A denotational semantics for PROMELA
addressing arbitrary jumps

Marco Comini\textsuperscript{1}\textsuperscript{[0000\textendash}0002\textendash}8069\textendash}3411, María del Mar Gallardo\textsuperscript{2}\textsuperscript{[0000\textendash}0003\textendash}3481\textendash}5307 \ast, and Alicia Villanueva\textsuperscript{3}\textsuperscript{[0000\textendash}0003\textendash}1090\textendash}5009 \ast\ast

\textsuperscript{1} Università degli Studi di Udine (Italy)
marco.comini@uniud.it
\textsuperscript{2} Universidad de Málaga, Andalucía Tech,
ITI-Software
mdgallardo@uma.es
\textsuperscript{3} VRAIN - Universitat Politècnica de València
alvilgal@upv.es

Abstract 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 tech-
niques based on abstract interpretation would be defined on top of such
semantics.

1 Introduction

Concurrent programming has posed additional difficulties to the problem of guar-
anteeing 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.

\ast This author has been partially supported by the Spanish MCIU under grant RTI2018-099777-B-I00.
\ast\ast This author has been partially supported by the EU (FEDER) and the Spanish MCIU under grant RTI2018-094403-B-C32, and by Generalitat Valenciana under grant PROMETEO/2019/098.
In this context, PROMELA and SPIN provide the tools for effectively model check 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. A tool to check some of the classical equivalences defined for PROMELA is presented in [5].

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. Depending on the kind of technique to be used, denotational semantics with some properties can be more convenient than others. On top of the concrete inductive, fixpoint semantics defined in this work, it will be possible to define appropriate abstract semantics which implement specific analysis, for instance related to the temporal behavior of the system. The experience with the semantics proposed for the tccp language [4] showed that good results are obtained when abstraction is based on a concrete inductive, fixpoint semantics. To the best of our knowledge, in the literature, we only find proposals for (transitional) operational semantics for PROMELA.

The semantics defined in this paper works in two phases. Intuitively, at a process level we first collect for each process declaration a set of hypothetical traces corresponding to the set of behaviors independent from the context. As one can imagine, some of these traces may become impossible to travel given a context (a set of processes run in parallel). Then, in a second phase at a system level, the traces from the processes running in parallel are combined to obtain global traces. During this phase, some traces might be discarded (for instance, when a synchronous communication is requested but there is no process able for handshaking).

Due to the PROMELA language spirit, our plan is to focus on global (concurrent) properties, namely on defining analyses on top of the sets obtained at the system level. We note that it would be possible to define analyses for local processes (on top of the sets obtained after the first phase) for analyzing the behavior of local variables. However, these would be in general of less interest since, 1) PROMELA defines concurrency only at the level of the system (in contrast to other concurrency models, there is no parallel operator among the basic constructs), and 2) the language is strongly designed to precisely capture the concurrent behavior and abstract as much as possible the local behavior of processes (as a means to reduce the number of states to explore).

One of the purposes of the PROMELA language is to facilitate the modeling of concurrent state machines. This is why it includes the goto statement that is intensively used by programmers to implement jumps from source to target states. It is well known that dealing with arbitrary jumps constitutes a technical difficulty when defining a semantics by induction on syntax. In this paper, for
the sake of conciseness, we build the process denotations based on the static graph provided by the process code.

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 [8]. Its specification is given on the SPIN website. Other important sources are [9,1]. 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 ([12]). 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, to consider 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 present 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 using the CSP notation [7].

2.1 Syntax and informal operational semantics of PROMELA

For space reasons, we do not describe the syntax and operational semantics of PROMELA constructors exhaustively in this paper. Instead, we give an intuitive description of the main features of the PROMELA language by means of examples. As it will be clear in the following discussion, the syntax of the language is similar to C. A PROMELA program consists of a sequence of global declarations (types and variables), process declarations (proctype) and the (optional) init process used to initialize the system, when required.

Consider as running example the PROMELA code given in Figure 1. The program corresponds to a variant of the well-known first attempt to the Peterson’s algorithm [11] to solve the mutual exclusion problem for two processes. The algorithm contains two processes $P(0)$ and $P(1)$ which are trying continuously
to enter in its critical section (CS). To avoid to violate the mutual exclusion property (that is, to ensure that always there is at most one process executing its critical section), each process uses a boolean flag \(f[0]\) and \(f[1]\). When a process wants to enter its CS, it changes its own boolean to true and waits for the other process to be outside its CS (its boolean is false). After executing the CS, each process sets its own boolean to false. Although this first attempt solves the mutual exclusion requirement, it contains many possible deadlocked executions.

As can be observed, the program starts by defining, in Line 1, a global array \(f\) with two components of the predefined type \(\text{bit}\). Both components are initialized to false \((0)\). Next, in Lines 2-8, the proctype \(P\) is declared. Proctypes are similar to C functions (they can contain local types, variables and executable code) except for, when invoked, their code is always executed in a new execution thread. When processes do not need initial values, PROMELA allows to directly declare and instantiate them placing the keyword \(\text{active}\) before \(\text{proctype}\).

The \(\text{init}\) process in Line 9 acts as the main function in C. In our example it is used to create by means of statement \(\text{run}\) two instances of the proctype \(P\), one with \(id=0\) and the other with \(id=1\). When processes are created, SPIN identifies them through its \(\text{pid}\) which is an integer different for each process instance.

The evolution of a PROMELA program proceeds by freely interleaving the instructions of the processes in execution. To completely understand this behavior is mandatory to introduce the notion of \textit{executability} in PROMELA. SPIN selects an instruction to be next executed only if the instruction is an executable instruction of a process, that is, a statement that does not suspend. If no executable statement can be selected, the system blocks. Assignments (as \(f[id]=1\) in Line 3) and \(\text{run}\) statements are always executable. However, boolean expressions (as \(!f[1-id]\) in Line 4) are only executable when they are true. This is the natural synchronization mechanism based in shared memory implemented in PROMELA. A similar behavior occurs for instance when processes communicate using shared channels. The send or receive channel operations are only executable if the corresponding channels are not full or not empty, respectively.
The atomic statement in Line 9 is used to avoid that the init process loses the execution control during the creation of the two new processes. When a statement inside an atomic statement is non-executable (as the boolean expression of Line 4 when it evaluates to false), SPIN breaks the atomic process execution to avoid the system deadlock. Thus, the atomic statement should be read as “execute the code in an atomic manner while it is possible”. An atomic statement is executable iff its first nested statement is. For instance, atomic in Line 3 is always executable since its first nested statement f[id] = 1 is an assignment that can be always be executed.

