An Operational Semantics for a Fragment of PRS

Lavindra de Silva¹, Felipe Meneguzzi² and Brian Logan³

¹ Institute for Advanced Manufacturing, University of Nottingham, Nottingham, UK
² Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil
³ School of Computer Science, University of Nottingham, Nottingham, UK
lavindra.desilva@nottingham.ac.uk, felipe.meneguzzi@pucrs.br, brian.logan@nottingham.ac.uk

Abstract
The Procedural Reasoning System (PRS) is arguably the first implementation of the Belief–Desire–Intention (BDI) approach to agent programming. PRS remains extremely influential, directly or indirectly inspiring the development of subsequent BDI agent programming languages. However, perhaps surprisingly given its centrality in the BDI paradigm, PRS lacks a formal operational semantics, making it difficult to determine its expressive power relative to other agent programming languages. This paper takes a first step towards closing this gap, by giving a formal semantics for a significant fragment of PRS. We prove key properties of the semantics relating to PRS-specific programming constructs, and show that even the fragment of PRS we consider is strictly more expressive than the plan constructs found in typical BDI languages.

1 Introduction
The Procedural Reasoning System (PRS) [Georgeff and Ingrand, 1989; Georgeff and Lansky, 1987; 1986] is generally recognised as one of the first implementations of the Belief–Desire–Intention (BDI) model of agency and practical reasoning. PRS has been extremely influential, and is still widely used [Ghallab et al., 2016], particularly in robotics, e.g., [Ingrand et al., 1996; Alami et al., 1998; Foughali et al., 2016; Niemueller et al., 2017; Lemaignan et al., 2017]. For example, PRS-based systems secured first and second places in recent RoboCup and ICAPS logistics competitions. PRS has also influenced the design of many subsequent BDI-based agent programming languages, e.g., [Rao, 1996; Huber, 1999; Busetta et al., 1999; Winikoff et al., 2002; Morley and Myers, 2004; Sardina et al., 2006; Sardina and Padgham, 2011], though most of these languages implement only a subset of the programming language features supported by PRS. Surprisingly, given its centrality in the BDI paradigm, PRS lacks a formal operational semantics, which makes it difficult to determine its expressive power relative to other BDI agent programming languages, or to verify the correctness of PRS programs. For example, there is a widespread “folk belief” in the agent programming community that the plan graphs used by PRS are more expressive than most if not all other BDI agent formalisations, yet no proof of this intuition exists.

In this paper, we give a formal semantics for a significant fragment of PRS. We focus on language features specific to PRS, and in particular, its graph-based representation for plans and its programming constructs for maintaining a condition (i.e., maintenance goals). We make three main contributions. First, we develop a formalisation for the syntax of PRS as a directed bipartite graph (Section 2). Second, we provide an operational semantics that accounts for graph-based plans and adopting, suspending, resuming, and aborting (nested) maintenance goals (Section 3). Third, we prove key properties of the semantics of these PRS-specific programming constructs, and show that PRS plan-graphs are strictly more expressive than the plan rules found in typical BDI languages (Section 4). In Section 5, we briefly discuss related work and conclude.

2 PRS Syntax
We briefly recall the syntax and deliberation cycle of PRS, as defined in [Ingrand, 1991]. In the interests of brevity, we omit some features of the language, including meta-level reasoning, ‘true concurrency’, semaphores, and features such as ‘if’ and ‘else’ statements expressible in terms of the fragment we define. Plans in PRS consist of graphs. For compactness, and to allow a precise specification of the semantics, we adopt a plan-rule notation similar to that used in other BDI-based agent programming languages to specify the triggering and context conditions of plans, and specify plan bodies using a textual representation of bipartite graphs. We assume a first-order language with a vocabulary consisting of mutually disjoint and infinite sets of variable, constant, predicate, node, event-goal, and action symbols.

A PRS agent is defined by a belief base B, an action-library A, and a plan-library P. A belief base is a set of ground atoms. An action-library is a set of action-rules specifying the actions available to the agent. An action is of the form act(ℓ), where act is an n-ary action symbol denoting an evaluable function that may change the agent’s environment, and ℓ = ℓ₁, . . . , ℓ_n are (possibly ground) terms. Action-rules are similar to STRIPS operators, and are of the form act(ℓ);ψ ← Φ⁺;Φ⁻, where ℓ = ℓ₁, . . . , ℓ_n are variables; ψ is a formula specifying the precondition of the action; and the
add-list $\Phi^+$ and delete-list $\Phi^-$ are sets of atoms that specify the effects of executing the action. A plan-library II consists of a set of plan-rules of the form $e(t):\varphi \vdash \psi \rightarrow G$, where $e$ is an n-ary event-goal symbol, $t = t_1, \ldots, t_n$ are terms, $\varphi$ and $\psi$ are formulas, and $G$ is a plan-body graph. The rule states that, if the context condition $\psi$ holds, $G$ is a ‘standard operating procedure’ for achieving the event-goal $e(t)$ or the goal-condition $\varphi$.

Plan-body graphs are built from the following set of user programs: actions; belief addition $+b$ adds the atom $b$ to $B$; belief removal $-b$ removes the atom $b$ from $B$; test $?\varphi$, where $\varphi$ is a formula, tests whether $\varphi$ holds in $B$; event-goal program or goal-condition program $\ev$, where $\ev \in \{e, 0\}$, specifies that $\ev$ needs to be achieved; wait $\WT(\varphi)$ waits until formula $\varphi$ holds in $B$; passive preserve $\PR_p(\ev, \varphi)$ specifies that $\ev$ needs to be achieved while monitoring $\varphi$ and aborted if $\ev$ fails or $\varphi$ does not hold; and active preserve $\PR_a(\ev, \varphi)$, which is similar to the former except that if condition $\varphi$ does not hold, the plan-body graph for $\ev$ is suspended and the re-achievement of $\varphi$ is attempted by posting the goal $\varphi$. If the goal succeeds and $\varphi$ is re-achieved then the active preserve is resumed, and otherwise it is aborted (if the goal fails) or re-suspended (if the goal succeeds but $\varphi$ is still not achieved). Since user program $\ev$ may generate subgoals, wait and preserve programs may also become ‘nested’ within one another, giving rise to potentially complex interactions. We use $\PR_a(\ev, \varphi)$ to denote either $\PR_a(\ev, \varphi)$ or $\PR_p(\ev, \varphi)$. Formally, a user program $P_u$ is a formula in the language defined by the grammar

$$P_u ::= \text{act} | ?\varphi | +b | -b | \ev | \WT(\varphi) | \PR_a(\ev, \varphi).$$

