longmapsto_{mod} stop, \gamma \overset{inst}{\longmapsto}_{mod} \gamma', inst \in InstL \cup Rend}{\gamma \overset{inst}{\longmapsto}_{sim} \gamma'}$$

$$\texttt{Stop-sim}\ \frac{\gamma \overset{inst}{\longmapsto}_{mod} stop}{\gamma \overset{inst}{\longmapsto}_{sim} stop}$$

Fig. 7. Observable steps in simulation

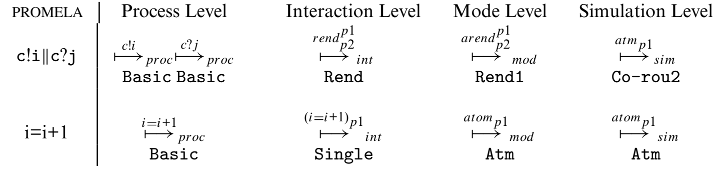
* `Dst-sim`
  The rule `Dst-sim` defines how to group single deterministic steps in the systems to be considered as a unique simulation step. The deterministic execution (D_Step) is executed without interruption. The relation $(\overset{dstep_j}{\longmapsto}_{mod})^+$ indicates the transitive closure of $\overset{dstep_j}{\longmapsto}_{mod}$; that is, given a process j, the rule $\overset{dstep_j}{\longmapsto}_{mod}$ is applied once or more times until this process cannot proceed with this rule.

* `Atm-sim`
  In general, the instructions inside $Atomic$ that can be executed without suspension are considered as a unique simulation step. There is no difference between the end of the $Atomic$ and the suspension. If a *Rendez* statement is the first instruction of the $Atomic$ sequence, this rule may be applied only if the process in execution j is the receiver, because the rule `Atm-mod` labels the transition with atm_j. In the other case, that is, when the process j is the sender, the rule `Rend1-mod` labels the transition with $arend_k^j$ (k being the receiver process) and therefore `Atm-sim` cannot be applied.

* `Co-rou1-sim` and `Co-rou2-sim`
  The rule `Co-rou1-sim` models the case in which the recipient of the rendezvous can continue the atomic execution. Note that the arrow $\overset{atm_j}{\longmapsto}_{sim}$ is labelled with the process which started the $Atomic$ execution. In addition, the rule `Co-rou2-sim` applies when the receiver finishes the $Atomic$ instruction because this construct has either ended or suspended. Note that now the relation $(\overset{atm_k}{\longmapsto}_{mod})^*$ (the reflexive-transitive closure of $\overset{atm_k}{\longmapsto}_{mod}$) is necessary because the process that has just gained control can immediately lose it (transition $(\overset{atm_k}{\longmapsto}_{mod})$ may be not applied).

* `Ilv-sim`
  The rule `Ilv2` defines other steps of the system; that is, when a process executes an instruction in interleaving mode. The execution is defined just as in the lower level $\overset{-}{\longmapsto}_{mod}$.

* `Stop-sim`
  The last rule produces the end of the execution. This can happen when all processes have correctly finished their execution, when the system has a deadlock, and also when there has been an execution error in some process. The kind of termination is given by the label. We apply this rule first, so the execution can only proceed if no error has been produced. The rest of the rules discard termination before being applied.



PROMELA	Process Level	Interaction Level	Mode Level	Simulation Level
c!i‖c?j	$\overset{c!i}{\longmapsto}_{proc}\overset{c?j}{\longmapsto}_{proc}$ Basic Basic	$\overset{rend^{p1}_{p2}}{\longmapsto}_{int}$ Rend	$\overset{arend^{p1}_{p2}}{\longmapsto}_{mod}$ Rend1	$\overset{atm_{p1}}{\longmapsto}_{sim}$ Co-rou2
i=i+1	$\overset{i=i+1}{\longmapsto}_{proc}$ Basic	$\overset{(i=i+1)_{p1}}{\longmapsto}_{int}$ Single	$\overset{atom_{p1}}{\longmapsto}_{mod}$ Atm	$\overset{atom_{p1}}{\longmapsto}_{sim}$ Atm

Fig. 8. Applying the semantic rules

3.3. Generalized semantics for simulation

Using the transition systems defined above, we now define the semantics for simulation of PROMELA models.

Definition 1 (Generalized semantics). We define the generalized semantics of a PROMELA model M with respect to functions τ and φ as the set $Gen(M, \tau, \varphi)$ of all maximal (possibly infinite) sequences of configurations generated by using $\longmapsto_{sim}$ from the initial configuration $initial(M, \tau, \varphi)$.

Different definitions for φ and τ give us different interpretations $Gen(M, \tau, \varphi)$ for a given model M. If φ^P and τ^P denote the standard definition of these functions (the one directly implemented by SPIN), then $Gen(M, \tau^P, \varphi^P)$ represent the standard trace-based semantics of PROMELA.

A definition of these two functions φ^P and τ^P may be found in the Appendix.

3.4. Example

We use the short code in Fig. 2 to illustrate the use of the semantic rules considering the standard semantics $Gen(M, \tau^P, \varphi^P)$ of PROMELA. As commented above, functions τ^P and φ^P implement the well-known meaning for the basic instructions in PROMELA (at the level used in the example, they are quite close to programming languages like C). This simple model contains three of the most complex PROMELA constructs, namely `atomic`, rendezvous and `unless`. Note that the whole code is labelled as explained in previous sections.

The use of some semantic rules for this model is shown in Fig. 8 where it is assumed that $j = 1$. The first column (PROMELA) contains the steps produced by the simulator SPIN. The other columns, from right to left, contain the rules that define the execution of these steps at the highest level ($\overset{-}{\longmapsto}_{sim}$), the mode level ($\overset{-}{\longmapsto}_{mod}$), the intermediate level ($\overset{-}{\longmapsto}_{int}$) and the process level ($\overset{-}{\longmapsto}_{proc}$), respectively. Each row contains the information about one simulation step, including the name of each rule below each arrow (we have omitted the suffix indicating the level of the rule applied to simplify the figure).

Figure 9 shows an execution sequence of the same model. The first column of this figure shows the identifier of the process in execution. The second column shows the set of instructions executed. Note that when the two processes execute the rendezvous, both appear in the first column. In addition, when process `p2` executes an *Atomic*, the sequence of instructions executed is shown on the second column.

The third column shows the sequence of configurations produced by applying the semantic rules at the simulation level. The initial configuration of the model, given by the $initial$ function, can be represented as $\gamma_0 = ((p1, \sigma_1), (p2, \sigma_2))$, where $\sigma_1 = \langle i = 1, L1, (L1, ilv)\rangle$ and $\sigma_2 = \langle j = 1, L8, (L11 \cdot L8, ilv)\rangle$ are the states of processes `p1` and `p2`, which are given as $\sigma = \langle \sigma_e, \sigma_l, (esc(\sigma_l), mode(\sigma_l))\rangle$ (we have added the implicit information $(esc(\sigma_l), mode(\sigma_l))$ for clarity). Note that one transition at the simulation level involves the application of several rules in the lower levels, as shown in Fig. 8.

4. Abstract simulation

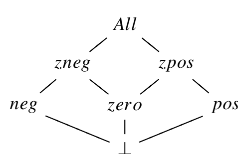
In this section, we present the *data abstraction* approach for abstracting PROMELA introduced in [GaM99, GMM02].

In order to abstract a model M, we must replace the original data domain with descriptions that are elements of some simpler abstract domain. Consequently, the abstraction of a model involves modifying (abstracting) the states of processes which include local and global variables and channels and the operations that work with them. Proving the correctness of the transformation process involves analyzing the relationship between the behavior of the original model and the abstract one.

Process(es)	Executed instruction(s)	Simulation steps
p1	`(i < N)`	$\gamma_0 = ((p1, \langle i = 1, L1, (L1, ilv)\rangle), (p2, \langle j = 1, L8, (L11 \cdot L8, ilv)\rangle)) \overset{(i<N)_{p1}}{\longmapsto}_{sim}$
p1 ∥ p2	`c!i ∥ c?j`	$\gamma_1 = ((p1, \langle i = 1, L4, (L4, atm)\rangle), (p2, \langle j = 1, L8, (L11 \cdot L8, ilv)\rangle)) \overset{atm_{p1}}{\longmapsto}_{sim}$
p2	`printf(j)`	$\gamma_2 = ((p1, \langle i = 1, L5, (L5, atm)\rangle), (p2, \langle j = 1, L9, (L11 \cdot L9, ilv) >\rangle) \overset{printf(j)_{p2}}{\longmapsto}_{sim}$
p1	`i = i + 1`	$\gamma_3 = ((p1, \langle i = 1, L5, (L5, atm)\rangle), (p2, \langle j = 1, L8, (L11 \cdot L8, ilv)\rangle >)) \overset{atm_{p1}}{\longmapsto}_{sim}$
p1	`i < N`	$\gamma_4 = ((p1, \langle i = 2, L1, (L1, ilv)\rangle), (p2(1), \langle j = 1, L8, (L11 \cdot L8, ilv)\rangle)) \overset{(i<N)_{p1}}{\longmapsto}_{sim}$
p1 ∥ p2	`c!i ∥ c?j`	$\gamma_5 = ((p1, \langle i = 2, L4, (L4, atm)\rangle)), (p2, \langle j = 1, L8, (L11 \cdot L8, ilv)\rangle)) \overset{atm_{p1}}{\longmapsto}_{sim}$
p2	`j%2 == 0`	$\gamma_6 = ((p1, \langle i = 2, L5, (L5, atm)\rangle), (p2, \langle j = 2, L9, (L11 \cdot L9, ilv)\rangle)) \overset{((j\%2)==0)_{p2}}{\longmapsto}_{sim}$
p2	`printf(''Even'');j = 1`	$\gamma_7 = ((p1, \langle i = 2, L5, (L5, atm)\rangle), (p2, \langle j = 2, L12, (L12, atm)\rangle)) \overset{atm_{p2}}{\longmapsto}_{sim}$
p1	`i = i + 1`	$\gamma_8 = ((p1, \langle i = 2, L5, (L5, atm)\rangle), (p2, \langle j = 1, L8, (L11 \cdot L8, ilv)\rangle)) \overset{atm_{p1}}{\longmapsto}_{sim}$
...	...	$\gamma_9 = ((p1, \langle i = 3, L1, (L1, ilv)\rangle), (p2, \langle j = 1, L8, (L11 \cdot L8, ilv)\rangle)) \overset{\cdots}{\longmapsto}_{sim} \cdots \cdots \cdots$

Fig. 9. A concrete simulation

if $n > 0$ then $\alpha_v(n) = pos$.
if $n = 0$ then $\alpha_v(n) = zero$.
if $n < 0$ then $\alpha_v(n) = neg$.



if $n\%2 = 0$ then $eo(n) = e$.
if $n\%2! = 0$ then $eo(n) = o$.