Channels are declared through variable declarations by using the chan type. The declaration chan c = [N] of {bit,mtype} defines a channel c of capacity N for storing N records composed of a bit and an mtype (enumerated types that users can freely define). The send and receive operations on channels are written using the CSP notation. For instance, c?x,y and c!1,m are operations on channel c defined above, where we are assuming that m is a value of type mtype, and x and y variables of appropriate type. If the capacity N is greater than zero, a buffered channel is 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.

The control statements in PROMELA are do/od and if/fi, although do/od is not really needed since it can be simulated with if/fi and goto. These statements introduce the non-determinism in the language. An if/fi statement such as the one in Lines 3-8 of Figure 1 starts with the keyword if, ends with fi and contains one or more branches that begin with symbols :: (the instruction in the example has only one branch). The execution of an if statement consists in the execution of one of its branches. To do this, SPIN selects one of the executable branches non-deterministically. A branch is executable if its first instruction (the guard) is executable. If no branch is executable, then the if/fi instruction suspends. In the case of the if/fi statement of Figure 1 as its guard (the atomic statement of Line 4) is always executable, the selection never suspends. It is also possible to add an unique ending else branch to statements if/fi. In this case, the if/fi never suspends, since when no branch is executable the else branch is selected.

As we observe in Figure 1, PROMELA statements can be labeled using identifiers. For instance, L3 is the label of instruction f[id]=0.

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). Other read-only predefined variables handled by the semantic engine are _pid, which is local to each process and contains the corresponding process identifier, and _nr_pr which stores the number of active processes in the system.
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 to execute and updates the global state. There are some statements that might globally influence executability and that are handled by means of other internal variables. For instance, a side effect of the execution of any statement in an atomic block (except the last one) is to set a variable exclusive to the pid of the executing process, thus preserving the exclusive privilege to execute. We leave the rest of non-local constructs (such as timeout or unless) as future work.

In the model, there is no explicit characterization of time. The semantics engine keeps executing transitions until no executable one remains. This happens if either the number of active processes is zero, or when a deadlock occurs. The interested reader can consult [9] 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 states 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 instances). 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 and channel instances. Due to space limitations, we do not formally introduce the different components of the system (basic types, arrays, records and their associated operators) but we informally introduce on demand the needed notation. In the following, we use $V$ for elements that can be stored in memory and $Channels$ for the denotations of channels.

**Definition 1 (System State).** A system state of a PROMELA program is an element of the domain $State = (\{Loc \rightarrow V\} \times \{ChanId \rightarrow Channels\})_\bot$ where Loc are locations of a memory capable to store denotable values $V$, that are basic values, structures, and arrays$^5$ and ChanId is the set of channel ids; Channels is the set of all channel instances; $\bot$ denotes an erroneous (inconsistent) state.

$^5$ For all set $D$, we write $D_\bot$ for the flat domain of values of $D$ with bottom $\bot$. 
In the sequel, we write \( \sigma = (\sigma_L, \sigma_C) \) as a typical element of \( \text{State} \). Moreover, for \( l \in \text{Loc} \) we write \( \sigma(l) \) as a shorthand of \( \sigma_L(l) \) or \( \bot \) when \( \sigma = \bot \). Analogously for \( c \in \text{ChanId} \).

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

**Definition 2 (Environment).** An environment is a partial function associating program identifiers \( \text{Ident} \) with locations or channel ids \( \text{Env} = [\text{Ident} \rightarrow \text{Loc} \cup \text{ChanId}] \).

In the sequel, we write \( \rho \) as a typical element of the set of \( \text{Env} \).

### 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 (Conditional step and conditional traces).** A conditional step is either

- a **conditional transition** \( p \triangleright t \) which consists of a state transformer \( t : \text{State} \rightarrow \text{State} \) guarded by a condition \( p : \text{State} \rightarrow \text{Bool} \), or
- a **finalizer**, written \( \blacksquare \), or
- a **process spawn**, written \( \text{run}(\Psi) \), where \( \Psi \) is a non-empty set of conditional traces.

For conditional transitions and process spawns inside atomic blocks, we can add the markers \( \overset{\text{p}}{\triangleright} t \) and \( \text{run}(\Psi) \). In the sequel, we refer to the marked or non-marked version of conditional steps and process spawns indifferently, except in those cases when it is necessary to distinguish them. Given a conditional transition \( \phi = p \triangleright t \) (or \( \phi = p \triangleright t \)) we use the notation \( \phi^C \) and \( \phi^T \) to denote \( p \) and \( t \).

A conditional trace is defined as

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

We use \( \varepsilon \) to denote the empty trace (of length zero). We denote by \( \mathcal{CT} \) the set of all 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 finite trace \( \psi \) and a trace \( \psi' \), we denote their concatenation by \( \psi \cdot \psi' \). Finally, given a set of traces \( \Psi \), we denote by \( \psi \cdot \Psi \) the set \( \{ \psi \cdot \psi' | \psi' \in \Psi \} \).

**Example 1 (Conditional trace).** Given an environment \( \rho \), a trace formed by two conditional steps is \( \psi = (\overset{\lambda \sigma. \text{True} \triangleright \lambda \sigma. \sigma[\sigma(\rho(y))/\rho(x)])}{\lambda \sigma. \sigma[\sigma(\rho(y))/\rho(x)]}) \cdot \blacksquare \). The first step is a conditional transition whose predicate is always satisfied, and transformer binds variable \( x \) to the value of variable \( y \) in \( \sigma \). The second component of the trace is the finalizer \( \blacksquare \). Another example of conditional trace is \( \text{run}(\{\psi, \blacksquare\}) \).
Intuitively, a conditional transition $p \triangleright t$ represents all possible computations where the current process state $\sigma$ satisfies the condition $p$ and then the process progresses to state $t(\sigma)$. The finalizer ■ represents the end of the execution. In the following section we show how the *process spawn* is used to handle process calls.