A plan-body graph is a directed bipartite graph representing a partially ordered set of user programs, where the set of nodes (i.e., node symbols) is split into state nodes, and transition nodes labelled with the user programs.

**Definition 1.** Let $P_u$ be the set of all user programs. A plan-body graph $G$ is a tuple $(S, T, E_{in}, E_{out}, L_0, N, s_0)$, where:

(i) $S$ is a set of state nodes and $s_0 \in S$ the initial node;
(ii) $T$ is a disjoint set of transition nodes; (iii) $E_{in} \subseteq S \times T$ is a set of input edges; (iv) $E_{out} \subseteq T \times S$ is a set of output edges; (v) function $L_0 : T \mapsto P_u$ represents the user programs labelling transition nodes; and (vi) $N \subseteq \text{Suc}(T)$ is a set of current nodes.

We stipulate that for any node $n \in S \cup T$ there exists a sequence $s_1 \cdot t_1 \cdot \ldots \cdot s_k-1 \cdot t_{k-1} \cdot s_k$, such that $s_0 = s_1$, node $n$ is in the sequence, and for each $i \in [1, k-1]$, $\langle s_i, t_i \rangle \in E_{in}$ and $\langle t_i, s_{i+1} \rangle \in E_{out}$.

The PRS deliberation cycle consists of three main steps: processing environment updates, e.g., new event-goals or belief updates; instantiating a plan-body graph to achieve an event-goal or goal condition, or executing a single step in an instantiated plan-body graph; and notifying plan-body graphs when conditions they are monitoring are established or violated. Execution of a plan-body graph starts in the initial node $(s_0)$, and progresses to one or more state/transition nodes $(N)$. Transition nodes initially represent user programs $(L_0)$ and evolve following the semantics in Section 3 until they reach the ‘empty program’ and terminate. The plan-body graph finishes executing (successfully) if no current state nodes $N$ of

$G$ lead anywhere. Formally, the plan-body graph is initial if $N = \{s_0\}$, and finished, denoted by $\text{finish}(G)$, if $N \subseteq S$, and for all $s \in S$ and $t \in T : (s, t) \not\in E_{in}$. All plan-body graphs occurring in a plan-library are initial.

**Example 1.** Consider an agent that has a goal to travel from its current location to a destination $d$ [Sardina and Padgham, 2011]. The goal can be achieved in various ways depending on the distance to the destination, represented by the following plan-rules (each omitted goal-condition $\varphi$ is $\top$):

$$\text{travel}(d) : At(x) \land \text{WalkDist}(x, d) \rightarrow G_{\text{walk}},$$

$$\text{travel}(d) : At(x) \land \exists y (\text{inCity}(x, y) \land \text{inCity}(y, d)) \rightarrow G_{\text{city}},$$

$$\text{travel}(d) : At(x) \land \neg \exists y (\text{inCity}(x, y) \land \text{inCity}(y, d)) \rightarrow G_{\text{far}}.$$

The first plan-rule refers to the plan-body graph $G_{\text{walk}}$ shown in Fig. 1.

**3 PRS Semantics**

Our semantics for PRS follows the approach adopted by the CAN [Winikoff et al., 2002] agent programming language, as defined in [Sardina and Padgham, 2011], where a transition relation on agent configurations is defined in terms of a set of derivation rules [Plotkin, 1981]. An agent configuration is a tuple $[\Pi, \Lambda, B, A, \Gamma]$ where $\Pi$ is a plan-library, $A$ is an action-library, $B$ is a belief base, $\Lambda$ is a sequence of executed actions, and $\Gamma$ is a set of intentions (as $\Pi$ and $\Lambda$ do not change during execution, we omit them from derivation rules). Agent configurations represent the execution state of a PRS agent, and intentions are the current states of full programs being pursued in order to achieve top-level event-goals. As in CAN, full programs extend the syntax of user programs to represent the current evolution of a user program, and may contain information used to decide the next transition, e.g., the plan-body graphs yet to be tried in achieving an event-goal.

Formally, a full program (or simply program) is a formula in the language defined by the grammar $P ::= \eta \mid \text{act} | ?\varphi | +b | -b | \ev | WT(\varphi) | \PR_a(P, \varphi) | \PR_p(P, \varphi) | \PR_a(\ev, \varphi)$

where $\eta$ (‘nil’) indicates that there is nothing left to execute; $\WT(\varphi)$ denotes either $\WT(\varphi)$ or $\WT(\varphi)$, where $\WT(\varphi)$ indicates that program $\WT(\varphi)$ has been adopted (i.e., its execution has started); $\ev : \psi_1 : G_1, \ldots, \psi_n : G_n$ represents the set of relevant plan-rules for achieving the event-goal or goal-condition $\ev$; $P \rightsquigarrow P'$ represents a suspended active preserve program $P'$ whose associated condition is being re-achieved by a recovery program $P$; $G \triangleright P$, where $P = \ev : \psi_1 : G_1, \ldots, \psi_n : G_n$, represents the default deliberation mechanism for ‘goal commitment’: achieve $\ev$ using an applicable plan-body graph $G$, but if that fails, try an

---

**Figure 1:** Plan-body graphs $G_{\text{walk}}$ and $G_{\text{pw}}$. $G_{\text{pw}}$ occurs in a plan-rule for the event-goal program $\text{P}_{\text{pw}}$ (prepare to walk) in $G_{\text{walk}}$. 