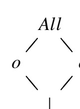


Fig. 10. Two abstractions of Z

4.1. Abstracting data and configurations

The data abstraction approach implemented in abstract model checking used to be a state-to-state abstraction, which means that normally the abstraction process is defined making use of an *abstraction function* which transforms the original data into abstractions (descriptions/approximations). In our case, data are the variables and channels, i.e., the set Var, declared in the model to be abstracted. In order to reduce the data domain of a given variable v, we have to define a new abstract domain $Const_v^\alpha$ simpler than the original one, $Const_v$, and an abstraction function $\alpha_v : Const_v \to Const_v^\alpha$ which relates each value in the original domain with its abstraction.

For instance, assume that variable v takes integer values, that is, $Const_v = Z$, and that we want to reduce this domain to the simpler set $Z^\alpha = \{neg, zero, pos, zneg, zpos, all\}$ by means of the abstraction function $\alpha_v : Z \to Z^\alpha$ defined in the left side of Fig. 10. Note that in the definition no integer value is initially associated to $zneg$, $zpos$ and all. These abstract values could be taken by variable v during execution due to the loss of information inherent in the abstraction. For instance, if $v = pos$ at a certain point and instruction $v = v - 1$ is executed, there are two possible results for v, pos and $zero$, which is represented by the abstract value $zpos$. This means abstract values are partially ordered with respect to the degree of precision. In the second column of Fig. 10 we find the lattice Z^α with its partial order. The bigger elements in the lattice correspond to the less precise ones.

Therefore, abstracting a model consists in defining a family of abstraction functions $\Lambda = \{\alpha_v\}_{v \in Var}$ such as $\alpha_v : Const_v \to Const_v^\alpha$, one for each model variable $v \in Var$ to be abstracted. In order to simplify the exposition we assume that

- $Const_v^\alpha$ is a partially ordered set (poset) of approximated data[2], representing the new domain where v will take its values during the abstract interpretation. The ordering relation $\leqslant_v^\alpha$ defined over $Const_v^\alpha$ gives the precision of the representation. As shown in the previous example, $c_1^\alpha \leqslant_v^\alpha c_2^\alpha$ means that the abstract value c_1^α is more precise than c_2^α.
- the map $\alpha_v : Const_v \to Const_v^\alpha$ defines the abstraction of each concrete value $d \in Const_v$ onto the most precise approximation $\alpha_v(d) \in Const_v^\alpha$. If we do not want to abstract $Const_v$, it is sufficient to define $Const_v^\alpha = Const_v$, $\leqslant_v^\alpha \equiv =$, and α_v as the identity function.

The family of abstractions Λ determines the abstraction of the environments $\alpha_e : Env \to Env^\alpha$ where:

[2] Abstract domains are usually finite lattices.

- $Const^\alpha = \bigcup_v Const_v^\alpha$ and $Env^\alpha = \{\sigma_e^\alpha | \sigma_e^\alpha : Var \to Const^\alpha\}$
- Env^α is a poset with the partial order $\leqslant_e^\alpha$ defined as:
$\forall \sigma_{e1}^\alpha, \sigma_{e2}^\alpha \in Env^\alpha.(\sigma_{e1}^\alpha \leqslant_e^\alpha \sigma_{e2}^\alpha \text{ iff } \forall v \,.\, \sigma_{e1}^\alpha(v) \leqslant_v^\alpha \sigma_{e2}^\alpha(v))$
- $\alpha_e(\sigma_e)(v) = \alpha_v(\sigma_e(v))$, for each $v \in Var$.

Since semantic rules handle configurations, we have to extend the environment abstraction to states and configurations. Thus, following the previous discussion, we may construct the set of abstract states as $State^\alpha = Env^\alpha \times Label$. $State^\alpha$ is a poset considering the partial order relation $\leqslant_s^\alpha$ defined as follows. Given two abstract states $\sigma_1^\alpha = \langle \sigma_{e1}^\alpha, \sigma_{l1} \rangle$ and $\sigma_2^\alpha = \langle \sigma_{e2}^\alpha, \sigma_{l2} \rangle$,

$$\sigma_1^\alpha \leqslant_s^\alpha \sigma_2^\alpha \iff \sigma_{e1}^\alpha \leqslant_e^\alpha \sigma_{e2}^\alpha, \sigma_{l1} = \sigma_{l2}$$

In addition, we may extend abstraction function α_e to states in the natural way: $\alpha_s : State \to State^\alpha$ is defined as $\alpha_s(\langle \sigma_e, \sigma_l \rangle) = \langle \alpha_e(\sigma_e), \sigma_l \rangle$.

Similarly, we may define the poset of abstract configurations as $\Gamma^\alpha = \{\gamma^\alpha | \gamma^\alpha : Pid \to State^\alpha\}$. The partial order $\leqslant_s^\alpha$ can also be extended to $\leqslant_\gamma^\alpha$ in order to relate abstract configurations as follows:

$$\gamma_1^\alpha \leqslant_\gamma^\alpha \gamma_2^\alpha \iff \forall j \in Pid.\gamma_1^\alpha(j) \leqslant_s^\alpha \gamma_2^\alpha(j)$$

In addition, α_s may be extended to the abstraction function α_γ as $\alpha_\gamma(\gamma)(j) = \alpha_s(\gamma(j))$, for each $j \in Pid$. Finally, given $t = \gamma_0 \stackrel{l^0}{\longmapsto}_{sim} \cdots$ and $t^\alpha = \gamma_0^\alpha \stackrel{l^0}{\longmapsto}_{sim} \cdots$, we write $\alpha(t)$ to denote the abstract trace $\alpha_\gamma(\gamma_0) \stackrel{l^0}{\longmapsto}_{sim} \cdots$, and write $\alpha(t) \leqslant^\alpha t^\alpha$ when $\forall m \geqslant 0.\alpha_\gamma(\gamma_m) \leqslant_\gamma^\alpha \gamma_m^\alpha$.

Once data have been abstracted, in order to realize the abstract interpretation of a model, we must also define an approximation of the language operations which involve data, that is, the instructions defined as $Basic$ in Sect. 2 to adapt them to the new data representation. We define this abstract behavior in terms of the functions $\tau^\alpha : BExp \times Env^\alpha \to \{false, true\}$ and $\varphi^\alpha : Basic \times Env^\alpha \to Env^\alpha$. Note that the domain Env used in the definition of these operations has been replaced by Env^α. Therefore, considering these functions and Definition 1, $Gen(M, \tau^\alpha, \varphi^\alpha)$ provides an abstract semantics for model M.

Example 2. For example, let function $eo : Z \to \{o, e, all\}$ be defined as shown in Fig. 10. This function (even/odd) allows us to define the family of abstractions $\Lambda = \{\alpha_i, \alpha_j\}$ for the model in Fig. 2, where $\alpha_i = \alpha_j = eo$ (that is, we abstract variables i and j in the model). The new abstract domain for these variables is $(Const_i^\alpha, \leqslant_i^\alpha) = (Const_j^\alpha, \leqslant_j^\alpha) = (\{o, e, all\}, \leqslant^\alpha)$ where the order relation is shown at the right side of Fig. 10. Functions τ^α and φ^α may be defined as follows. Given $\sigma_e^\alpha \in Env^\alpha$

- $\tau^\alpha(i < N, \sigma_e^\alpha) = true$
- $\tau^\alpha(i >= N, \sigma_e^\alpha) = true$
- $\tau^\alpha(j\%\,2 == 0, \sigma_e^\alpha) = (\sigma_e^\alpha(j) == e)$
- $\varphi^\alpha(i = 1, \sigma_e^\alpha) = \sigma_e^\alpha[o/i]$.[3]
- $\varphi^\alpha(j = 1, \sigma_e^\alpha) = \sigma_e^\alpha[o/j]$.
- $\varphi^\alpha(i = i + 1, \sigma_e^\alpha) = \sigma_e^\alpha[e/i]$ iff $\sigma_e^\alpha(i) = o$.
- $\varphi^\alpha(i = i + 1, \sigma_e^\alpha) = \sigma_e^\alpha[o/i]$ iff $\sigma_e^\alpha(i) = e$.
- $\varphi^\alpha(inst, \sigma_e^\alpha) = \sigma_e^\alpha$, for the rest of basic instructions $inst \in Basic$.

Figure 11 shows a simulation of the model $Even/Odd$ in Fig. 2 when using the new functions τ^α and φ^α to apply the semantic rules at process level as defined in Fig. 4. This trace is an abstraction of the trace shown in Fig. 9. In each configuration of the abstract sequence, variables i and j have been abstracted using the abstraction function eo. Note that Fig. 11 only differs from Fig. 9 in the values of these variables. In addition, in this figure the configurations γ_0 and γ_9 coincide. This cycle is detected by the verifier and the analysis of this trace of execution ends, saving time and memory for verification.

We can arbitrarily define the φ^α and τ^α functions, but the interest of the approach is in preserving some correction properties relating a given semantics $Gen(M, \tau, \varphi)$ with the abstract semantics $Gen(M, \tau^\alpha, \varphi^\alpha)$. In the next section, we study the conditions that τ^α and φ^α must verify for $Gen(M, \tau^\alpha, \varphi^\alpha)$ to achieve correctness. From

[3] Recall that notation $\sigma[c/v]$ represents the function which is equal to σ for all the elements in its domain except for v that is substituted by c.

