Towards a bottom-up fixpoint semantics that models the behavior of PROMELA programs

Marco Comini
Università degli Studi di Udine
marco.comini@uniud.it

Francesco S. Comisso
Università degli Studi di Udine
comisso.francescosaverio@spes.uniud.it

Alicia Villanueva*
vrAIn - Universitat Politècnica de València
alvilga1@upv.es

PROMELA (Process Meta Language) is a high-level specification language designed for modeling interactions in distributed systems. PROMELA is used as the input language for the model checker SPIN (Simple Promela INterpreter). The main characteristics of PROMELA are non-determinism, process communication through synchronous as well as asynchronous channels, and the possibility to dynamically create instances of processes.

In this paper, we introduce a bottom-up, fixpoint semantics that aims to model the behavior of PROMELA programs. This work is the first step towards a more ambitious goal where analysis and verification techniques based on abstract interpretation would be defined on top of such semantics.

1 Introduction

Concurrent programming has posed additional difficulties to the problem of guarantying correctness of programs. In fact, it may be almost impossible to replicate and debug an error caught by testing due to the fact that the different threads or processes can interact with each other at a different pace (depending on, for instance, processors speed). Not to mention the lack of coverage that usually testing techniques suffer.

Program verification is needed to ensure the correctness of a program in the case when, first, testing is simply not reliable enough due to a large number of possible inputs and, second, the consequences of failure are too costly in terms either of economic losses or of consequences that affect our lives.

In this context, PROMELA and SPIN ([7]) provide the tools for effectively model check ([2, 11]) the concurrent behavior of systems (specified in PROMELA). SPIN incorporates powerful techniques to handle big (finite) search spaces, some of them based on approximations or on heuristic searches. However, no complementary techniques (different from model checking) for the formal verification of PROMELA models have been proposed in the literature. [5] describes a tool to check some of the classical equivalences defined for PROMELA.

This work constitutes the first step of a project that aims to provide an approach to verification of properties of PROMELA programs different from model checking. More specifically, we are interested in analysis and verification techniques based on approximation via abstract-interpretation techniques built upon a denotational bottom-up fixpoint semantics, similar to those proposed for the tccp language [4]. Such denotational semantics must faithfully represent the *program behavior*, that is a suitable observation

*This work has been partially supported by the EU (FEDER) and the Spanish MINECO under grants TIN 2015-69175-C4-1-R and RTI2018-094403-B-C32 and by Generalitat Valenciana Deeptrust PROMETEO/2019/098.

Submitted to:
PROLE 2019

of the traces produced by the (official) operational semantics. This is achieved by proving full-abstraction of the denotational semantics w.r.t. the behavior. Full-abstraction allows to guarantee correctness of the analysis or verification results built upon such denotational semantics.

It can be deduced that, in addition to the denotational semantics introduced in this work, a formal representation of the behavior of the program is needed. Since the behavior of PROMELA programs is well settled by the SPIN semantics, in this work we focus on the denotational model, leaving the formalization of the behavior as future work, linked to the proof of full-abstraction.

Depending on the kind of technique to be used, denotational semantics with some properties can be more convenient than others. Since we are interested in abstract-interpretation based techniques, we would need a condensed, bottom-up, fixpoint semantics that is fully abstract w.r.t. the behavior of PROMELA programs. To the best of our knowledge, in the literature, we only find proposals for (top-down) operational semantics for PROMELA.

This work is organized as follows. In the following section we introduce the PROMELA language. Section 3 is devoted to the presentation of the denotational domain. The formalization of the fixpoint denotational semantics and also some examples are shown in Section 4. Finally, we conclude by discussing some relevant properties of the semantics and describe our plans for future work.

2 Promela

PROMELA is a high-level specification language designed for modeling interactions in concurrent and distributed systems [7]. Its specification is given on the SPIN website[1]. Other important sources are [8, 1]. Given a system specified in PROMELA, SPIN can either simulate the system's behavior or it can generate a C program that carries out the verification of the system against some specified correctness properties. These properties can be defined within the model or as temporal LTL properties ([10]). In order to achieve effectiveness, PROMELA's core design philosophy is to limit the size of the state space of models. This is enforced by introducing some restrictions on the language and trying to create a model where the optimal model abstraction is desired (e.g., by avoiding both to underspecify and to overspecify the system). However, considering only finite sized models is often not enough to overcome the state-explosion problem.

The emphasis of the language is not on computation details but on the modeling of process synchronization and coordination. Therefore, some common features of implementation languages such as input-output constructs or floating-point arithmetic are just not considered in PROMELA. In contrast, PROMELA is non-deterministic, allows for the dynamic creation of concurrent processes and processes can communicate via message channels that can be either synchronous or asynchronous.

We show a simple example of PROMELA program borrowed from [1] in Figure 1. It consists of the declaration of a global variable `n` and two process types (`P` and `Q`). Since the process declarations are preceded by the `active` directive, an instance of each process type will be created at the beginning of the execution. There are six possible interleavings for this simple example and SPIN is able to explore all six possibilities for verification. In the rest of this section, we introduce the aspects of the language that are relevant for our work.

[1] http://spinroot.com

```
byte n = 0;
active proctype p() {
  n = 5;
  printf (``Process P, n = %d\n'', n)
}
```

```
active proctype q() {
  n = 8;
  printf (``Process Q, n = %d\n'', n)
}
```

Figure 1: A simple PROMELA model.

2.1 Syntax and informal operational semantics of PROMELA

In PROMELA, we can distinguish two categories of constructs, behavioral constructs, and verification constructs. Behavioral constructs are those that may have an effect over the system state, in contrast to verification constructs. The latter are constructs (such as assertions, special labels or embedded LTL formulas) that SPIN uses to express properties over the system. Since our goal is to model the behavior of PROMELA programs, in this paper we consider only behavioral constructs.

The syntax of the language is shown in Figure 2. Similarly to [6], we adopt the convention that a non-terminal like `<Basic>` represents both the rule in the grammar and the set of instructions it generates. A PROMELA program consists of a sequence of user-defined types `<TypeDefDecl>`, (partial) declarations of the enumerate type `<MTypeDecl>`, declarations of variables `<VarDecl>`, or process declarations (`<Process>`). The syntax for user-defined structures `<TypeDefDecl>` resemble that used by the C language.[2] Elements of a structure are called fields. It is important to remark that recursive types are not allowed in PROMELA. `<MTypeDecl>` defines the enumerated type by using the keyword `mtype`. There can be multiple `<MTypeDecl>` declarations in a program, but every symbol is added to a single enumerated set. Variable declarations `<VarDecl>` consists of a variable type `<TypeName>` and a list of (maybe initialized) variable names `<Ivar>`. If no initialization is declared, variables are initialized with default values.

In PROMELA, types are used to specify the size allocation of the variables. Boolean values are aliases for integers, thus *false* represents the value 0 whereas *true* is 1 (actually, any value different from 0 is interpreted as *true*). There are no explicit type conversions. Types of smaller size are implicitly converted to bigger ones when needed. PROMELA introduces the type *pid*, which is an 8-bit sized type to store process identifiers and cannot be converted to numeric types. Note that every variable declaration `<Ivar>` may include a literal `<Const>` between square brackets. This allows the declaration of unidimensional arrays of fixed length. Expressions `<Expr>` are arithmetic and boolean expressions with standard syntax. They have to be free from side effects since they are the mechanism that determines whether a statement is executable. We do not include the definition for `<Const>` and `<Expr>` in Figure 2 (see [8] for a complete description).

Channels are declared through variable declarations by using the *chan* type that has value -1 associated by default. The *chan* type is not type-compatible with any other type. The channel declaration `<ChanInit>` consists of the capacity of the channel (between square brackets) and the structure of messages (a tuple of basic values). If the capacity is greater than zero, a buffered channel will be created. Otherwise, a rendezvous channel (synchronous channel) is created. Rendezvous channels can pass messages only through handshake communication between sender and receiver and cannot store messages. By default, buffered channels manage messages in first-in-first-out (FIFO) order. Every channel is in

