# A Formal Executable Semantics of PROMELA

Byoungho Son and Kyungmin Bae

Pohang University of Science and Technology, Pohang, South Korea

**Abstract.** PROMELA, the modeling language of SPIN, is widely used to specify and model check finite-state concurrent systems but lacks support for deductive verification. This paper presents a faithful, executable semantics of PROMELA in the $\mathbb{K}$ framework that enables code-level deductive verification. To address the nontrivial interactions between guarded nondeterminism and concurrency, we introduce LOAD-AND-FIRE, an elegant semantic pattern that yields a modular, uniform treatment of guarded nondeterminism, cross-process interference, and atomicity in $\mathbb{K}$. Our semantics enables the full suite of analyses provided by $\mathbb{K}$, including deductive verification of PROMELA programs with infinite state spaces, a capability previously unavailable for PROMELA models. We illustrate the approach with a case study in deductive verification of an infinite-state concurrent system.

## 1 Introduction

PROMELA (PROcess MEta LAnguage) [1], the modeling language of the SPIN model checker, has been widely used for decades to specify concurrent systems. It combines imperative constructs—such as assignments, conditionals, and loops—with first-class nondeterministic choice and message passing via buffered and rendezvous channels. PROMELA is broadly adopted across academia and industry for teaching, prototyping, and validating concurrency designs in domains including protocols, operating systems, and multithreaded software [16,5,10].

Given PROMELA's widespread use, many verification tasks increasingly demand *deductive* reasoning: parametric invariants (e.g., "for all buffer capacities $N$"), proofs for infinite-state systems, compositional arguments, and reusable proof artifacts. SPIN's explicit-state LTL model checking [13] is highly effective for analyzing finite-state concurrent systems, yet it does not directly address these needs. What is missing is a mechanized, executable semantics of PROMELA that provides both a precise, runnable reference for the language and a basis for code-level deductive verification.

Prior semantic definitions of PROMELA, such as structural operational semantics [38,17], labeled transition systems [2,24], and denotational semantics [18], provide insight but typically lack mechanized formalizations that support deductive verification of PROMELA programs. SPIN remains the reference implementation, yet it is informally specified [23]. To our knowledge, no prior line of work offers, in a single framework, (i) an executable interpreter for PROMELA; (ii) a proof system that reasons about PROMELA programs; and (iii) modular extensibility that scales to the language's difficult features.

In this paper, we present a faithful and executable semantics of PROMELA in $\mathbb{K}$ that also supports deductive verification. $\mathbb{K}$ [30] is a rewriting-based semantic framework that enables modular, machine-readable definitions of programming languages and automatically derives tools such as interpreters, model checkers, and deductive verifiers from the semantics. Concretely, our semantics (i) runs PROMELA models as an interpreter, (ii) provides a proof engine (`kprove`) to establish safety properties for both finite-state and infinite-state models, and (iii) constitutes a precise, machine-readable reference for the language.

Defining a faithful, executable semantics for PROMELA is challenging because the language exhibits nontrivial interactions between guarded nondeterminism and concurrency. For example, PROMELA permits nondeterministic choice whose options are guarded by arbitrary statements rather than Boolean conditions, and those statements may themselves nest additional choices. A satisfactory semantics must surface enabledness at the frontier before committing to a step and integrate blocking, nondeterminism, and atomicity in a uniform way. Formalizing these interactions in an elegant, executable form is difficult; consequently, prior work on PROMELA often targets restricted subsets that avoid these complications.

To address these challenges, we propose the LOAD-AND-FIRE semantic pattern. *Load* normalizes nested guarded constructs (including statement-guarded options) into a canonical multiset of enabled frontiers and is purely structural and produces no effects. *Fire* then commits atomically to one enabled frontier. A simple lock mechanism enforces atomic blocks, and a lock-tossing rule preserves atomicity across rendezvous (e.g., a send inside an atomic block transfers the lock to the receiver). This pattern yields modular rules and a uniform treatment of nested guarded nondeterminism, cross-process interference, and atomicity.

Our semantics enables the full suite of analyses provided by the $\mathbb{K}$ framework [15], including deductive verification of PROMELA programs with infinite state spaces, a capability previously unavailable for PROMELA models. It is worth noting that the executability of our semantics enables *validation* by differential execution against the reference implementation, SPIN [13]. We use benchmark examples–including those from the SPIN repository–as a conformance suite, also covering corner cases that exercise the nontrivial features described above.

The contributions of this paper are summarized as follows. (1) We present the first mechanized, executable semantics of PROMELA in $\mathbb{K}$ that enables code-level deductive verification. (2) We introduce the LOAD-AND-FIRE semantic pattern for guarded nondeterminism with arbitrary statement guards, applicable to languages with similar features (e.g., Go's `select`). We demonstrate the approach with a case study in deductive verification of an infinite-state concurrent system.

The rest of the paper is organized as follows. Section 2 provides background on the $\mathbb{K}$ framework and PROMELA. For readability, we present the semantics incrementally: Section 3 considers a core subset without nested guards, atomic blocks, or impure expressions; Section 4 extends this subset with nested guards; and Section 5 adds atomic blocks and impure expressions. Section 6 presents a case study in deductive verification. Section 7 discusses related work. Finally, Section 8 presents some concluding remarks.

# 2 Preliminaries

## 2.1 The $\mathbb{K}$ Semantics Framework

$\mathbb{K}$ [30] is a semantic framework for programming languages, based on rewriting logic [19]. It has been widely used to formalize a variety of languages, including C [9], Java [3], JavaScript [25], and so on. $\mathbb{K}$ provides several analysis tools, such as deductive verifiers and symbolic execution engines.

**Semantics Definition in $\mathbb{K}$.** In $\mathbb{K}$, program states are represented as multisets of nested cells, called *configurations*. Each cell represents a component of a program state, such as computations and stores. Transitions between configurations are specified as (labeled) $\mathbb{K}$ rules, specifying only the relevant parts of these cells.

A computation in $\mathbb{K}$ (called the $\mathbb{K}$ continuation) is defined as a $\curvearrowright$-separated sequence (of sort $K$) of computational tasks (of sort *KItem*). For example, $t_1 \curvearrowright t_2 \curvearrowright \ldots \curvearrowright t_n$ represents the computation consisting of $t_1$ followed by $t_2$, and so on. A task can be decomposed into simpler tasks, and the result of a task is forwarded to the subsequent tasks. E.g., $(5+x)*2$ is decomposed into $x \curvearrowright 5+\square \curvearrowright \square * 2$, where $\square$ is a placeholder for the result of a previous task. If $x$ evaluates to some value, say 4, then $4 \curvearrowright 5+\square \curvearrowright \square * 2$ becomes $5+4 \curvearrowright \square * 2$, which eventually becomes 18. We denote the empty task as "$.K$".

The following shows a typical example of $\mathbb{K}$ rules for variable lookup, where `lookup` is a label, the $k$ cell contains a computation, *env* contains a map from variables to locations, and *store* contains a map from locations to values:

$$\texttt{lookup:}\ \langle \frac{x}{v} \curvearrowright \ldots\rangle_k\ \langle \ldots x \mapsto l \ldots\rangle_{env}\ \langle \ldots l \mapsto v \ldots\rangle_{store}$$

A horizontal line represents a state change, and "..." indicates irrelevant parts. A cell without horizontal lines is not changed by the rule. By the `lookup` rule, if the first item in $k$ is $x$, then $x$ is replaced by the value $v$ of $x$ in its location $l$.

$\mathbb{K}$ is effective at defining nontrivial features of real-world languages regarding control flow, concurrency, and nondeterminism. This allowed many features in C (e.g., pointers, `goto`, `malloc`) to be defined near completely in $\mathbb{K}$ [9]. Likewise, $\mathbb{K}$'s inherent concurrency and nondeterminism enabled elegant treatments of Java multithreading [3] and JavaScript `for-in` enumeration [25].

For example, `goto` statements can be straightforwardly defined in $\mathbb{K}$, by treating control as explicit data. A standard approach [9] is to preprocess a *gotoMap* cell containing a map from labels to continuations, encoding the local control-flow for each label. The following shows a simple $\mathbb{K}$ rule for `goto` statements:

$$\texttt{goto:}\ \frac{\langle \texttt{goto}\ X \curvearrowright \ldots\rangle_k}{K}\ \langle \ldots X \mapsto K \ldots\rangle_{gotoMap}$$

This rules matches the label $X$ from the first item `goto` $X$ in the $k$ cell, looks up for the continuation $K$ bound to $X$ in *gotoMap*, and then replaces the current continuation by $K$.

```
1  requires "IMP.k"
2  (...)
3  module SPEC
4    imports IMP // IMP semantics
5    (...)
6    claim
7      <k> while ( 0 < n )
8          { s = s + n; n = n - 1; }
9      => .K ...</k>
10     <store>
11     s |-> (
12       S:Int
13       =>
14       S +Int ((N +Int 1) *Int N /Int 2));
15     n |-> (N:Int => 0)
16     </store>
17   requires N >=Int 0
18 endmodule
```

Fig. 1: An example reachability claim in `spec.k`.

**Deductive Verification in $\mathbb{K}$.** The $\mathbb{K}$ framework provides a language-agnostic deductive verifier based on Reachability Logic (RL) [6], which is parameterized by the user-defined $\mathbb{K}$ semantics of a language. It has been widely used to verify real-world programs, such as smart contracts [11,27,26].

RL is a proof system for proving partial correctness of a program, whose properties are specified as (all-path) reachability claims of the form $\phi \Rightarrow \psi$, where $\phi$ and $\psi$ are Matching Logic [29] *patterns* describing a *set* of states (i.e., $\mathbb{K}$ configurations). It means: any *terminating* path from a state satisfying $\phi$ reaches a state satisfying $\psi$. Skeirik et al. [32] extended the use of reachability claims as partial correctness to specify invariant properties of nonterminating systems, by "freezing" every reachable state of a nonterminating path to induce a corresponding finite prefix for which reachability claims apply.

Figure 1 shows a concrete example of a reachability claim, excerpted from $\mathbb{K}$'s tutorial[1]. $\mathbb{K}$ configurations in the claim are written in an xml-like notation, where the reachability relation "`=>`" is written locally within a cell to denote the difference between the initial and goal patterns of the claim. The claim asserts: upon termination of the `while` loop (line 9), the inital value of `s` is incremented by $((N+1) * N/2)$ (line 14), assuming the initial value of `n` is $N > 0$ (line 17).

This claim can be proved automatically via the following process: first, one writes the claim in the file `spec.k`, importing the semantics definition (e.g., `imp.k`). Then, running `kprove` for `spec.k` completes the proof as follows:

```
$ kprove spec.k
(output) #Top // success
```

Typically, `kprove` may require auxilary claims to prove the main claim $\phi \Rightarrow \psi$. Auxilary claims can be used to ignore the cycles reachable from $\phi$, as they correspond to infinite paths. When such cycles involve an intermediate pattern $\phi'$ but not $\phi$, they should be discharged via the auxilary claim $\phi' \Rightarrow \psi$.