**Definition 4 (Preorder on CT).** We order conditional traces by their information content. Namely, for all conditional steps $\phi_1$ and $\phi_2$, for all $\phi_1, \psi_1, \phi_2, \psi_2$, $\psi \in CT$ and for all $\Psi_1, \Psi_2 \subseteq CT$, we define $\varepsilon \leq \psi$, $\models \models$, and

\[
\phi_1 \cdot \psi_1 \leq \phi_2 \cdot \psi_2 \iff \psi_1 \leq \psi_2 \land \forall \sigma \in \text{State}, \phi_1^\sigma(\sigma) \Rightarrow \phi_2^\sigma(\sigma) \land \phi_1^\sigma(\sigma) = \phi_2^\sigma(\sigma)
\]

\[
\text{run}(\Psi_1) \leq \text{run}(\Psi_2) \iff \Psi_1 \subseteq \Psi_2
\]

where $\Psi_1 \subseteq \Psi_2 \iff \forall \psi_1 \in \Psi_1 \exists \psi_2 \in \Psi_2$, $\psi_1 \leq \psi_2$

**Example 2.** Given the conditional trace $\psi$ in Example 1 and the conditional trace $\psi' = (\lambda \sigma. \rho(x) > 0 \triangleright \lambda \sigma. \rho(\sigma(\rho(y))/\rho(x)))$. ■ we can observe that $\psi' \leq \psi$ since the predicate at the first position of $\psi'$ implies the one at the first position of $\psi$, and the remaining components are equal. This also means that the behaviors represented by $\psi'$ are also represented by $\psi$.

Note that $\leq$ is not anti-symmetric; for instance, $\{\phi \cdot \psi\} \notin \{\phi, \phi \cdot \psi\}$ and $\{\phi, \phi \cdot \psi\} \notin \{\phi, \psi\}$. Therefore, in order to obtain a partial order, we use equivalence classes w.r.t. the equivalence relation induced by $\leq$.

**Definition 5 (Semantic domain).** Given $M_1, M_2 \subseteq CT$, we define $M_1 \bowtie M_2$ as $M_1 \bowtie M_2 \equiv M_1 \bowtie M_2$. We denote by $C$ the class of non-empty sets of conditional traces modulo $\bowtie$. Formally, $C = \{[M]_{\bowtie} | M \subseteq CT\}$. We abuse notation and denote by $\leq$ also the partial order induced by the preorder $\bowtie$ on equivalence classes of $C$.

$(C, \subseteq, \cup, \cap, \tau, 1)$ is a complete lattice, where $\tau = [CT]_{\bowtie}$, $1 = \{[\varepsilon]\}_{\bowtie}$ and, for all $M \in C$, $\cup M = \bigcup_{[M], \bowtie M} [M]_{\bowtie}$ and $\cap M = \{[\psi \in CT | \forall [M]_{\bowtie} \bowtie M \exists \psi' \in M, \psi \leq \psi']\}_{\bowtie}$

It is worthy noting that we exclude $\emptyset$ from $C$ since the absence of progression is already represented by $\{\varepsilon\}$.

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

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

**Definition 6 (Interpretations).** Let $PlD$ be the set of process names and $\text{MGC}_{PlD} := \{p(\overrightarrow{T}) \mid p \in PlD, \overrightarrow{T}$ are distinct locations} (or simply $\text{MGC}$ when clear from the context).

Two functions $I, J: \text{MGC}_{PlD} \rightarrow C$ are variants, denoted by $I \equiv J$, if for each $\kappa \in \text{MGC}_{PlD}$ there exists a location remapping $\rho$ such that $(I(\kappa))\rho = (J(\kappa))\rho$.

An interpretation is a function $I: \text{MGC}_{PlD} \rightarrow C$ modulo variance.

\[\text{In other words, a family of elements of } C \text{ indexed by } \text{MGC} \text{ modulo variance.}\]
The semantic domain $\Pi_{Id}$ (or simply $\Pi$ when clear from the context) is the set of all interpretations ordered by the point-wise extension of $\preceq$ (which by abuse of notation we also denote by $\preceq$).

Essentially, the semantics of each process in $\Pi_{Id}$ is given over formal parameters that will be stored in a location that is actually irrelevant. It is important to note that $\text{MGC}_{\Pi_{Id}}$ has the same cardinality of $\Pi_{Id}$ (and is thus finite) and therefore each interpretation is a finite collection (of possibly infinite elements).

The partial order on $\Pi$ formalizes the evolution of the process computation. $(\Pi, \preceq)$ 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 $\cup$ and $\cap$, respectively. We abuse notation and use that of $\mathbb{C}$ for $\Pi$ as well.

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

4 The inductive bottom-up fixpoint semantics

In the literature, there exist some works that introduce formal definitions of an operational semantics for PROMELA. For instance, [14] proposes a clear structural operational semantics defined in an incremental style based upon [10]. However, such semantics models an old version of the language and more recent features such as the new behavior of atomic statement is missing. In our opinion, the most relevant proposal is [6], which provides a parametric structural operational semantics defined as the basis of the $\alpha$-SPIN tool for abstract model checking. In order to design our denotational semantics, we have took into consideration only the official specification of [8]. When any aspect was not defined there, we made our own assumptions and then we validated them with the semantics in [14,6] and with the latest version of SPIN.

As the reader might have noted, we have restricted ourselves to a significant fragment of PROMELA and made some assumptions that we recapitulate here for clarity. For simplicity, some constructs are not considered, for instance timeout or unless, and the ordered insertion and random removal of channel messages are not handled. Moreover, we treat the primitive expression run as a statement (and therefore it cannot be used into expressions). Statements that can be translated into the included ones (od/do or the for statement) are also omitted.

The semantics is defined as the least fixpoint of a semantics operator $D : \Pi \rightarrow \Pi$ that transforms interpretations. Informally, such fixpoint is computed as the limit $\bigcup \{ D^k(\lambda \kappa. \bot) \mid k \in \mathbb{N} \}$ where $\kappa \in \text{MGC}_{\Pi_{Id}}$. Function $D$ is defined in terms of auxiliary semantics evaluation functions described in the following sections.
4.1 Denotation of Expressions

Due to space limitations, we assume to have two functions to evaluate expressions. The first one is used for expressions on the right hand side of an assignment and it returns the value of a given r-expression in a given state $R : r\text{-expressions} \times Env \times State \to V$. The second function, for l-expressions, returns the modifier for the location associated to the expression. Given an environment $\rho$, a state $\sigma$ and then a value-updating function $f : V \to V$, the function $\llbracket l\text{-expressions} \times Env \times State \to (V \to V) \to State \rrbracket$ builds a function to update the location identified by the l-expression with modifier $f$.