[2] Identifiers such as `<TypeDefIdent>`, `<Ident>`, `<ProcessTypeName>` or `<ChannelId>` are sequences of alphanumeric symbols starting by a letter or _. We do not include the grammar for those elements.

```
<Model> ::= <Decl> [ ‘;’ <Decl> ]*
<Decl>  ::= <TypeDefDecl> | <MTypeDecl> | <VarDecl> | <Process>
<TypeDefDecl> ::= ‘typedef’ <TypeDefIdent> ‘{’ <VarDecl> [‘,’ <VarDecl> ]* ‘}’
<MTypeDecl> ::= ‘mtype’ ‘=’ ‘{’ <Ident> [‘,’ <Ident> ]* ‘}’
<VarDecl> ::= <TypeName> <Ivar> [‘,’ <Ivar> ]*
<TypeName> ::= <BasicTypeName> | <TypeDefIdent>
<BasicTypeName> ::= ‘bit’ | ‘bool’ | ‘byte’ | ‘short’ | ‘int’ | ‘chan’ | ‘mtype’ | ‘pid’
<Ivar> ::= <Ident> [ ‘[’ <Const> ‘]’ ] [ ‘=’ <Expr> | ‘=’ <ChanInit> | ‘=’ <ArrayInit> ]
<ChanInit> ::= ‘[’ <Const> ‘]’ ‘of’ ‘’ <TypeName> [ ‘,’ <TypeName> ]* ‘’
<ArrayInit> ::= ‘{’ <Expr> [‘,’ <Expr> ]* ‘}’
<Process> ::= ‘init’ ‘{’ <Sequence> ‘}’ | <ProcType>
<ProcType> ::= [ ‘active’ [‘[’ <Const> ‘]’ ] ] ‘proctype’ <ProcessTypeName> ‘(’ <Params> ‘)’
                         [‘priority’ <Val> ] [‘provided’ <Expr> ] ‘{’ <Sequence> ‘}’
<Sequence> ::= <Step> [ ‘;’ <Step>]*
<Step> ::= <VarDecl> | <Stmt> [ ‘unless’ <Stmt> ]
<Stmt> ::= <Basic> | <Jump> | <If> | <Do> | <Run> | <Atomic>
<Basic> ::= <Expr> | <Assign> | <Send> | <Receive> | <Run> | <Print> | <Assert>
<Assign> ::= <LVal> ‘=’ <Expr> | <LVal>‘++’ | <LVal>‘--’
<LVal> ::= <Ident> [ ‘[’ <Expr> ‘]’ ] [ ‘.’  <LVal> ]
<Send> ::= <ChannelId> ‘!’  <ExpSeq>
<Receive> ::= <ChannelId>‘?’<ExpSeq> | <ChannelId>‘?’‘<’ <ExpSeq> ‘>’
<ExpSeq> ::= <Arg> [ ‘,’ <Arg>]*
<Arg> ::= <LVal> | <Expr> | ‘eval’ ‘(’ <LVal> ‘)’
<Jump> ::= ‘break’ | <Goto>
<If> ::= ‘if’ <Options> ‘fi’
<Options> ::= ‘::’  <Sequence> [ ‘::’  <Sequence> ]* [‘::’  ‘else’ (‘;’ | ‘->’) <Sequence>]
<Do> ::= ‘do’ <Options> ‘od’
<Run> ::= ‘run’ <ProcessTypeId> ‘(’ <ActualParameters> ‘)’ [‘priority’ <Expr>]
<Atomic> ::= ‘atomic’ ‘{’ <Sequence> ‘}’
```

Figure 2: Syntax grammar of PROMELA

principle accessible from every active process. It is sufficient the knowledge of the channel id, regardless of the fact that the channel was created locally.

All syntactically legal process declarations are produced from non-terminal `<Process>`. The special `init` process can be declared once in each program. When execution starts, an instance of this process is created first. `<ProcType>` allows the definition of types of processes and instances of such processes can be created during execution through either the `active` directive or the `run` instruction; in both cases a process identifier *pid* that is assigned upon creation is accessible through the predefined local variable `_pid`. PROMELA limits the number of process instances to 255 but we do not impose this restriction in our semantics. A `<ProcType>` declaration consists of

1. the optional directive `active` that might also contain the number of instances to be created at the beginning of the execution;
2. the keyword `proctype` followed by a name (that must be unique within the model) followed by a list of parameters (a list of variable declarations where structs and arrays are not allowed);
3. the optional directives `priority` and `provided` that allows to specify, respectively, a priority level that affects the scheduler and a necessary condition to proceed with the execution of each process transition;[3] and

[3] In this work we do not handle priority among processes.

4. the *body* of the process, formed by a non-empty list of local variable declarations `<VarDecl>` and statements `<Stmt>` maybe followed by the *unless* construct.

The construction $stmt_1$ `unless` $stmt_2$ states that $stmt_1$ (*main sequence*) is run only when $stmt_2$ (*escape sequence*) is not enabled. When enabled, execution goes to the escape sequence discarding the main sequence.

For the sake of simplicity, we have decided to consider the `run` operator as a statement instead of an expression.[4] Therefore, we consider six kinds of statements, `<Basic>`, `<Jump>`, `<If>`, `<Do>`, `<Run>` and `<Atomic>`. Block declarations and compound statements (i.e., *do*, *if* and *atomic*) create nested static scope.

In PROMELA expressions are basic statements. They act as conditions since any expression evaluated to 0 (*false*) would block the execution. There are some predefined basic tests related to channels such as *nempty*, *nfull*, etc. Other basic statements are assignments, channel communication and run. An assignment is composed of a left-expression and a right-expression. Statements such as `v++` are syntactic sugar for `v=v+1`. The `<Send>` and `<Receive>` statements are those that implement communication of processes through channels (by using `!` and `?` for sending and receiving, respectively). Buffered channels are FIFO by default. Although there are variants of communication primitives that allow access to channels in different order, we plan to handle them as future work. With the aim to keep the presentation clear, we do not include in the paper the case for synchronous (*rendezvous*) communication, however we discuss in Section 5 how we handle it.

The semicolon symbol is a separator between statements, thus identifies a sequence of statements `<Sequence>`.[5] PROMELA admits two kinds of jumps: `break`, the statement that exits from the innermost *do* loop, and `goto`. We have decided to not deal with *goto* jumps in this work, but we sketch how we plan to handle them in Section 5.

The (compound) statements `<If>` and `<Do>` contain one or more options, each one starting by ‘`::`’. The first basic statement of each option is called the guard. An option is enabled if its guard is enabled (*true*). If more than one option is enabled, the scheduler chooses one. No assumptions are made about the possible choices of the scheduler so (following a conservative approach) all possible alternatives of execution are considered. Thus, these statements are a source of non-determinism. If no option is enabled, then the else-option is considered (if present). Otherwise, the local control blocks (at the front of these statements).

The run operator `<Run>` creates a new process passing the list of actual parameters (a list of expressions) to the new process. It is not required to specify values for all parameters, thus default values are passed when no value is defined.

Finally, a key statement in PROMELA is “`atomic`{ *stmts* }”. It avoids (whenever possible) interleaving during the execution of the statements *stmts*. The only case when a different process can interleave with *stmts* is when one of these statements blocks. A very similar statement is the `d_step` statement which aborts execution if its body suspends, in contrast to allowing interleaving. Since this statement is commonly used as a verification construct (it is not a behavioral one), we do not consider it in our semantics. Nevertheless, having modeled the *atomic* construct, we think it would be quite straightforward to extend the semantics for it.

[4]PROMELA considers that `run` is the only expression with side effects. This is because it can be used as a blocking expression when the maximum number of processes has been reached. Since we remove such restriction in our semantics, we prefer to consider `run` as one specific kind of statement.

[5]The `->` symbol is an alias for the semicolon and is used just for readability reasons.

Operational Engine. The official semantics of PROMELA is defined in terms of an operational model which contains one or more processes, zero or more variables, zero or more channels, and a *semantics engine* to handle the non-determinism (due to processes interleaving or to selection statements) and some internal variables such as *exclusive* (for atomic) or *handshake* (for *rendezvous*).

The local control of each process is represented by a labeled transition system (LTS). Each LTS models the local behavior of a single process. The semantics engine runs the system in a stepwise manner: selecting and executing one basic transition from some LTS at the time. In brief, the behavior of the engine is as follows. While there exist executable (enabled) statements, it selects one of them, updates the global state and then checks whether there are synchronous channel operations waiting to be executed and try to execute one of them.

A transition is executable if its executability clause is satisfied in the current system state. In addition to this *local* decision, an *exclusive* variable is used to handle atomic sequences. A side effect of the execution of any statement in an atomic block (except the last one) is to set variable *exclusive* to the *pid* of the executing process, thus preserving the exclusive privilege to execute. So once an atomic segment regains control executing a transition, the exclusive privilege is regained. There are *global* considerations (when there are no executable statements for instance) for which the *timeout* or *else* variables are used. Also, restrictions given by *atomic* statements must be considered. In order to handle *unless* statements, the priority of transitions is taken into consideration when deciding executable statements within a process.

In the model, there is no explicit characterization of time, thus concurrent transitions are interleaved. The semantics engine keeps executing transitions until no executable one remains. This happens if either the number of active processes turns to zero, or when a deadlock occurs. The interested reader can consult [8] for the pseudo-code of this process.

3 The semantic domain

The definition of our denotational semantics is inspired by the semantics for the tccp language of [3]. In this work, denotations are based on so-called hypothetical computations; i.e., traces of conditional steps that ideally model actual computations that would be obtained by feeding them with initial system stores that validate every condition.

The semantics domain for PROMELA essentially is formed by sequences of possible evolutions of the stored information (i.e., the system memory and the channels instancies). As such, we first need to formalize how the information stored at each execution point is represented (Section 3.1) and then we formalize the notion of traces of conditional evolutions (Section 3.2) that are the basic constituent of our denotations.

3.1 The system state

The *system state* of a PROMELA program contains a representation of the memory plus channel instances.

Definition 3.1 (System State) *A system state of a PROMELA program is an element of the domain State* $= ([Loc \to \mathbb{V}_\perp] \times [ChanId \to Channels_\perp])_\perp$ *where Loc are locations of a memory capable to store denotable values* $\mathbb{V}$*, that are basic values, structures, and arrays;*[6] *ChanId is the set of channel ids; Channels is the set of all channel instances, and* $\perp$ *denotes an erroneous (inconsistent) state.*

[6]$\mathbb{V}_\perp$ is the natural extension for adding the $\perp$ value.

In the sequel, we write σ as a typical element of *State*. Moreover, for $l \in Loc$ we write $\sigma(l)$ as a shorthand of $\bot$ when $\sigma = \bot$ or $\sigma_L(l)$ when $\sigma = (\sigma_L, \sigma_C)$. Analogously for $c \in ChanId$.

Let us now formalize the domains for the different components of the system state.