### 2.2 The PROMELA Language

PROMELA is a popular high-level modeling language for specifying concurrent and distributed systems. It is used as the input language for the SPIN [13]

[1] `https://kframework.org/k-distribution/k-tutorial/1_basic/22_proofs/`

$$
\begin{array}{lll}
\mathit{Seq} & ::= & \mathit{Stmt}^{+} \\
\mathit{Stmt} & ::= & \mathit{BStmt} \mid \mathit{CStmt} \mid \mathit{Decl} \\
\mathit{Decl} & ::= & \mathit{Type}\ \mathit{Id}\texttt{[}\mathit{Int}\texttt{]} \\
 & \mid & \texttt{chan}\ \mathit{Id}\texttt{[}\mathit{Int}\texttt{]}\ \texttt{=}\ \texttt{[}\mathit{Int}\texttt{]}\ \texttt{of}\ \texttt{\{}\ \mathit{Type}^{+}\ \texttt{\}} \\
\mathit{Type} & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{chan} \\
\mathit{BStmt} & ::= & \mathit{Expr} \\
 & \mid & \mathit{Id}\texttt{[}\mathit{Expr}\texttt{]}\ \texttt{=}\ \mathit{Expr} \\
 & \mid & \mathit{Id}\texttt{[}\mathit{Expr}\texttt{]}\ \texttt{!}\ \mathit{Expr}^{+} \\
 & \mid & \mathit{Id}\texttt{[}\mathit{Expr}\texttt{]}\ \texttt{?}\ \mathit{Arg}^{+} \\
\mathit{Arg} & ::= & \mathit{Id}\texttt{[}\mathit{Expr}\texttt{]} \mid \mathit{Int} \mid \mathit{Bool}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{CStmt} & ::= & \texttt{if}\ \mathit{Option}^{+}\ \texttt{fi} \\
 & \mid & \texttt{do}\ \mathit{Option}^{+}\ \texttt{od} \\
 & \mid & \texttt{goto}\ \mathit{Id} \mid \texttt{break} \\
 & \mid & \texttt{atomic}\ \texttt{\{}\ \mathit{Seq}\ \texttt{\}} \\
\mathit{Expr} & ::= & \texttt{run}\ \mathit{Id}\texttt{(}\mathit{Expr}^{*}\texttt{)} \\
 & \mid & \mathit{Id}\texttt{[}\mathit{Expr}\texttt{]} \mid \mathit{Int} \mid \mathit{Bool} \\
 & \mid & \texttt{nfull(}\mathit{Id}\texttt{[}\mathit{Expr}\texttt{])} \\
 & \mid & \mathit{Id}\texttt{[}\mathit{Expr}\texttt{]}\ \texttt{?}\ \texttt{[}\mathit{Arg}^{+}\texttt{]} \\
 & \mid & \ominus \mathit{Expr} \mid \mathit{Expr} \odot \mathit{Expr} \\
\mathit{Option} & ::= & \texttt{::}\ \mathit{Expr}\ \texttt{->}\ \mathit{Seq}
\end{array}
$$

Fig. 2: An abstract syntactic subset of the full PROMELA. $+/*$ denote repetitions in EBNF notation; unary/binary operators are denoted $\ominus/\odot$, resp.

model checker—which received the 2001 ACM Software System Award [1]—for verification and simulation. SPIN/PROMELA has been applied to a wide range of systems, including cryptographic protocols [16], Linux synchronization primitives [10], and flight-guidance systems [5].

A PROMELA program consists of declarations of global data (e.g., integers, booleans, channels) and processes (`proctype`). Like C functions, process declarations provide arguments and a body containing a `;`-separated sequence (*Seq*) of statements, whose syntax considered in this paper is given in Figure 2. Statements include *basic* statements (*BStmt*) which define one-step atomic action, and *control* statements (*CStmt*) which organize the control-flow. All variables are considered as arrays; e.g., a variable `x` is syntactic sugar for `x[0]` (unlike C). An informal description of PROMELA is available on SPIN's official website[2].

Basic statements include condition statements, assignments, and channel operations. A condition statement is a standalone expression that gets skipped when it evaluates to true. It is typically used to guard the subsequent sequence. Channel operations are send/receive operations through a channel.

Similar to related formalisms (e.g., Hoare's CSP [12], Dijkstra's GCL [8]), PROMELA provides features for nondeterminism with `if`/`do` statements, whose inner options (*Option*) are selected nondeterministically when enabled. Unlike CSP/GCL, there is no syntactic restriction for the guards in each option sequence; options may start with arbitrary statements, allowing nested options.

A PROMELA program models a concurrent system of processes instantiated from `proctype` declarations. Processes may be active at startup (by annotating the `proctype` with `active`) or spawned by `run` expressions (e.g., in the initializer process `init`). Concurrent processes execute as interleavings of locally executed basic statements. Enclosing a sequence *SL* in `atomic {`*SL*`}` enforces atomic execution. Processes communicate via global variables or channels. Channels are either buffered or handshake: buffered channels provide asynchronous FIFO queues of finite capacity, while handshake channels synchronize sender and

[2] `https://spinroot.com`

```
1  int x = 0, turn = 0;
2  chan c = [10] of { int };
3  active proctype producer() {
4    do
5      :: turn == 0 -> c ! 42; turn = 1
6      :: turn != 0 -> skip
7    od
8  }
9  active proctype consumer() {
10   do
11     :: turn == 1 -> c ? x; turn = 0
12     :: turn != 1 -> skip
13   od
14 }
```

Fig. 3: A producer and consumer model written in PROMELA.

receiver. A channel `c` is declared as `chan c = [`$N$`] of {`$TL$`}`, where $N$ is the capacity ($N = 0$ for handshake) and $TL$ specifies the message format.

In PROMELA, basic statements can only be executed if their *enabledness* (or *executablility* in PROMELA term) condition holds; otherwise, they block until they become enabled by global state changes. For example, the buffered receive `c ? x` is enabled iff the channel `c` is nonempty and the arguments match the message. Likewise, a sequence of statements is enabled iff its first statement is enabled; an `atomic` statement is enabled iff its inner sequence is enabled; an `if`/`do` statement is enabled iff either one of the options is enabled. Accordingly, `atomic` acquires atomicity only when the inner sequence is enabled; `if`/`do` chooses an option when it is enabled.

Figure 3 shows a producer–consumer model written in PROMELA. Two `active` processes, `producer` and `consumer`, run concurrently and communicate via the buffered channel `c`. In its `do` loop, `producer` either (when `turn == 0`) sends 42 on `c` and sets `turn` to 1 (line 5), or (when `turn != 0`) executes `skip` (line 6). `consumer` behaves symmetrically: when `turn == 1` it receives 42 from `c` into `x` and sets `turn` to 0 (line 11). Here, "`->`" and `skip` are syntactic sugar for `;` and `true` (as a condition statement), respectively.

## 3 A Basic Semantics of PROMELA in $\mathbb{K}$

In this section, we give an overview of the semantics of PROMELA in $\mathbb{K}$. Rather than covering the full syntax presented in Figure 2, we focus on a simplified subset for which it is straightforward to define $\mathbb{K}$ rules. In subsequent sections, we use these rules as a baseline and extend them modularly to cover the full syntax within our scope.

### 3.1 Syntactic Subset

We impose three syntactic restrictions on the full syntax shown in Figure 2.

(Expression-Guarded Options) We require that each nondeterministic option in `if`/`do` start with an explicit expression as its guard. Namely, we restrict the syntax of *Option* to be:

$$\textit{Option} ::= \texttt{::}\ \textit{Expr}\ \texttt{->}\ \textit{Seq}$$

$$\langle\langle\langle Id\rangle_{ptName}\ \langle List\{Decl\}\rangle_{params}\ \langle Seq\rangle_{code}\ \langle Id \hookrightarrow K\rangle_{gotoMap}\rangle_{ptype^*}\rangle_{ptypes}$$
$$\langle\langle\langle Int\rangle_{pid}\ \langle Id\rangle_{pName}\ \langle K\rangle_{k}\ \langle Id \times Int \hookrightarrow Ref\rangle_{env}\ \langle Set\{Id\}\rangle_{lVars}\rangle_{proc^*}\rangle_{procs}$$
$$\langle Int \hookrightarrow PVal\rangle_{str}\ \langle Int \hookrightarrow Queue\rangle_{net}\ \langle Int_{\perp}\rangle_{lock}$$
$$\langle Int\rangle_{nextPid}\ \langle Int\rangle_{nextLoc}\ \langle Int\rangle_{nextBCid}\ \langle Int\rangle_{nextHCid}$$

$$\begin{array}{lll@{\qquad}lll}
BChan & ::= & bch(Int) & Value & ::= & PVal \mid CVal \\
HChan & ::= & hch(Int) & Ref & ::= & loc(Int) \mid CVal \\
CVal & ::= & uch \mid BChan \mid HChan & Queue & ::= & q(Int, Msg^*) \\
PVal & ::= & Int \mid Bool & Msg & ::= & m(Value^*)
\end{array}$$

Fig. 4: (a) Top-Level Configuration (b) Semantic Domains

Under this restriction, the enabledness conditions of options are explicitly tied to the outer `if`/`do` wrappers, so that nondeterministic selection is defined in a straightforward way: an option can be selected if its guard evaluates to true. This restriction will be dropped in Section 4.

(Basic Granularity) We assume a simple setting in which basic statements are the sole granularity of atomic execution in PROMELA: interleaving *cannot* occur during the execution of a basic statement, and *can* occur whenever a basic statement completes. This is enforced by excluding `atomic` statements from the syntactic category *CStmt*. This restriction will be dropped in Section 5.

(Pure Expressions) We assume that every expression used in PROMELA code is pure: evaluating an expression never produces side effects. To enforce this, we exclude `run` expressions from the syntactic category *Expr*, where expressions containing `run` as a sub-expression carry the side effect of spawning a process. This restriction will be dropped in Section 5.

### 3.2 Semantic Domains and Configuration

We describe the structure of our configuration in Figure 4a upon which the $\mathbb{K}$ rules are defined. Some core semantic domains are shown in Figure 4b.

The configuration contains core semantic components as nested cells such as: a set of `proctype` declarations (*ptypes*); a set of active processes (*procs*); a store (*store*) mapping locations to primitive values (*PVal*); a network (*net*) mapping buffered channel ids to the corresponding queues (*Queue*) of messages (*Msg*).