**Example 3 (Evaluation of expressions).** Given $\rho := \{x \to l_2\}$ and $\sigma := \{(l_x \to 4),\{\}\}$, then $R[x + 1 > 3]_\sigma = R[x + 1]_\sigma > R[3]_\sigma = R[x]_\sigma + R[1]_\sigma > 3 = \sigma(\rho(x)) + 1 > 3 = 1$ (which corresponds to $True$).

4.2 Denotation of Basic Statements of PROMELA

We start by showing the denotation for basic statements. Given a basic statement $s$, an environment $\rho$ and an interpretation $\mathcal{I}$, the function $\mathcal{S} : stmt \times Env \times \mathbb{R} \to CT$, written $\mathcal{S}[s]^{\rho}_{\mathcal{I}}$, builds the conditional trace corresponding to the effects of $s$. We show below the rules corresponding to the case when the constructs are not within the scope of an atomic block. We use the variants $\triangleright$ and $run$ replacing constructs $\triangleright$ and $run$ in the given rules when the basic statement is within the scope of an atomic block. The information derived from the use of these variants will be used during the second phase of the semantics computation (when synchronization among processes is dealt with). More specifically, following the operational behavior, the first statement in an atomic block behaves as a guard to enter the atomic block itself; if such guard is satisfied all statements of the block will be run as an indivisible sequence as far as they do not suspend. There are some scenarios in which the statements inside an atomic block may suspend however. In particular, when a synchronous send statement occurs in an atomic block, the control is lost, thus the behavior is equivalent to having an independent atomic block starting at the statement right after the send statement. Hence, as has been advanced, to later reproduce this behavior, when a statement $s$ is inside an atomic block (except if it is the first one of the block or it is the first one after a synchronous send statement) the used operator is the one labeled with the arrow.

$$
\mathcal{S}[\text{goto } L]^{\rho}_{\mathcal{I}} := \varepsilon
$$

$$
\mathcal{S}[\text{Expr}]^{\rho}_{\mathcal{I}} := \lambda \sigma. R[\text{Expr}]^{\rho}_{\mathcal{I}} \triangleright id
$$

where $id$ is the identity function.

$$
\mathcal{S}[\text{if } t \text{ then } e \text{ else } e']^{\rho}_{\mathcal{I}} := \lambda \sigma. \text{True} \triangleright \lambda \sigma. \mathcal{L}[e]^{\rho}_{\mathcal{I}}(\lambda \nu. R[e']^{\rho}_{\mathcal{I}})
$$

$$
\mathcal{S}[\text{type } x = e]^{\rho}_{\mathcal{I}} := \lambda \sigma. \text{True} \triangleright \lambda \sigma. \sigma[\mathcal{R}[e]^{\rho}_{\mathcal{I}}/\rho(x)]
$$

For asynchronous channels $c$ (i.e., whose capacity is greater than 0)

$$
\mathcal{S}[\text{c! } c']^{\rho}_{\mathcal{I}} := \lambda \sigma. (\sigma(\rho(c)) \neq 0 \land \text{full}(\sigma(\rho(c)))) \triangleright

\lambda \sigma. \sigma[\text{push}(\sigma(\rho(c)), \mathcal{R}[c]^{\rho}_{\mathcal{I}})/\rho(c)]
$$

For synchronous channels $c$ (i.e., whose capacity is equal to 0)

$$
\mathcal{S}[\text{c? } c']^{\rho}_{\mathcal{I}} := \lambda \sigma. (\sigma(\rho(c)) = 0 \land \text{empty}(\sigma(\rho(c)))) \triangleright

\lambda \sigma. \sigma[\text{poll}(\sigma(\rho(c)), \mathcal{R}[c]^{\rho}_{\mathcal{I}})/\rho(c)]
$$

For synchronous channels $c$ (i.e., whose capacity is equal to 0)

$$
\mathcal{S}[\text{c? } c']^{\rho}_{\mathcal{I}} := \lambda \sigma. (\sigma(\rho(c)) = 0 \land \text{empty}(\sigma(\rho(c)))) \triangleright

\lambda \sigma. \sigma[\text{poll}(\sigma(\rho(c)), \mathcal{R}[c]^{\rho}_{\mathcal{I}})/\rho(c)]
$$
Section 4.4, make use of these guards to guarantee that both processes can only communicate via the synchronous channel. At system level, functions wantsynch and synch, given in Equation (4.1f) model asynchronous communication. The conditional transition is executable only if the channel is currently defined and not empty (for receiving). The channel valuation is updated as expected. Functions push, pop and head are part of the semantic domain for system states.

Equations (4.1g) and (4.1h) model synchronous communication. We treat synchronous channels as channels with size one and synchronization is achieved by using the predefined variable handshake. The behavior for synchronous communication is similar to the asynchronous one, the particular point is that the send statement updates the handshake variable with the channel identifier so that the receive statement can check if the channel it is listening to coincides with that in handshake. Upon completion, handshake is set to its default value (−1). Observe that guards in Equations (4.1g) and (4.1h) establish the mandatory conditions needed for the sender/receiver processes to communicate via the synchronous channel. At system level, functions wantsynch and synch, given in Section 4.4, make use of these guards to guarantee that both processes can only progress synchronously.

Finally, Equation (4.1i) models a process call of the run statement for the interpretation $I$. All conditional traces built by this rule start with a conditional transition always enabled that allocates the needed space for the local variables of the new process; 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” are those settled in the first transition. The interleaving behavior is handled in a second phase of the semantics computation.