Definition 3.2 (Memory values) *Denotable values are elements that can be stored in memory and are defined as follows*

$$\mathbb{V} = \mathbb{B} \uplus \bigcup_{n\in\mathbb{N}} Array(\mathbb{B},n) \uplus \mathbb{R} \uplus \bigcup_{n\in\mathbb{N}} Array(\mathbb{R},n) \qquad \mathbb{R} = Record\{f_1 : \mathbb{V},\dots,f_n : \mathbb{V}\}$$

$$\mathbb{B} = Int \uplus Short \uplus Byte \uplus Bit \uplus MType \uplus ChanId \uplus Pid$$

The set $\mathbb{B}$ represents basic denotable values for the PROMELA basic types, which are all the elements storable in a variable of type $\tau \in BasicTypes$. The set BasicTypes is $\{int, short, byte, bit, mtype, chan, pid\}$.

In the following, we write a type as a superscript of a set to denote that the expected values for that set are of the given type; like $\mathbb{D}^{int}$ to specify that the values in $\mathbb{D}$ are meant to be of type *int*. Thus $\mathbb{B}^{int} = Int$.

Definition 3.3 (Array values) *Given a type τ and a positive integer n, by $\tau[n]$ we denote the type of arrays of n values of type τ. With $Array(\mathbb{D}^\tau, n)$ we denote the set of all arrays of size n with values in $\mathbb{D}$ of type τ. $Array(\mathbb{D}^\tau, n)$ is equipped with the following auxiliary functions.*

Creation. *For all $n \in \mathbb{N}$, $array_n : \mathbb{D}^\tau \times \cdots \times \mathbb{D}^\tau \to Array(\mathbb{D}^\tau, n)$ takes n values of type τ and returns an array of type $\tau[n]$.*

Read. *For all $n \in \mathbb{N}$, given an array $v \in Array(\mathbb{D}^\tau, n)$ and an integer k, $read_n : Array(\mathbb{D}_\tau, n) \to Int \to \mathbb{D}_\tau$ returns the denotable value in the k-th position of v. Reading off the bounds of an array gives back the default value of type τ.*

Update. *For all $n \in \mathbb{N}$, given $v \in Array(\mathbb{D}^\tau, n)$, an integer k and a modifier function $m : \mathbb{D}^\tau \to \mathbb{D}^\tau$, $update_n : Array(\mathbb{D}^\tau, n) \to Int \to (\mathbb{D}^\tau \to \mathbb{D}^\tau) \to Array(\mathbb{D}^\tau, n)$ returns v with the k-th position updated with function m. Updating off the bounds of an array has no effect.*

Definition 3.4 (Structures values) *We denote the structured type with field names $f_1,\dots,f_n$ of type $\tau_1,\dots,\tau_n$ by typedef$\{f_1 : \tau_1,\dots,f_n : \tau_n\}$. $Record\{f_1 : \mathbb{D}_1^{\tau_1},\dots,f_n : \mathbb{D}_n^{\tau_n}\}$ denotes the set of all structures with field names $f_1,\dots,f_n$ with values in domains $\mathbb{D}_1,\dots,\mathbb{D}_n$ of type $\tau_1,\dots,\tau_n$. Structures are equipped with the following auxiliary functions.*

Creation. *For all field names $f_1,\dots,f_n$, $record_{f_1,\dots f_n} : \mathbb{D}^{\tau_1} \times \cdots \times \mathbb{D}^{\tau_n} \to Record\{f_1 : \mathbb{D}_1^{\tau_1},\dots,f_n : \mathbb{D}_n^{\tau_n}\}$ takes n values of the corresponding type and returns a structure with fields $f_1,\dots,f_n$ instantiated to the provided values.*

Read. *For $k \in \{1,\dots,n\}$, projection $\pi_{f_k} : Record\{f_1 : \mathbb{D}_1^{\tau_1},\dots,f_n : \mathbb{D}_n^{\tau_n}\} \to \mathbb{D}_k^{\tau_k}$ that extracts field f_k.*

Update. *For $k \in \{1,\dots,n\}$, given a record value and a value modifier, $u_{f_k} : Record\{f_1 : \mathbb{D}_1^{\tau_1},\dots,f_n : \mathbb{D}_n^{\tau_n}\} \to (\mathbb{D}_k^{\tau_k} \to Record\{f_1 : \mathbb{D}_1^{\tau_1},\dots,f_n : \mathbb{D}_n^{\tau_n}\})$ returns a record with field f_k updated.*

Definition 3.5 (Channel instances) *Channels denote the set of all channel instances. A channel is a list of messages with a maximum size. A message is an element of Message $= \mathbb{B}^{\tau_1} \times \cdots \times \mathbb{B}^{\tau_n}$ where, $\tau_1,\dots,\tau_n$ are basic types. We denote by $Channel(k, \tau_1 \times \cdots \times \tau_n)$ the set of all channels of maximum size $k \in \mathbb{N}$ containing messages from $\mathbb{B}^{\tau_1} \times \cdots \times \mathbb{B}^{\tau_n}$. Moreover, $Channels := \bigcup_{k\in\mathbb{N},\tau_1,\dots,\tau_n \in BasicTypes} Channel(k, \tau_1 \times \cdots \times \tau_n)$. Channel instances and messages are not denotable values (thus not included in $\mathbb{V}$). Channels are equipped with some auxiliary functions.*

Creation. *Given a list of types for message components and a channel capacity n,* $channel_n : \tau_1 \times \cdots \times \tau_k \to Channel(n, \tau_1 \times \cdots \times \tau_k)$ *returns new empty channel of the given size and type.*

Channel inquiries. *These tests are defined in correspondence to those defined in the language:* $nfull : Channels \to Bit$, $full : Channels \to Bit$, $nempty : Channels \to Bit$ *and* $empty : Channels \to Bit$.

Access. *Functions* $push : Channels \to Message \to Channels$ *and* $pop : Channels \to Channels$ *allow to, respectively, insert a new message into a given channel's tail and remove a message from its head.* $head : Channels \to Message$ *returns the message in the head of the channel. Interacting with a channel with less or more arguments does not result in an error.*[7]

As usual the scope is managed by environments, which are not required to be part of the state since PROMELA has static scope.

Definition 3.6 (Environment) *An environment is a partial function associating identifiers with locations* $Env = [Ident \rightharpoonup Loc]$.

In the sequel, we write ρ as a typical element of *Env*.

3.2 The Domain of Denotations

Let us now define the domain upon which our denotations are built. Intuitively, each component in the semantics, called *conditional trace* aims to represent a sequence of hypothetical (conditional) computational steps.

Definition 3.7 (Conditional step and conditional traces) *A* conditional step *is*

a conditional transition $p \rhd t$ *which, given a predicate* $p : State \to Bool$ *and a state transformer* $t : State \to State$*, consists of the state modifier t guarded by condition p,*

a finalizer, *written* $\blacksquare$*, or*

a process spawn, *written* $\boldsymbol{run}(S)$*, where S is a non-empty set of conditional traces.*

Given a conditional transition $\phi = p \rhd t$ *we use the notation* ϕ^P *and* ϕ^T *for p and t.*

A conditional trace *is defined as*

- *a finite sequence of conditional steps (except* $\blacksquare$*), possibly ending with* $\blacksquare$*, or*
- *an infinite sequence of conditional steps (except* $\blacksquare$*).*

An infinite conditional trace or a finite one terminated by $\blacksquare$ *is called* maximal*; Otherwise, it is called* partial.

We use ε *to denote the empty trace (of length zero). We denote by* $\mathcal{CT}$ *the domain of all non-empty sets of conditional traces. We denote by* $\phi_1 \cdot \ldots \cdot \phi_n$ *the conditional trace whose conditional steps are* $\phi_1, \ldots, \phi_n$*. Moreover, given a partial trace s and a (partial or maximal) trace* s'*, we denote their concatenation by* $s \cdot s'$*. Finally, given a set of traces S, we denote by* $s \cdot S$ *the set* $\{s \cdot s' \mid s' \in S\}$.

Intuitively, a conditional transition $p \rhd t$ represents all possible computations where the current system state σ satisfies the condition p and then the system progress to state $t(\sigma)$. The finalizer $\blacksquare$ represents the end of the execution. This construct allows in the following to distinguish between traces where the execution has terminated from those that are blocked at some point. In PROMELA, a process terminates when it reaches the *end* of its body. Finally, in the following section we show how the *process spawn* is used to handle process calls.

It is worthy to note that we exclude $\varnothing$ from $\mathcal{CT}$ since the absence of progression is already represented by $\{\varepsilon\}$.

[7] Extra arguments are dropped and default values are used for missing ones.

Definition 3.8 (Preorder on $\mathcal{CT}$) *We order conditional traces by their information content. Namely, for all $\phi_1 \cdot s_1$, $\phi_2 \cdot s_2$, $s \in \mathcal{CT}$ and for all S_1, $S_2 \subseteq \mathcal{CT}$, $\varepsilon \le s$, $\blacksquare \le \blacksquare$, and*

$$\phi_1 \cdot s_1 \le \phi_2 \cdot s_2 \iff s_1 \le s_2 \wedge \forall \sigma \in State.\, \phi_1^P(\sigma) \Rightarrow \phi_2^P(\sigma) \wedge \phi_1^T(\sigma) = \phi_2^T(\sigma)$$
$$\boldsymbol{run}(S_1) \le \boldsymbol{run}(S_2) \iff S_1 \sqsubseteq S_2 \text{ where } S_1 \sqsubseteq S_2 \iff \forall s_1 \in S_1\, \exists s_2 \in S_2 . s_1 \le s_2$$

Note that $\sqsubseteq$ is not antisymmetric; for instance, $\{\phi \cdot s\} \sqsubseteq \{\phi, \phi \cdot s\}$ and $\{\phi, \phi \cdot s\} \sqsubseteq \{\phi \cdot s\}$. Therefore, in order to obtain a partial order, we use equivalence classes w.r.t. the equivalence relation induced by $\sqsubseteq$.