A *ptype* cell contains static informations of a process: e.g., its name (*ptName*), parameters (*params*), code (*code*), and the gotomap (*gotoMap*). A *proc* cell contains dynamic informations of a process: e.g., its continuation ($k$), pid (*pid*), local variables (*lVars*), and environment (*env*). *env* cell contains a mapping from integer-indexed variables (e.g., `x[0]`) to a reference (*Ref*), either to a store location, or to a buffered channel. The *lock* cell contains a global lock used for enforcing atomic execution of a given process; it may contain a pid to which exclusive execution is granted, or $\perp$ indicating that the lock is free. The cells *nextPid*, *nextLoc*, *nextBCid*, and *nextHCid* are used to supply fresh identifiers.

$$\left\langle \left\langle \begin{matrix} \langle 0\rangle_{pid}\ \langle\texttt{producer}\rangle_{pName}\ \langle\texttt{turn = 1} \curvearrowright \ldots\rangle_{k}\ \langle\emptyset\rangle_{lVars} \\ \langle \texttt{x[0]} \mapsto \mathit{loc}(0); \texttt{turn[0]} \mapsto \mathit{loc}(1); \texttt{c[0]} \mapsto \mathit{bch}(0)\rangle_{env} \end{matrix} \right\rangle_{proc} \left\langle \begin{matrix} \langle 1\rangle_{pid}\ \langle\texttt{consumer}\rangle_{pName}\ \langle\texttt{do ... od}\rangle_{k}\ \langle\emptyset\rangle_{lVars} \\ \langle \texttt{x[0]} \mapsto \mathit{loc}(0); \texttt{turn[0]} \mapsto \mathit{loc}(1); \texttt{c[0]} \mapsto \mathit{bch}(0)\rangle_{env} \end{matrix} \right\rangle_{proc} \right\rangle_{procs}$$

$$\langle\langle(\cdots)\rangle_{ptype}\langle(\cdots)\rangle_{ptype}\rangle_{ptypes}\langle 0 \mapsto 0; 1 \mapsto 0\rangle_{str}\ \langle 0 \mapsto \mathit{queue}(10, m(42))\rangle_{net}$$

$$\langle\bot\rangle_{lock}\ \langle 2\rangle_{nextPid}\ \langle 2\rangle_{nextLoc}\ \langle 1\rangle_{nextBCid}\ \langle 0\rangle_{nextHCid}$$

Fig. 5: A $\mathbb{K}$ configuration of the producer-consumer model

Figure 5 shows the $\mathbb{K}$ configuration of the produce-consumer model from Section 2, after executing two basic statements `turn == 0; c ! 42` in `producer`. Note that the global variables `x` and `turn` point to the values stored in *str*, and the channel `c` points to the queue (capacity 10) with a message 42 in *net*.

Following the notational conventions of $\mathbb{K}$, we omit nested structure when unambiguous: e.g., when *k* and *env* cells appear in $\mathbb{K}$ rules without mentioning any *proc*, they are implicitly assumed to be contained in the same *proc* cell.

### 3.3 Basic Statements Under Concurrency

As noted, basic statements should execute atomically. Ideally, we can enforce this in the $\mathbb{K}$ framework by encoding each complete one-step behavior of basic statements as a *single* $\mathbb{K}$ rule. This would immediately enforce atomic execution for each basic statement.

However, because PROMELA is a high-level language, the behavior of basic statements often decomposes into several orthogonal steps that are more modular to specify as separate rules. For example, a buffered receive `c ? x, y` naturally expands into: "enabledness check; dequeue; assign `x`; assign `y`", where $\mathbb{K}$ rules are modularly defined for each of these intermediate steps.

Under concurrency, this modular style raises consistency concerns. For instance, if two processes `p1` and `p2` execute `c ? x, y` concurrently when the channel `c` holds a single message, an interleaving such as "enabledness check (`p1`); enabledness check (`p2`); dequeue (`p1`)" would abruptly block `p2` from dequeueing the message.

To prevent such inconsistencies while retaining a modular style, we introduce two special $\mathbb{K}$ rules, $\mathsf{fire}$ and $\mathsf{rel}$, which enforce mutual exclusion between concurrent applications of *execution rules* performing intermediate computations of basic statements. Informally, for a basic statement *BS*, $\mathsf{fire}$ initiates the execution of *BS* by granting a free lock to the owner process, upon checking the enabledness of *BS*. $\mathsf{rel}$ then completes the execution by releasing the lock. The following rules formalize this behavior:

$$\text{fire: } \langle P\rangle_{pid}\ \langle \frac{BS}{\mathit{effect}(BS) \curvearrowright \texttt{\#rel}} \curvearrowright ...\rangle_k\ \langle \frac{\bot}{P}\rangle_{lock}\ \langle ENV\rangle_{env}\ \langle STR\rangle_{str}\ \langle NET\rangle_{net}$$

$$\text{if } [\![\mathit{enabled}(BS)]\!]_{ENV,STR,NET} = \top$$

$$\text{rel: } \langle \frac{\texttt{\#rel}}{.K} \curvearrowright ...\rangle_k\ \langle \frac{\cdot}{\bot}\rangle_{lock}$$

Here, $\mathit{effect}(BS)$ is defined to be a $\mathbb{K}$ continuation representing intermediate computations for $BS$, while $\mathit{enabled}(BS)$ denotes the enabledness condition. fire rule checks $\mathit{enabled}(BS)$ as a side-condition[3], via evaluation function $[\![\cdot]\!]_{(\cdot,\cdot,\cdot)}$ parameterized by the local environment, the store, and the network. Note that by Basic Granularity, the lock is released immediatly after $\mathit{effect}(BS)$ completes.

The side-effects and the enabledness conditions for the four basic statements– namely, condition/assignment/buffered send/buffered receive– are defined as follows, via the functions $\mathit{effect} : \mathit{BStmt} \to K$ and $\mathit{enabled} : \mathit{BStmt} \to \mathit{Expr}$.

$$\begin{array}{ll}
\mathit{effect}(E) = .K & \mathit{enabled}(E) = E \\
\mathit{effect}(X\texttt{[}E\texttt{] = }E') = \texttt{\#assign(}X\texttt{,}E\texttt{,}E'\texttt{)} & \mathit{enabled}(X\texttt{[}E\texttt{] = }E') = \mathit{true} \\
\mathit{effect}(X\texttt{[}E\texttt{] ! }EL) = \texttt{\#send(}X\texttt{,}E\texttt{,}EL\texttt{)} & \mathit{enabled}(X\texttt{[}E\texttt{] ! }EL) = \mathit{nfull}(X[E]) \\
\mathit{effect}(X\texttt{[}E\texttt{] ? }AL) = \texttt{\#recv(}X\texttt{,}E\texttt{,}AL\texttt{)} & \mathit{enabled}(X\texttt{[}E\texttt{] ? }AL) = X[E]?[AL]
\end{array}$$

Here, the side-effects are represented by *effect markers* (e.g., `#assign`), which is reduced via the (multi-step) execution rules, *atomically*. Handshake operations are excluded here; they are handled specially via a dedicated firing rule below.

**Example (Assignment).** We demonstrate concretely how fire and rel help define execution rules for assignments in a modular way. The following rules reduce the effect marker $\texttt{\#assign(}X\texttt{,}E\texttt{,}E'\texttt{)}$ in two steps:

$$\text{assign-eval: } \langle \texttt{\#assign(}X\texttt{,}\frac{E}{[\![E]\!]_{ENV,STR,NET}}\texttt{,}\frac{E'}{[\![E']\!]_{ENV,STR,NET}}\texttt{)} \curvearrowright ...\rangle_k$$

$$\langle ENV\rangle_{env}\ \langle STR\rangle_{str}\ \langle NET\rangle_{net} \quad \text{if at least one of } E \text{ or } E' \text{ is a non-value}$$

$$\text{assign-prim: } \langle \texttt{\#assign(}X\texttt{,}I : \mathit{Int}\texttt{,}V : \mathit{PVal}\texttt{)} \curvearrowright ...\rangle_k\ \langle ... X[I] \mapsto \mathit{loc}(J) ...\rangle_{env}\ \langle ... J \mapsto \frac{\cdot}{V} ...\rangle_{str}$$

$$\text{assign-chan: } \langle \texttt{\#assign(}X\texttt{,}I : \mathit{Int}\texttt{,}V : \mathit{CVal}\texttt{)} \curvearrowright ...\rangle_k\ \langle ... X[I] \mapsto \frac{\cdot}{V} ...\rangle_{env}$$

The assign-eval rule evaluates the index $E$ to an integer $I$ and $E'$ to a value $V$. If $V$ is a primitive value, assign-prim updates the store by $V$ at the location pointed by $X[]$; otherwise, if $V$ is a channel value, assign-chan updates the environment so that $X[I]$ points to the channel $V$.

Now consider the assignment `x = x + y` in the process with pid 1, where $\texttt{x} = 4$ and $\texttt{y} = 2$. This executes in lock-step via the sequence "fire; assign-eval;

[3] In the paper, we use $\top/\bot$ and `true`/`false` interchangeably for boolean values.

assign-prim; rel", as shown by the trace below (mutated portions are underlined):

$\langle 1\rangle_{pid}\ \langle \underline{\texttt{x = x + y}} \curvearrowright K\rangle_k\ \langle \texttt{x}[0]\mapsto 0; \texttt{y}[0]\mapsto 1\rangle_{env}\ \langle 0\mapsto 4; 1\mapsto 2\rangle_{str}\ \langle \underline{\bot}\rangle_{lock}$
$\to \langle 1\rangle_{pid}\ \langle \underline{\texttt{\#assign}}\texttt{(x, 0, }\underline{\texttt{x + y}}\texttt{)} \curvearrowright \texttt{\#rel} \curvearrowright K\rangle_k\ \langle \texttt{x}[0]\mapsto 0; \texttt{y}[0]\mapsto 1\rangle_{env}\ \langle 0\mapsto 4; 1\mapsto 2\rangle_{str}\ \langle 1\rangle_{lock}$
$\to \langle 1\rangle_{pid}\ \langle \underline{\texttt{\#assign(x, 0, 6)}} \curvearrowright \texttt{\#rel} \curvearrowright K\rangle_k\ \langle \texttt{x}[0]\mapsto 0; \texttt{y}[0]\mapsto 1\rangle_{env}\ \langle 0\mapsto \underline{4}; 1\mapsto 2\rangle_{str}\ \langle 1\rangle_{lock}$
$\to \langle 1\rangle_{pid}\ \langle \underline{\texttt{\#rel}} \curvearrowright K\rangle_k\ \langle \texttt{x}[0]\mapsto 0; \texttt{y}[0]\mapsto 1\rangle_{env}\ \langle 0\mapsto 6; 1\mapsto 2\rangle_{str}\ \langle \underline{1}\rangle_{lock}$
$\to \langle 1\rangle_{pid}\ \langle K\rangle_k\ \langle \texttt{x}[0]\mapsto 0; \texttt{y}[0]\mapsto 1\rangle_{env}\ \langle 0\mapsto 6; 1\mapsto 2\rangle_{str}\ \langle \bot\rangle_{lock}$

### 3.4 Communication via Channels

Following the approach above, we present the execution rules for channel operations. This concludes our execution rules for all four basic statements; condition statements do not require any execution rules by PURE EXPRESSIONS.