---

196
alternative applicable graph from those appearing in P; and 
\( \eta \triangleright P \) indicates that a plan-body graph for ev has succeeded. Note that full programs are more general than those yielded by our semantics.

We first give derivation rules for configurations of the form \([I, A, B, A, P]\), where \( P \) is a full program, i.e., for a single

single action. We give the derivation rules for actions, belief operations, goal and plan adoption, and goal commitment in Section 3.1, for advancing plan-body graphs in Section 3.2; for PRS-specific wait and preserve programs in Section 3.3; and for agents with configurations of the form \([I, A, B, A, P]\), i.e., multiple intentions, in Section 3.4.

3.1 Semantics for Actions, Belief Updates, Goal and Plan Adoption, and Goal Commitment

Fig. 2 shows the derivation rules for actions and belief operations. Rules add and del simply update \( B \), replacing programs +b and -b with \( \eta \) (\( A \) remains unchanged). The semantics for actions is given by rule A. The antecedent checks whether there is a relevant action-rule for act (i.e., one whose 'head' act' matches act under substitution \( \theta \)), and whether the action is applicable (i.e., its precondition holds in \( B \)).

Rule EV1 adopts the event-goal program \( e \) by creating the set \( \Delta \) of plan-rules relevant for the event, i.e., the rules in \( \Pi \) with event-goals matching \( e \) via a most general unifier \( \text{mgu}(e, e') \).

\[
\Delta = \{ \psi : G \theta | e' : \varphi; \psi \leftarrow G \in \Pi, \theta = \text{mgu}(e, e') \} \neq \emptyset \quad EV1
\]

Example 2. Processing an event-goal program of the form travel(Uni) using rule EV1 yields the following (full) program, encoding all relevant options for this event:

\[
\text{travel}(\text{Uni}) : \{ \psi_1 : \text{G.walk}, \psi_2 : \text{G.city}, \psi_3 : \text{G.\text{travel}} \}
\]

with \( \psi_1 = \text{At}(x) \land \text{WalkDist}(x, \text{Uni}) \);
\( \psi_2 = \text{At}(x) \land \exists y(\text{InCity}(x, y) \land \text{InCity}(\text{Uni}, y)) \); and \( \psi_3 = \text{At}(x) \land \neg \exists y(\text{InCity}(x, y) \land \text{InCity}(\text{Uni}, y)) \).

Similarly, rule EV2 adopts the goal-condition program \( \phi \) by creating the set \( \Delta \) of plan-rules in \( \Pi \) that can achieve \( \phi \).

\[
\Delta = \{ \psi : G \theta | e : \varphi; \psi \leftarrow G \in \Pi, \varphi \vdash \phi \} \neq \emptyset \quad EV2
\]

Rule Sel selects an applicable plan-rule for event-goal or goal-condition ev from the set of relevant rules, and schedules the associated plan-body graph for execution.

\[
\psi : G \in \Delta \quad \text{B} \vdash \psi \theta \quad \text{Sel}
\]

Rules for goal commitment are shown in Fig. 2. Rule \( \triangleright_{st} \) executes a single step in a program \( G \triangleright P \) if the plan-body graph \( G \) has neither failed nor finished (see Section 3.2). Rule \( \triangleright_{end} \) discards the alternative program \( P \) in a program \( \eta \triangleright P \) (where \( \eta \) here represents a completed graph). Finally, rule \( \triangleright_f \) schedules the alternative program \( P = \text{ev} : \{ \Delta \} \) for execution and executes a single step in it, provided \( G \) has failed and \( P \) has not, i.e., an applicable plan-rule exists for \( \text{ev} \).

3.2 Semantics for Plan-Body Graphs

We extend the definition of a plan-body graph in Definition 1 to a full plan-body graph, representing the current ‘state’ in the evolution of an initial plan-body graph. A full plan-body graph is of the form \( G = (S, T, E_{in}, E_{out}, L_0, L_c, N, s_0) \) where \( L_0 : T \rightarrow P \) is a function that maps each transition node \( t \in T \) to a full program \( P \in \text{P} \), which represents the current form of the possibly evolved (initial) user program \( L_0(t) \). We use the following auxiliary definitions: \( \text{bef}(t) = \{ s | (s, t) \in E_{in} \} \) are the input state nodes of \( t \); \( \text{af}(t) = \{ s | (t, s) \in E_{out} \} \) are its output state nodes; and the update to \( L_c \) with a (new) program \( P \) for \( t \) is

\[
\text{UPD}(L_c, t, P) = \{ L_c \{ \{ t, L_c(t) \} \} \cup \{ t, P \} \}.
\]

The first rules state that a transition node \( t \), in the case where it is not initially associated with a test condition, becomes active if it is not already active but all of its input states are. Becoming active includes (re-)initialising \( t \) to correspond to its user program. This is done in case \( t \) is part of a cycle.

\[
t \in T \quad \text{t} \not\in N \quad \text{bef}(t) \subseteq N \quad L_0(t) \neq \emptyset \quad \text{G}^{P}\text{start}
\]

with \( G' = (S, T, E_{in}, E_{out}, L_0, L_c', N', s_0) \);
\( L_c' = \text{UPD}(L_c, t, L_0(t)) \);
\( N' = (N \setminus \text{bef}(t)) \cup \{ t \} \).

Once a transition node \( t \) is active, it can perform a single execution step in its associated (current) program \( L_c(t) \).

\[
t \in N \quad [B, A, L_c(t)] \rightarrow [B', A', P] \quad \text{G}^{P}\text{step}
\]

\( G' = (S, T, E_{in}, E_{out}, L_0, L_c', N', s_0) \);
\( L_c' = \text{UPD}(L_c, t, P) \).