Definition 3.9 (Semantic domain) *Given M_1, $M_2 \subseteq \mathcal{CT}$, we define $M_1 \doteqdot M_2$ as $M_1 \sqsubseteq M_2 \wedge M_1 \sqsupseteq M_2$. We denote by $\mathbb{C}$ the class of non-empty sets of conditional traces modulo $\doteqdot$. Formally, $\mathbb{C} = \{[M]_{\doteqdot} \mid M \in \mathcal{CT}\}$. We abuse notation and denote by $\sqsubseteq$ also the partial order induced by the preorder $\sqsubseteq$ on equivalence classes of $\mathbb{C}$.*

$(\mathbb{C}, \sqsubseteq, \bigsqcup, \bigsqcap, \top, \bot)$ *is a complete lattice, where $\top = [\mathcal{CT}]_{\doteqdot}$, $\bot = [\{\varepsilon\}]_{\doteqdot}$ and, for all $\mathcal{M} \subseteq \mathbb{C}$, $\bigsqcup \mathcal{M} = [\bigcup_{[M]_{\doteqdot} \in \mathcal{M}} M]_{\doteqdot}$ and $\bigsqcap \mathcal{M} = [\{s \in \mathcal{CT} \mid \forall [M]_{\doteqdot} \in \mathcal{M}\, \exists s' \in M . s \le s'\}]_{\doteqdot}$*

The normal form of an equivalence class M is the smallest set in M. In the sequel, any $M \in \mathbb{C}$ is implicitly considered to be a set in $\mathcal{CT}$ that is obtained by choosing an arbitrary representative of the elements of the equivalence class M. Actually, all the operators that we use on $\mathbb{C}$ are independent of the choice of the representative. Therefore, we can define any operator on $\mathbb{C}$ in terms of its counterpart defined on sets of $\mathcal{CT}$.

To give meaning to process calls we use the following notion of interpretation that associates to each process symbol an element of $\mathbb{C}$ modulo variance.

Definition 3.10 (Interpretations) *Let $\mathbb{MGC}_{PIdent} := \{p(\overrightarrow{x}) \mid p \in PIdent, \overrightarrow{x}$ are distinct variables$\}$ (or simply $\mathbb{MGC}$ when clear from the context).*

Two functions $I, J : \mathbb{MGC}_{PIdent} \to \mathbb{C}$ are variants*, denoted by $I \cong J$, if for each $\kappa \in \mathbb{MGC}_{PIdent}$ there exists a variable renaming ρ such that $(I(\kappa))\rho = J(\kappa\rho)$.*

An interpretation *is a function $\mathcal{I} : \mathbb{MGC}_{PIdent} \to \mathbb{C}$ modulo variance*[8].

The semantic domain $\mathbb{I}_{\mathbb{C}}^{PIdent}$ (or simply $\mathbb{I}_{\mathbb{C}}$ when clear from the context) is the set of all interpretations ordered by the point-wise extension of $\sqsubseteq$ (which by abuse of notation we also denote by $\sqsubseteq$).

Essentially, the semantics of each predicate in *PIdent* is given over formal parameters whose names are actually irrelevant. It is important to note that $\mathbb{MGC}_{PIdent}$ has the same cardinality of *PIdent* (and is thus finite) and therefore each interpretation is a finite collection (of possibly infinite elements).

The partial order on $\mathbb{I}_{\mathbb{C}}$ formalizes the evolution of the process computation. $(\mathbb{I}_{\mathbb{C}}, \sqsubseteq)$ is a complete lattice with bottom element $\lambda\kappa.\bot$, top element $\lambda\kappa.\top$, and the least upper bound and greatest lower bound are the point-wise extension of $\bigsqcup$ and $\bigsqcap$, respectively. We abuse notation and use that of $\mathbb{C}$ for $\mathbb{I}_{\mathbb{C}}$ as well.

Any $\mathcal{I} \in \mathbb{I}_{\mathbb{C}}$ is implicitly considered to be a function $\mathbb{MGC} \to \mathbb{C}$ that is obtained by choosing an arbitrary representative of the elements of $\mathcal{I}$ generated by $\cong$. Therefore, we can define any operator on $\mathbb{I}_{\mathbb{C}}$ in terms of its counterpart defined on functions $\mathbb{MGC} \to \mathbb{C}$. Moreover, we also implicitly assume that the application of an interpretation $\mathcal{I}$ to a process call κ, denoted by $\mathcal{I}(\kappa)$, is the application $I(\kappa)$ of any representative I of $\mathcal{I}$ that is defined exactly on κ.

4 The bottom-up fixpoint semantics

We finally present the bottom-up fixpoint semantics for PROMELA.

[8] In other words, a family of elements of $\mathbb{C}$ indexed by $\mathbb{MGC}$ modulo variance.

In the literature, there exist some works that introduce formal definitions of an operational semantics for PROMELA. For instance, [12] proposes a clear structural operational semantics defined in an incremental style based upon [9]. However, such semantics models an old version of the language and more recent features such as the new behavior of *atomic* is missing. In our opinion, the most relevant proposal is [6], which provides a parametric structural operational semantics defined as the basis of the α-SPIN tool for abstract model checking. In order to design our denotational semantics, we have took into consideration only the official specification of [7]. When any aspect was not defined there, we made our own assumptions and then we validated them with the semantics in [12, 6] and with the latest version of SPIN.

As the reader might have noted, we have restricted ourselves to a significative fragment of PROMELA and made some assumptions that we recapitulate here for clarity. For simplicity, some constructs are not considered: *goto* jumps (and user-defined labels), the *timeout* (system defined) variable; ordered insertion and random removal of channel messages. Moreover, we treat the primitive expression *run* as a statement (and therefore it cannot be used into expressions). In this presentation of the semantics we consider only asynchronous communications. We do not force the active processes to terminate in reverse order of birth. We do not cover print statements nor statements that can be translated into the included ones (e.g. the *for* statement). Finally, we choose not to cover *d_step* in our work since is a SPIN related construct.

The semantics is defined as the least fixpoint of a semantics operator $\mathbb{D}:\mathbb{I}_{\mathbb{C}}\to\mathbb{I}_{\mathbb{C}}$ that transforms interpretations. Informally, such fixpoint is computed as the limit $\bigsqcup\{\mathbb{D}^k(\lambda\kappa.\bot)\mid k\in\mathbb{N}\}$ where $\kappa\in\mathbb{MGC}_{PIdent}$. Function $\mathbb{D}$ is defined in terms of auxiliary semantics evaluation functions that we describe in the following sections.

4.1 Denotation of Expressions

We define the semantics function to evaluate r-expressions $\mathbb{R}: r\text{-}expressions\times Env\times State\to\mathbb{B}$ as follows. $\widehat{e}$ notation represents the mapping into values for literals, arithmetic and boolean operators.

$$\mathbb{R}[\![e_1\,binop\,e_2]\!]_{\rho,\sigma}:=\widehat{binop}(\mathbb{R}[\![e_1]\!]_{\rho,\sigma},\mathbb{R}[\![e_2]\!]_{\rho,\sigma})\qquad \mathbb{R}[\![const]\!]_{\rho,\sigma}:=\widehat{const}$$
$$\mathbb{R}[\![unop\,e_1]\!]_{\rho,\sigma}:=\widehat{unop}(\mathbb{R}[\![e_1]\!]_{\rho,\sigma})\qquad \mathbb{R}[\![id]\!]_{\rho,\sigma}=\sigma(\rho(id))$$
$$\mathbb{R}[\![e_1[e_2]]\!]_{\rho,\sigma}=read(\mathbb{R}[\![e_1]\!]_{\rho,\sigma},\mathbb{R}[\![e_2]\!]_{\rho,\sigma})\qquad \mathbb{R}[\![l.field]\!]_{\rho,\sigma}=\pi_{field}(\mathbb{R}[\![l]\!]_{\rho,\sigma})$$
$$\mathbb{R}[\![e_1,\dots,e_n]\!]_{\rho,\sigma}:=(\mathbb{R}[\![e_1]\!]_{\rho,\sigma},\dots,\mathbb{R}[\![e_n]\!]_{\rho,\sigma})$$
$$\mathbb{R}[\![c\text{->}e_1:e_2]\!]_{\rho,\sigma}:=\text{if }\mathbb{R}[\![c]\!]_{\rho,\sigma}=0\text{ then }\mathbb{R}[\![e_1]\!]_{\rho,\sigma}\text{ else }\mathbb{R}[\![e_2]\!]_{\rho,\sigma}$$

Moreover we define the semantics function $\mathbb{L}: l\text{-}expressions\times Env\times State\to((\mathbb{B}\to\mathbb{B})\to State)$ that given an environment ρ, a state σ and then a value updating function m, modifies the location identified by the l-expression with updater m.

$$\mathbb{L}[\![id]\!]_{\rho,\sigma}=\lambda f.\sigma\big[f(\sigma(\rho(id)))/\rho(id)\big]\qquad \mathbb{L}[\![l.field]\!]_{\rho,\sigma}=\lambda f.\mathbb{L}[\![l]\!]_{\rho,\sigma}(\lambda s.u_{field}(s,f))$$
$$\mathbb{L}[\![e_1[e_2]]\!]_{\rho,\sigma}=\lambda f.\mathbb{L}[\![e_1]\!]_{\rho,\sigma}(\lambda a.update(a,\mathbb{R}[\![e_2]\!]_{\rho,\sigma},f))$$

EXAMPLE 4.1 (EVALUATION OF EXPRESSIONS)

Given $\rho:=\{\mathtt{x}\rightharpoonup l_x,\ \mathtt{a}\rightharpoonup l_a\}$ and $\sigma:=(\{l_x\rightharpoonup 4,\ l_a\rightharpoonup array_3(1,2,3)\},\{\})$, then $\mathbb{R}[\![\mathtt{x+1>3}]\!]_{\rho,\sigma}=\mathbb{R}[\![\mathtt{x+1}]\!]_{\rho,\sigma}>\mathbb{R}[\![\mathtt{3}]\!]_{\rho,\sigma}=\mathbb{R}[\![\mathtt{x}]\!]_{\rho,\sigma}+\mathbb{R}[\![\mathtt{1}]\!]_{\rho,\sigma}>3=\sigma(\rho(\mathtt{x}))+1>3=1$ (which corresponds to *True*).