`Process(es)`	`Executed instruction(s)`	`Simulation steps`
p1	`(i < N)`	$\gamma_0^\alpha = ((p1, \langle i = o, L1, (L1, ilv)\rangle), (p2, \langle j = o, L8, (L11 \cdot L8, ilv)\rangle)) \stackrel{(i<N)_{p1}}{\longmapsto}_{sim}$
p1 ∥ p2	`c!i ∥ c?j`	$\gamma_1^\alpha = ((p1, \langle i = o, L4, (L4, atm)\rangle), (p2, \langle j = o, L8, (L11 \cdot L8, ilv)\rangle)) \stackrel{atm_{p1}}{\longmapsto}_{sim}$
p2	`printf(j)`	$\gamma_2^\alpha = ((p1, \langle i = o, L5, (L5, atm)\rangle), (p2, \langle j = o, L9, (L11 \cdot L9, ilv) >\rangle) \stackrel{printf(j)_{p2}}{\longmapsto}_{sim}$
p1	`i = i + 1`	$\gamma_3^\alpha = ((p1, \langle i = o, L5, (L5, atm)\rangle), (p2, \langle j = o, L8, (L11 \cdot L8, ilv)\rangle >)) \stackrel{atm_{p1}}{\longmapsto}_{sim}$
p1	`i < N`	$\gamma_4^\alpha = ((p1, \langle i = e, L1, (L1, ilv)\rangle), (p2(1), \langle j = o, L8, (L11 \cdot L8, ilv)\rangle)) \stackrel{(i<N)_{p1}}{\longmapsto}_{sim}$
p1 ∥ p2	`c!i ∥ c?j`	$\gamma_5^\alpha = ((p1, \langle i = e, L4, (L4, atm))\rangle), (p2, \langle j = o, L8, (L11 \cdot L8, ilv)\rangle)) \stackrel{atm_{p1}}{\longmapsto}_{sim}$
p2	`j%2 == 0`	$\gamma_6^\alpha = ((p1, \langle i = e, L5, (L5, atm)\rangle), (p2, \langle j = e, L9, (L11 \cdot L9, ilv)\rangle)) \stackrel{((j\%2)==0)_{p2}}{\longmapsto}_{sim}$
p2	`printf(''Even'');j = 1`	$\gamma_7^\alpha = ((p1, \langle i = e, L5, (L5, atm)\rangle), (p2, \langle j = e, L12, (L12, atm)\rangle)) \stackrel{atm_{p2}}{\longmapsto}_{sim}$
p1	`i = i + 1`	$\gamma_8^\alpha = ((p1, \langle i = e, L5, (L5, atm)\rangle), (p2, \langle j = o, L8, (L11 \cdot L8, ilv)\rangle)) \stackrel{atm_{p1}}{\longmapsto}_{sim}$
...	...	$\gamma_9^\alpha = ((p1, \langle i = o, L1, (L1, ilv)\rangle), (p2, \langle j = o, L8, (L11 \cdot L8, ilv)\rangle)) \stackrel{\cdots}{\longmapsto}_{sim} \cdots \cdots \cdots$

Fig. 11. An abstract simulation

a practical point of view, semantics $Gen(M, \tau, \varphi)$ used to be the standard PROMELA semantics $Gen(M, \tau^P, \varphi^P)$. However, the theoretical framework allows for considering any initial semantics, even those corresponding to previously abstracted models. This allows for a successive abstraction process.

4.2. Soundness of the abstraction

An abstract model $Gen(M, \tau^\alpha, \varphi^\alpha)$ is correct if it can simulate all execution traces of the original model ($Gen(M, \tau, \varphi)$). Simulation is formalized by making use of the abstraction function α and the partial ordering $\leqslant^\alpha$ defined over the concrete traces in $Gen(M, \tau, \varphi)$. In this section, we impose some *correctness conditions* when defining functions τ^α and φ^α in order to correctly simulate the behavior of the model when using the original functions τ and φ. Abstraction process directly affects the evaluation of Boolean expressions. In particular, the negation of Boolean expressions is especially modified when abstracting a model. Therefore, from now on, we assume that no *If*/*Do* instruction in the model to be abstracted has an `else` branch. This is not a real restriction since each `else` branch may be substituted by an ordinary branch having as a guard the negation of the guards of the other branches.

The semantic structure presented in Sect. 3 facilitates the correct abstraction of models, since it constrains the issue of correctness to comparing the concrete and abstract versions of functions τ and φ. An abstract model is correct if it is an *over*-approximation of the original (concrete) one, that is, if each trace produced by the concrete model is simulated/abstraced by a trace in the abstract model. The following definition states the conditions assuring that the abstraction of a model is correct.

Definition 2 (Correctness conditions). Given a family of abstractions Λ, we say that τ^α and φ^α are α-*correct* with respect to τ and φ, when the following conditions hold:

Mτ τ^α is monotonic wrt its second argument. Given $b \in BExp$
$\forall \sigma_{e1}^\alpha, \sigma_{e2}^\alpha \in Env^\alpha$:
if $\sigma_{e1}^\alpha \leqslant_e^\alpha \sigma_{e2}^\alpha$ *then* $\tau^\alpha(b, \sigma_{e1}^\alpha) \Rightarrow \tau^\alpha(b, \sigma_{e2}^\alpha)$

Mφ φ^α is monotonic wrt its second argument. Given $inst \in Basic$
$\forall \sigma_{e1}^\alpha, \sigma_{e2}^\alpha \in Env^\alpha$:
if $\sigma_{e1}^\alpha \leqslant_e^\alpha \sigma_{e2}^\alpha$ *then* $\varphi^\alpha(inst, \sigma_{e1}^\alpha) \leqslant_e^\alpha \varphi^\alpha(inst, \sigma_{e2}^\alpha)$.

LCτ Given $b \in BExp$, $\sigma_e \in Env$ and $\sigma_e^\alpha \in Env^\alpha$ such that $\alpha_e(\sigma_e) \leqslant_e^\alpha \sigma_e^\alpha$: $\tau(b, \sigma_e) \Rightarrow \tau^\alpha(b, \sigma_e^\alpha)$.

LCφ Given $inst \in Basic$, $\sigma_e \in Env$ and $\sigma_e^\alpha \in Env^\alpha$ such that $\alpha_e(\sigma_e) \leqslant_e^\alpha \sigma_e^\alpha$:
$\alpha_e(\varphi(inst, \sigma_e)) \leqslant_e^\alpha \varphi^\alpha(inst, \sigma_e^\alpha)$.

Cτ Given $b \in BExp$ and $\sigma_e^\alpha \in Env^\alpha$: *if* $\tau^\alpha(b, \sigma_e^\alpha)$ *then* $\exists \sigma_e \in Env$:
$\alpha_e(\sigma_e) \leqslant_e^\alpha \sigma_e^\alpha$ *and* $\tau(b, \sigma_e)$.

Conditions $M\tau$ and $M\varphi$ assure that τ^α and φ^α preserve the loss of information during the abstraction process. Conditions $LC\tau$ and $LC\varphi$ assure that everything that can happen in the execution of the original model is *locally* approximated by the abstract execution. The condition $LC\tau$ is operationally important because the non-satisfaction of a Boolean expression in PROMELA implies suspension. Thus, $LC\tau$ establishes that an abstract execution

suspends at a given program point because of non-satisfaction of a Boolean expression only if every concrete execution also suspends at that point. Finally, condition $C\tau$ assures that the evaluation of an abstract test is *false* only if it is known that during any standard execution of the model the evaluation of the test is never true. Since false tests are not executable, this means that the abstract interpretation of the model does not unnecessarily add execution traces. Note that $M\tau$ may be deduced from $LC\tau$ and $C\tau$, but we explicitly define this condition for clarity.

An interesting point is that τ^α satisfying these correctness conditions will be used in the next section to introduce the over-approximation method for abstract model checking. For this purpose, we use the following relation between τ^α and τ. Given $b \in BExp$, and $\sigma_e^\alpha \in Env^\alpha$, function τ^α is defined as

$$\tau^\alpha(p, \sigma_e^\alpha) = \bigvee_{\{\sigma_e \in Env.\alpha(\sigma_e) \leqslant_e^\alpha \sigma_e^\alpha\}} \tau(p, \sigma_e) \qquad (Over)$$

Correctness conditions assure that for each trace t in the original model, there exists a trace t^α in the abstract model such that t^α is an abstract simulation of t, that is, such that $\alpha(t) \leqslant^\alpha t^\alpha$. Since abstract models are over-approximations, finite traces $\gamma_0 \overset{lb^0}{\longmapsto}_{sim} \cdots \overset{lb^m}{\longmapsto}_{sim} stop$ produced by the original model which end with a label $lb^m \in ProcError \cup \{ies\}$ indicating an execution error or deadlock may be not simulated in the abstract models. In the following theorem these traces are called *erroneous*. Definition 6 in Sect. 5.3 extends this notion with other kinds of errors. In addition, we assume that the original model does not include `timeout` instructions. The reason for this will be commented in the proof of the theorem.

Theorem 1. Let Λ be a family of abstractions for the model M. Let $Gen(M, \tau, \varphi)$ and $Gen(M, \tau^\alpha, \varphi^\alpha)$ be two semantics of the model M, τ^α and φ^α being α-*correct* with respect to τ and φ. Then, for each non erroneous trace $t \in Gen(M, \tau, \varphi)$, an abstract non erroneous sequence $t^\alpha \in Gen(M, \tau^\alpha, \varphi^\alpha)$ exists, such that $\alpha(t) \leqslant^\alpha t^\alpha$.

Proof. The proof is based on the following assertions.

1. Abstraction does not modify the process program counters.
2. Given two states for a process $\sigma = \langle \sigma_e, \sigma_l \rangle \in State$ and $\sigma^\alpha = \langle \sigma_e^\alpha, \sigma_l \rangle \in State^\alpha$ such that $\alpha_s(\sigma) \leqslant_s^\alpha \sigma^\alpha$, then $exec(\sigma_l, \sigma_e) \Rightarrow exec(\sigma_l, \sigma_e^\alpha)$. To prove this assertion, it suffices to use condition $\mathbf{LC}\tau$, because function *exec* is basically defined using τ (see Appendix).
3. Given two process states $\sigma_1, \sigma_2 \in State$ such that $\sigma_1 \overset{lb}{\longmapsto}_{proc} \sigma_2$, and an abstract state $\sigma_1^\alpha \in State^\alpha$ verifying that $\alpha_s(\sigma_1) \leqslant_s^\alpha \sigma_1^\alpha$, there exists an abstract state σ_2^α such that (a) $\alpha_s(\sigma_2) \leqslant_s^\alpha \sigma_2^\alpha$ and (b) $\sigma_1^\alpha \overset{lb}{\longmapsto}_{proc} \sigma_2^\alpha$. To prove this assertion, we use points 1 and 2 above and Conditions $\mathbf{M}\varphi$ and $\mathbf{LC}\varphi$.