If a transition node’s program has finished execution, the node becomes inactive and its outgoing nodes become active.

\[
t \in N \quad L_0(t) = \emptyset \quad \text{G}^{P}\text{end}
\]

\( G' = (S, T, E_{in}, E_{out}, L_0, L_c, N', s_0) \);
\( N' = (N \setminus \{ t \}) \cup \text{af}(t) \).

\text{1We treat the functions } L_0 \text{ and } L_c \text{ as relations, i.e., as sets of ordered pairs of the form } (t, P).
Example 3. Suppose that the agent believes it is currently at home, which is walking distance to the university. In this case, the Sel rule transforms the set of relevant options represented by the program in Example 2 into the program $G_{walk} \triangleright travel(Univ) : \{\psi_2 : G_{city}, \psi_1 : G_{far}\}$, i.e., the agent selects the $G_{walk}$ plan-body graph while keeping the other graphs as backup alternatives, represented by the right-hand side of the $\triangleright$ operator. When the agent starts executing the $G_{walk}$ graph, the rule $G_{start}^P$ removes $s_0$ from $N$ and adds the transition node associated with subgoal $\text{lpw}$. This is then executed using rule $G_{stp}$, whose antecedent uses rule $E_{V_1}$ to resolve the subgoal.

If a transition node is initially associated with a test condition, then the node becomes active only if the condition holds in the current belief base. The chosen transition node also becomes inactive at the same execution step, as once the condition is tested there is nothing left to execute. Under this semantics, PRS allows choices in execution within the graph: either a non-deterministic choice when multiple transition nodes, with non ‘mutex’ test programs, exit the same state node, or deterministic choices induced by such tests.

$$t \in T \quad \text{bef}(t) \subseteq N \quad L_0(t) = \psi \quad B \models \phi$$

$$[B, A, G] \rightarrow [B, A, G']$$

$G' = \langle S, T, E_{in}, E_{out}, L_0', L_e', N', s_0\rangle$;

$L_e' = \text{upd}(L_e, t, \eta)$;

$N' = (N \setminus \text{bef}(t)) \cup \text{aft}(t)$.

The case where $B \models \phi$ holds represents failure, i.e., the inability to execute a step in $t$.

Finally, if a plan-body graph has finished (Section 2), rule $G_{end}$ replaces it with program $\eta$.

$$\text{fin}(G)$$

$$[B, A, G] \rightarrow [B, A, \eta]$$

$G_{end}$

Example 4. Consider the evolution $G_{pw} \triangleright pw \cdot \langle \Delta_{pw} \rangle$ of subgoal $\text{lpw}$. Achieving the former using the graph in Figure 1 involves the parallel execution of the programs $P_{1\text{pw}}$ and $P_{2\text{pw}}$. This ‘split’ is represented by the outgoing edges from the transition node associated with the ‘?’ user program, and results in state nodes $s_4$ and $s_5$ being added to $N$, using rule $G_{stp}$. Note that for the transition node associated with $\text{lev}$ to become active, both $P_{1\text{pw}}$ and $P_{2\text{pw}}$ must complete execution, and transition to $s_6$ and $s_7$, respectively.

3.3 Semantics for Wait and Preserve Programs

We now give a semantics for wait and preserve programs of the form $\text{WT}(\phi), \text{PR}_b(P, \phi)$, and $\text{PR}_a(P, \phi)$, and suspended active preserve programs of the form $P_1 \leadsto \text{PR}_a(P_2, \phi)$, where $P_1$ is the recovery program. In all cases, we assume that condition $\phi$ is ground when the program is adopted.

Rule $W_{adopt}$ adopts a wait program, i.e., changes its form to indicate that condition $\phi$ is now being monitored. Rule $W$ specifies that the wait for $\phi$ should continue if $\phi$ does not hold in the belief base. Finally, rule $W_{end}$ specifies that the wait should end if $\phi$ does hold. In all cases, $C = [B, A, \text{WT}(\phi)]$.

$$[B, A, \text{WT}(\phi)] \rightarrow C$$

$$[B, A] \rightarrow W_{adopt}$$

$$C \rightarrow W \quad B \not\models \phi$$

The first set of derivation rules for preserve programs apply to both passive and active preserves. Rule $P_{\text{radopt}}$ specifies the adoption of a preserve program, i.e., the adoption of its event-goal or goal-condition program $\text{lev}$. Rule $P_{\text{stp}}$ executes a single step in a preserve program if $\phi$ is not violated, and $P_{\text{succe}}$ removes a completed preserve program.

$$[B, A, \text{lev}] \rightarrow [B, A, \text{ev} : \langle \Delta \rangle]$$

$$[B, A, \text{PR}_a(\text{lev}, \phi)] \rightarrow [B, A, \text{PR}_b(\text{ev} : \langle \Delta \rangle, \phi)]$$

$$P \not\models \text{lev} \quad B \models \phi \rightarrow [B, A, P] \rightarrow [B', A', P']$$

$$[B, A, \text{PR}_b(P, \phi)] \rightarrow [B', A', \text{PR}_a(P, \phi)]$$

Rule $P_{\text{fail}}$ specifies that the passive preserve fails if $\phi$ is violated or $P$ is blocked.

$$P \not\in \{\text{lev}, \text{ev}\}$$

$$[B, A, \text{PR}_a(P, \phi)] \rightarrow [B, A, \text{false}]$$

Rules $AP_{\text{fail}}$ to $AP_{\text{stp}2}$ operationalise adopted active and suspended preserve programs. We define three special multisets.$^2$ Given an expression $\mathcal{E}$ that is either a program or plan-body graph $G = \langle S, T, E_{in}, E_{out}, L_0, L_e, N, s_0\rangle$, we define a ‘path’ of ‘nested’ (adopted) preserve and wait programs as any element in the multiset $\mathcal{T}(\mathcal{E})$, defined as $\mathcal{T}(\mathcal{E}) = \{\mathcal{E} \cdot \tau : \tau \in T(\mathcal{E})\}$. The conditions $\mathcal{E}$ are given by the following cases:

1. $\mathcal{E} = \mathcal{P} \leadsto \mathcal{P}'$, then

$$\mathcal{T}(\mathcal{E}) = \{\mathcal{E} \cdot \tau : \tau \in T(\mathcal{E})\}$$

2. $\mathcal{E} = \mathcal{P} \leadsto \mathcal{P}'$, then

$$\mathcal{T}(\mathcal{E}) = \{\mathcal{E} \cdot \tau : \tau \in T(\mathcal{E})\}$$

3. $\mathcal{E} = \mathcal{P} \leadsto \mathcal{P}'$, then

$$\mathcal{T}(\mathcal{E}) = \{\mathcal{E} \cdot \tau : \tau \in T(\mathcal{E})\}$$

When $\mathcal{E} = G$ we take the multiset union of the sequences corresponding to transition nodes in $G$ that are executed in parallel. The first element in such a sequence is a ‘most abstract’ preserve or wait program occurring in $\mathcal{E}$, and the last element is a ‘most deeply’ nested preserve or wait program occurring in $\mathcal{E}$. We use $\mathcal{S}_c(\mathcal{E})$ and $\mathcal{P}_c(\mathcal{E})$ to denote the multisets of all the elements in all the sequences in $\mathcal{T}(\mathcal{E})$ that are, and are not, of the form $P \leadsto P'$, respectively; i.e., any element in $\mathcal{S}_c(\mathcal{E})$ is a suspended (adopted) active preserve program, and any element in $\mathcal{P}_c(\mathcal{E})$ is an adopted wait, passive preserve, or active preserve program that is not suspended. We use $\mathcal{T}(\mathcal{E})$ to check whether a wait/preserve program in some ‘path’ in $\mathcal{E}$ may be ‘pruned’ by a more abstract preserve program in the path, and we use $\mathcal{S}_c(\mathcal{E})$ and $\mathcal{P}_c(\mathcal{E})$ to count the number of suspended and unsuspended programs occurring in $\mathcal{E}$.

Example 5. Suppose $P_{1\text{pw}}$ and $P_{2\text{pw}}$ have evolved to, respectively, adopted programs $\text{WT}(\phi)$ and $\text{PR}_b(P, \phi')$, with $P = e : \langle \Delta \rangle$ (i.e., one parallel branch waits for a condition $\phi$ while the other branch tries to achieve an event-goal program $\text{lev}$ while preserving $\phi'$). Then, $\mathcal{T}(G_{walk}) = \{\text{WT}(\phi), \text{PR}_b(P, \phi')\}$. Moreover, if $P$ evolves to $P' = G_e \triangleright e : \langle \Delta' \rangle$, then $\mathcal{T}(G_{walk}) = \{\text{WT}(\phi), \text{PR}_b(P', \phi') \cdot \text{WT}(\phi'')\}$. $^2$Adapted from [Sardina and Padgham, 2011].
Rule $APr_{fail}$ specifies that the adopted active preserve program $Pr_a(P, \phi)$ fails if $P$ fails and the monitored condition $\phi$ is not violated. If $\phi$ is violated, $APr_{sus}$ suspends the preserve program and attempts to re-establish $\phi$ using the recovery (goal-condition) program $!\phi$. Rules $APr_{fail}$ and $APr_{sus}$ specify, respectively, that a suspended preserve program fails if the recovery program fails, and is resumed if the recovery program completes.

$$P \not\in \eta, \forall \{\text{lev} \} \quad B = \phi \quad [B, A, P] \not\rightarrow [B, A, \text{false}] \quad APr_{fail}$$

$$P \not\in \eta, \forall \{\text{lev} \} \quad B \neq \phi \quad [B, A, Pr_a(P, \phi)] \rightarrow [B, A, !\phi] \quad APr_{sus}$$

Finally, rules $APr_{sus1}^{step1}$ and $APr_{sus2}^{step2}$ define the execution of recovery programs and suspended (active preserve) programs. Rule $APr_{sus2}^{step2}$ executes a single ‘cleanup’ or ‘notification’ step in the suspended program. Such an execution step amounts to a program $P_1$ evolving to a program $P_2$ that has fewer suspended programs, or fewer unsuspended preserve or wait programs, e.g., due to a failed passive preserve that had a ‘nested’ wait program. The relation $P \prec P'$ is defined for programs $P$ and $P'$ as: $P \prec P' \iff |P_r(P)| < |P_r(P')| \lor |S_r(P)| = |S_r(P')|$.

$$[B, A, P_1] \rightarrow [B', A', P'_1] \quad APr_{sus1}^{step1}$$

$$[B, A, P_2] \rightarrow [B', A', P'_2] \quad APr_{sus2}^{step2}$$

Note that rule $APr_{sus2}^{step2}$ implies that if an active preserve is suspended, all of the (possibly adopted) ‘nested’ programs occurring in $P_2$ are ‘implicitly suspended’, i.e., they can only perform ‘cleanup’ or ‘notification’ steps, so we are guaranteed to terminate.

**Proposition 1.** Any sequence of configurations $[B_1, A_1, P_1]$, $\ldots$, $[B_n, A_n, P_n]$ is finite if for all $i \in [1, n - 1]$, we have that $P_{i+1} \prec P_i$ and $[B_i, A_i, P_i] \rightarrow [B_{i+1}, A_{i+1}, P_{i+1}]$.

**Proof.** Since each such execution step from $P_i$ to $P_{i+1}$ must yield fewer wait programs, preserve programs, and/or programs of the form $P \rightarrow Pr_a(P, \phi)$, it is sufficient to consider whether a ‘switch’ of the latter to resume $Pr_a(P', \phi)$ can lead to it (possibly with an ‘evolved’ $P'$) being suspended again, and whether this can continue indefinitely.