Example 4 (Semantics for assignment statement). Given an environment $\rho$ and an interpretation $I$, the semantics for an assignment $x = y$ is

$$S[x = y]_I^\rho = \lambda \sigma. True \triangleright \lambda \sigma. L[x]_\rho \sigma (\lambda v. R[y]_\rho \sigma) = \lambda \sigma. True \triangleright \lambda \sigma. \sigma(\rho(y))/\rho(x)$$

This expression can be read as a conditional transition that is always enabled and, as expected, transforms the state $\sigma$ by copying the value (stored in the location) of variable $y$ (i.e., $\sigma(\rho(y))$) to (the location of) variable $x$ (i.e., $\rho(x)$).

4.3 Denotation of process definitions

Since we have arbitrary goto instructions in the programs, the flow of control does not follow predefined patterns like when we have only structured constructs. To define a semantics by “straight” (one-to-one) induction on syntax one has to resort to the use of some sort of higher-order constructions (like continuations). However, the development of abstract semantics based on this type of concrete semantics certainly becomes more complicated.

In order to build “first-order” denotations we choose to define the semantics of process definitions $\text{proctype } p(\overline{x})\{\text{body}\}$ by induction on the structure of the Control Flow Graph (CFG) $\Pi_p$ of its body. Such CFG $\Pi_p$ can be easily constructed from the process syntax, likewise the labeled transition systems of the operational engine. More specifically, each node of a CFG is what we call a process point, i.e., a point that is referred by the process counters of the operational engine as well as a goto statement. We follow the typical convention to name process points as the statement labels (when they are present in the program). Furthermore, we interchange each label with the corresponding process point. In the sequel, we denote by $\mathcal{L}_p$ the set of process points of process $p$ (the nodes of $\Pi_p$).

Edges of the CFG are labeled by the basic statement that generates such edge. The construction of the CFG is standard, except for the case of statement $i\text{ }/f\text{i}$. For a generic $L:\text{if}:: \text{grd}_i;L_i::\text{stmts}_i$\text{elseif}::$\text{opt}_i;\text{stmts}_i\text{else}::L'$: $\cdots$, for all branches in $\Lambda$ we have an edge labeled with $\text{grd}_i$ from process point $L$ to process point $L_i$ and then we have the path (starting at point $L_i$) corresponding to (the translation of) the statements $\text{stmts}_i$ (if the last statement of $\text{stmts}_i$ is reached, the path ends in point $L'$). When the else branch is present, we add also a path where the first arc is a special one $l \Rightarrow \bar{l}$ and then we proceed with (the translation of) the statements $\text{stmts}_e$. In the sequel we denote by $\mathcal{E}_p$ the set of edges $l \xrightarrow{\text{opt}_i} l'$ or $l \xrightarrow{\text{else}} l'$ of $\Pi_p$.

\footnote{Note that due to the operational semantics of goto statements, nodes in the CFG might not correspond to states of the PROMELA process since, in the language, process counters never point to goto statements. However, because of the same, the steps in traces generated by the semantics do correspond to system states.}
Example 5. The CFG of the process $p$ defined in Figure 1 is the following.

The denotation of a process definition is built by using the following immediate consequences operator $\mathcal{P} : \text{procs} \times \text{Env} \times \mathbb{I}_C \times [L_p \rightarrow C] \rightarrow [L_p \rightarrow C]$. This operator uses a family of sets of conditional traces indexed by process points (coded with a function $\eta \in [L_p \rightarrow C]$) that convey the behavior from each process point “to the end” and then builds (backward) the behaviour of all statements that after one execution step proceed like specified by $\eta$.

\[
\mathcal{P}[\Pi_p]^{\rho}_{L,\eta} := \lambda l. \begin{cases} 
\text{addElse} \left( \bigcup_{t' \in \mathcal{E}_p} \mathcal{S}[s]^{\rho}_{L} \cdot \eta(t'), \bigcup_{t' \in \mathcal{E}_p} \mathcal{S}[s]^{\rho}_{L} \cdot \eta(t') \right) & \text{if } l \xrightarrow{\sigma} t' \in \mathcal{E}_p \\
\text{otherwise} & 
\end{cases}
\]

where $\text{addElse}(E, T) := T \cup (\wedge_{\psi \in \mathcal{E}} \neg \phi^C) \triangleright id \cdot E$. The addElse semantic operator is used for $\textbf{else}$ branches and it adds the negation of each guard of the other branches in order to make the $\textbf{else}$ branch executable only when all other branches are not. Note that conjunction $\wedge_{\psi \in \mathcal{E}} \neg \phi^C$ is finite since we have finitely many different first conditional states $\phi$ in any $T$.

Given a process definition $\Pi_p$, an environment $\rho$ and an interpretation $I$, the function $\mathcal{D} : \text{procs} \times \text{Env} \times \mathbb{I}_C \rightarrow [L_p \rightarrow C]$, written $\mathcal{D}[\Pi_p]^{\rho}_{L}$, builds the conditional traces corresponding to the executions in isolation of $p$.

\[
\mathcal{D}[\Pi_p]^{\rho}_{L} := \lambda l. \text{fixAtoms}(\mathcal{F}(l)) \quad \text{where} \quad \mathcal{F} := \text{lfp}(\lambda \eta. \mathcal{P}[\Pi_p]^{\rho}_{L,\eta})
\]

The semantics operator fixAtoms replaces the marked conditional steps with all possible executions that correspond to the prescribed semantics of the atomic regions. It is defined as $\text{fixAtoms}(\psi) := \bigcup_{\psi \psi} \text{fixAt}(\psi)$ where

- $\text{fixAt}(\psi) := \{\psi\}$ if $\psi$ does not contain marked conditional steps.
- Otherwise, without loss of generality, let $\psi = \psi' \cdot \phi_1 \cdot \rho_1 \cdot \overset{\phi_2}{\rho_2} \cdot \psi''$ where $\phi_1, \phi_2$ are conditional transitions; $\rho_1, \overset{\phi_2}{\rho_2}$ are a possibly empty sequences of process spawns; $\psi'$, $\phi_1 \cdot \rho_1$ has no marks and $\overset{\phi_2}{\rho_2}$ are marked. Then

\[
\text{fixAt}(\psi) := \text{fixAtoms}(\psi' \cdot (\lambda \sigma. \phi_1^C(\sigma) \land \phi_2^C(\phi_1^T(\sigma))) \triangleright \lambda \sigma. \phi_2^C(\phi_1^T(\sigma)) \cdot \rho_1 \cdot \rho_2 \cdot \psi'') \cup \psi' \cdot (\lambda \sigma. \phi_1^C(\sigma) \land \neg \phi_2^C(\phi_1^T(\sigma))) \triangleright \phi_1^T \cdot \rho_1 \cdot \text{fixAtoms}(\phi_2 \cdot \rho_2 \cdot \psi'')
\]