The soundness of the abstraction at simulation-level may be inductively proved considering that initially $\alpha_\gamma(initial(M, \tau, \varphi)) \leqslant_\gamma^\alpha initial(M, \tau^\alpha, \varphi^\alpha)$, and using the soundness of the abstraction at process-level given by point 3 above. Note that abstraction process does not preserve the *invalid end states* or `timeout` condition, because in general given $\sigma \in State$ and $\sigma^\alpha \in State^\alpha$ such that $\alpha_s(\sigma) \leqslant_s^\alpha \sigma^\alpha$, then $\neg exec(\sigma_l, \sigma_e) \not\Rightarrow \neg exec(\sigma_l, \sigma_e^\alpha)$. □

Given an abstraction, including τ^α and φ^α, it is possible to mechanize the analysis of the correctness conditions with the help of a theorem prover like PVS (see [GrS97] for a related work). Furthermore, the analysis is independent of the model to be abstracted, so we could put a description of α in an *abstraction library* to be employed in the future. This is the approach followed in the tool αSPIN [αSPIN], although the current abstraction library is not automatically analyzed for correctness.

In the following sections we always assume that the semantic functions τ and φ used to build the concrete and abstract semantics verify the hypothesis of the previous theorem.

5. Extending the generalized semantics for model checking

In this section, we study the extension of the generalized semantics presented in Sect. 3 to include the PROMELA elements for verification. This extended semantics preserves functions τ and φ as parameters, which means that it may also be utilized when using abstract interpretations of the model to be verified. In the next section, we will study how to interpret the results produced by SPIN when analyzing abstract models.

Model checking techniques allow us to decide whether one or all executions produced by a model of a system satisfy a temporal logic formula. By default, given a temporal formula, SPIN translates it into an automaton that represents an undesirable behavior (which is claimed to be impossible). Then, verification consists of an

exhaustive exploration of the state space searching for executions that satisfy the acceptance conditions of the automaton. If such an execution exists, then the tool reports it as a counterexample for the property. If the model is explored and a counterexample is not found, then the model satisfies the property as a *universal property*. The same verification scheme can be employed to check whether a formula cannot be satisfied by any path (*refutation of existential properties*).

The automata obtained from temporal formulas are included in models as the special process *never* (also referred to as *never claim*). The execution of this process must be interleaved with each execution step of the system. Process *never* is automatically constructed by the tool following the PROMELA syntax with some restrictions: only tests on global variables and channels are allowed, along with the *If/Do/Jump* constructors. In addition, process *never* may contain some special test on the labels of the model processes described below.

Besides process *never claim*, verification in SPIN also comprises checking erroneous cycles which are characterized in Sect. 5.2.

5.1. New rules for verification

In order to define the verification semantics of PROMELA, we employ the same approach and reuse the simulation-oriented SOS rules as follows. Verification semantics comprises three levels of rules, the same two functions τ and φ, and the definition of cycles (acceptance and non-progress). The top level transition relation $\overset{-}{\longmapsto}_{ver}$ defines the synchronous execution of the *never* automaton with the rest of the system, and the ways of finding erroneous execution. This level employs the $\overset{-}{\longmapsto}_{sim}$ relation when a system step is required. The second level ($\overset{-}{\longmapsto}_{nev}$) is included to deal with the different kinds of *never* instructions (mainly the instructions to inspect process counters). Finally, the process level is the same as in the simulation mode (the *never* process is basically executed as a normal process).

5.1.1. Rules for never claim

In what follows, we use $n \in Pid$ as a special identifier for the *never claim* process. We assume that the state of this process contains the usual information in σ, including two extra predefined variables in σ_e. The `_last` variable stores the identifier of the process that executed the last instruction (excluding the *never claim* process). The `np_` Boolean variable represents whether the current execution is included in a *non-progress cycle*. Variable `np_` is not modelled by the semantics. *Non-progress cycles* will be described in the following section. In addition, if *never claim* is present in M, $initial(M, \tau, \varphi)$ initializes this state. This representation allows us to employ the same rules used for standard PROMELA processes, as shown in Fig. 12.

The transition relation $\overset{-}{\longmapsto}_{nev} \subseteq \Gamma \times L_{proc} \times (\Gamma \cup \{stop\})$ is used to control the steps in the $never\ claim$. Each step can provoke the suspension, termination or evolution of the process. The evolution is due to the successful evaluation of a Boolean expression including the special Boolean expressions that contain references to the process counters in the system ($pc_value(pid) == L$, $enabled(pid)$). We denote expression $process_type[j]@L$ with $pc_value(j) == L$. We assume that the set of labels L_{proc} (defined in Sect. 3.2.1) includes these two special instructions, and also that they are always executable. Note that since *never claim* has a restricted syntax, the transition relations $\longmapsto_{int}$, $\longmapsto_{mod}$ and $\longmapsto_{sim}$ are not necessary to model it.

The $\overset{-}{\longmapsto}_{nev}$ relation in Fig. 12 is defined as follows.

* `Basic-nev`
  This rule is used when the Boolean expression to be executed by *never claim* does not contain a test on the executability or the process counter of another particular process.
* `Enabled-nev` and `Pcvalue-nev`
  These rules are used when the Boolean expression to be executed contains a test on the executability of a particular process or a test on its program counter.
* `End-nev`
  This rule represents the case where *never claim* finishes its code.

`Basic-nev`	$\dfrac{\gamma(n)\overset{inst}{\longmapsto}_{proc}\sigma, inst\notin\cup_j\{pc_value(j), enabled(j)\}}{\gamma\overset{inst_n}{\longmapsto}_{nev}\gamma[\sigma/n]}$	`End-nev`	$\dfrac{\gamma(n).\sigma_l\in End}{\gamma\overset{end}{\longmapsto}_{nev}stop}$
`Enabled-nev`	$\dfrac{\gamma(n)\overset{enabled(j)}{\longmapsto}_{proc}\sigma, \gamma\overset{inst_j}{\longmapsto}_{int}\gamma'}{\gamma\overset{enabled(j)_n}{\longmapsto}_{nev}\gamma[\sigma/n]}$	`Pcvalue-nev`	$\dfrac{\gamma(n)\overset{pc_value(j)=L}{\longmapsto}_{proc}\sigma, \gamma(j).\sigma_l=L}{\gamma\overset{(pc_value(j)=L)_n}{\longmapsto}_{nev}\gamma[\sigma/n]}$

Fig. 12. Never rules

`Discard-ver`	$\dfrac{\gamma(n)\neq\perp, \gamma\not\longmapsto_{nev}}{\gamma\overset{discard}{\longmapsto}_{ver}stop}$	`Err-ver`	$\dfrac{\gamma[\perp/n]\overset{inst}{\longmapsto}_{sim}stop,\ inst\in ProcError\cup\{ies\}}{\gamma\overset{inst}{\longmapsto}_{ver}stop}$
`End-ver`	$\dfrac{\gamma\overset{end}{\not\longmapsto}_{nev}stop, \gamma[\perp/n]\overset{end}{\longmapsto}_{sim}stop}{\gamma\overset{end}{\longmapsto}_{ver}stop}$	`Synch-ver`	$\dfrac{\gamma\overset{inst_n}{\longmapsto}_{nev}\gamma', \gamma[\perp/n]\overset{inst_j}{\longmapsto}_{sim}\ \gamma''}{\gamma\overset{synch_j}{\longmapsto}_{ver}\gamma''[\gamma'(n).\sigma_e[j/_last]/n]}$
`Simu-ver`	$\dfrac{\gamma(n)=\perp, \gamma\overset{inst_j}{\longmapsto}_{sim}\ \gamma'}{\gamma\overset{sim_j}{\longmapsto}_{ver}\gamma'}$	`Claim-ver`	$\dfrac{\gamma\overset{end}{\longmapsto}_{nev}stop}{\gamma\overset{claim_v}{\longmapsto}_{ver}stop}$
`Ends-ver`	$\dfrac{\gamma(n)=\perp, \gamma\overset{end}{\longmapsto}_{sim}stop}{\gamma\overset{end}{\longmapsto}_{ver}stop}$		

Fig. 13. Verification-level rules

5.1.2. Verification-level rules

Although the $never\ claim$ is an optional construct in verification, in order to obtain a clear and general semantics we consider that verification without the $never\ claim$ is a special case. The transition relation $\longmapsto_{ver}\subseteq \Gamma\times L_{ver}\times(\Gamma\cup\{stop\})$ defines the different ways to execute the pair $(system, never\ claim)$[4] and also the execution of models without $never\ claim$. The labels of the relation $\longmapsto_{ver}$ are elements of

$$L_{ver} = L_{sim}\cup(\cup_{j\in Pid}\{synch_j, sim_j\})\cup\{discard, claim_v\}.$$

The index j always represents the process instance that executes code in the system part.

* `Discard-ver`, `End-ver`
  These two rules apply when the system execution is not recognized by the *never claim* process. In the first case (`Discard-ver`) the execution is discarded because the current configuration of the system is not the one expected by *never claim*. In the second one (`End-ver`), the system stops, and consequently *never claim* cannot continue.
* `Simu-ver` and `Ends-ver`
  The simulation-level rules are used when no *never claim* process exists ($\gamma(n)=\perp$).
* `Err-ver`
  This rule is used when the system part has an execution error (independently of the *never claim* process).
* `Synch-ver`
  This rule models the synchronized execution. When *never claim* can execute a process-level step and the system part can also proceed, there is a step at this level. The new configuration has updated the states for the process at the system level (*j*) which has evolved, and for *never claim* also. In this process, the value for variable `_last` has changed to *j*. Note that $\gamma[\perp/n]$ represents the system part without *never claim*.
* `Claim-ver` This rule recognizes a particular erroneous system execution.

5.2. Cycles

The execution of PROMELA models can produce infinite (non-terminating) executions. In these cases, the verification-oriented part of the language considers two kinds of errors on infinite executions: acceptance cycles and

[4] Note that $system$ represents the PROMELA model excluding $never\ claim$

non-progress cycles. To check cycles, the language considers two special sets of instructions labels denoted as $Accept$ and $Progress$. We assume that these labels are elements of the set $Label$. Now we give a more general definition of these errors.

Definition 3 (Accepting execution). An accepting execution in process j is an infinite sequence $t = \gamma_0 \overset{lb^0}{\longmapsto}_{ver} \gamma_1 \overset{lb^1}{\longmapsto}_{ver} \ldots \overset{lb^{k-1}}{\longmapsto}_{ver} \gamma_k \ldots$ such that $\exists L \in Accept.(\forall m \geqslant 0.(\exists s > m.\gamma_s(j).\sigma_l = L))$.