**Buffered Channels.** Communication via a buffered channel proceeds by processing the markers $\texttt{\#send(}X\texttt{,}E\texttt{,}EL\texttt{)}$ and $\texttt{\#recv(}X\texttt{,}E\texttt{,}AL\texttt{)}$, where $EL : Expr^*$ and $AL : Arg^*$ are sender's massage payloads and receiver's arguments, respectively. We list the execution rules for buffered channels below.

send-eval: $\langle \texttt{\#send(}X\texttt{,}\dfrac{E}{[\![E]\!]_{ENV,STR,NET}}\texttt{,}\dfrac{EL}{[\![EL]\!]_{ENV,STR,NET}}\texttt{)} \curvearrowright \ldots\rangle_k$
$\langle ENV\rangle_{env}\ \langle STR\rangle_{str}\ \langle NET\rangle_{net}$ if $E$ is a non-value or $EL$ is not a list of values

send-msg: $\langle \dfrac{\texttt{\#send(}X\texttt{,}I\texttt{,}VL\texttt{)}}{.K} \curvearrowright \ldots\rangle_k\ \langle \ldots X_I \mapsto bch(C)\ldots\rangle_{env}\ \langle \ldots C\mapsto q(\cdot, ML; \dfrac{nil}{m(VL)})\ldots\rangle_{net}$

recv-eval: $\langle \texttt{\#recv(}X\texttt{,}\dfrac{E}{[\![E]\!]_{ENV,STR,NET}}\texttt{,}AL\texttt{)} \curvearrowright \ldots\rangle_k\ \langle ENV\rangle_{env}\ \langle STR\rangle_{str}\ \langle NET\rangle_{net}$ if $\neg E : Int$

recv-msg: $\langle \dfrac{\texttt{\#recv(}X\texttt{,}I : Int\texttt{,}AL\texttt{)}}{\texttt{\#assignMsg(}AL\texttt{,}VL\texttt{)}} \curvearrowright \ldots\rangle_k\ \langle \ldots X_I \mapsto bch(C)\ldots\rangle_{env}\ \langle \ldots C\mapsto q(I, \dfrac{m(VL);}{nil}\ ML)\ldots\rangle_{net}$

assign-msg1: $\langle \dfrac{.K}{\texttt{\#assign(}X\texttt{,}I\texttt{,}V\texttt{)}} \curvearrowright \texttt{\#assignMsg(}\dfrac{X[I]}{nil}, AL, \dfrac{V}{nil}, VL\texttt{)} \curvearrowright \ldots\rangle_k$

assign-msg2: $\langle \texttt{\#assignMsg(}\dfrac{V}{nil}, AL, \dfrac{V}{nil}, VL\texttt{)} \curvearrowright \ldots\rangle_k$ assign-msg3: $\langle \dfrac{\texttt{\#assignMsg(}nil, nil\texttt{)}}{.K} \curvearrowright \ldots\rangle_k$

The rules send-eval and recv-eval work similarly to assign-eval. By slight abuse of notation in send-eval, the evaluation function $[\![\cdot]\!]_{(\cdot,\cdot,\cdot)}$ is used as a pointwise extension, returning $Value^*$ from $Expr^*$. send-msg enqueues the message $m(VL)$ to the queue pointed by $X_I$ (through a buffered channel $bch(C)$). In a similar way, recv-msg dequeues the message from the queue, and then receives the message to the arguments by iterative assignments via the rules assign-msg1 – assign-msg3.

**Handshake Channels.** As mentioned, the firing rule for handshake is separately defined, as it involves two $k$ cells for inter-process synchronization:

$$\textsf{hs-fire:}\ \langle ...\langle \frac{X\texttt{[}E\texttt{]}\ \texttt{!}\ EL}{.K} \curvearrowright ...\rangle_k\ \langle ENV\rangle_{env}\ ...\rangle_{proc}\ \langle STR\rangle_{str}\ \langle NET\rangle_{net}\ \langle \frac{\bot}{P}\rangle_{lock}$$
$$\langle ...\langle P\rangle_{pid}\ \langle \frac{X'\texttt{[}E'\texttt{]}\ \texttt{?}\ AL}{\texttt{\#assignMsg(}AL\texttt{,}EL\texttt{)} \curvearrowright \texttt{\#rel}} \curvearrowright ...\rangle_k\ \langle ENV'\rangle_{env}\ ...\rangle_{proc}$$
$$\text{if } [\![X\texttt{[}E\texttt{]}]\!]_{ENV,STR,NET} =_{HChan} [\![X'\texttt{[}E'\texttt{]}]\!]_{ENV',STR,NET}\ \wedge\ AL = [\![EL]\!]_{ENV,STR,NET}$$

The side condition checks if $X\texttt{[}E\texttt{]}$ and $X'\texttt{[}E'\texttt{]}$ denote the same handshake channel and if the sender's payload $EL$ matches the receiver arguments $AL$. After handshake, the control is given to the receiver ($P$), who continues to receive the message via assign-msg1 – assign-msg3 defined above.

### 3.5 Other $\mathbb{K}$ Rules

We list other $\mathbb{K}$ rules that are worth mentioning. The rules for local variable declaration can be found in Appendix A.

**Nondeterminism.** As Expression-Guarded Options require each option's enabledness condition to appear as its guard, the nondeterministic selection rule (select) for `if` statements can be defined straightforwardly, which selects the option whose guard evaluates to true and yields its sequence.

$$\textsf{select:}\ \langle \frac{\texttt{if}\ OL\ \texttt{(::}\ E\ \texttt{->}\ SL\texttt{)}\ OL'\ \texttt{fi}}{SL} \curvearrowright ...\rangle_k\ \langle ENV\rangle_{env}\ \langle STR\rangle_{str}\ \langle NET\rangle_{net}\ \langle \bot\rangle_{lock}$$
$$\text{if } [\![E]\!]_{ENV,STR,NET} = \top$$

Note that this rule requires the global lock to be free, thereby preventing inconsistent guard evaluation during the execution of a basic statement.

**Structural Rules.** Below we list the structural rules, which rearrange the configuration or control without changing the program's observable state.

$$\textsf{seq:}\ \langle \frac{S\ \texttt{;}\ SL}{S \curvearrowright SL} \curvearrowright ...\rangle_k \qquad \textsf{loop:}\ \langle \frac{.K}{\texttt{if}\ OL\ \texttt{fi}} \curvearrowright \texttt{do}\ OL\ \texttt{od} \curvearrowright ...\rangle_k$$

$$\textsf{goto:}\ \langle ...\langle X\rangle_{pName}\ \langle \frac{\texttt{goto}\ Y\ \curvearrowright ...}{K}\rangle_k\ ...\rangle_{proc}\ \langle ...\langle X\rangle_{ptName}\ \langle ...\ Y \mapsto K\ ...\rangle_{gotoMap}\ ...\rangle_{ptype}$$

$$\textsf{break1:}\ \langle \frac{\texttt{break}}{.K} \curvearrowright \texttt{do}\ OL\ \texttt{od} \curvearrowright ...\rangle_k \qquad \textsf{break2:}\ \langle \texttt{break} \curvearrowright \frac{KI : KItem}{.K} \curvearrowright ...\rangle_k\ \ \text{otherwise}$$

The seq rule decomposes a sequence $SL$ into $\mathbb{K}$ continuations. The goto rule performs an unconditional jump to the destination label, as covered in Section 2. We assume *gotoMap* is appropriately preprocessed, eliminating all the labels in the code. The loop rule produces `if` indefinitely, whenever `do` is encountered.

```
active proctype p1() {
  if
    :: A
    :: do :: B :: C od ; D
  fi
} // A,B,C : any basic statement
```

Fig. 6: Nested Options

```
int x = 0, y = 0;
chan c = [0] of { int };
active proctype p1()
{ if :: c ! 1 :: y = 1 fi }
active proctype p2()
{ if :: y = 1 :: c ? x fi }
```

Fig. 7: Cross-Process Interference

The loop breaks by the rules break1 and break2 (the side-condition "otherwise" means no other rules match).

# 4 Handling Nondeterminism

In this section, we drop the Expression-Guarded Options restriction introduced in the previous section. This syntactic extension allows options to be guarded by arbitrary statements, as in Figure 2. Consequently, the previous select rule no longer applies under the extended syntax. Devising a new set of rules for this setting is nontrivial. We begin with motivating examples that lead to our Load-and-Fire approach, which we develop at the end of this section.

In Figure 6, the outer `if` involves nested options: the second option is guarded by a whole `do` block. Consequently, `if` is guarded by the three *leading* basic statements `A`, `B`, and `C`, each of which can be selected when enabled.

In Figure 7, both of the `if` statements in processes `p1`, `p2` are guarded by send/receive operations via a handshake channel `c`. This raises cross-process interference among two nondeterministic choices: selecting the first option in `p1` forces `p2` to take its second (the handshake must occur), whereas selecting the second option in `p1` forces `p2` to take its first (the handshake cannot occur).

As motivated by the examples, `if` statements may involve (i) nested options, (ii) cross-process interference, or (iii) their combinations. Consequently, unlike select, their behavior cannot be specified by simply matching a guard (i.e., a leading basic statement) among the options, since guards may appear at *arbitrary nesting depth* within options.

## 4.1 The Load-and-Fire Approach

We address the challenge above by introducing *loading* rules, which are structural normalization rules that flatten nested `if` options into a canonical form. The canonical form is a *multiset* of $\mathbb{K}$ continuations, each being the flattened computation for each local branch of `if`.

For example, a continuation of the form `if ... fi` $\curvearrowright K_{global}$ is normalized by loading rules, into the canonical form $[BS_1 \curvearrowright K_1 \mid \cdots \mid BS_n \curvearrowright K_n] \curvearrowright K_{global}$ for each leading basic statement $BS_i$ appearing in `if ... fi`, where $[\_ \mid \_]$ is the multiset constructor. Intuitively, this canonical form represents a "superposition" of local continuations $BS_i \curvearrowright K_i \curvearrowright K_{global}$, where each $BS_i$ is available for syntactic matching (modulo associativity/commutativity for multisets) at the top-level of the continuation.

Accordingly, we can lift the previous fire and hs-fire rules to match under this set-lifted context, organized by the loading rules. This allows the firing rules to match against a basic statement under arbitrarily nested options. Below we give concrete definitions of the (lifted) firing rules and the loading rules, and revisit the motivating examples to demonstrate how they work.