For $Pr_a(P, \phi)$ to resume, we must have $P = \eta$, and then if $\phi$ still does not hold, the program will indeed evolve to $!\phi \leadsto Pr_a(P', \phi)$. Moreover, recovery program $!\phi$ does have at least one relevant plan-rule, as it was once able to evolve to $\eta$ (recall that $\phi$ has been ground from the moment its associated active preserve program was adopted). However, the only possible execution step on $!\phi$ cannot reduce the number of aforementioned programs.

We use the notation $C \rightarrow C'$ to denote that there exists a non-empty sequence of configurations from $C$ to $C'$.

### 3.4 Agent-Level (‘Top-Level’) Semantics

We now give the derivation rules for the top-level execution of an agent program. Transitions between agent configurations are defined by the derivation rules in Fig. 3: an expression $C \rightarrow C'$ denotes a transition of type $t \in \{prs, event, cond, int\}$.

Rule $A_{prs}$ is the top-level rule, and represents the PRS deliberation cycle. A single PRS-type execution step comprises three things: progressing an intention by one step, or removing a completed intention (i.e., $P = \eta$) or a failed one (using rules $A_{int}$ or $A_{rem}$, respectively); processing newly observed event-goals (using rule $A_{ev}$), i.e., creating an intention for each new event-goal that is observed from the (external) environment; and finally, performing all the necessary ‘notification’ and ‘cleanup’ steps on wait, preserve, suspended, and recovery programs (using rule $A_{cond}$) to leave the agent in a ‘sound’ or ‘stable’ configuration. More specifically, $A_{cond}$ takes a single step in an intention if the step will yield an intention with fewer suspended programs, or fewer unsuspended (adopted) preserve or wait programs.

### 4 Properties of the Semantics

We now prove key properties of the semantics and show the greater expressivity of our PRS fragment compared to CAN. In what follows, we use $[B, A, P] \rightarrow$ as an abbreviation for $\exists B', A', P' : [B, A, P] \rightarrow [B', A', P']$.

**Theorem 1.** Let $C_1$ and $C_2$ be as above. Then, $C_2$ is sound.

**Proof Sketch.** Observe from the antecedent of derivation rule $A_{prs}$ that only one step of type INT is performed on $C_1$. Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)
followed by one of type EVENT, and zero or more of type COND. Assume the theorem does not hold because there is a passive preserve \( \text{PR}_P(P, \phi) \in \mathbb{P}_r(\Gamma_2) \), for some \( P \), such that \( B_2 \not\models \phi \) or \( [B_2, A_2, P] \not\models \phi \). Then, since \( \text{PR}_P(P, \phi) \) is an adopted program appearing in some intention \( P_1 \in \Gamma_2 \), either rule \( \text{PR}_{\text{fail}} \) or \( \text{PR}_{\text{sus}} \) can be applied to configuration \( [B_2, A_2, \text{PR}_P(P, \phi)] \) to yield an intention \( P' \). Since \( P' \models P_1 \), rule \( \text{Acond} \) will be applied (possibly multiple times) to \( P' \) until \( \text{PR}_P(P', \phi) \not\in \mathbb{P}_r(\Gamma_2) \) for all \( P' \), which contradicts our assumption. Assume instead that the theorem does not hold because there is a program \( P \rightarrow P' \in \mathbb{S}_r(\Gamma_2) \) such that \( [B_2, A_2, P] \not\models \phi \). Then, either rule \( \text{APR}_{\text{fail}} \) or \( \text{APR}_{\text{sus}} \) will be applied to configuration \( [B_2, A_2, P] \rightarrow P' \) or its 'evolution', again resulting in a contradiction. The remaining cases are proved similarly.

Theorem 2. Let \( C_1 \) and \( C_2 \) be as before. For each \( \mathbb{W}(\phi) \in \mathbb{P}_r(\Gamma_1) \) such that \( \mathbb{W}(\phi) \not\in \mathbb{P}_r(\Gamma_2) \), we have that \( B_2 \not\models \phi \), or \( \mathbb{W}(\phi) \) is pruned between \( C_1 \) and \( C_2 \).

Proof Sketch. Let \( \mathbb{W}(\phi) \in \mathbb{P}_r(\Gamma_1) \) be a program such that \( \mathbb{W}(\phi) \not\in \mathbb{P}_r(\Gamma_2) \). Let \( t \) be a transition node currently labelled with \( \mathbb{W}(\phi) \), where \( T = \{ t, \ldots \} \) and \( N = \{ \ldots \} \) are the transition nodes and current nodes in a (‘partially executed’) plan-body graph \( G \). We consider the case where if \( B_2 \not\models \phi \), then \( \mathbb{W}(\phi) \) must have been pruned between \( C_1 \) and \( C_2 \), because no other derivation rules can ‘remove’ \( \mathbb{W}(\phi) \). First, rule \( G_{\text{end}} \) (which, if applicable, ‘removes’ \( G \)) requires \( \text{fin}(G) \) to hold, which cannot be the case as \( t \in N \); for the same reason, rule \( G_{\text{start}} \) cannot ‘reset’ \( \mathbb{W}(\phi) \). Second, the antecedent of rule \( \triangleright \phi \) requires that \( [B, A, G] \not\models \phi \) (for some \( A \) and \( B \in \{ B_1, B_2 \} \)), which cannot hold as we can take a step on \( \mathbb{W}(\phi) \) via rule \( W \). Similarly, if \( P \) is an intention in which \( G \) occurs, the antecedent of agent-level rule \( \text{Arc} \) cannot hold (i.e., \( [B_1, A, P] \not\models \phi \) cannot hold for any \( A \)). Then, it follows that \( \mathbb{W}(\phi) \) is pruned between \( C_1 \) and \( C_2 \).