That is, an accepting execution passes through a given accepting label infinitely. Note that we do not distinguish between whether the model contains a *never claim* (the use of $Accept$ labels in *never claim* is also considered).

Definition 4 (Non-progress execution). A non-progress execution is an infinite sequence $t = \gamma_0 \overset{lb^0}{\longmapsto}_{ver} \gamma_1 \overset{lb^1}{\longmapsto}_{ver} \ldots \overset{lb^{k-1}}{\longmapsto}_{ver} \gamma_k \ldots$ such that $\neg(\exists j \in Pid, \exists L \in Progress.(\forall m \geqslant 0.(\exists s > m.\gamma_s(j).\sigma_l = L)))$.

That is, a non-progress execution does not pass through any progress label infinitely.

5.3. Operational semantics for verification

We now present several views of the verification of M depending on the goal of the user. We first define erroneous executions due to a *never claim* violation.

Definition 5 (Violation of *never claim*). Given a model M containing a *never claim*, $Never^-(M, \tau, \varphi)$ represents the set of execution sequences that violate the never claim, and it is defined as:

$t \in Never^-(M, \tau, \varphi)$ iff one of the following conditions holds:

1. $\exists k > 0\ t = \gamma_0 \overset{lb^0}{\longmapsto}_{ver} \gamma_1 \overset{lb^1}{\longmapsto}_{ver} \ldots \overset{lb^{k-1}}{\longmapsto}_{ver} \gamma_k \overset{claim_v}{\longmapsto}_{ver} stop$.
2. t is an accepting execution in n (*never claim*).
3. $\exists k > 0\ t = \gamma_0 \overset{lb^0}{\longmapsto}_{ver} \gamma_1 \overset{lb^1}{\longmapsto}_{ver} \ldots \overset{lb^{k-1}}{\longmapsto}_{ver} \gamma_k \overset{end}{\longmapsto}_{ver}$ and $\gamma_k(n).\sigma_l \in Accept$.

The third condition occurs when the system part finishes correctly, and the *never claim* process is in an *accept* label at this point. When this happens, SPIN enforces the system part to infinitely repeat the last configuration, producing an accepting execution as modelled by the second condition. This consideration corresponds to the current scheme to translate temporal formulas into *never claims*.

Definition 6. An erroneous execution is a sequence of configurations $t = \gamma_0 \overset{lb^0}{\longmapsto}_{ver} \gamma_1 \overset{lb^1}{\longmapsto}_{ver} \ldots \overset{lb^{k-1}}{\longmapsto}_{ver} \gamma_k \ldots$ satisfying some of the following conditions:

- $\exists k > 0$ such that $\gamma_k \overset{lb}{\longmapsto}_{ver} stop$, and $lb \in \{ies\} \bigcup_j \{error_j\}$.
- t is an accepting execution in a process $j \neq n$.
- t is a non-progress execution.
- A violation of *never claim*.

The next two definitions model the sets of erroneous and correct executions, that is, the operational semantics for verification. Both definitions may be applied even when the *never claim* process exists.

Definition 7. The set of erroneous execution sequences of a model M with respect to the functions τ and φ is the set $Ver^-(M, \tau, \varphi)$ of all erroneous execution sequences generated using $\longmapsto_{ver}$ from the initial configuration $initial(M, \tau, \varphi)$.

Definition 8. The set of execution sequences verified in a model M with respect to the functions τ and φ is the set $Ver^+(M, \tau, \varphi)$ of all maximal (possible infinite) non-erroneous execution sequences generated using $\longmapsto_{ver}$ from the initial configuration $initial(M, \tau, \varphi)$.

Note that the previous definition is close to the verification method in SPIN. In the presence of *never claim*, the number and length of visited (verified) paths could be less than the real non-erroneous paths, if the rule `Discard-ver` is used.

6. Abstract model checking

One of the most exciting applications of abstraction to engineering design languages is the construction of abstract models to verify temporal properties. The classic approach to this problem (i.e. presented in [CGL94, LGS95, DGG97]) is formulated as follows. Assume that we cannot check whether a temporal formula f is satisfied by all traces in model M ($M \models \forall f$) due to the *state explosion problem*. In this case, we may construct an abstract/reduced model M^α and redefine the satisfiability relation $\models$ into a new relation $\models_u^\alpha$ according to the new data definitions. We use the subindex u to indicate that the classic approach for abstracting properties *under*-approximates them. This way of defining $\models_u^\alpha$ preserves universal properties from the abstract to the concrete model, that is, $M^\alpha \models_u^\alpha \forall f \Rightarrow M \models \forall f$ holds.

In [GMP02a, GMP02b], we have developed the dual approach $\models_o^\alpha$ for abstracting properties based on *over*-approximation. In contrast to the classic method, this approach preserves the refutation of existential properties from the abstract to the concrete model, which may be formulated as $M^\alpha \not\models_o^\alpha \exists f \Rightarrow M \not\models \exists f$. Both approaches were developed inside the temporal logic framework, without taking into account implementation details. In [GMP02b] we showed that they are dual and equivalent ways to solve abstract model checking, and that the selection of the method depends on user preferences and the availability of tools. Note that both methods work with the same abstract model M^α, which is always an over-approximation of the original one (Theorem 1).

In this section, we present a complementary view of the problem which highlights in the relation between the abstraction approach and the on-the-fly model checking implemented by SPIN. As commented in the previous section, SPIN is able to refute properties (described by *never claim* processes) against models. To achieve this, it realizes the synchronized product of two automata, one representing the model, and the other one the *never claim* process. In the context of abstract model checking, models used to be correct over-approximations of the original models. Following the previous discussion, these abstract versions of the models are converted into *over-approximated* automata before being analyzed by SPIN. Therefore, in order to correctly synchronize the abstract automaton (the abstract model) and the automaton corresponding to *never claim*, this process must be abstracted using the same methodology, i.e. it must be over-approximated. Note that since *never claim* is a correct PROMELA process, we may use the same technique for abstracting the processes in the models and the *never claims*. From this point of view, over-approximation seems to be the most natural approach for abstracting properties in on-the-fly model checkers.

6.1. The automata theoretic approach

The problem of abstract model checking in the context of pure PROMELA (without temporal logic) can be defined as follows: given a PROMELA specification $M \parallel N$, where M represents a system and N is the *never claim* process, how can the analysis of $Never^-(M \parallel N, \tau, \varphi) = \emptyset$ be reduced by abstraction?

As commented above, our approach consists in abstracting the *never claim* process N as the other processes in the system and ensuring that the synchronous product of the abstract versions $(M \parallel N)^\alpha$ gives enough information to discard that $Never^-(M \parallel N, \tau, \varphi) \neq \emptyset$.

In the rest of the section, we assume that Λ is a family of abstractions and that M^α and N^α are the abstractions of M and N obtained as explained in Sect. 4. Note that M^α really represents the set of traces $Gen(M, \tau^\alpha, \varphi^\alpha)$. So we can directly use the definition of $Never^-$ given in Definition 5. Therefore, the set $Never^-(M \parallel N, \tau^\alpha, \varphi^\alpha)$ is the set of abstract sequences in M^α violating N^α

The following proposition states that the synchronized execution of the abstract system and the abstract *never claim* process can ensure the absence of errors in the concrete system.

Proposition 1 (Preserving non-violation).. $Never^-(M \parallel N, \tau^\alpha, \varphi^\alpha) = \emptyset \Rightarrow Never^-(M \parallel N, \tau, \varphi) = \emptyset$.

Proof. Assume that $t \in Never^-(M \parallel N, \tau, \varphi)$ satisfies Condition 1 of Definition 5. This means that *never claim* has finished its execution. Since Condition **LC**τ of Definition 2 is held, and in addition, by construction, abstraction does not modify the program counters, using a similar argument than the one used in the proof of Theorem 1, we have that there exists an abstract trace t^α such that $\alpha(t) \leqslant^\alpha t^\alpha$ and the abstract *never claim* process also reaches its end point in t^α. Therefore, $t^\alpha \in Never^-(M \parallel N, \tau^\alpha, \varphi^\alpha)$. The other two conditions for violation of *never claim* are similarly analyzed. □

Although the use of τ^α to execute the *never claim* guaranties the implication in this proposition, it also produces spurious errors as shown in the next example.

Example 3. Consider the model Even/Odd given in Fig. 2. The following *never claim* process checks whether variable i is always even (this is a non-desirable behavior).

```
never {
accept_even:
if::
 (i % 2 == 0): goto accept_even
fi;
}
```

The synchronous execution of this process, following the rules described above, will produce an acceptance cycle for an abstract trace $t^\alpha = \gamma_0^\alpha \stackrel{lb_0}{\longmapsto}_{ver} \cdots$ only if every state of *never claim* in t^α satisfies the test `i % 2 == 0`, that is, if $\forall k \geqslant 0$, $\tau^\alpha(\texttt{i\%2 == 0}, \gamma_k^\alpha(n))$. Note that due to the loss of information involved in the abstraction, for all abstract states σ^α, we have defined the abstract evaluation of guards in p1 as $\tau^\alpha(i < N, \sigma^\alpha) = true$ and $\tau^\alpha(i >= N, \sigma^\alpha) = true$. Therefore, it is possible to have an abstract trace $t^\alpha = \gamma_0^\alpha \stackrel{lb_0}{\longmapsto}_{ver} \cdots$ for which $\tau^\alpha(\texttt{i\%2 == 0}, \gamma_k^\alpha(n))$ in every state, because it is always possible to choose the second guard in p1. This is a spurious error, because clearly it does not occur in the initial model.

The existence of spurious errors is common to both the classic and the over-approximation approaches to model checking (see [GMP02b]), and their elimination is one active line of research (see [GMP02c] for our approach to this problem).

6.2. Extension to temporal logic

Although *never claim* may be used to specify properties to be refuted against models, users prefer a higher level notation like temporal logic. As mentioned in Sect. 2, SPIN supports its translation into *never claims*. The aim of this section is to relate the preservation results in temporal logic based abstract model checking with Proposition 1.

The syntax and semantics of LTL formulas to be evaluated against PROMELA models were presented in Sect. 2 (see Fig. 3). Atomic propositions in LTL formulas regarding PROMELA models are Boolean expressions. Thus, considering the standard notion of satisfiability given in Fig. 3, and following the same idea used for abstracting the model, we may assert that in order to define the abstract satisfaction of a temporal formula it suffices to define the abstract satisfaction of the atomic propositions. The way of defining this relation is the main difference between the classic and the over-approximation methods.