Moreover $\mathbb{L}[\![\mathtt{a[x-3]}]\!]_{\rho,\sigma}(\lambda v.v+4)=\mathbb{L}[\![\mathtt{a}]\!]_{\rho,\sigma}(\lambda a.update(a,\mathbb{R}[\![\mathtt{x-3}]\!]_{\rho,\sigma},(\lambda v.v+4)))=$
$\sigma\big[(\lambda a.update(a,1,(\lambda v.v+4)))(\sigma(\rho(\mathtt{a})))/\rho(\mathtt{a})\big]=\sigma\big[update(\sigma(l_a),1,(\lambda v.v+4))/l_a\big]=(\{l_x\rightharpoonup 4,\ l_a\rightharpoonup array_3(1,6,3)\},\{\})$.

4.2 Denotation of Statements

Given a sequence of statements *stmts*, an environment ρ, an interpretation $\mathcal{I}$ and a set of traces S, the function $\mathbb{S} : \mathit{stmts} \times \mathit{Env} \times \mathbb{I}_{\mathbb{C}} \times \mathbb{C} \to \mathbb{C}$, written $\mathbb{S}[\![\mathit{stmts}]\!]^{S}_{\rho,\mathcal{I}}$ builds the set of traces corresponding to the effects of *stmts* followed by traces of S, except for the few cases when the control flow does not reach the end of *stmts*. All conditional steps computed for a statement correspond to transitions performed by a process executing that statement, thus describe the *local* behavior (do not take into consideration interleaving with other processes). The denotations corresponding to a process activation are stored into a process spawn **run**. Global behavior is handled after the computation of the fixpoint denotation of process declarations.

We start by showing the denotation for basic PROMELA statements.

$$\mathbb{S}[\![\mathit{Expr}]\!]^{S}_{\rho,\mathcal{I}} := (\lambda\sigma.\mathbb{R}[\![\mathit{Expr}]\!]_{\rho,\sigma} \rhd \mathit{id}) \cdot S \tag{4.1}$$

$$\mathbb{S}[\![l = r]\!]^{S}_{\rho,\mathcal{I}} := \big(\lambda\sigma.\mathit{True} \rhd \lambda\sigma.\mathbb{L}[\![l]\!]_{\rho,\sigma}(\lambda v.\mathbb{R}[\![r]\!]_{\rho,\sigma})\big) \cdot S \tag{4.2}$$

$$\mathbb{S}[\![\mathit{type}\ x = e]\!]^{S}_{\rho,\mathcal{I}} := \big(\lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma\big[\mathbb{R}[\![e]\!]_{\rho,\sigma}/\rho(x)\big]\big) \cdot S \tag{4.3}$$

$$\mathbb{S}[\![\mathit{chan}\ c = [n]\ \mathit{of}\ \{\mathit{types}\}]\!]^{S}_{\rho,\mathcal{I}} := \big(\lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma\big[\mathit{channel}(\mathbb{R}[\![n]\!]_{\rho,\sigma}, \{\mathit{types}\})/\rho(c)\big]\big) \cdot S \tag{4.4}$$

$$\mathbb{S}[\![\mathit{type}\ a[n] = \{e_1,\dots,e_n\}]\!]^{S}_{\rho,\mathcal{I}} := \big(\lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma\big[\mathit{array}_n(\mathbb{R}[\![e_1]\!]_{\rho,\sigma},\dots,\mathbb{R}[\![e_n]\!]_{\rho,\sigma})/\rho(a)\big]\big) \cdot S \tag{4.5}$$

$$\begin{aligned}\mathbb{S}[\![c!\overrightarrow{e}]\!]^{S}_{\rho,\mathcal{I}} := \big(&\lambda\sigma.\sigma(\rho(c)) \neq \bot \wedge \mathit{nfull}(\sigma(\rho(c))) \rhd \\ &\lambda\sigma.\sigma\big[\mathit{push}(\sigma(\rho(c)), \mathbb{R}[\![\overrightarrow{e}]\!]_{\rho,\sigma})/\rho(c)\big]\big) \cdot S\end{aligned} \tag{4.6}$$

$$\begin{aligned}&\mathbb{S}[\![c?x_1,\dots,x_n]\!]^{S}_{\rho,\mathcal{I}} := \big(\lambda\sigma.\sigma(\rho(c)) \neq \bot \wedge \mathit{nempty}(\sigma(\rho(c))) \rhd \\ &\quad\lambda\sigma.\sigma\big[v_1/\rho(x_1)\big]\dots\big[v_n/\rho(x_n)\big]\big[\mathit{pop}(\sigma(\rho(c)))/\rho(c)\big]\big) \cdot S \\ &\quad\text{where } (v_1,\dots,v_n) = \mathit{head}(\sigma(\rho(c)))\end{aligned} \tag{4.7}$$

$$\begin{aligned}&\mathbb{S}[\![\mathtt{run}\ p(\overrightarrow{e})]\!]^{S}_{\rho,\mathcal{I}} := \big(\lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma\big[\overrightarrow{v}/\overrightarrow{l}\,\big]\big[\mathit{pid}/l_{\mathit{pid}}\big]\big[\sigma(\rho(\mathtt{\_nr\_pr}))+1/\rho(\mathtt{\_nr\_pr})\big]\big) \cdot \\ &\quad\mathbf{run}(\mathcal{I}(p(\overrightarrow{l}, \overrightarrow{l_{loc}}, l_{\mathit{pid}}))) \cdot S \\ &\quad\text{where } \overrightarrow{v} = \mathbb{R}[\![\overrightarrow{e}]\!]_{\rho,\sigma},\ \mathit{pid} \text{ is a fresh pid in } \sigma \text{ and } \overrightarrow{l}, \overrightarrow{l_{loc}}, l_{\mathit{pid}} \text{ are fresh locations in } \sigma\end{aligned} \tag{4.8}$$

Equation (4.1) models the evaluation of an expression, whose right-evaluation is used as the condition for the continuation of the trace. Equation (4.2) models an assignment (always enabled) and might alter the state σ. Equations (4.3), (4.4) and (4.5) model, respectively, basic-type variable initialization, channel initialization, and array initialization. These actions are always enabled and modify the system state. Equations (4.6) and (4.7) model asynchronous communication. The conditional transition is executable only if the channel is currently defined and not full (for sending) or not empty (for receiving). The channel valuation is updated as expected.

Finally, Equation (4.8) models a process call of the *run* statement for the interpretation $\mathcal{I}$. All conditional traces start with a conditional transition always enabled that allocates the needed space for the local variables of the new process;[9] stores the values of the actual parameters into (the locations of) the formal parameters; creates the value for the `_pid` variable of the new process; and increments the global variable `_nr_pr` (the number of active processes). The second conditional step is a process spawn that contains the suitable instance of the denotation of the process definition in $\mathcal{I}$, whose "working locations"

[9] It provides fresh locations for the formal parameters, for the variables declared in the body of the proctype declaration and for the variable `_pid`.

are those settled in the first transition. Then it proceeds with a trace of S. The interleaving behavior is handled in a second phase of the semantics computation.

Example 4.2 (Semantics for assignment statement)
Given an environment ρ, a set of traces S and an interpretation $\mathcal{I}$, the semantics for an assignment `x=y` is

$$\mathbb{S}[\![\mathtt{x}=\mathtt{y}]\!]^{S}_{\rho,\mathcal{I}} = \big(\lambda\sigma.\mathit{True} \triangleright \lambda\sigma.\mathbb{L}[\![\mathtt{x}]\!]_{\rho,\sigma}(\lambda v.\mathbb{R}[\![\mathtt{y}]\!]_{\rho,\sigma})\big)\cdot S = \big(\lambda\sigma.\mathit{True} \triangleright \lambda\sigma.\sigma\big[\sigma(\rho(\mathtt{y}))/\rho(\mathtt{x})\big]\big)\cdot S$$

This expression can be read as a set of conditional traces where the first conditional transition of each trace is always enabled and, as expected, it will transform the state σ by copying the value (stored in the location) of variable `y` (i.e., $\sigma(\rho(\mathtt{y}))$) to (the location of) variable `x` (i.e., $\rho(\mathtt{x})$). Then, the rest of the trace is a trace in S.