**Firing Rules.** The following rules are the lifted versions of `fire` and `hs-fire`:

fire: $\langle P\rangle_{pid}\ \langle \frac{[BS \curvearrowright K_{local} \mid K_{rest}]}{effect(BS) \curvearrowright K_{local} \curvearrowright \texttt{\#rel}} \curvearrowright ...\rangle_k\ \langle ENV\rangle_{env}\ \langle STR\rangle_{str}\ \langle NET\rangle_{net}\ \langle \frac{\bot}{P}\rangle_{lock}$

if $[\![enabled(BS)]\!]_{ENV,STR,NET} = \top$

hs-fire: $\langle ...\langle \frac{[X\texttt{[}E\texttt{]}\ \texttt{!}\ EL \curvearrowright K_{local} \mid K_{rest}]}{K_{local}} \curvearrowright ...\rangle_k\ \langle ENV\rangle_{env} ...\rangle_{proc}\ \langle STR\rangle_{str}\ \langle NET\rangle_{net}$

$\langle ...\langle P\rangle_{pid}\ \langle \frac{[X'\texttt{[}E'\texttt{]}\ \texttt{?}\ AL \curvearrowright K'_{local} \mid K'_{rest}]}{\texttt{\#assignMsg(}AL\texttt{,}EL\texttt{)} \curvearrowright K'_{local} \curvearrowright \texttt{\#rel}} \curvearrowright ...\rangle_k\ \langle ENV'\rangle_{env} ...\rangle_{proc}\ \langle \frac{\bot}{P}\rangle_{lock}$

if $[\![X\texttt{[}E\texttt{]}]\!]_{ENV,STR,NET} =_{HChan} [\![X'\texttt{[}E'\texttt{]}]\!]_{ENV',STR,NET} \ \wedge\ AL = [\![EL]\!]_{ENV,STR,NET}$

Compared to the previous versions, the changes are found only in the $k$ cells: A basic statement $BS$ is matched under the set-lifted context $[BS \curvearrowright K_{local} \mid K_{rest}]$, where $K_{\text{rest}}$ matches the other branches. When $BS$ fires, the set-lifted continuation *collapses* back into a linear form, reflecting a nondeterministic selection.

**Loading rules.** Loading rules decompose a continuation of the form `if ... fi` $\curvearrowright K$ into $[BS_1 \curvearrowright K_1 \mid \cdots \mid BS_m \curvearrowright K_n] \curvearrowright K$, so that enabledness becomes visible at the front of each branch. As a special case, $BS \curvearrowright K$ is also lifted to $[BS] \curvearrowright K$, viewing $BS \equiv$ `if :: ` $BS$ ` fi`. The loading rules are defined as follows:

load-lift: $\langle \frac{S}{\texttt{\#load} \curvearrowright [S]} \curvearrowright ...\rangle_k$ if $Loadable(S) = \top$

load-bs: $\frac{\texttt{\#load}}{.K} \curvearrowright [BS \curvearrowright K]$

load-seq: $LT \curvearrowright [\frac{S\texttt{;}SL}{S \curvearrowright SL} \curvearrowright K]$

load-do: $LT \curvearrowright [\frac{.K}{\texttt{if}\ OL\ \texttt{fi}} \curvearrowright \texttt{do}\ OL\ \texttt{od} \curvearrowright K]$

load-if1: $\frac{LT \curvearrowright [\texttt{if (::}\ SL\texttt{)}\ OL\ \texttt{fi} \curvearrowright K]}{[LT \curvearrowright [SL \curvearrowright K] \mid LT \curvearrowright [\texttt{if}\ OL\ \texttt{fi} \curvearrowright K]]}$

load-if2: $\frac{LT \curvearrowright [\texttt{if (::}\ SL\texttt{)}\ \texttt{fi} \curvearrowright K]}{[LT \curvearrowright [SL \curvearrowright K]]}$

load-goto: $LT \curvearrowright [\frac{.K}{\texttt{true}} \curvearrowright \texttt{goto}\ X \curvearrowright K]$

load-break: $LT \curvearrowright [\frac{.K}{\texttt{true}} \curvearrowright \texttt{break} \curvearrowright K]$

Loading begins by lifting *Loadable* statements (i.e. basic and `if` statements) in a $k$ cell into multisets, with a special loader token `#load`. `#load` (bound to the metavariable $LT$) guides the local rewrite within the multiset, by recursively decomposing each inner options until a leading basic statement is reached.

### 4.2 Examples

Our Load-and-Fire semantics handle nested options and cross-process interference in an elegant way, as illustrated by the running examples below.

*Example 1.* The nested `if` appearing in Figure 6 is loaded so that the basic statements `A`, `B`, and `C` appear upfront, exposing each branch's enabledness for firing (via fire/hs-fire). We underline the portion reduced by the loading rules:

$$
\begin{array}{l}
\langle \underline{\texttt{if :: A :: do :: B :: C od fi; D}}\rangle_k \\
\to \langle \underline{\texttt{if :: A :: do :: B :: C od fi}} \curvearrowright \texttt{D}\rangle_k \\
\to \langle \underline{\texttt{\#load} \curvearrowright [\texttt{if :: A :: do :: B :: C od fi}]} \curvearrowright \texttt{D}\rangle_k \\
\to \langle [\texttt{\#load} \curvearrowright [\texttt{A}] \mid \underline{\texttt{\#load} \curvearrowright [\texttt{if :: do :: B :: C od fi}]}] \curvearrowright \texttt{D}\rangle_k \\
\to \langle [\texttt{\#load} \curvearrowright [\texttt{A}] \mid \underline{\texttt{\#load} \curvearrowright [\texttt{do :: B :: C od}]}] \curvearrowright \texttt{D}\rangle_k \\
\to \langle [\texttt{\#load} \curvearrowright [\texttt{A}] \mid \underline{\texttt{\#load} \curvearrowright [\texttt{if :: B :: C fi} \curvearrowright \texttt{do :: B :: C od}]}] \curvearrowright \texttt{D}\rangle_k \\
\to \langle [\texttt{\#load} \curvearrowright [\texttt{A}] \mid \texttt{\#load} \curvearrowright [\texttt{B} \curvearrowright \texttt{do :: B :: C od}] \mid \underline{\texttt{\#load} \curvearrowright [\texttt{if :: C fi} \curvearrowright \texttt{do :: B :: C od}]}] \curvearrowright \texttt{D}\rangle_k \\
\to \langle [\underline{\texttt{\#load} \curvearrowright [\texttt{A}]} \mid \texttt{\#load} \curvearrowright [\texttt{B} \curvearrowright \texttt{do :: B :: C od}] \mid \texttt{\#load} \curvearrowright [\texttt{C} \curvearrowright \texttt{do :: B :: C od}]] \curvearrowright \texttt{D}\rangle_k \\
\to \langle [\texttt{A} \mid \underline{\texttt{\#load} \curvearrowright [\texttt{B} \curvearrowright \texttt{do :: B :: C od}]} \mid \texttt{\#load} \curvearrowright [\texttt{C} \curvearrowright \texttt{do :: B :: C od}]] \curvearrowright \texttt{D}\rangle_k \\
\to \langle [\texttt{A} \mid \texttt{B} \curvearrowright \texttt{do :: B :: C od} \mid \underline{\texttt{\#load} \curvearrowright [\texttt{C} \curvearrowright \texttt{do :: B :: C od}]}] \curvearrowright \texttt{D}\rangle_k \\
\to \langle [\texttt{A} \mid \texttt{B} \curvearrowright \texttt{do :: B :: C od} \mid \texttt{C} \curvearrowright \texttt{do :: B :: C od}] \curvearrowright \texttt{D}\rangle_k
\end{array}
$$

*Example 2.* Cross-process interference in Figure 7 is resolved in the fully loaded form (intermediate loading steps omitted):

$$
\begin{array}{l}
\langle \texttt{if :: c ! 1 :: y = 1 fi}\rangle_{k_1} \langle \texttt{if :: y = 1 :: d ? x fi}\rangle_{k_2} \\
\to ! \langle [\texttt{c ! 1} \mid \texttt{y = 1}]\rangle_{k_1} \langle [\texttt{d ? x} \mid \texttt{y = 1}]\rangle_{k_2}
\end{array}
$$

Here, $k_1$ and $k_2$ denote (by abuse of notation) the $k$ cells of `p1` and `p2`, respectively. The rule `hs-fire` applies to this loaded form, making a joint nondeterministic choice across the two processes and thus capturing the interference.

## 5 More Extensions

In this section, we further extend the Load-and-Fire to incrementally add `atomic` blocks and `run`-expressions to our syntax, in a modular way. We briefly present the added/revised rules.

### 5.1 Adding Atomic Blocks

We add `atomic` blocks to our syntax (i.e., we drop Basic Granularity). Sequences in an `atomic` block execute atomically; e.g., `atomic{ x++; x++; }` is equivalent to `x = x + 2`. Intermixing `atomic` with constructs such as goto and handshake introduces subtle control-flow issues (Figure 8, 9) to be elaborated below. We outline extensions to the relevant $\mathbb{K}$ rules that cover these cases.

```
active proctype p() {
  goto L;
  atomic {
    A; L: B
  }
}
```

Fig. 8: Goto under atomicity.

```
chan q = [0] of { bool };
active proctype p1() {
  atomic { A; q!0; B }
}
active proctype p2() {
  atomic { q?0 ; C }
}
```

Fig. 9: Handshake under atomicity.

**Loading Rules.** We modularly extend the loading rules to handle atomicity. Since `if`, `do`, and `atomic` can be nested in arbitrary combinations, `atomic` blocks may hide leading basic statements and must therefore be decomposed by loading. We present the extended portion, treating `atomic` blocks as *Loadable*. The old load-lift and load-bs are deprecated; all other rules remain unchanged.

load-lift: $\langle P\rangle_{pid}\ \langle\dfrac{S}{\texttt{\#load}(P = L) \curvearrowright [S]} \curvearrowright \ldots\rangle_k\ \langle L\rangle_{lock}$ if $Loadable(S)$

load-bs1: $\dfrac{\texttt{\#load}(\top) \curvearrowright [BS \curvearrowright K]}{.K}$ load-bs2: $\dfrac{\texttt{\#load}(\bot)}{.K} \curvearrowright [BS \curvearrowright \dfrac{.K}{\texttt{\#rel}} \curvearrowright K]$

load-at1: $\dfrac{\texttt{\#load}(\top) \curvearrowright [\texttt{atomic \{ } SL \texttt{ \}} \curvearrowright K]}{SL}$ load-at2: $\texttt{\#load}(\dfrac{\bot}{\top}) \curvearrowright [\dfrac{\texttt{atomic \{ } SL \texttt{ \}}}{SL \curvearrowright \texttt{\#rel}} \curvearrowright K]$

The purpose of these rules is to insert a single `#rel` at the end of the *outermost* `atomic` block, thereby enforcing atomic execution within it. We refine the loader token with a Boolean flag (written as $\top/\bot$ inside of `#load(·)`) that records whether `#rel` has already been inserted to prevent incorrect early release. Together with the other loading rules, `#load(·)` is propagated along the nested structure, inserting `#rel` at each valid release points.