In the context of PROMELA our method employs the function τ^α (*Over*) to evaluate the atomic propositions, whereas the classic method should employ the function τ_u^α, which is defined as follows.

Definition 9 (Under-approximation). Given $p \in BExp$ and $\sigma_e^\alpha \in Env^\alpha$, τ_u^α is defined as

$$\tau_u^\alpha(p, \sigma_e^\alpha) = \bigwedge_{\{\sigma_e \in Env.\alpha(\sigma_e) \leqslant_e^\alpha \sigma_e^\alpha\}} \tau(p, \sigma_e) \qquad (Under)$$

Using Definitions *Over* and *Under* the satisfaction of a proposition in a given state is defined as follows:

1. $\sigma_e \models p$ when $\tau(p, \sigma_e)$ holds,
2. $\sigma_e^\alpha \models_o^\alpha p$ when $\tau^\alpha(p, \sigma_e^\alpha)$ holds, and
3. $\sigma_e^\alpha \models_u^\alpha p$ when $\tau_u^\alpha(p, \sigma_e^\alpha)$ holds.

We can extend this notation to define the satisfaction of temporal formula just by replacing the standard satisfaction by $\models_u^\alpha$ or $\models_o^\alpha$ in Fig. 3. The following theorem presents two direct results of the previous definitions. In the theorem, we assume that M does not include a *never claim*, $Gen(M, \varphi^\alpha, \tau^\alpha)$ is a correct over-approximation of model $Gen(M, \varphi, \tau)$ in the sense described in the Sect. 4.2.

Theorem 2. Given a temporal formula f
(1) $M^\alpha \models_u^\alpha \forall f \Rightarrow M \models \forall f$
(2) $M^\alpha \not\models_o^\alpha \exists f \Rightarrow M \not\models \exists f$
(3) $M^\alpha \models_u^\alpha \forall f \Leftrightarrow M^\alpha \not\models_o^\alpha \exists \neg f$

Note that implications (1) and (2) cannot be converted into equivalences because the two approaches modify the standard meaning of the negation. Given a proposition p and an abstract environment σ_e^α, using definition (*Under*), it is possible that for the classic method neither $\tau_u^\alpha(p, \sigma_e^\alpha)$ nor $\tau_u^\alpha(\neg p, \sigma_e^\alpha)$ hold. In contrast, due to definition (*Over*) for the over-approximation method, it is possible that both $\tau^\alpha(p, \sigma_e^\alpha)$ and $\tau^\alpha(\neg p, \sigma_e^\alpha)$ hold. This

is why we skipped the negation rule from Fig. 3. In addition the third result of the previous theorem establishes the duality of both approaches. In this theorem, we assume that $\neg f$ is in negation normal form.

We now relate the automata based and temporal logic based abstract model checking. Taking into account the previous discussion on abstraction and automata based verification, this theorem, and in particular the last result, supports the implementation of $\models_u^\alpha$ and $\models_o^\alpha$ in αSPIN.

Clearly, using Theorem 2, verification and refutation may be implemented by translation of the temporal formulas into *never claims*. In the first case (verification of universal properties) the implementation consists of the following steps:

1. Construct $\neg f$ in negation normal form.
2. Translate $\neg f$ into the *never claim* $N_{\neg f}$.
3. Abstract $(M \parallel N_{\neg f})$ with Λ, τ^α and φ^α.
4. Check $Never^-(M \parallel N_{\neg f}, \tau^\alpha, \varphi^\alpha) = \emptyset$
5. Use $Never^-(M \parallel N_{\neg f}, \tau^\alpha, \varphi^\alpha) = \emptyset \Rightarrow M^\alpha \not\models_o^\alpha \exists \neg f (\Leftrightarrow M^\alpha \models_u^\alpha \forall f)$.

The second case (refutation of existential properties) starts at step 2. Steps 1 and 2 are implemented in SPIN following [GPV95]. Steps 3 to 5 are implemented in αSPIN.

7. Discussion and related work

There are a number of works on the use of parameters to have several semantics (instances) in the same framework. Ferrari and Montanari consider a standard format for SOS (the De Simone format) and study how many theoretical results for this format can be extended to the parameterized framework [FeM98]. The same idea although in a different context is also presented by Cleaveland and Henesy [ClH93]. More recently, Bauer and Huuck give a parameterized semantics to Sequential Function Charts [BaH02]. The relation between generalized semantics and abstract interpretation was initially studied for constraint logic programs [GDL95, CoC95]. As far as we know, generalization for abstract model checking has not been employed before.

From the point of view of abstract model checking, the abstract interpretation approach to model checking was employed in [CGL94, LGS95, DGG97] for the verification of temporal properties expressed with CTL and μ-calculus. In [Lev01], the author presents a symbolic semantics of value-passing concurrent processes in order to achieve a more precise result than [DGG97]. Recently, Graf and Saïdi used the theorem prover PVS for the automatic construction of the abstract model [GrS97]. This work has been extended to obtain a textual specification, which can be abstracted again [BLO98]. This transformation capability is also supported in αSPIN, where it is also possible to abstract a previously abstracted model. The conditions presented in this paper guaranty correctness.

The study of abstraction at the automata level is also presented in [KPV01]. The authors employ the under-approximation and over-approximation of propositions, but their aim is to construct a temporal formula that represents information about a model. They do not work with abstract model checking.

Recently, the abstract interpretation technique has been applied to PROMELA [FeJ00]; however, no semantic basis is employed to reason about correctness. The authors discuss a manual abstraction of an specific protocol (Five Packet Handshake Protocol). They employ predicate abstractions instead of the data abstraction approach followed by our semantics.

The only operational semantics given for the whole language is due to Gerald Holzmann, and is available with the SPIN documentation [Spin]. This semantics is given in terms of an algorithm for the global evolution of the system for simulation mode and details of the *executable()* function. The precise definition of the system components (transitions, system states, channels, processes, etc.) is very suitable for giving an implementation guide. The whole semantics, indeed, seems oriented to implementation and the lack of SOS rules may introduce difficulties for extensibility or to give an overview of the language to new users.

There have been other attempts to give a precise semantics for the language. The proposal in [NaH97] is based on a number of functions to define the transition relation between states. Although this proposal is more declarative than [Spin], it is also more suitable to support implementation than to describe the language for users. Furthermore, verification aspects are not considered.

The approach in [Wei97] is closer to our proposal. In this paper, the author considers both modelling and verification-oriented features, and gives SOS rules. However, some key points that are not considered that can be easily solved in our approach. The use of only one process identifier in the rules impedes features like suspension

in atomic or allowing several processes to execute atomic sequences (as co-routines). Due to the lack of an explicit executability function, it is impossible to consider `else` or to discard executions when *never claim* suspends. The `unless` sentence is defined by giving a high priority to the rule that evaluates the escapes, but the concept of priority is not considered in other rules. Although Weisse's semantics was given for PROMELA 2.0, the structure of the rules makes it very difficult to extend these improvements to PROMELA 3.0.

Finally, the PROMELA semantics given in [Bev97] is described using the common Lisp-based logic of ACL2. The use of this logic provides several advantages. On the one hand, it enables proving the consistence of the semantic rules and, on the other, it makes the model executable. However, users interested in PROMELA must know ACL2 in order to completely understand the rules. In addition, some important language aspects such as the `d_step` instructions, *never claims* and correctness properties expressed by means of *progress* and *accept labels* are not handled by this approach.

Anyway, in this paper, we have also made some simplifications that do not affect the idea of supporting abstraction: variable and channel declarations, deterministic processes ($D_proctype$), enabling conditions in processes ($provided$), priorities, sending names of channels in messages and assertion violations. The declarative part related to variables and channels and the correct use of types, structures and arrays is omitted because these declarations are very similar to declarations in usual languages such as C. The runtime errors when using variables (such as violation of mode in xr or xs variables) or the effects over special variables (hidden and predefined variables) may be included in functions such as φ^P, $initP$ or $initial$. We have also considered a fixed set of options for simulation and verification, i.e. blocking output to full channels, checking for invalid end states, disabling partial order reduction and enabling fair execution sequences. The well-structured semantics for simulation and verification makes extension with new characteristics and new execution options practical. These advantages have been fully exploited in our definition to give certain sentences the formal meaning. For example, `skip` was included in the last step inside the rule `Basic-proc`; `timeout` instruction has been included as a very simple rule in the interaction level. In this sense, another substantial contribution with respect to previous proposals is the formalization of rendezvous inside `atomic` sentences for modelling co-routines. This feature was introduced in the semantics by just adding four new rules: `Co-rou1-sim`, `Co-rou2-sim` and `Co-rou3-sim` in the highest level and `Rend3-mod` in the interaction level.

8. Conclusions

The main contribution of this work is the definition of a parameterized semantics for PROMELA that support the correct implementation of abstract model checking on top of SPIN. This semantics allows us to define, in a common semantic framework, different levels of abstraction from an initial model, and to easily compare them for soundness conditions. In particular, these conditions are the key point for supporting correct abstractions by program transformation, as implemented in αSPIN. We have also presented how the theoretical framework provides a method to implement under-approximation and over-approximation approaches to abstract model checking. Both approaches have been also implemented in αSPIN.

Further application-oriented generalizations are possible. The parameters τ and φ are suitable to define and implement *data abstraction*. A new generalized semantics for PROMELA could be defined to develop *control abstraction*.

Acknowledgments. We are grateful to the reviewers for their very useful suggestions to improve both the contents and the readability of this paper. Many questions about PROMELA were solved by the quick and precise replies of Gerald Holzmann. The authors also would like to thank him for reading a first draft of this paper and providing very helpful comments.

Appendix: Detailed definitions

A.1. Details of executability functions

Function next makes use of function *deref* defined as follows:

Definition 10. The function $deref : Label \to Label$ dereferences labels until it finds a $Basic/Do/If/Run$ instruction.

- $deref(L) = L$, if $I(L) \in Basic \cup If \cup Do \cup Run$
- $deref(L) = deref(L')$, if $I(L) = goto\ L'$
- $deref(L) = deref(L')$, if $I(L) = \texttt{break}$, $\texttt{do} \cdots :: \ldots \texttt{L} : \texttt{break}; \ldots \texttt{od}; \texttt{L}' : \texttt{i}$
- $deref(L) = deref(L')$, if $I(L) = \texttt{atomic}\{\texttt{L}' : \texttt{i}; \ldots\}$
- $deref(L) = deref(L')$, if $I(L) = \texttt{d_step}\{\texttt{L}' : \texttt{i}; \ldots\}$
- $deref(L) = deref(L')$ if $I(L) = \{\texttt{L}' : \ldots\}\texttt{unless}\{\ldots\};$

Definition 11 (Next instruction). Let $next : Label \to Label$ be the function which, given a label L in the code of a process i, returns the label pointing to the $Basic/If/Do/Run$ instruction that must be executed after $I(L)$ in the code of i. Function $next$ is only applied to labels of $Basic/Run$ sentences.