Theorem 3 states that an adopted preserve program is only ‘removed’ if: its condition becomes violated; its associated program becomes blocked; or the preserve program is pruned.

**Theorem 3.** If \( C_1 \) and \( C_2 \) are as before, and \( \text{PR}_x(P, \phi) \in \mathbb{P}_r(\Gamma_1) \) is a program s.t. \( \text{PR}_x(P_2, \phi) \not\in \mathbb{P}_r(\Gamma_2) \) for any \( P_2 \):

1. \( B_2 \not\models \phi \) or \( [B_2, A_2, P] \rightarrow [B_2, A_2, P'] \not\models \phi \);
2. \( [B_2, A_1, P] \rightarrow [B_2, A_2, P'] \rightarrow [B_3, A_3, P'' \not\models \phi] \); or
3. \( \text{PR}_x(P, \phi) \) is pruned between \( C_1 \) and \( C_2 \).

Proof Sketch. We show that if two of the conditions above do not hold, then the third will. For example, if (2) and (3) do not hold, then \( \text{PR}_x(P, \phi) \) must have been ‘removed’ by rule \( \text{PR}_{\text{fail}} \) or suspended by \( \text{APR}_{\text{sus}} \), whose antecedents entail (1). Similarly, if (1) and (2) do not hold, then \( \text{PR}_x(P, \phi) \) must have been pruned between \( C_1 \) and \( C_2 \) for the reasons given in the proof of Theorem 2.

Theorem 4 states that a suspended preserve program is only ‘removed’ during a PRS step if: the recovery program completes and the preserve program’s condition is re-established; the recovery program becomes blocked; or both are pruned.

**Theorem 4.** If \( C_1 \) and \( C_2 \) are as before, and \( P_1 \rightarrow \text{PR}_a(P_2, \phi) \in \mathbb{S}_r(\Gamma_1) \) is a program s.t. \( P_1 \rightarrow \text{PR}_a(P', \phi) \not\in \mathbb{S}_r(\Gamma_2) \) for any \( P' \) and \( P' \):

1. \( [B_2, A_1, P] \rightarrow [B_2, A_1, \eta] \) and \( B_1 \models \phi \);
2. \( [B_2, A_2, P_1] \rightarrow [B_2, A_2, P' \not\models \phi] \); or
3. \( [B_1, A_1, P] \rightarrow [B_2, A_2, P'] \rightarrow [B_3, A_3, P'' \not\models \phi] \);
4. or \( P_1 \rightarrow \text{PR}_a(P_2, \phi) \) is pruned between \( C_1 \) and \( C_2 \).

The proof of Theorem 4 is similar to that of Theorem 3.

Theorems 3 and 4 considered the case where all occurrences of preserve programs associated with the same condition \( \phi \), e.g., in multiple intentions in \( \Gamma_1 \), are removed in a PRS step. There is also the case where only some of them are removed in such a step. It is not difficult to develop theorems for this case, though we would need additional formal machinery. Similarly, we can show that active preserve programs are suspended and resumed for the right reasons. For example, with a minor extension to Theorem 3, we can show that if \( \text{PR}_a(P, \phi) \) becomes suspended (i.e., \( B \not\models \text{PR}_a(P, \phi) \in \mathbb{S}_r(\Gamma_2) \)), then \( B_2 \not\models \phi \) or \( (B_2, A_2, \eta) \not\rightarrow \phi \).

In the remainder of this section, we characterise the relative expressivity of our PRS fragment and the CAN formalism as defined in [Sardina and Padgham, 2011]. A CAN plan-rule is of the form \( e : \psi \rightarrow P \), where \( e \) is an event-goal, \( \psi \) is a context condition, and \( P \) is a plan-body, i.e., a formula in the language defined by the grammar \( P := \)

\[
\text{act} \mid ? \phi \mid + b \mid - b \mid \text{le} \mid P_1 \mid P_2 \mid P_1 \parallel P_2 \mid \text{Goal}(\phi_\alpha, P', \phi_\alpha),
\]

where \( P_1; P_2 \) is sequential composition, \( P_1 \parallel P_2 \) is parallel composition, and \( \text{Goal}(\phi_\alpha, P', \phi_\alpha) \) is a declarative goal specifying that formula \( \phi_\alpha \) (the goal) should be achieved using program \( P' \), failing if \( \phi_\alpha \) becomes true. The remaining programs are defined as for PRS user programs.

To state our results, we need the notions of execution traces and solutions. We define these only for PRS as the definitions for CAN are analogous.

**Definition 2.** An execution trace of an agent configuration \( C = [A, P, \Pi, B, A, \Gamma] \) is a finite sequence of agent configurations \( C_1 \cdots C_n \) such that \( C = C_1 \) and \( C_i \dashv \vdash C_{i+1} \) for all

---

3The definitions of \( \mathbb{P}_r(\mathcal{E}) \) and \( \mathbb{S}_r(\mathcal{E}) \), for some expression \( \mathcal{E} \), can be generalised similarly.
Theorem 5. If \( \Pi^- \) is a CAN library and \( \Lambda \) an action-library, there exists a PRS library \( \Pi^+ \), s.t. for any event-goal \( i.e. \) and beliefs \( B \): \( \text{sol}(\Lambda, \Pi^- \cdot \mathcal{B}, \{i.e.\}) \neq \text{sol}(\Lambda, \Pi^+ \cdot \mathcal{B}, \{i.e.\}) \).