**Firing Rules.** We lift the firing rules to operate under atomicity, in accordance with the extended loading rules. The revised firing rules (i) no longer inserts `#rel`, as it is inserted by loading rules; and (ii) also matches a self-acquired lock, since basic statements may fire consecutively within an `atomic` block without releasing the lock.

fire: $\langle P\rangle_{pid}\ \langle\dfrac{[BS \curvearrowright K_{local} \mid K_{rest}]}{\mathit{effect}(BS) \curvearrowright K_{local}} \curvearrowright \ldots\rangle_k\ \langle ENV\rangle_{env}\ \langle STR\rangle_{str}\ \langle NET\rangle_{net}\ \langle\dfrac{L}{P}\rangle_{lock}$

if $L \in \{\bot, P\}\ \wedge\ [\![enabled(BS)]\!]_{ENV,STR,NET} = \top$

$$\textsf{hs-fire:}\ \langle ...\langle P\rangle_{pid}\ \langle \frac{[X\texttt{[}E\texttt{]}\ \texttt{!}\ EL \curvearrowright K_{local} \mid K_{rest}]}{[\texttt{true} \curvearrowright K_{local}]} \curvearrowright ...\rangle_k\ \langle ENV\rangle_{env} ...\rangle_{proc}\ \langle STR\rangle_{str}\ \langle NET\rangle_{net}$$

$$\langle ...\langle P'\rangle_{pid}\ \langle \frac{[X'\texttt{[}E'\texttt{]}\ \texttt{?}\ AL \curvearrowright K'_{local} \mid K'_{rest}]}{(L = P\ ?\ \texttt{\#toss(}P\texttt{)} : .K) \curvearrowright \texttt{\#assignMsg(}AL\texttt{,}EL\texttt{)} \curvearrowright K'_{local}} \curvearrowright ...\rangle_k\ \langle ENV'\rangle_{env} ...\rangle_{proc}$$

$$\langle \frac{L}{P'}\rangle_{lock} \qquad \text{if } [\![X\texttt{[}E\texttt{]}]\!]_{ENV,STR,NET} =_{HChan} [\![X'\texttt{[}E'\texttt{]}]\!]_{ENV',STR,NET} \wedge L \in \{\bot, P, P'\} \wedge AL = [\![EL]\!]_{ENV,STR,NET}$$

Additionally, under atomicity, handshake operations chained across multiple processes induce a global atomic chain of executions, as shown in Figure 9. If a handshake occurs while `p1` is executing atomically, atomicity is transferred to `p2`, which executes block `C`; upon completion, atomicity returns to `p1`. This mechanism is implemented in hs-fire via lock tossing: if the sender already holds the lock ($L = P$), the lock is tossed back to the sender upon releasing the lock. The relevant rules for lock-tossing appear below.

**Execution Rules.** We list the execution rules updated for atomicity. Irrelevant execution rules from Section 3 remain unchanged.

$$\textsf{toss1:}\ \langle \frac{\texttt{\#toss(}P\texttt{)} \curvearrowright K}{K \curvearrowright \texttt{\#toss(}P\texttt{)}} \curvearrowright \texttt{\#rel} \curvearrowright ...\rangle_k \qquad \textsf{toss2:}\ \langle \frac{\texttt{\#toss(}P\texttt{)} \curvearrowright \texttt{\#rel}}{.K} \curvearrowright ...\rangle_k\ \langle \frac{\cdot}{P}\rangle_{lock}$$

$$\textsf{loose:}\ \langle P\rangle_{pid}\ \langle ... \curvearrowright \frac{KI : KItem}{(KI = \texttt{\#toss(}\cdot\texttt{)}\ ?\ .K : KI)} \curvearrowright \texttt{\#rel}\ ...\rangle_k\ \langle \frac{P}{\bot}\rangle_{lock} \quad \text{otherwise}$$

$$\textsf{goto:}\ \langle ...\langle P\rangle_{pid}\ \langle X\rangle_{pName}\ \langle \frac{\texttt{goto}\ Y}{K} \curvearrowright ...\rangle_k ...\rangle_{proc}\ \langle \frac{L}{(B\ ?\ L : \bot)}\rangle_{lock}$$
$$\langle ...\langle X\rangle_{ptName}\ \langle ...\ Y \mapsto (B, K)\ ...\rangle_{gotoMap} ...\rangle_{ptype}$$

The rules toss1 and toss2 implement lock tossing used in hs-fire. toss1 propagates the marker `#toss(`$P$`)` (with $P$ the sender's pid) through the continuation until `#rel`. toss2 then sets the lock to $P$ instead of releasing it.

When execution blocks inside an `atomic` block, atomicity is lost temporarily. The loose rule implements this by releasing the lock when no (firing) rules match, via the side condition "otherwise". Any pending toss marker, if present, is invalidated. Note that `#rel` still remains, preserving the original atomicity.

Finally, atomicity may switch abruptly via `goto` when the atomicity of the source and destination differs (Figure 8). The revised `goto` rule reflects the atomicity of the destination $Y$: if $Y$ is (resp., is not) within an `atomic` block, indicated by $B = \top$ (resp., $B = \bot$), the lock is preserved (resp., released). Accordingly, the *gotoMap* is augmented with atomicity flags $B$ for each label.

### 5.2 Adding Impure Expressions

As our final language extension, we add `run` expressions (i.e., we drop Pure Expressions). For simplicity, we only consider `run` expressions used in condition statements, and in the righthand-side of the assignment.

Evaluating `run` expressions incurs side effects of spawning new processes. For example, assuming the *nextPid* cell contains 42, the side effect of $X$ `= run p1() + run p2()` is to spawn two processes `p1` and `p2` and to assign $X$ `= 42 + 43`. To reflect this, we extend the function $\mathit{effect} : \mathit{BStmt} \times \mathit{Int} \to K$ to be parameterized by *nextPid* as the second argument:

$$
\begin{aligned}
&\mathit{effect}(E, I) = \texttt{\#run(}E\texttt{)}\\
&\mathit{effect}(X\texttt{[}E\texttt{] = }E', I) = \texttt{\#run(}E'\texttt{)} \curvearrowright \texttt{\#assign(}X\texttt{,}E\texttt{,}E' \restriction_I\texttt{)}\\
&\mathit{effect}(E, I) = \mathit{effect}(E) \quad \text{(otherwise, define as before)}
\end{aligned}
$$

, where the purification function $(\cdot) \restriction_{(\cdot)}$: $\mathit{BStmt} \times \mathit{Int} \to \mathit{BStmt}$ substitutes all occurrences of `run` expressions in *BS* by the given pid's.

Accordingly, the fire rule is revised to use the updated effect and enabledness for basic statements that may involve `run` as subexpressions:

$$
\textsf{fire:}\quad \langle P\rangle_{pid}\ \langle \frac{[BS \curvearrowright K_{local} \mid K_{rest}]}{\mathit{effect}(BS, P') \curvearrowright K_{local}} \curvearrowright ...\rangle_k\ \langle ENV\rangle_{env}\ \langle STR\rangle_{str}\ \langle NET\rangle_{net}
$$
$$
\langle \frac{\bot}{P}\rangle_{lock}\ \langle P'\rangle_{nextPid} \quad \text{if } L \in \{\bot, P\} \ \wedge\ [\![\mathit{enabled}(BS \restriction_{P'})]\!]_{ENV,STR,NET} = \top
$$

For execution rules, we add rules that process the `#run(·)` markers. These rules decompose `#run(`$E$`)` into `#run(run `$X$`(`$EL$`))`'s for each `run `$X$`(`$EL$`)` contained in $E$. The run rule spawns a new process upon each `#run(run `$X$`(`$EL$`))`.

$$
\textsf{run:}\quad \langle \frac{\texttt{\#run(run }X\texttt{(}EL\texttt{))}}{\texttt{\#wait}} \curvearrowright ...\rangle_k\ \langle ENV\rangle_{env}\ \langle XS\rangle_{lVars}
$$
$$
\langle ...\ \langle X\rangle_{ptName}\ \langle DL\rangle_{params}\ \langle SL\rangle_{code}\ ...\rangle_{ptype}\ \langle \frac{P}{P+1}\rangle_{nextPid}
$$
$$
\frac{.\mathit{Cell}}{\langle\langle P\rangle_{pid}\ \langle X\rangle_{pName}\ \langle \texttt{\#init(}DL\texttt{,}EL\texttt{)} \curvearrowright \texttt{\#done} \curvearrowright SL\rangle_k\ \langle ENV \setminus XS\rangle_{env}\ \langle\emptyset\rangle_{local}\rangle_{proc}}
$$

Auxilary rules for the run rule can be found in Appendix B.

## 6 Case Study: Application to Deductive Verification

We implemented a prototype[4] $\mathbb{K}$ semantics of PROMELA. We validated our semantics via several examples with respect to the SPIN implementation. Using this prototype, we present a case study of using the $\mathbb{K}$ deductive verifier to prove reachability claims for PROMELA programs under our $\mathbb{K}$ semantics. Our case study includes examples for which SPIN fails to verify.

As a simple case, we revisit the reachability spec (Fig. 1) discussed in Section 2. Following the similar approach there, we can also prove the same claim for the counterpart program written in PROMELA shown in Figure 10. Note that leaving `N` as a parameter in the code yields an infinite family of models: each fixed `N` has a finite reachable state space (checkable by SPIN), but SPIN cannot verify the property for arbitrary `N`.

[4] The semantics definition, along with the case study and benchmark examples for validation, are available at `https://tinyurl.com/vmcai26-k-promela`

```
int n = N, s = 0 // N is parameter
active proctype sum() {
  do
    :: 0 < n -> s = s + n ; n--
    :: 0 >= n -> break
  od
}
```

Fig. 10: A PROMELA code for summation.

```
int disp = 0, serv = 0, crit = 0;
active proctype p1() {
  int tick;
  do
    :: atomic { tick = disp;
                disp = disp + 1 }
      ; atomic { tick == serv;
                 crit = crit + 1 }
      ; atomic { serv = serv + 1;
                 crit = crit - 1 }
  od
}
```

```
active proctype p2() {
  int tick;
  do
    :: atomic { tick = disp;
                disp = disp + 1 }
      ; atomic { tick == serv;
                 crit = crit + 1 }
      ; atomic { serv = serv + 1;
                 crit = crit - 1 }
  od
}
active proctype monitor() { freeze }
```

Fig. 11: A PROMELA code for bakery algorithm with 2 processes with a monitor.

As a nontrivial example, we verify Lamport's Bakery algorithm [14] with two processes (having an infinite state space) for which SPIN cannot verify. Figure 11 presents the PROMELA program for the Bakery Algorithm involving two concurrent processes `p1` and `p2`. Analogous to a real bakery (or bank), each process repeatedly: (i) obtains a ticket (`tick`) from the dispenser (`disp`), (ii) enters the critical section when the server (`serv`) calls its ticket, and then (iii) exits, via LOOP := `do ... od`. Each stage is denoted by the code fragment:

$$\begin{aligned}
\texttt{WAIT} &:= \texttt{atomic \{ tick = disp; disp = disp + 1 \}; ENTER}\\
\texttt{ENTER} &:= \texttt{atomic \{ tick == serv; crit = crit + 1 \}; EXIT}\\
\texttt{EXIT} &:= \texttt{atomic \{ serv = serv + 1; crit = crit - 1 \}}
\end{aligned}$$

Note that `tick` can grow indefinitely, inducing an infinite number of states.

We verify mutual exclusion (mutex) between `p1` and `p2`, by asserting the invariant $\texttt{crit} \leq 1$ for all reachable states. To express this property for nonterminating systems in reachability logic, we follow the similar approach proposed in [32]: we introduce a `monitor` process that nondeterministically "freezes" the global state via a pseudo-statement `freeze`, by setting the lock to a special value `#frozen` (also denoted "$\circledast$"). By defining the goal pattern $\phi_{goal}$ as the set of "frozen" states where $\texttt{crit} \leq 1$ holds, the main claim $\phi_{init} \Rightarrow \phi_{goal}$ asserts mutex for all finite prefix of the original bakery code without `monitor`, from the initial pattern $\phi_{init}$ with $\texttt{disp}=\texttt{serv}=N$ for some integer $N$.