- $next(L) = deref(L')$ if $I(L) \in Basic \cup Run$, $\texttt{L} : \texttt{i}_1; \texttt{L}' : \texttt{i}_2; \ldots$

Definition 12 (Guards). 1 Let $g : Label \to \wp(Label)$ be defined as:

– $g(L) = \{L\}$ if $I(L) \in Basic \cup Run$
– $g(L) = \cup_{i=1}^{k} g(deref(L_i))$, if $I(L) \in If \cup Do$ and

· $I(L) = \texttt{if} :: \texttt{L}_1 : \texttt{i}_1; \ldots \texttt{L}_\texttt{k} : \texttt{i}_\texttt{k};\ \texttt{fi}$ or
· $I(L) = \texttt{do} :: \texttt{L}_1 : \texttt{i}_1; \ldots \texttt{L}_\texttt{k} : \texttt{i}_\texttt{k};\ \texttt{od}$ or
· $I(L) = \texttt{if} :: \texttt{L}_1 : \texttt{i}_1; \ldots \texttt{L}_\texttt{k} : \texttt{i}_\texttt{k};\ \texttt{else}\ \texttt{L}_{\texttt{k+1}} : \texttt{i}_{\texttt{k+1}}\ \texttt{fi}$ or
· $I(L) = \texttt{do} :: \texttt{L}_1 : \texttt{i}_1; \ldots \texttt{L}_\texttt{k} : \texttt{i}_\texttt{k};\ \texttt{else}\ \texttt{L}_{\texttt{k+1}} : \texttt{i}_{\texttt{k+1}}\ \texttt{do}$

Function g associates each label L with the set of labels pointing to the $Basic/Run$ instructions which could be executed if $I(L)$ is executable.

2 Let $g_{else} : Label \to Label_\perp$ be defined as

– $g_{else}(L) = deref(L_{k+1})$ if $I(L) \in If \cup Do$ and

· $I(L) = \texttt{if} :: \texttt{L}_1 : \texttt{i}_1; \ldots \texttt{L}_\texttt{k} : \texttt{i}_\texttt{k};\ \texttt{else}\ \texttt{L}_{\texttt{k+1}} : \texttt{i}_{\texttt{k+1}}\ \texttt{fi}$ or

· $I(L) = \texttt{do} :: \texttt{L}_1 : \texttt{i}_1; \ldots \texttt{L}_\texttt{k} : \texttt{i}_\texttt{k};\ \texttt{else}\ \texttt{L}_{\texttt{k+1}} : \texttt{i}_{\texttt{k+1}}\ \texttt{do}$.

– $g_{else}(L) = \perp$, otherwise.

Definition 13 (Executable function). Let $exec : Label \times Env \to \{false, true\}$ be the executable function. Given a label L, and a local environment σ_e, $exec(L, \sigma_e)$ is equal to:

- $\tau(I(L), \sigma_e)$, if $I(L) \in BExp \cup Input$.
- $\tau(nfull(c), \sigma_e)$, if $I(L) = c!v_1, \ldots, v_s \in Output$.
- $\tau(nfull(c), \sigma_e)$, if $I(l) = c!!v_1, \ldots, v_s \in Output$.
- $\bigvee_{m \in g(L)} exec(m, \sigma_e)$ if $I(L) \in If \cup Do$ and $g_{else}(L) = \perp$.
- $exec(L', \sigma_e)$, if $I(L) = \texttt{atomic}\{\texttt{L}' : \texttt{i}; \ldots\}$ or $I(L) = \texttt{d_step}\{\texttt{l}' : \texttt{i}; \ldots\}$.
- $true$, otherwise.

In short, $exec(L, \sigma_e)$ returns $true$ if the instruction $I(L)$ of a process does not suspend in the local environment σ_e and returns $false$, otherwise. Note that if $I(L) \in Rendez$, then $exec(L, \sigma_e) = true$.

Before executing a *Rendez* instruction, it is necessary to check if the data sent match in the input instruction. Assume that function $matching : \Gamma \times Pid \times Pid \to \{false, true\}$ checks if data sent and received through the shared channel match. In particular, $matching(\gamma, i, j)$ returns $true$ if the current instructions for i and j are the two sides of a rendezvous, that is, if $rdv(\gamma(i).\sigma_l, \gamma(j).\sigma_l)$ and, in addition, instructions $I(\gamma(i).\sigma_l) = c!m_1$, $I(\gamma(j).\sigma_l) = c?m_2$, and $match(m_1, m_2)$. Function $match$ will be detailed in the following section.

The next three functions are used in the definition of function $nextexec$. The first function $fhandshk$ returns the set of the labels $fhandshk(L, j, k, L_0 \cdot \ldots \cdot L_m, \gamma)$ of the input instructions allowed in process k to continue the *Rendez* statement labelled by L in process j.

Definition 14 (fhandshk function). Let function $fhandshk : Label \times Pid \times Pid \times Label^* \times \Gamma \to \wp(Label)$ be defined as:

- $fhandshk(L, j, k, L_0 \cdot \ldots \cdot L_m, \gamma) = fhandshk(L, j, k, L_1 \cdot \ldots \cdot L_m, \gamma)$ iff $\neg exec(L_0, \gamma(k).\sigma_e)$, otherwise
- $fhandshk(L, j, k, L_0 \cdot \ldots \cdot L_m, \gamma) = fhandshk(L, j, k, L_1 \cdot \ldots \cdot L_m, \gamma)$ iff $\forall L' \in g(L_0), (\neg exec(L', \gamma(k).\sigma_e) \vee \neg rdv(L, L'))$, otherwise
- $fhandshk(L, j, k, L_0 \cdot \ldots \cdot L_m, \gamma) = \{L' \in g(L_0) | rdv(L, L'), matching(\gamma[L/\gamma(j).\sigma_l, L'/\gamma(k).\sigma_l], j, k)\}$.
- $fhandshk(L, j, k, \epsilon, \gamma) = \emptyset$.

The following function uses the previous one as an auxiliary function.

Definition 15 (handshk function). Let $handshk : Label \times Pid \times Pid \times \Gamma \to \wp(Label)$ be defined as

$$handshk(L, j, k, \gamma) = fhandshk(L, j, k, esc(\gamma(k).\sigma_l), \gamma)$$

The following function returns the labels of all basic instructions that can be used to continue the execution of a given process j. It iterates for all process in the system to find all labels satisfying the conditions described in the two previous ones.

Definition 16 (fexec function). Let function $fexec : Pid \times Label^* \times \Gamma \to \wp(Label \cup Label \times Pid \times Label)$ be defined as

- $fexec(j, L_0 \cdot \ldots \cdot L_m, \gamma) = fexec(j, L_1 \cdot \ldots \cdot L_m, \gamma)$ iff $\neg exec(L_0, \gamma(j).\sigma_e)$, otherwise
- $fexec(j, L_0 \cdot \ldots \cdot L_m, \gamma) = \{L \in g(L_0) | exec(L, \gamma(j).\sigma_l)\} \bigcup_{k \in Pid} \{(L, k, L') | L \in g(L_0), L' \in handshk(L, j, k, \gamma)\}$, otherwise
- $fexec(j, \epsilon, \gamma) = \emptyset$.

Finally, function $nextexec$ returns the set of labels (possibly 3-uples) through which a given process j may continue its execution

Definition 17 (nextexec function). Let function $nextexec : Pid \times \Gamma \to \wp(Label \cup (Label \times (Pid \times Label)))$ be defined as

$$nextexec(j, \gamma) = fexec(j, esc(\gamma(j).\sigma_l), \gamma)$$

A.2. Details of PROMELA instructions for modeling/simulation: τ, φ

The following definition applies over variables, constants or a sequence of constants.

- Let $Channel \subset Var$ be the set of channels declared in the model.
- Let $car : Const^* \to Const_\perp$ be defined as $car(c_1 \cdot \cdots \cdot c_n) = c_1$ if $n > 0$ and $car(\epsilon) = \perp$.
- Let $cdr : Const^* \to Const^*$ be defined as $cdr(c_1 \cdot \cdots \cdot c_n) = c_2 \cdot \cdots \cdot c_n$ if $n \geqslant 1$ and $cdr(\epsilon) = \epsilon$.
- Let $c_1 \cdot c$ denote the sequence obtained by appending symbol c_1 and the sequence c.
- Let $length : Const^* \to N$ be defined as $length(c_1 \cdot \cdots \cdot c_n) = n$.
- Let $size(c)$ denote the size of channel $c \in Channel$ in the model declaration.
- Let Exp be the set of arithmetical and Boolean expressions that can be built with the program variables and constants.
- Let $value : Exp \times Env \to Const \cup \{error\}$ be the function that given an expression returns its value.
- Let $match : (Const)^* \times (Var \cup Const)^* \to \{false, true\}$ be the function that check if one tuple of constants (data sent by the sender) and one tuple composed of constants or variables (data expected by the recipient) match, that is, $match(d_1 \cdot \cdots \cdot d_n, e_1 \cdot \cdots \cdot e_n)$ holds iff $\forall 0 \leqslant i \leqslant n.e_i \in Var$ or $d_i == e_i$.

With these auxiliary functions, we could define the execution of the PROMELA instructions using the following functions.