In the sequel we define the denotation for the various compound statements.

$$\mathbb{S}[\![\mathtt{if}[::\mathit{opt}_i]_{i\in\Lambda}\mathtt{fi}]\!]^{S}_{\rho,\mathcal{I}} := \bigsqcup_{i\in\Lambda}\mathbb{S}[\![\mathit{opt}_i]\!]^{S}_{\rho,\mathcal{I}} \tag{4.9}$$

For the sake of simplicity we present the simplest version of the selection *if* statement without the else-option. It assumes that Λ is the set of indexes identifying the different options (in the *if* statement). Its denotation is the union of all conditional traces associated with each option. The else-option is handled by adding its traces but with the first conditional transition modified with the conjunction of the negations of the conditions of the initial transitions of the traces of all the other options.

$$\begin{aligned}\mathbb{S}[\![\mathtt{do}[::\mathit{opt}_i]_{i\in\Lambda}\mathtt{od}]\!]^{S}_{\rho,\mathcal{I}} &:= \{s_0\cdot\ldots\cdot s_m\cdot s \mid m\in\mathbb{N}, 0\le j\le m, s_j\in S_L, s'\cdot\bullet\in S_\bullet, s\in s'\cdot S\}\sqcup\\ &\quad\{s_0\cdot\ldots\cdot s_m \mid m\in\mathbb{N}, 0\le j\le m, s_j\in S_L\}\cdot S_\infty\sqcup\{s_0\cdot\ldots\cdot s_j\cdot\ldots \mid \forall j\in\mathbb{N}, s_j\in S_L\}\\ &\quad\text{where } (S_\infty,S_L,S_\bullet) = \mathit{split}\Big(\bigsqcup_{i\in\Lambda}\mathbb{S}[\![\mathit{opt}_i]\!]^{\{\bullet\}}_{\rho,\mathcal{I}}\Big)\end{aligned} \tag{4.10}$$

We present the simplest version of the iteration statement *do* without the else-option. The $\bullet$ marker is used to identify when a *break* statement is reached. We introduce an auxiliary function *split* which, given a set of traces S, classifies them into three disjoint sets. More specifically, $\mathit{split}(\bar{S}) = (S_\infty, S_L, S_\bullet)$ where S_∞ has all infinite traces of $\bar{S}$, S_L contains finite traces in $\bar{S}$ not containing the $\bullet$ marker (which are traces corresponding to an iteration of the loop), and $S_\bullet$ contains the rest of finite traces in $\bar{S}$ (ending with $\bullet$ by construction). We abuse notation and allow the join of a set of traces that might contain $\bullet$. We assume to extend the definition of $\le$ as $\bullet\le\bullet$. Thus the denotation of the iteration statement is the union of an arbitrary (finite) repetition of the (partial) traces of S_L followed by an infinite trace; an infinite repetition of the traces of S_L; an arbitrary (finite) repetition of the traces of S_L followed by a trace of $S_\bullet$ where the (ending) $\bullet$ is replaced by a trace of the continuation S.

The denotation of a sequence of statements is defined to support Equation (4.10).

$$\mathbb{S}[\![\mathit{stmt};\mathit{stmts}]\!]^{S}_{\rho,\mathcal{I}} := \begin{cases} S & \text{if } \mathit{stmt} = \mathtt{break}\\ \mathbb{S}[\![\mathit{stmt}]\!]^{\mathbb{S}[\![\mathit{stmts}]\!]^{S}_{\rho,\mathcal{I}}}_{\rho,\mathcal{I}} & \text{otherwise}\end{cases} \qquad \mathbb{S}[\![\varepsilon]\!]^{S}_{\rho,\mathcal{I}} := \begin{cases}\{\varepsilon\} & \text{if } S=\{\bullet\}\\ S & \text{otherwise}\end{cases} \tag{4.11}$$

The denotation of a sequence of statements without `break` with a “proper” continuation S is the concatenation of the traces of the statements of the sequence, followed by traces in S. In case the continuation is “improper” (i.e., $\{\bullet\}$) then we will have only the traces of the statements of the sequence (that will populate the set S_L of Equation (4.10)). If instead we have a `break` then we proceed immediately with the continuation S.

Example 4.3 (Semantics for the Sequence statement)

Let us show how $\mathbb{S}$ uses the continuation parameter by means of an example. Given the sequence `x++;y=8;break;y=2` and a set of traces $S \neq \{\bullet\}$, its denotations is computed as follows:
$\mathbb{S}[\![\texttt{x++;y=8;break;y=2}]\!]^{S}_{\rho,\mathcal{I}} = \mathbb{S}[\![\texttt{x++}]\!]^{\mathbb{S}[\![\texttt{y=8;break;y=2}]\!]^{S}_{\rho,\mathcal{I}}}_{\rho,\mathcal{I}} = \lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma\big[\sigma(\rho(\texttt{x}))+1/\rho(\texttt{x})\big]\cdot$
$\mathbb{S}[\![\texttt{y=8;break;y=2}]\!]^{S}_{\rho,\mathcal{I}} = \lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma\big[\sigma(\rho(\texttt{x}))+1/\rho(\texttt{x})\big]\cdot \mathbb{S}[\![\texttt{y=8}]\!]^{\mathbb{S}[\![\texttt{break;y=2}]\!]^{S}_{\rho,\mathcal{I}}}_{\rho,\mathcal{I}} =$
$\lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma\big[\sigma(\rho(\texttt{x}))+1/\rho(\texttt{x})\big]\cdot \lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma\big[8/\rho(\texttt{y})\big]\cdot \mathbb{S}[\![\texttt{break;y=2}]\!]^{S}_{\rho,\mathcal{I}} =$
$\lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma\big[\sigma(\rho(\texttt{x}))+1/\rho(\texttt{x})\big]\cdot \lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma\big[8/\rho(\texttt{y})\big]\cdot S.$

$$\mathbb{S}[\![\texttt{atomic\{}\mathit{body}\texttt{\}}]\!]^{S}_{\rho,\mathcal{I}} := \bigsqcup\big\{s\cdot S \,\big|\, s \in \mathit{Atomics}(R)\big\} \quad \text{where} \tag{4.12}$$
$$R := \mathit{MoveSpawnsBack}(\mathbb{S}[\![\mathit{body}]\!]^{\perp}_{\rho,\mathcal{I}})$$
$\mathit{MoveSpawnsBack}$ moves all **run** to the end of conditional traces
$$\mathit{Atomics}(S) := \big\{(\phi_1\circ\dots\circ\phi_n)\cdot rs \,\big|\, \phi_1\cdot\ldots\cdot\phi_n\cdot rs \in S,\ rs \text{ can contain only } \textbf{run}\big\}\sqcup$$
$$\bigsqcup\big\{(\lambda\sigma.\phi^C(\sigma)\wedge\neg\phi^C_{k+1}(\sigma) \rhd \phi^T)\cdot \mathit{Atomics}(\{\phi_{k+1}\cdot\ldots\cdot\phi_n\cdot rs\}) \,\big|$$
$$\phi_1\cdot\ldots\cdot\phi_n\cdot rs \in S,\ rs \text{ can contain only } \textbf{run},\ 0\le k<n,\ \phi=\phi_1\circ\dots\circ\phi_k\big\}$$
$$(\phi^C_1 \rhd \phi^T_1)\circ(\phi^C_2 \rhd \phi^T_2) := \lambda\sigma.\phi^C_1(\sigma)\wedge\phi^C_2(\sigma) \rhd \lambda\sigma.\phi^T_2(\phi^T_1(\sigma))$$

We assume that the value of expression "$\phi_1\circ\dots\circ\phi_k$" when $k=0$ is $\lambda\sigma.\mathit{True} \rhd \mathit{id}$. Note that the sentence "*rs* can contain only **run**" includes also the case $rs=\varepsilon$ (when there are no spawns in the trace).

The atomic construct is defined by performing at once all consecutive enabled statements in its body. To formalize this, the definition considers all traces generated by $\mathbb{S}[\![\mathit{body}]\!]^{\perp}_{\rho,\mathcal{I}}$, where all process spawns are postponed at the end,[10] and constructs the sub-traces $\phi_1,\ldots,\phi_k$ whose steps are enabled, by creating a condition such that all conditions of $\phi_1,\ldots,\phi_k$ are true and the condition of ϕ_{k+1} is false. We use the operator $\circ$ that sequentially combines the effects of two different conditional transitions into a single transition.