Figure 12 shows the concrete spec for our main claim. We succintly write this main claim in the following high-level notation:

$$\langle \texttt{LOOP}, \texttt{LOOP}, \texttt{freeze}, \texttt{N}, \texttt{N}, \texttt{0}, \bot \rangle \Rightarrow \langle \texttt{?\_}, \texttt{?\_}, \texttt{.K}, \texttt{?\_}, \texttt{?\_}, \texttt{?C}{\leq}\texttt{1}, \circledast \rangle \tag{1}$$

```
1  claim:
2    <procs>
3      <proc>... // proc p1
4        <k> LOOP => ?_ </k>
5        <env>
6    disp[0] |-> loc(0); serv[0] |-> loc(1)
7    crit[0] |-> loc(2); tick[0] |-> loc(3)
8        </env>
9   ...</proc>
10     <proc>... // proc p2
11       <k> LOOP => ?_ </k>
12       <env>
13   disp[0] |-> loc(0); serv[0] |-> loc(1)
14   crit[0] |-> loc(2); tick[0] |-> loc(4)
15       </env>
16  ...</proc>
17     <proc>... // proc monitor
18       <k> freeze => .K </k>
19  ...</proc>
20   </procs>
21   <str>
22     0 |-> (N:Int => ?_); 1 |-> (N => ?_);
23     2 |-> (0 => ?C);
24     3 |-> (0 => ?_); 4 |-> (0 => ?_)
25   </str>
26   <lock> #none => #frozen </lock>
27 ensures ?C <=Int 1 // MUTEX
```

Fig. 12: The main claim

```
1  claim:
2    <procs>
3      <proc>... // proc p1
4        <k> ENTER! => ?_ </k>
5        <env>
6    disp[0] |-> loc(0); serv[0] |-> loc(1)
7    crit[0] |-> loc(2); tick[0] |-> loc(3)
8        </env>
9   ...</proc>
10     <proc>... // proc p2
11       <k> WAIT! => ?_ </k>
12       <env>
13   disp[0] |-> loc(0); serv[0] |-> loc(1)
14   crit[0] |-> loc(2); tick[0] |-> loc(4)
15       </env>
16  ...</proc>
17     <proc>... // proc monitor
18       <k> freeze => .K </k>
19  ...</proc>
20   </procs>
21   <str>
22     0 |-> (N +Int 1 => ?_);
23     1 |-> (N:Int => ?_); 2 |-> (0 => ?C);
24     3 |-> (N => ?_); 4 |-> (0 => ?_)
25   </str>
26   <lock> #none => #frozen </lock>
27 ensures ?C <=Int 1 // MUTEX
```

Fig. 13: An auxiliary claim

The seven components in the tuple correspond to: (the continuations of) `p1`,`p2`, `monitor`, (the values of) `disp` (line 22), `serv` (line 22), `crit` (line 23), and the global lock (line 26). In the goal pattern, "`?_`" denote "don't care" variables, and "`?C`$\leq$`1`" means `crit` $\leq 1$, where "`?`" indicate newly introduced variables.

During the proof of the main claim, we found that the bakery code induces three cycles that do not contain $\phi_{init}$. As noted in Section 2, they can be ignored via adding auxiliary claims, listed as follows:

$$\langle \mathtt{WAIT!(T)}, \mathtt{WAIT!(T)}, \mathtt{freeze}, \mathtt{N}, \mathtt{N}, 0, \bot \rangle \Rightarrow \langle \mathtt{?\_}, \mathtt{?\_}, \mathtt{.K}, \mathtt{?\_}, \mathtt{?\_}, \mathtt{?C}{\leq}\mathtt{1}, \circledast \rangle \tag{2}$$

$$\langle \mathtt{ENTER!(N)}, \mathtt{WAIT!}(\cdot), \mathtt{freeze}, \mathtt{N}+1, \mathtt{N}, 0, \bot \rangle \Rightarrow \langle \mathtt{?\_}, \mathtt{?\_}, \mathtt{.K}, \mathtt{?\_}, \mathtt{?\_}, \mathtt{?C}{\leq}\mathtt{1}, \circledast \rangle \tag{3}$$

$$\langle \mathtt{WAIT!}(\cdot), \mathtt{ENTER!(N)}, \mathtt{freeze}, \mathtt{N}+1, \mathtt{N}, 0, \bot \rangle \Rightarrow \langle \mathtt{?\_}, \mathtt{?\_}, \mathtt{.K}, \mathtt{?\_}, \mathtt{?\_}, \mathtt{?C}{\leq}\mathtt{1}, \circledast \rangle \tag{4}$$

Here, `WAIT!(T)` denotes the fully-loaded continuation for `WAIT`, under the local variable `tick`=`T`, and likewise for `ENTER!(T)`. Indeed, the initial pattern of the claim 2 is simply the fully-loaded version of the initial pattern of the claim 1. The initial patterns of the claims 3 and 4 are obtained by executing the `atomic` block of `p1` and `p2`, respetively, from the initial pattern of the claim 2. Figure 13 shows the concrete spec for the claim 3.

Encoding the four claims (1 main / 3 auxiliary) in `spec.k` and running it with `kprove`, it successfully terminates with output `#Top` in approximately 5 minutes in a laptop computer (i5-1335U 2.50 GHz / 16 GB RAM).

## 7 Related Work

Substantial work exists on establishing a formal semantics of PROMELA. In [23,2,24], the semantics is given by a low-level labeled transition system. [38,17,31] give structural operational semantics (SOS) [28], while [18] gives a denotational semantics. From the analysis perspective, [38,24] remain in the scope of SPIN's LTL verification. Other lines of work extend the analysis to other techniques such as abstract interpretation [17,18], and PROMELA-to-C refinement [31]. We identified two works that establish mechanized, executable semantics of PROMELA—one in ACL2 [2] and another in Isabelle/HOL [24]. However, to the best of our knowledge, no prior work provides an executable semantics that enables code-level deductive verification of PROMELA.

$\mathbb{K}$ [30] is an executable semantic framework for defining programming languages. It emphasizes modularity—addressing a well-known limitation of SOS—so new features can be added without revising unrelated rules. $\mathbb{K}$ has proved effective for control-intensive features (e.g., exceptions, call/cc) and has been used to formalize several real-world languages [9,3,25,37]. The framework was later formalized as Matching Logic [29], yielding a deductive system [6] with commercial applications, notably in smart-contract verification [27,26,11]. This work presents the first executable PROMELA semantics defined in $\mathbb{K}$.

$\mathbb{K}$ is grounded in Rewriting Logic [20], which is introduced as a general formalism for specifying concurrent systems [19]. Classic process calculi—including CCS [21] and the Pi-calculus [22]— have been studied within Rewriting Logic [4,7,35,36,33,34]. Rewriting Logic's inherent nondeterminism and true concurrency make such semantics natural to specify; we exploit this in our Load-and-Fire semantics via multiset matching.

## 8 Concluding Remarks

We have presented a faithful, executable semantics of PROMELA in the $\mathbb{K}$ framework. Our semantics enables code-level deductive verification of PROMELA models, including infinite-state systems, a capability previously unavailable for PROMELA. Beyond providing a precise, machine-readable reference, it bridges model checking and deductive reasoning: properties beyond explicit-state model checking can now be proved directly on PROMELA programs.

We have also introduced Load-and-Fire, an elegant semantic pattern that yields a modular, uniform treatment of guarded nondeterminism, cross-process interference, and atomicity in $\mathbb{K}$. Beyond PROMELA, this constitutes a reusable methodology for $\mathbb{K}$-based language semantics of guarded concurrency, with natural applications to constructs such as Go's `select` and Erlang's `receive`.

Future work includes supporting temporal reasoning for deductive verification (e.g., LTL) strengthening proof automation for PROMELA (e.g., reusable lemma libraries), and conducting additional case studies on complex PROMELA models. We also plan to apply Load-and-Fire to other programming languages with guarded choice and synchronous communication.

## References


1. Association for Computing Machinery: Acm software system award — gerard j. holzmann (for spin) (2001), https://awards.acm.org/award-recipients/holzmann_1625680, award citation and year. Accessed 2025-09-10
2. Bevier, W.: Toward an operational semantics of promela in acl2. In: Proceedings of the Third SPIN Workshop (1997)
3. Bogdanas, D., Roşu, G.: K-Java: A complete semantics of Java. In: Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 445–456. ACM (2015). https://doi.org/10.1145/2676726.2676982
4. Braga, C., Meseguer, J.: Modular rewriting semantics in practice. Electronic Notes in Theoretical Computer Science **117**, 393–416 (2005). https://doi.org/10.1016/j.entcs.2004.06.019, proceedings of the 5th International Workshop on Rewriting Logic and its Applications (WRLA'04)
5. Choi, Y.: From nusmv to spin: Experiences with model checking flight guidance systems. Form. Methods Syst. Des. **30**(3), 199–216 (Jun 2007). https://doi.org/10.1007/s10703-006-0027-9, https://doi.org/10.1007/s10703-006-0027-9
6. Ştefănescu, A., Ştefan Ciobâcă, Mereuţă, R., Moore, B.M., Şerbănuţă, T.F., Roşu, G.: All-path reachability logic. In: Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA'14). LNCS, vol. 8560, pp. 425–440. Springer (July 2014). https://doi.org/http://dx.doi.org/10.1007/978-3-319-08918-8_29
7. Degano, P., Gadducci, F., Priami, C.: A causal semantics for CCS via rewriting logic. Theoretical Computer Science **275**(1–2), 259–282 (2002). https://doi.org/10.1016/S0304-3975(01)00165-7
8. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (Aug 1975). https://doi.org/10.1145/360933.360975, https://doi.org/10.1145/360933.360975
9. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. In: Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. vol. 47, pp. 533–544. ACM (2012). https://doi.org/10.1145/2103656.2103719
10. Evrard, H., Donaldson, A.F.: Model checking futexes. In: Model Checking Software: 29th International Symposium, SPIN 2023, Paris, France, April 26–27, 2023, Proceedings. p. 41–58. Springer-Verlag, Berlin, Heidelberg (2023). https://doi.org/10.1007/978-3-031-32157-3_3, https://doi.org/10.1007/978-3-031-32157-3_3
11. Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., Moore, B.M., Zhang, Y., Park, D., Ştefănescu, A., Roşu, G.: Kevm: A complete formal semantics of the ethereum virtual machine. In: 31st IEEE Computer Security Foundations Symposium (CSF). pp. 204–217. IEEE Computer Society (2018). https://doi.org/10.1109/CSF.2018.00022
12. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (Aug 1978). https://doi.org/10.1145/359576.359585, https://doi.org/10.1145/359576.359585
13. Holzmann, G.: The model checker spin. IEEE Transactions on Software Engineering **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521
14. Lamport, L.: A new solution of dijkstra's concurrent programming problem. Commun. ACM **17**(8), 453–455 (Aug 1974). https://doi.org/10.1145/361082.361093, https://doi.org/10.1145/361082.361093