Definition 18 (Function τ^P). Given $\tau^P : BExp \times Env \to \{false, true\}$, $\tau^P(B, \sigma_e) = true$ iff one of the following holds:

- $B \in \{true, \texttt{timeout}, \texttt{skip}\}$
- $B = B1 \&\& B2$, $\tau^P(B1, \sigma_e)$ and $\tau^P(B2, \sigma_e)$
- $B = B1 \| B2$, $\tau^P(B1, \sigma_e)$ or $\tau^P(B2, \sigma_e)$

- $B = !B, \neg\ \tau^P(B, \sigma_e)$
- $B = c\ ?[\ v_1, \ldots, v_n\ ], c \in\ Channel, car(\sigma_e(c)) = \{d_1, \ldots, d_n\}, \forall 1 \leqslant s \leqslant n.match(d_s, v_s)$.
- $B = c\ ??[\ v_1, \ldots, v_n\ ], c \in\ Channel, \tau^P(c\ ?[\ v_1, \ldots, v_n\ ], \sigma_e)$ or $\tau^P(c\ ??[\ v_1, \ldots, v_n\ ], \sigma_e[cdr(\sigma_e(c))/c])$.
- $B = full(c), c \in\ Channel, length(\sigma_e(c)) = size(c)$.
- $B = nfull(c), \neg\tau^P(full(c), \sigma_e)$.
- $B = empty(c), c \in\ Channel, length(\sigma_e(c)) = 0$.
- $B = nempty(c), \neg\tau^P(empty(c), \sigma_e)$

Definition 19 (Function φ^P). Let $\varphi^P : Basic \times Env \to State \cup \{error\}$ be defined as follows:

- $\varphi^P(i, \sigma_e) = \sigma_e$, if $i \in BExp \cup Print$.
- $\varphi^P(v = e, \sigma_e) = \sigma_e[value(e, \sigma_e)/v]$, if $value(e, \sigma_e) \neq error$.
- $\varphi^P(v = e, \sigma_e) = error$, if $value(e, \sigma_e) = error$.
- $\varphi^P(c\ ?\ v_1, \ldots, v_n, \sigma_e) = \sigma_e[cdr(\sigma_e(c))/c, d_{i_1}/v_{i_1}, \ldots, d_{i_s}/v_{i_s}]$ where $\{v_{i_1}, \ldots, v_{i_s}\} = \{v_1, \ldots, v_n\} \cap Var$.
- $\varphi^P(c\ ?<\ v_1, \ldots, v_n >, \sigma_e) = \sigma_e[d_{i_1}/v_{i_1}, \ldots, d_{i_s}/v_{i_s}]$ where $\{v_{i_1}, \ldots, v_{i_s}\} = \{v_1, \ldots, v_n\} \cap Var$.
- $\varphi^P(c\ ??\ v_1, \ldots, v_n, \sigma_e) =$
  - $\varphi^P(c\ ?\ v_1, \ldots, v_n, \sigma_e)$, if $\tau^P(c\ ?[\ v_1, \ldots, v_n], \sigma_e)$, or
  - $\varphi^P(c\ ??\ v_1, \ldots, v_n, \sigma_e[cdr(\sigma_e(c))/c])$, otherwise.
- $\varphi^P(c\ ??\ < v_1, \ldots, v_n >, \sigma_e) = \sigma'_e[\sigma_e(c)/c]$ where $\sigma'_e = \varphi^P(c\ ??\ v_1, \ldots, v_n, \sigma_e)$.
- $\varphi^P(c\ !\ d_1, \ldots, d_n, \sigma_e) = \sigma_e[\sigma_e(c).\{value(d_1, \sigma_e), \ldots, value(d_n, \sigma_e)\}/c]$.
- $\varphi^P(c\ !!\ d_1, \ldots, d_n, \sigma_e) =$
  - $\varphi^P(c\ !!\ d_1, \ldots, d_n, \sigma_e) = \varphi^P(c!\ d_1, \ldots, d_n, \sigma_e)$, if $length(\sigma_e(c)) = 0$.
  - $\varphi^P(c\ !!\ d_1, \ldots, d_n, \sigma_e) = \sigma_e[\{value(d_1, \sigma_e), \ldots, value(d_n, \sigma_e)\} \cdot \sigma_e(c)/c\ ]$, if $car(\sigma_e(c)) > \{value(d_1, \sigma_e), \ldots, value(d_n, \sigma_e)\}$.
  - $\varphi^P(c\ !!\ d_1, \ldots, d_n, \sigma_e) = \sigma'_e[car(\sigma_e(c)) \cdot \sigma'_e(c)/c]$ if $car(\sigma_e(c)) \leqslant \{value(d_1, \sigma_e), \ldots, value(d_n, \sigma_e)\}$, and $\varphi^P(c!!\ d_1, \ldots, d_n, \sigma_e[cdr(\sigma_e(c))/c]) = \sigma'_e$.

Note that these definitions are not exhaustive, and should be completed with other existing or new PROMELA instructions.

References

[BaH02] Bauer N, Huuck R (2002) A parameterized semantics for sequential function charts. In: the proceedings of SFEDL (Semantic Foundations of Engineering Design Languages), Satellite Event of ETAPS 2002. pp 69–83.

[BLO98] Bensalem S, Lakhnech Y, Owre S (1998) Computing abstractions of infinite state systems compositionally and automatically. In: Computer aided verification, LNCS-1427, Springer, Berlin Heidelberg New York, pp 319–331

[Bev97] Bevier WR (1997) Toward an operational semantics of promela in Acl2. In: Proceedings of the third SPIN workshop, SPIN97. http://netlib.bell-labs.com/netlib/spin/ws97/papers.html

[CES86] Clarke EM, Emerson EA, Sistla AP (1986) Automatic verification of finite-state concurrent systems using temporal logic specifications. In: ACM trans programming lang syst 8(2):244–263

[CGL94] Clarke EM, Grumberg O, Long DE (1994) Model checking and abstraction. In: ACM trans programming lang syst 16(5):1512–1542

[CGP00] Clarke E, Grumberg O, Peled D (2000) Model checking, The MIT Press

[ClH93] Cleaveland R, Henesy M (1993) Testing equivalence as a bisimulation equivalence. Formal Aspects Comput 5:1–20

[CoC95] Codognet C, Codognet P (1995) A generalized semantics for concurrent constraint languages and their abstract interpretation. In: Constraint processing, selected papers. pp 39–49

[CoC77] Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the conference record of the 4th ACM symposium on principles of programming languages. pp 238–252

[DGG97] Dams D, Gerth R, Grumberg O (1997) Abstract interpretation of reactive systems. ACM Trans programming lang syst 19(2):253–291

[FeJ00] Fersman E, Jonsson, B (2000) Abstraction of communication channels in promela: a study case. In: SPIN model checking and software verification, LNCS-1885, Springer, Berlin Heidelberg New York, pp 187–204

[FeM98] Ferrari L, Montanari U (1998) Parameterized structured operational semantics. Fundamenta Informaticae 34:1–31
[GaM99] Gallardo MM, Merino P (1999) A framework for automatic construction of abstract PROMELA models. In: theoretical and practical aspects of SPIN model checking, LNCS-1680, Springer, Berlin Heidelberg New York, pp 184–199
[GMP01] Gallardo MM, Merino P, Pimentel E (2001) An operational semantics of PROMELA for simulation, verification and abstraction. In: Technical Report LCC ITI-01/9. Dpto. de Lenguajes y Ciencias de la Computacion. University of Malaga
[GMP02a] Gallardo MM, Merino P, Pimentel E (2002) Verifying abstract LTL properties on concurrent systems. In: Proceedings of the 6th world conference on integrated design and process technology
[GMM02] Gallardo MM, Martínez J, Merino P, Pimentel E (2002) A tool for abstraction in model checking. In: Proceedings of the 7th international workshop on formal methods for industrial critical systems (July 2002) electronic notes in theoretical computer science 66.2
[GMP02b] Gallardo MM, Merino P, Pimentel E (2002) Comparing under and over-approximations of LTL properties for model checking. In: Proceedings of the 11th international workshop on functional and (constraint) logic programming, electronic notes in theoretical computer science, vol 76
[GMP02c] Gallardo MM, Merino P, Pimentel E (2002) Refinement of LTL formulas for abstract model checking. In: Proceedings of the 9th international static analysis symposium SAS '02. LNCS-2477, Springer, Berlin Heidelberg New York, pp 395–410
[GDL95] Giacobazzi R, Debray SK, Levi G (1995) A generalized semantics and abstract interpretation for constraint logic programs. J Logic Programming 25(3):191–248
[GrS97] Graf S, Saïdi H (1997) Construction of abstract state graphs with PVS. In: Proceedings of the computer aided verification, LNCS-1254, Springer, Berlin Heidelberg New York, pp 72–83
[Hol91] Holzmann GJ (1991) Design and validation of computer protocols, Prentice-Hall
[Hol97] Holzmann GJ (1997) The model checker SPIN. IEEE Trans Softw Engin 23(5):279–295
[Hol03] Holzmann GJ (2003) The SPIN model checker. Primer and Reference Manual, Addison-Wesley
[LGS95] Loiseaux C, Graf S, Sifakis J, Boujjani A, Bensalem S (1995) Property preserving abstractions for the verification of concurrent systems. Formal Methods Syst Des 6:1–36
[MaP92] Manna Z, Pnueli A (1992) The temporal logic of reactive and concurrent systems—Specification, Springer, Berlin Heidelberg New York
[NaH97] Natarajan V, Holzmann GJ (1997) Outline for an operational semantics of promela. In: the SPIN Verification Systems. DIMACS Ser Discrete Math Theor Comput Sci 32:133–152
[GPV95] Gerth R, Peled D, Vardi MY, Wolper P (1995) Simple on-the-fly automatic verification of linear temporal logic. In: Proceedings of the PSTV conference. pp 3–18
[KPV01] Kesten Y, Pnueli A, Vardi MY (2001) Verification by augmented abstraction: the automata-theoretic view. J Comput Syst Sci 62(4):668–690
[Lev01] Levi F (2001) A symbolic semantics for abstract model checking. Sci Comput Programming 39:133–152
[Plo81] Plotkin GD (1981) A structural approach to operational semantics. In: Technical Report DAIMI FN-19, Computer Science Department, Aarhus University
[RuS99] Rusu V, Singerman E (1999) On proving safety properties by integrating static analysis, theorem proving and abstraction. In: Proceedings of tools and algorithms for the construction and analysis of systems, LNCS-1579, Springer, Berlin Heidelberg New York, pp 178–192
[VaW86] Vardi MY, Wolper P (1986) An automata-theoretic approach to automatic program verification. In: Proceedings of the Symposium on logic in computer science. pp 332–344
[Wei97] Weise C (1997) An incremental formal semantics for PROMELA. In: Proceedings the third SPIN workshop. http://netlib.bell-labs.com/netlib/spin/ws97/papers.html
[Spin] *On-the-fly LTL Model Checking with SPIN*. http://spinroot.com/spin/whatispin.html
[αSPIN] αSPIN *project*. University of Málaga. http://www.lcc.uma.es/gisum/fmse/tools

Received 6 October 2002
Accepted in revised form 24 November 2003 by M. Broy, G. Lüttgen and M. Mendler
Published online 20 May 2004