Example 4.4 (Semantics for atomic sequence of statements)

Consider the sequence of statements stmts = `x>0->n=1; y<3->m=2`. Then, $S := \mathbb{S}[\![\mathit{stmts}]\!]^{\perp}_{\rho,\mathcal{I}} = \lambda\sigma.\sigma(\rho(\texttt{x}))>0 \rhd \mathit{id}\cdot\lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma\big[1/\rho(\texttt{n})\big]\cdot\lambda\sigma.\sigma(\rho(\texttt{y}))<3 \rhd \mathit{id}\cdot\lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma\big[2/\rho(\texttt{m})\big]$. Since no trace in S contains **run**, $\mathit{MoveSpawnsBack}(S) = S$. We have

$$\mathit{Atomics}(S) = \{\phi_0\cdot\ldots\}\sqcup\bigsqcup_{k\in\mathbb{N}}\underbrace{\phi_0\cdot\ldots\cdot\phi_0}_{k}\cdot\big(\{\phi_1\}\sqcup\phi_3\cdot\big(\{\phi_2\cdot\ldots\}\sqcup\{\underbrace{\phi_2\cdot\ldots\cdot\phi_2}_{j}\cdot\phi_4 \mid j\in\mathbb{N}\}\big)\big) \text{ where}$$

$$\phi_0 := \lambda\sigma.\sigma(\rho(\texttt{x}))\le 0 \rhd \mathit{id} \qquad \phi_1 := \lambda\sigma.\sigma(\rho(\texttt{x}))>0\wedge\sigma(\rho(\texttt{y}))<3 \rhd \lambda\sigma.\sigma\big[1/\rho(\texttt{n})\big]\big[2/\rho(\texttt{m})\big]$$
$$\phi_2 := \lambda\sigma.\sigma(\rho(\texttt{y}))\ge 3 \rhd \mathit{id} \qquad \phi_3 := \lambda\sigma.\sigma(\rho(\texttt{x}))>0\wedge\sigma(\rho(\texttt{y}))\ge 3 \rhd \lambda\sigma.\sigma\big[1/\rho(\texttt{n})\big]$$
$$\phi_4 := \lambda\sigma.\sigma(\rho(\texttt{y}))<3 \rhd \lambda\sigma.\sigma\big[2/\rho(\texttt{m})\big]$$

Hence,

$$\mathbb{S}[\![\texttt{atomic\{}\mathit{stmts}\texttt{\}}]\!]^{R}_{\rho,\mathcal{I}} = \{\phi_0\cdot\ldots\}\sqcup\bigsqcup_{k\in\mathbb{N}}\underbrace{\phi_0\cdot\ldots\cdot\phi_0}_{k}\cdot\big(\phi_1\cdot R\sqcup\phi_3\cdot\big(\{\phi_2\cdot\ldots\}\sqcup\{\underbrace{\phi_2\cdot\ldots\cdot\phi_2}_{j}\cdot\phi_4\cdot R \mid j\in\mathbb{N}\}\big)\big)$$

[10] This definition follows PROMELA's behavior that postpones the creation of processes to the end of atomic blocks.

4.3 Denotation of a PROMELA Program

We are ready to provide the semantic function that computes the traces associated with a PROMELA program. In order to simplify the definitions but without loss of generality, we assume that programs accepted by this function are normalized. Essentially, for all processes declared as active we add a suitable run statement in the init process, incapsulated into an atomic block. Also, we add an initialization to the proper default value to all local variable declarations without an initialization. We move all global variable declarations to the top. Moreover, for global variable declarations with initialization, we remove such initialization and add a corresponding initialization assignment at the top of the init process. We remove *printf* statements. For instance, the program in Figure 1 in normalized form is:

```
byte n;                                proctype p() {
                                         n = 5
init {                                 }
  n = 0;                               proctype q() {
  atomic { run p(); run q() }            n = 8
}                                      }
```

In order to define the semantic function for programs, let us start by defining the denotation for process declarations. Given a list of process declarations, an environment, and an interpretation, the function $\mathbb{D}: Proctypes \times Env \times \mathbb{I}_{\mathbb{C}} \to \mathbb{I}_{\mathbb{C}}$ (defined below) applies the effect of one step of the computation to the input interpretation.

$$\mathbb{D}[\![proctype\ p_1(\overrightarrow{x_1})\{body_1\};\ldots;proctype\ p_n(\overrightarrow{x_n})\{body_n\}]\!]_{\rho_g,\mathcal{I}} := \begin{cases} p_1(\overrightarrow{l_1},\overrightarrow{l_1^v},l_1^p) \mapsto \mathbb{S}[\![body_1]\!]^{End}_{\rho_1,\mathcal{I}} \\ \vdots \\ p_n(\overrightarrow{l_n},\overrightarrow{l_n^v},l_n^p) \mapsto \mathbb{S}[\![body_n]\!]^{End}_{\rho_n,\mathcal{I}} \end{cases}$$

where $\rho_i := \rho_g\big[\overrightarrow{l_i}/\overrightarrow{x_i}\big]\big[\overrightarrow{l_i^v}/\overrightarrow{v_i}\big]\big[l_i^p/\texttt{\_pid}\big]$ (4.13)

$\overrightarrow{v_i}$ are the local variables of $body_i$

$End := \{\lambda\sigma.True \rhd \lambda\sigma.\sigma\big[\sigma(\rho(\texttt{\_nr\_pr}))-1/\rho(\texttt{\_nr\_pr})\big]\cdot\blacksquare\}$

End is the set of traces representing the final step of each process, i.e., is responsible for decreasing the counter of process instances and then stop.

We can now introduce $\mathbb{P}: Prog \times State \to \mathbb{T}$ where $\mathbb{T}$ is the domain of sets of sequences of states in *State*. Note that the definition uses three auxiliary functions that are formalized later. Given a normalized program $P \in Prog$ with init process *init*, global variable declarations *vDecls* and a list of process declarations *pDecls*, we define

$$\mathbb{P}[\![vDecls;pDecls]\!]_{\sigma_0} := cmpct\Big(prpgt\Big(\sigma_0, intrlv\Big(\big(\bigsqcup_{k\in\mathbb{N}}(\lambda\mathcal{I}.\mathbb{D}[\![pDecls]\!]_{\rho_v,\mathcal{I}})^k(\lambda\kappa.\bot)\big)(init(l_0))\Big)\Big)\Big)$$

where l_0 is the location reserved for the `_pid` variable of *init* and ρ_v is the environment populated with associations, of both the language predefined variables and the global variables declared in *vDecls*, to suitable (different fresh) locations.

$\mathbb{P}$ first computes the least fixpoint $\mathcal{F}$ of the function $\mathbb{D}$; the result is used to compute the interleaving of spawned processes and then the initial state σ_0 is propagated to the conditional traces of $\mathcal{F}(init)$; Finally, transitions that do not change the state are removed. $\mathbb{P}$ is well defined since $\mathbb{D}$ is continuous.

Given a set of conditional traces, $intrlv : \mathbb{C} \to \mathbb{C}$ replaces the process spawns $\mathbf{run}(S)$ of each conditional trace with all possible interleavings of S with the rest of the trace. It uses the function $shuffle : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ that takes two sets of conditional traces and returns the set of all possible interleavings, having care of removing all $\blacksquare$ that can appear in the middle of an interleaving.

$$intrlv(S) := \bigsqcup\{inter(s) \mid s \in S\} \text{ where}$$

$$inter(\blacksquare) := \{\blacksquare\} \qquad inter(\phi \cdot s) := \phi \cdot inter(s)$$

$$inter(\varepsilon) := \{\varepsilon\} \qquad inter(\mathbf{run}(S) \cdot s) := \bigsqcup shuffle(intrlv(S), inter(s))$$

Given a set S of conditional traces *not containing process spawns*, $prpgt : State \times \mathbb{C} \to \mathbb{T}$ propagates the initial state σ_0 through each conditional trace in S, giving as a result sequences of states. If S is not $\perp$ then it can be partitioned depending on the initial conditional step of all its conditional traces. By construction we can possibly have just one trace that starts with $\blacksquare$. Thus all traces of $S \setminus \{\blacksquare\}$ must begin with a finite number of conditional transitions. Thus we can formally define $prpgt$ as

$$prpgt(\sigma, \perp) = \{\psi\}$$

$$prpgt(\sigma, \{\blacksquare\} \cup S) = \{\xi\} \cup prpgt(\sigma, S)$$

$$prpgt(\sigma, \phi_1 \cdot S_1 \cup \cdots \cup \phi_k \cdot S_k) = \bigcup\left\{\phi_i^T(\sigma) \cdot prpgt(\phi_i^T(\sigma), S_i) \,\middle|\, \phi_i^P(\sigma) = True\right\}$$

Here ξ denotes the end state and ψ blocking states. At each step, the next conditional transition of each trace is evaluated, obtaining a new state in the collected sequence. When a trace has no conditional transition it is not considered as input for the following iteration.

Finally, the auxiliary function $cmpct : \mathbb{T} \to \mathbb{T}$, given a set of sequences of states, gets rid of consecutive equal states.

Let us illustrate the semantics computation by means of an example.

EXAMPLE 4.5 (SIMPLE INTERLEAVING EXAMPLE)