15. Lazar, D., Arusoaie, A., ŞerbănuŢă, T.F., Ellison, C., Mereuta, R., Lucanu, D., Roşu, G.: Executing formal semantics with the K tool. In: Proceedings of the International Symposium on Formal Methods. Lecture Notes in Computer Science, vol. 7436, pp. 267–271. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_23
16. Maggi, P., Sisto, R.: Using spin to verify security properties of cryptographic protocols. In: Proceedings of the 9th International SPIN Workshop on Model Checking of Software. p. 187–204. Springer-Verlag, Berlin, Heidelberg (2002)
17. María del Mar Gallardo, Pedro Merino, E.P.: A generalized semantics of promela for abstract model checking. Formal Aspects of Computing **16**(3), 166–193 (2004). https://doi.org/https://doi.org/10.1007/s00165-004-0040-y
18. Marco Comini, Maria del Mar Gallardo, A.V.: A denotational semantics for promela addressing arbitrary jumps. In: Proceedings of LOPSTR 2021 (2021)
19. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992). https://doi.org/10.1016/0304-3975(92)90182-F
20. Meseguer, J.: Twenty years of rewriting logic. J. Log. Algebraic Methods Program. **81**(7-8), 721–781 (2012). https://doi.org/10.1016/J.JLAP.2012.06.003, https://doi.org/10.1016/j.jlap.2012.06.003
21. Milner, R.: A Calculus of Communicating Systems, Lecture Notes in Computer Science, vol. 92. Springer Berlin Heidelberg, Berlin, Heidelberg (1980). https://doi.org/10.1007/3-540-10235-3, http://link.springer.com/10.1007/3-540-10235-3
22. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I. Information and Computation **100**(1), 1–40 (Sep 1992). https://doi.org/10.1016/0890-5401(92)90008-4
23. Natarajan, V., Holzmann, G.: Outline for an operational semantics of promela. In: Proceedings of the Second SPIN Workshop (1996)
24. Neumann, R.: Using promela in a fully verified executable ltl model checker. In: Verified Software: Theories, Tools and Experiments. Springer, Cham (2014). https://doi.org/https://doi.org/10.1007/978-3-319-12154-3_7
25. Park, D., Stefănescu, A., Roşu, G.: KJS: A complete formal semantics of JavaScript. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 346–356. ACM (2015). https://doi.org/10.1145/2737924.2737991
26. Park, D., Zhang, Y., Roşu, G.: End-to-end formal verification of ethereum 2.0 deposit smart contract. In: Computer Aided Verification – 32nd International Conference, CAV 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12224, pp. 151–164. Springer (2020). https://doi.org/10.1007/978-3-030-53288-8_8
27. Park, D., Zhang, Y., Saxena, M., Daian, P., Roşu, G.: A formal verification tool for ethereum vm bytecode. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018). pp. 912–915. ACM (2018). https://doi.org/10.1145/3236024.3264591
28. Plotkin, G.D.: A structural approach to operational semantics. DAIMI Report FN-19, Department of Computer Science, Aarhus University, Aarhus, Denmark (1981)
29. Roşu, G.: Matching logic. Logical Methods in Computer Science **13**(4), 1–61 (December 2017). https://doi.org/http://arxiv.org/abs/1705.06312

30. Rosu, G., Serbănută, T.F.: An overview of the K semantic framework. The Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
31. Sharma, A.: A refinement calculus for promela. In: 2013 18th International Conference on Engineering of Complex Computer Systems (ICECCS). pp. 75–84. IEEE (2013). https://doi.org/10.1109/ICECCS.2013.20
32. Skeirik, S., Stefanescu, A., Meseguer, J.: A constructor-based reachability logic for rewrite theories. In: Fioravanti, F., Gallagher, J.P. (eds.) Logic-Based Program Synthesis and Transformation. pp. 201–217. Springer International Publishing, Cham (2018)
33. Stehr, M.O.: Cinni – a generic calculus of explicit substitutions and its application to $\lambda$-, $\varsigma$- and $\pi$-calculi. In: Futatsugi, K. (ed.) Proceedings of the 3rd International Workshop on Rewriting Logic and its Applications (WRLA 2000). Electronic Notes in Theoretical Computer Science, vol. 36, pp. 70–92. Elsevier, Amsterdam, The Netherlands (2000). https://doi.org/10.1016/S1571-0661(05)80125-2
34. Thati, P., Sen, K., Martí-Oliet, N.: An executable specification of asynchronous $\pi$-calculus semantics and may testing in maude 2.0. In: Gadducci, F., Montanari, U. (eds.) Rewriting Logic and its Applications: Fourth International Workshop, Pisa, Italy, September 19–21, 2002, Electronic Notes in Theoretical Computer Science, vol. 71, pp. 261–281. Elsevier, Amsterdam, The Netherlands (2004). https://doi.org/10.1016/S1571-0661(05)82539-3
35. Verdejo, A., Martí-Oliet, N.: Two case studies of semantics execution in maude: {CCS} and {LOTOS}. Formal Methods in System Design **27**(1–2), 113–172 (2005). https://doi.org/10.1007/s10703-005-2254-x
36. Viry, P.: Input/output for elan. In: Meseguer, J. (ed.) Proceedings of the 1st International Workshop on Rewriting Logic and its Applications (WRLA '96), Electronic Notes in Theoretical Computer Science, vol. 4, pp. 51–64. Elsevier, Amsterdam, The Netherlands (1996). https://doi.org/10.1016/S1571-0661(04)00033-7
37. Wang, K., Wang, J., Poskitt, C.M., Chen, X., Sun, J., Cheng, P.: K-ST: A formal executable semantics of the Structured Text language for plcs. IEEE Transactions on Software Engineering **49**(10), 4796–4813 (2023). https://doi.org/10.1109/TSE.2023.3315292
38. Weise, C.: An incremental formal semantics for promela. In: Proceedings of the Third SPIN Workshop (1997)


# Appendix

## A Rules for Local Declaration

The statement $T\ \ X[I]$ declares a local (array) variable $X$ of (integer) size $I$, initialized by default values defined for each typename $T \in \{\texttt{int}, \texttt{bool}, \texttt{chan}\}$.

Declaration for primitive types is defined by the `decl-prim` rule, where *str* is initialized to the default value $\mathit{initVal}(PT)$ for $PT \in \{\texttt{int}, \texttt{bool}\}$:

`decl-prim`:

$$\langle \frac{PT\ \ X\texttt{[}I\texttt{]}}{.K} \curvearrowright \ldots \rangle_k \ \langle \frac{ENV}{ENV[loc(J)/X[0], \ldots, loc(J+I-1)/X[I-1]]} \rangle_{env} \ \langle \frac{XS}{XS \cup \{X\}} \rangle_{lVars}$$
$$\langle \frac{STR}{STR[\mathit{initVal}(PT)/J, \ldots, \mathit{initVal}(PT)/J+I-1]} \rangle_{str} \ \langle \frac{J}{J+I} \rangle_{nextLoc}$$

We write $M[V_1, \ldots, V_n/X_0, \ldots, X_n]$ to denote parallel substitution for a map $M$. In the rule, the environment binds each indexed variable to location `loc(·)`.

For channel declarations, the variables are directly bound to channel objects. The `decl-uch` rule for uninitialized channel declaration binds the variable to the undefined channel `uch`.

$$\texttt{decl-uch:}\ \ \langle \frac{\texttt{chan}\ X\texttt{[}I\texttt{]}}{.K} \curvearrowright \ldots\rangle_k\ \langle \frac{ENV}{ENV[uch/X[0], \ldots, uch/X[I-1]]}\rangle_{env}\ \langle \frac{XS}{XS \cup \{X\}}\rangle_{lVars}$$

For initialized channel declarations, The rules `decl-bch` and `decl-hch` binds each variable to a buffered channel `bch(·)` and handshake channel `hch(·)`, resp.

`decl-bch:`

$$\langle \frac{\texttt{chan}\ X\texttt{[}I\texttt{] = [}N\texttt{] of \{}TL\texttt{\}}}{.K} \curvearrowright \ldots\rangle_k\ \langle \frac{ENV}{ENV[bch(J)/X[0], \ldots, bch(J+I-1)/X[I-1]]}\rangle_{env}$$
$$\langle \frac{XS}{XS \cup \{X\}}\rangle_{lVars}\ \langle \frac{STR}{[q(N,\epsilon)/J, \ldots, q(N,\epsilon)/J+I-1]}\rangle_{net}\ \langle \frac{J}{J+I}\rangle_{nextBCid}\quad \text{if } N > 0$$

`decl-hch:`

$$\langle \frac{\texttt{chan}\ X\texttt{[}I\texttt{] = [0] of \{}TL\texttt{\}}}{.K} \curvearrowright \ldots\rangle_k\ \langle \frac{ENV}{ENV[hch(J)/X[0], \ldots, hch(J+I-1)/X[I-1]]}\rangle_{env}$$
$$\langle \frac{XS}{XS \cup \{X\}}\rangle_{lVars}\ \langle \frac{J}{J+I}\rangle_{nextHCid}$$

## B Rules for Run Expressions

A composite `run` expression is decomposed via the rules `run1` – `run3`.

$$\texttt{run1:}\ \langle \frac{\texttt{\#run(}E \odot E'\texttt{)}}{\texttt{\#run(}E\texttt{)} \curvearrowright \texttt{\#run(}E'\texttt{)}} \curvearrowright \ldots\rangle_k \qquad \texttt{run2:}\ \langle \frac{\texttt{\#run(}\ominus E\texttt{)}}{\texttt{\#run(}E\texttt{)}} \curvearrowright \ldots\rangle_k$$

$$\texttt{run3:}\ \langle \frac{\texttt{\#run(}E\texttt{)}}{.K} \curvearrowright \ldots\rangle_k \quad \text{otherwise}$$

The rules `sync`,`init1`, and `init2` are used in the main `run` rule to initialize a spawned process in a synchronized way.

$$\texttt{sync:}\ \ \langle \frac{\texttt{\#wait}}{.K} \curvearrowright \ldots\rangle_k\ \langle \frac{\texttt{\#done}}{.K} \curvearrowright \ldots\rangle_k \qquad \texttt{init1:}\ \langle \frac{\texttt{\#init(}nil\texttt{,}\,nil\texttt{)}}{.K} \curvearrowright \ldots\rangle_k$$

$$\texttt{init2:}\ \langle \frac{.K}{T\ \ X\texttt{[}E\texttt{]} \curvearrowright \texttt{\#assign(}X\texttt{,}E\texttt{,}E'\texttt{)}} \curvearrowright \texttt{\#init(}\frac{T\ \ X\texttt{[}E\texttt{]}}{nil}\texttt{,}\,DL\texttt{,}\,\frac{E'}{nil}\texttt{,}\,EL\texttt{)} \curvearrowright \ldots\rangle_k$$