This operator generates traces for the different scenarios in an atomic region. It composes the conditions and effects of two atomic steps $\phi_1$ and $\phi_2$ in a single step. But the alternative trace when $\phi_2$ suspends is also generated $(\phi_1^C(\sigma) \land \neg \phi_2^C(\phi_1^T(\sigma)))$. This strategy is applied recursively.
**Example 6 (Process denotations).** We start by showing the output of the fixpoint computation for the \textit{init} process in Figure 1. \text{lfp}(\lambda \eta. \mathcal{P}[\Pi_{\text{init}}]_{\llbracket \eta \rrbracket}^{\mathcal{P}}) (L.5) is

\[ (\lambda \sigma. \text{True} \triangleright \lambda \sigma. \sigma([\nu_{1}/l_{1}] [1/l_{1}] \sigma(\rho(\text{nr.pr})) + 1/\rho(\text{nr.pr})) \cdot \text{Fin}(\mathcal{I}(\mathcal{P}(0, . 1))) \cdot \\
(\lambda \sigma. \text{True} \triangleright \lambda \sigma. \sigma([\nu_{1}/l_{2}] [2/l_{2}] \sigma(\rho(\text{nr.pr})) + 1/\rho(\text{nr.pr})) \cdot \text{Fin}(\mathcal{I}(\mathcal{P}(1, . 2))) \cdot \quad \blacksquare \]

The atomic block ensures that the two processes are created before any of them starts executing its body. Since it is not possible that those two \text{run} statements suspend, the \text{fixAtomics} function compacts the two steps and prevents the interleaving with the two newly created processes. In fact \( \mathcal{D}[\Pi_{\text{init}}]_{\llbracket \eta \rrbracket}^{\mathcal{P}}(L.5) \) is

\[ (\lambda \sigma. \text{True} \triangleright \lambda \sigma. \sigma([\nu_{1}/l_{1}] [1/l_{1}] [\nu_{2}/l_{2}] [2/l_{2}] \sigma(\rho(\text{nr.pr})) + 2/\rho(\text{nr.pr})) \cdot \\
\text{run}(\mathcal{I}(\mathcal{P}(0, . 1))) \cdot \text{run}(\mathcal{I}(\mathcal{P}(1, . 2))) \cdot \quad \blacksquare \]  

(4.2)

The fixpoint computation for the \( \mathcal{P} \) process type results in a single infinite trace in which the four conditional steps actually shown below are infinitely repeated:

\[ \text{lfp}(\lambda \eta. \mathcal{P}[\Pi_{\text{P}}]_{\llbracket \eta \rrbracket}^{\mathcal{P}}(L.0) = \{ (\lambda \sigma. \text{True} \triangleright \lambda \sigma. \sigma[1/\sigma(\rho(\text{f[id]}))] \cdot \lambda \sigma. \sigma(\rho(\text{f[id]})) \triangleright \text{id} \\
(\lambda \sigma. \text{True} \triangleright \text{id} \cdot \lambda \sigma. \text{True} \triangleright \lambda \sigma. \sigma[0/\sigma(\rho(\text{f[id]}))] \cdot \ldots \} \]

The second step in the loop is marked as atomic whereas the third one corresponds to the execution of the \text{skip} statement representing the critical section. The atomic region in this process prevents any interleaving between the first and second step if the condition of the second step is satisfied. However, if that condition does not hold, then interleaving is possible. This is why the \text{fixAtomics} function generates an infinite set of infinite traces. The following set only shows the two possible beginnings for traces in \( \mathcal{D}[\Pi_{\text{P}}]_{\llbracket \eta \rrbracket}^{\mathcal{P}}(L.0) \). Each trace will consist of an arbitrary combination of these fragments.

\[ \{ (\lambda \sigma. \sigma(\rho(\text{f[id]})) \triangleright \lambda \sigma. \sigma[1/\sigma(\rho(\text{f[id]}))] \cdot \ldots \} \]

\( \lambda \sigma. \text{True} \triangleright \text{id} \cdot \lambda \sigma. \text{True} \triangleright \lambda \sigma. \sigma[0/\sigma(\rho(\text{f[id]}))] \cdot \ldots \),

\[ (\lambda \sigma. \sigma(\rho(\text{f[id]})) \triangleright \lambda \sigma. \sigma[1/\sigma(\rho(\text{f[id]}))] \cdot \lambda \sigma. \sigma(\rho(\text{f[id]})) \triangleright \text{id} \cdot \\
(\lambda \sigma. \text{True} \triangleright \text{id} \cdot \lambda \sigma. \text{True} \triangleright \lambda \sigma. \sigma[0/\sigma(\rho(\text{f[id]}))] \cdot \ldots \ldots \ldots \} \]

**4.4 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 \text{run} statement in the init process, encapsulated 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 and remove \text{printf} statements.

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
applies the effect of one step of the computation to the input interpretation.

\[
\mathcal{D}[[\text{proctype } p_1(x_1)](L_1 : \text{body}_1), \ldots ; \text{proctype } p_n(x_n)](L_n : \text{body}_n)]_\mathcal{P}^n := \begin{cases} 
\{ p_1(t_1, l_1^1, l_1^2) \mapsto \mathcal{D}[[\text{body}_1]]_\mathcal{P}^n (L_1) \cdot \text{End} \\
\vdots \\
\{ p_n(t_n, l_n^1, l_n^2) \mapsto \mathcal{D}[[\text{body}_n]]_\mathcal{P}^n (L_n) \cdot \text{End} 
\end{cases}
\]

where \( \rho_i := \rho_0[t_i / x_i][l_i^1 / v_i^1][l_i^2 / \text{pid}], \quad \overrightarrow{v_i} \) are the local variables of \( \text{body}_1 \),

\[ \text{End} := (\lambda \sigma. \text{True} \triangleright \lambda \sigma. \sigma[\sigma(\rho_{\text{nr \_pr}}) - 1/\rho_{\text{nr \_pr}}]) \quad \blacksquare \]