Let us call P the normalized program shown fo far. Let us call $body_{init}$ the body of the init process and let D be the process declarations, i.e., $D = \texttt{init}\{body_{init}\};\texttt{proctype p}()\{\texttt{n} = 5\};\texttt{proctype q}()\{\texttt{n} = 8\}$. Then, the semantics of the whole program is

$$\mathbb{P}[\![P]\!]_{\sigma_0} := cmpct\Big(prpgt\Big(\sigma_0, intrlv\Big(\big(\bigsqcup_{k\in\mathbb{N}}(\lambda\mathcal{I}.\mathbb{D}[\![D]\!]_{\rho_G,\mathcal{I}})^k(\lambda\kappa.\perp)\big)(init(l_0))\Big)\Big)\Big)$$

where we assume that the global environment ρ_G binds `n` to l_n and `_nr_pr` to l_c. Moreover l_0 is the location reserved for the `_pid` variable of *init*. By Equation (4.13),

$$\mathbb{D}[\![D]\!]_{\rho_G,\mathcal{I}} = \begin{cases} p(l_p) \mapsto \mathbb{S}[\![\texttt{n=5}]\!]^{\{end\}}_{\rho_G\left[{l_p}/{\texttt{\_pid}}\right],\mathcal{I}} \\ q(l_p) \mapsto \mathbb{S}[\![\texttt{n=8}]\!]^{\{end\}}_{\rho_G\left[{l_p}/{\texttt{\_pid}}\right],\mathcal{I}} \\ init(l_p) \mapsto \mathbb{S}[\![body_{init}]\!]^{\{end\}}_{\rho_G\left[{l_p}/{\texttt{\_pid}}\right],\mathcal{I}} \end{cases}$$

where $end = \lambda\sigma.True \rhd \lambda\sigma.\sigma\left[\sigma(l_c) - 1/l_c\right] \cdot \blacksquare$. Since for all $k \geq 2$,

$$(\lambda\mathcal{I}.\mathbb{D}[\![D]\!]_{\rho_G,\mathcal{I}})^k(\lambda\kappa.\perp) = \begin{cases} p(l_p) \mapsto \{\lambda\sigma.True \rhd \lambda\sigma.\sigma\left[5/l_n\right] \cdot end\} \\ q(l_p) \mapsto \{\lambda\sigma.True \rhd \lambda\sigma.\sigma\left[8/l_n\right] \cdot end\} \\ init(l_p) \mapsto \{\lambda\sigma.True \rhd \lambda\sigma.\sigma\left[0/l_n\right] \cdot \\ \qquad \lambda\sigma.True \rhd \lambda\sigma.\sigma\left[1/l_{pPid}\right]\left[2/l_{qPid}\right]\left[\sigma(l_c) + 2/l_c\right] \cdot \\ \qquad \mathbf{run}(\{\lambda\sigma.True \rhd \lambda\sigma.\sigma\left[5/l_n\right] \cdot end\}) \cdot \\ \qquad \mathbf{run}(\{\lambda\sigma.True \rhd \lambda\sigma.\sigma\left[8/l_n\right] \cdot end\})\} \end{cases}$$

this is also the fixpoint. Let us call it F and let $\phi_e := \lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma[\sigma(l_c)-1/l_c]$. We have

$$\begin{aligned}
&\mathit{intrlv}(F(\mathit{init}(l_0))) = \lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma[0/l_n]\cdot\lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma[1/l_{pPid}][2/l_{qPid}][\sigma(l_c)+2/l_c]\cdot\\
&\quad\{\lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma[5/l_n]\cdot\phi_e\cdot\lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma[8/l_n]\cdot\phi_e\cdot\blacksquare,\\
&\quad\ \lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma[5/l_n]\cdot\lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma[8/l_n]\cdot\phi_e\cdot\phi_e\cdot\blacksquare,\\
&\quad\ \lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma[8/l_n]\cdot\phi_e\cdot\lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma[5/l_n]\cdot\phi_e\cdot\blacksquare,\\
&\quad\ \lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma[8/l_n]\cdot\lambda\sigma.\mathit{True} \rhd \lambda\sigma.\sigma[5/l_n]\cdot\phi_e\cdot\phi_e\cdot\blacksquare\}
\end{aligned}$$

Then, for any σ_0 where $\sigma_0(l_c) = 0$,

$$\begin{aligned}
&\mathit{prpgt}(\sigma_0, \mathit{intrlv}(F(\mathit{init}(l_0)))) = \sigma_0\cdot\underbrace{\sigma_0[0/l_n]}_{\sigma_1}\cdot\underbrace{\sigma_1[1/l_{pPid}][2/l_{qPid}][2/l_c]}_{\sigma_2}\cdot\\
&\quad\{\underbrace{\sigma_2[5/l_n]}_{\sigma_3}\cdot\underbrace{\sigma_3[1/l_c]}_{\sigma_4}\cdot\underbrace{\sigma_4[8/l_n]}_{\sigma_5}\cdot\underbrace{\sigma_5[0/l_c]}_{\sigma_6},\\
&\quad\ \underbrace{\sigma_2[5/l_n]}_{\sigma_3}\cdot\underbrace{\sigma_3[8/l_n]}_{\sigma_4}\cdot\underbrace{\sigma_4[1/l_c]}_{\sigma_5}\cdot\underbrace{\sigma_5[0/l_c]}_{\sigma_6},\\
&\quad\ \underbrace{\sigma_2[8/l_n]}_{\sigma_3}\cdot\underbrace{\sigma_3[1/l_c]}_{\sigma_4}\cdot\underbrace{\sigma_4[5/l_n]}_{\sigma_5}\cdot\underbrace{\sigma_5[0/l_c]}_{\sigma_6},\\
&\quad\ \underbrace{\sigma_2[8/l_n]}_{\sigma_3}\cdot\underbrace{\sigma_3[5/l_n]}_{\sigma_4}\cdot\underbrace{\sigma_4[1/l_c]}_{\sigma_5}\cdot\underbrace{\sigma_5[0/l_c]}_{\sigma_6}\}
\end{aligned}$$

This is also $\mathbb{P}[\![P]\!]_{\sigma_0}$ because all consecutive states are different.

5 Conclusion and future work

This work presents a first proposal of a bottom-up fixpoint semantics for PROMELA, which is a widely used modeling language for concurrent, reactive systems. The presented semantics enjoys the good properties to be the basis for the implementation of verification and analysis techniques. First, it is goal-independent, which allows us to collect all possible behaviors from most general calls, avoiding the need of computing the semantics individually for each possible goal. Second, it focusses on being condensed in order to make computation of the semantics possible. Finally, it is bottom-up which improves both convergence and precision in the abstract-interpretation setting for analysis and verification. This is because the *join* operator of the abstract domain is used less times than in a top-down approach.

However, so far we have not formally proved its full abstraction w.r.t. the behavior of the language, which is the most relevant property to guarantee correctness of the analyses. The semantics is designed with the goal of being fully abstract. It has been refined guided by quite demanding examples and counterexamples, thus we are optimistic that after some additional work we can provide the results and proofs of full abstraction.

For the sake of clarity we have presented the version of the semantics without synchronous communications. Actually, we handle rendezvous by mimicking the strategy of the *semantics engine*, that uses the *handshake* variable to prioritize the execution of a message passing through a synchronous channel. Namely, in the semantics formalization, to all conditions (of conditional transitions) we add the requirement for variable *handshake* to be 0 (no synchronous communication is *pending*). Clearly, the rules for the send/receive of messages through synchronous channels update/consult the *handshake* variable appropriately.

We plan to extend the semantics to handle *goto*s, which are extensively used by the PROMELA community. The idea is to follow the classical approach with continuations used to define the semantics of imperative languages with *goto*.

As we just mentioned, the definition of a condensed, bottom-up, fixpoint semantics is the first step towards the definition of abstract interpretation-based analysis and verification techniques on this semantics. This implies that there is large room for future applications of this work. In the longer term we aim to define verification techniques and tools inspired in those defined for the tccp language in [4].

References

[1] M. Ben-Ari (2008): *Principles of the Spin Model Checker*, 1 edition. Springer-Verlag, London, doi:10.1007/978-1-84628-770-1. 2

[2] E. M. Clarke & E. A. Emerson (1982): *Design and synthesis of synchronization skeletons using branching time temporal logic*. In D. Kozen, editor: Logics of Programs 1981, Lecture Notes in Computer Science 131, Springer, Berlin, Heidelberg. 1

[3] M. Comini, L. Titolo & A. Villanueva (2013): *A Condensed Goal-Independent Bottom-Up Fixpoint Modeling the Behavior of tccp*. Available at http://www.dimi.uniud.it/comini/Papers/. Submitted for publication. 3

[4] M. Comini, L. Titolo & A. Villanueva (2014): *Abstract Diagnosis for tccp using a Linear Temporal Logic*. Theory and Practice of Logic Programming 14(4-5), pp. 787–801, doi:10.1017/S1471068414000349. Available at http://www.scopus.com/inward/record.url?eid=2-s2.0-84904677112&partnerID=40&md5=7b5f838e06ff4894949cd38c0789e225. 1, 5

[5] H. Erdogmus (1996): *Verifying Semantic Relations in SPIN*. In: Proceedings of the First SPIN Workshop. Available at https://www.researchgate.net/publication/2929105_Verifying_Semantic_Relations_in_SPIN/. 1

[6] M.M. Gallardo, P. Merino & E. Pimentel (2004): *A generalized semantics of PROMELA for abstract model checking*. Formal Asp. Comput. 16(3), pp. 166–193, doi:10.1007/s00165-004-0040-y. Available at https://doi.org/10.1007/s00165-004-0040-y. 2.1, 4

[7] G. J. Holzmann (2004): *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley. Available at https://books.google.it/books?id=NpBdZKmOHO8C. 1, 2, 4

[8] G. J. Holzmann (2012): *Spin Manual*. Available at http://spinroot.com/spin/Man/promela.html. 2, 2.1, 2.1

[9] V. Natarajan & G. J. Holzmann (1996): *Outline for an Operational Semantics of PROMELA*. In: The SPIN Verification System. Proceedings of the Second SPIN Workshop, DIMACS 32, AMS. 4

[10] A Pnueli (1971): *The temporal logic of programs*. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science (Providence, RI.), IEEE, New York, pp. 46–57. 2

[11] J.-P. Queille & J. Sifakis (1982): *Specification and Verification of Concurrent Systems in CESAR*. In: Proceedings of the 5th Colloquium on International Symposium on Programming, Springer-Verlag, London, UK, pp. 337–351. Available at http://dl.acm.org/citation.cfm?id=647325.721668. 1

[12] C. Weise (1997): *An incremental formal semantics for PROMELA*. In: Proceedings of the Third SPIN Workshop, SPIN97. 4