Proof Sketch. Given a CAN plan-rule \( e \cdot \psi \rightarrow P \in \Pi^- \), the first step is to obtain the corresponding PRS plan-rule \( e : \top ; \psi \rightarrow G \). We define three functions: \( g(P, n)_{in} \), \( g(P, n)_{out} \), and \( g(P, n)_{L} \), where \( n \in \mathbb{N} \). When \( n = 1 \), these functions represent respectively the elements \( E_{in}, E_{out} \), and \( L_0 \) in \( G \). If \( P = \text{act} \), we define \( g(P, n)_{in} = \{n \cdot s, n \cdot t\} \); \( g(P, n)_{out} = \{n \cdot t, n \cdot s\} \); and \( g(P, n)_{L} = \{n \cdot t, P\} \), where \( s, s' \) and \( t \) are unique symbols (and thus, \( n \cdot t \), for example, is a string). We also define \( g(P, n)_{start} = n \cdot s \) and \( g(P, n)_{end} = n \cdot s' \). Intuitively, uniquely identifies the PRS plan-body ‘subgraph’ corresponding to \( P \). If \( P = P_1 \cdot P_2 \) is the sequential composition of CAN programs \( P_1 \) and \( P_2 \),

\[
\begin{align*}
g(P, n)_{out} &= g(P_1, n \cdot 1)_{out} \cup g(P_2, n \cdot 2)_{out} \cup \{n \cdot t'' \cdot s''\}; \\
g(P, n)_{in} &= g(P_1, n \cdot 1)_{in} \cup g(P_2, n \cdot 2)_{in} \cup \{n \cdot s, n \cdot t\}; \\
g(P, n)_{L} &= g(P_1, n \cdot 1)_{L} \cup g(P_2, n \cdot 2)_{L} \cup \{n \cdot t', \top\}; \\
g(P, n)_{end} &= n \cdot s' \cdot t' \cdot t'' \cdot s''
\end{align*}
\]

where transition node \( n \cdot t' \) ‘connects’ the subgraphs corresponding to \( P_1 \) and \( P_2 \). As before, we define \( g(P, n)_{start} = n \cdot s \) and \( g(P, n)_{end} = n \cdot s'' \), and \( s, s', t, t' \) and \( t'' \) are unique symbols. We similarly define the subgraph corresponding to a CAN parallel composition \( P_1 \parallel P_2 \), test program, etc.

We then show by induction on the structures of \( P \) and \( G \) above that their traces yield the same solutions. For example, the derivation rules for CAN’s sequential composition can be simulated by repeatedly applying the \( G_{start}^P \), \( G_{stp}^P \) and \( G_{out}^P \) rules of PRS, and vice versa.

Theorem 6 states that the converse does not hold: even if we ignore programs that have no counterparts in CAN, PRS plan-libraries cannot be ‘directly mapped’ to CAN libraries. This result follows from a similar one in de Silva, 2017 which showed that the ‘plan-body’ representation used in HTN planning allows a more fine-grained interleaving of steps than do CAN plan-bodies: a CAN plan-body must specify steps in a ‘series-parallel’ manner, whereas HTN ‘plan-bodies’ (and PRS plan-body graphs) do not.

In what follows, \( \Pi^- \) denotes PRS plan-libraries that do not mention goal-condition, wait, nor preserve programs, and \( \Pi_\phi \in \text{CAN}(\Pi^-) \) denotes a directly mapped CAN library, i.e., one obtained from \( \Pi^- \) by ignoring the goal-condition \( \phi \) in each plan-rule, and replacing each graph \( G \) appearing in \( \Pi^- \) with a CAN plan-body \( P \) such that the multisets of (user) programs occurring in \( G \) and \( P \) are the same.

Theorem 6. There exists a PRS library \( \Pi^-_\phi \), an action-library \( \Lambda \), and event-goal \( i.e. \), s.t. for any CAN library \( \Pi_\phi \in \text{CAN}(\Pi^-_\phi) \) and beliefs \( B \): \( \text{sol}(\Lambda, \Pi^-_\phi \cdot \mathcal{B}, \{i.e.\}) \neq \text{sol}(\Lambda, \Pi_\phi \cdot \mathcal{B}, \{i.e.\}) \).

Proof Sketch. We translate an example HTN ‘plan-body’ provided in de Silva, 2017 into a PRS plan-body graph. First, we create the PRS plan-rule \( e_{top} : \top ; \psi \rightarrow G \), in particular by taking the following input and output edges for \( G \):

\[
\begin{align*}
E_{in} &= \{(s_1, t_1), (s_2, t_2), (s_3, t_3), (s_4, t_4), \\
& (s_5, t_5), (s_5, t_6), (s_7, t_6), (s_4, t_4)\}; \\
E_{out} &= \{(t_1, s_2), (t_1, s_3), (t_2, s_1), (t_3, s_5), \\
& (t_3, s_6), (t_4, s_7), (t_5, s_8), (t_6, s_9)\}.
\end{align*}
\]

Second, we set the initial program \( L_0(t_i) \) for \( l_i \in \{t_1, \ldots, t_6\} \) to a different event-goal \( e_{i_{top}} \), each of which is associated with a single plan-body graph representing the sequence of unique actions \( a_1^t \cdots a_{t_6}^t \). Finally, we show that the following sequence of actions:

\[
\begin{align*}
a_1^t \cdot a_2^t \cdot a_3^t \cdot a_4^t \cdot a_5^t \cdot a_6^t \cdot a_1^t \cdot a_2^t \cdot a_3^t \cdot a_4^t \cdot a_5^t \cdot a_6^t \cdot a_1^t \cdot a_2^t \\
\end{align*}
\]

cannot be produced by any CAN trace of program \( e_{top} \), relative to any CAN plan-library that is directly mapped from the above set of PRS plan-rules: e.g., the CAN body \( e_{i_1} \cdot e_{i_2} \parallel e_{i_3} \parallel e_{i_4} \parallel e_{i_5} \); \( e_{i_2} \parallel e_{i_6} \) does not allow \( a_6^t \) to follow \( a_5^t \).
Acknowledgements

Lavindra is grateful to Félix Ingrand for useful discussions about OpenPRS while Lavindra was a researcher on the GOAC/MARAE projects (2009-2011) at LAAS-CNRS. Felipe thanks CNPq for partial financial support under its PQ fellowship, grant number 305969/2016-1.

References