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 \( P : \text{Prog} \times \text{State} \to T \) where \( T \) is the domain of sets of sequences of states in State. Note that the definition uses two auxiliary functions \( \text{prpt} \) and \( \text{unroll} \) that are formalized later. Given a normalized program \( P \in \text{Prog} \) with init process \( \text{init} \), global variable declarations \( gv \) and a list of process declarations \( pd \), we define

\[
P[gv; pd]_\sigma := \text{prpt}(\sigma, \text{unroll}\left(\bigcup_{k \in \mathbb{N}} (\lambda I. \mathcal{D}[[pd]]_I^n)^k (\lambda T. \bot)\right)(\text{init}(l_0)))
\]  \hspace{1cm} (4.5)

where \( l_0 \) is the location reserved for the \( \text{pid} \) variable of \( \text{init} \) and \( \rho_0 \) is the environment populated with associations, of both the language predefined variables and the global variables declared in \( gv \), to suitable (different fresh) locations.

\( P \) first computes the least fixpoint \( F \) of the function \( \mathcal{D} \); the result is used to compute the interleaving of spawned processes, and finally the initial state \( \sigma \) is propagated to the conditional traces of \( F(\text{init}) \). \( P \) is well defined since \( \mathcal{D} \) is continuous.

Given a set of conditional traces, \( \text{unroll} : \mathcal{C} \to \mathcal{C} \) replaces the process spawns \( \text{run}(\Psi) \) of each conditional trace with all possible interleavings of \( \Psi \) with the rest of the trace. To define \( \text{unroll}(\Psi) \), first observe that if \( \Psi \) is not \( \bot \) then it can be partitioned depending on the initial conditional step of all its conditional traces \( \phi_1, \ldots, \phi_n \). Then

\[
\text{unroll}(\phi_1 \cdot \Psi_1 \cup \ldots \cup \phi_n \cdot \Psi_n) = \bigcup_{i \leq i \leq n} \text{unroll'}(\phi_i, \Psi_i \cup \bigcup_{j \neq i} \phi_j \cdot \Psi_j) \cup
\]

\[
\bigcup \left\{ \text{synch}(\phi_j, \phi_k) \cdot \Psi \cup \text{synch}(\phi_k, \phi_j) \cdot \Psi \mid \text{wantsynch}(\phi_j, \phi_k), \Psi = \text{unroll}(\Psi_j \cup \Psi_k \cup \bigcup_{i \neq j, i \neq k} \phi_i \cdot \Psi_i) \right\}
\]

where \( \text{unroll'}(\phi, \Psi) = \Psi \), \( \text{unroll'}(c \triangleright t, \Psi) = c \triangleright t \cdot \text{unroll}(\Psi) \), \( \text{unroll'}(\text{run}(\Psi'), \Psi) = \text{unroll}(\Psi \cup \Psi') \)

The second part of the definition uses predicate \( \text{wantsynch}(\phi_x, \phi_v) \) to detect if conditional state \( \phi_x \) wants to perform a synchronous send over a channel, and \( \phi_v \) wants to perform a synchronous receive over that channel. Moreover \( \text{synch}(\phi, \phi') \) is the conditional step that passes the message content and combines
the effects on the state of $\phi$ and then $\phi'$. Both actual combinations $\text{synch}(\phi_s, \phi_r)$ and $\text{synch}(\phi_r, \phi_s)$ are possible.

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

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

$$\text{prpgt}(\sigma, \phi_1 \cdot \Psi_1 \cup \cdots \cup \phi_k \cdot \Psi_k) = \bigcup \left\{ \phi^T_i(\sigma) \cdot \text{prpgt}(\phi^T_i(\sigma), \Psi_i) \mid \phi^C_i(\sigma) = \text{True} \right\}$$

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. Note that the second equation characterizes traces whose conditions become true after propagation. Since the unroll function generates non-real traces, we cannot distinguish blocking behaviors from wrong behaviors and we discard any of them.

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

**Example 7 (Computation of the program semantics).** Let $P$ the program in Figure 1 and $\text{body}_{\text{init}}$ the body of the init process. Let $D$ be the process declarations, i.e., $D = \text{init}\{\text{body}_{\text{init}}\}; \text{proctype} P(\text{bit id})\{\text{body}_P\}$ and $\Pi_{\text{init}}, \Pi_p$ their associated CFGs. Then, the semantics of the whole program is

$$P[\sigma_0] = \text{prpgt}(\sigma_0, \text{unroll}\left(\bigcup_{k \in \mathbb{N}} (\lambda \Delta \cdot \mathbb{D}[\mathbb{K}_D^\sigma]^k(\lambda \sigma \cdot \bot))(\text{init}(l_0))\right))$$

where the global environment $\rho_G$ binds $\xi [0]$ to $l_0$, $\xi [1]$ to $l_1$, $i$ to $id$, and $\lambda$ to $l_c$. Moreover $l_0$ is the location reserved for the $\lambda$ variable of $\text{init}$. Let $\rho_P := \rho_G[l_p/\lambda id][l_id/id]$ and $\rho_i := \rho_G[l_p/\lambda id]$. By Equation (1.3),

$$\mathbb{D}[\mathbb{K}_D^\sigma] = \begin{cases} p(id, l_0) & \Rightarrow D[\Pi_p]_P^\sigma(\text{init}(l_0)) \Rightarrow D[\Pi_{\text{init}}]_P^\sigma \Rightarrow \text{End} \\ l_1 & \Rightarrow D[\Pi_{\text{init}}]_P^\sigma \Rightarrow \text{End} \end{cases}$$

We now illustrate how the unrolling process of the trace generated by the init process works. For conciseness, let $\phi_\sigma$ be the starting conditional step $\lambda \sigma.\text{True} \triangleright \lambda \sigma.\sigma [v_1/l_1][v_2/l_2][\sigma(\rho(\lambda \sigma.pr)) + 2/\rho(\lambda \sigma.pr)]$. Then,

$$\text{unroll}\left(\{\phi_\sigma \cdot \text{run}(I(P(0, \bot 1)) \cdot \text{run}(I(P(1, \bot 2))) \cdot \text{End})\}\right) =$$

$$\{\phi_\sigma \cdot \text{unroll}\left(\{\text{run}(I(P(0, \bot 1)) \cdot \text{run}(I(P(1, \bot 2))) \cdot \text{End})\}\right) =$$

$$\{\phi_\sigma \cdot \text{unroll}\left(\{\text{run}(I(P(1, \bot 2))) \cdot \text{End}) \cup D_{p3}\right)\} = \{\phi_\sigma \cdot \{\text{unroll}\left(\{\text{End}) \cup D_{p1} \cup D_{p2}\right)\} \cup \Psi\}$$

where $D_{p1}$ and $D_{p2}$ are the sets of traces for $P(0, \bot 1)$ and $P(1, \bot 2)$ resulting from Equation (1.3), and $\Psi$ represents the rest of the unrolling of the traces in
$D_{p1}/D_{p2}$. Below we show the set $D_{p1}$. $D_{p2}$ is similar to $D_{p1}$, substituting every $f[0]$ by $f[1]$ and every $f[1]$ by $f[0]$.

$$D_{p1} = \{ \lambda \sigma.\sigma(\rho(lf[1])) \triangleright \lambda \sigma.\sigma[1/\sigma(\rho(lf[0]))] \triangleright \lambda \sigma.\lambda \sigma.\sigma(0/\sigma(\rho(f[0]))) \}$, 
$$\lambda \sigma.\sigma(\rho(lf[1])) \triangleright \lambda \sigma.\sigma[1/\sigma(\rho(lf[0]))] \triangleright \lambda \sigma.\sigma(0/\sigma(\rho(f[0]))) \triangleright \lambda \sigma.\sigma(\rho(f[1])) \triangleright \lambda \sigma.\sigma(\rho(f[1])) \triangleright \lambda \sigma.\sigma(0/\sigma(\rho(f[0]))) \} \} \} \} \} \} \} \}$

The unroll function takes each trace in these sets and computes all possible interleavings between each pair of traces. We show the beginning of two computed traces, one in which the first process executes once the loop followed by the execution of the loop by the second process, and another in which after the first two steps of the first process, the second one executes its first condition and then assumes the second step is not executable.

$$\phi_{l_{0}} \cdot (4.6) \cdot \{ \lambda \sigma.\sigma(\rho(lf[1])) \triangleright \lambda \sigma.\sigma[1/\sigma(\rho(f[0]))] \triangleright \lambda \sigma.\sigma(0/\sigma(\rho(lf[0]))) \}$, 
$$\lambda \sigma.\sigma(\rho(lf[1])) \triangleright \lambda \sigma.\sigma[1/\sigma(\rho(f[0]))] \triangleright \lambda \sigma.\sigma(0/\sigma(\rho(lf[0]))) \triangleright \lambda \sigma.\sigma(\rho(lf[1])) \triangleright \lambda \sigma.\sigma(\rho(lf[1])) \triangleright \lambda \sigma.\sigma(0/\sigma(\rho(f[0]))) \}$ 

After the interleaving process, the propagate operator accumulates the effects of the executed steps. For conciseness, let $\alpha(a,b)$ be the state $\{ l_{0} \triangleright l_{0}, l_{1} \triangleright b \}$, $\{ f[0] \triangleright l_{0}, f[1] \triangleright l_{1} \}$. The semantics for our program example includes all traces in which each process runs the complete loop before passing the control (or not) to the other process:

$$\text{prpgt} \left( \alpha(0,0), \text{unroll} \left( \text{init}(l_{0}) \right) \right) = \{ \alpha(0,0) \cdot \alpha(1,0) \cdot \alpha(1,0) \cdot \alpha(0,0) \cdot \alpha(0,1) \cdot \alpha(0,1) \cdot \alpha(0,0) \cdot \alpha(0,0) \cdot \alpha(0,0) \cdot \alpha(0,0) \cdot \alpha(0,0) \}$

Note that when propagating the second trace in Equation (4.6), we have that we reach a point in which the condition of the following step is false, thus the behavior trace is discarded. This particular trace, included in the set of traces generated by the unroll function, corresponds with a deadlocking trace in PROMELA, namely, $\sigma(0,0) \cdot \sigma(1,0) \cdot \sigma(1,0) \cdot \sigma(0,1) \cdot \sigma(1,1) \cdot \sigma(1,1)$. The (propagated) state $\sigma(1,1)$ falsifies the conditional trace $\lambda \sigma.\sigma(\rho(lf[0])) \triangleright \lambda \sigma.\sigma(\rho(lf[0])) \}$

5 Conclusion and future work

This work presents a first proposal of an inductive bottom-up fixpoint semantics for PROMELA, which is a widely used modeling language for concurrent, reactive systems. It does not cover the full language, but the considered fragment is very significant. In particular, it handles arbitrary goto, which are extensively used by the PROMELA community and both synchronous and asynchronous.
communications. 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 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.

As noted, the denotational semantics in this work does not capture deadlock- ing traces. As a future work we plan to overcome this limitation by defining a new unroll semantic operator that propagates the conditions of previous conditional states within the conditions of successive ones, in order to discard impossible sequentializations and retain only the finite traces that stop either on successful termination or deadlock.

References

1. M. Ben-Ari (2008): Principles of the Spin Model Checker, 1 edition. Springer-Verlag, London. [https://doi.org/10.1007/978-1-84628-770-1]
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.
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.
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. [https://doi.org/10.1017/S1471068414000349]
5. H. Erdogmus (1996): Verifying Semantic Relations in SPIN. In: Proceedings of the First SPIN Workshop.
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. [https://doi.org/10.1007/s00165-004-0040-y]
7. C. A. R. Hoare (1985): Communicating Sequential Processes. Prentice-Hall.
8. G. J. Holzmann (2004): The Spin Model Checker: Primer and Reference Manual. Addison-Wesley. Available at [https://books.google.it/books?id=NpBdZKmOHO8C]
9. G. J. Holzmann (2012): Spin Manual. Available at [http://spinroot.com/spin/Man/promela.html]
10. 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.
11. G. L. Peterson (1981): Myths about the mutual exclusion problem. Information Processing Letters 12(3), pp. 115–116. [https://doi.org/10.1016/0020-0190(81)90106-X]
12. 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.
13. 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

14. C. Weise (1997): An incremental formal semantics for PROMELA. In: Proceedings of the Third SPIN Workshop, SPIN97.