An Operational Semantics for True Concurrency in BDI Agent Systems

Lavindra de Silva
Department of Engineering
University of Cambridge, UK
Lavindra.deSilva@eng.cam.ac.uk

Abstract

Agent programming languages have proved useful for formally modelling implemented systems such as PRS and JACK, and for reasoning about their behaviour. Over the past decades, many agent programming languages and extensions have been developed. A key feature in some of them is their support for the specification of 'concurrent' actions and programs. However, their notion of concurrency is still limited, as it amounts to a nondeterministic choice between (sequential) action interleavings. Thus, the notion does not represent 'true concurrency', which can more naturally exploit multi-core computers and multi-robot manufacturing cells. This paper provides a true concurrency operational semantics for a BDI agent programming language, allowing actions to overlap in execution. We prove key properties of the semantics, relating to true concurrency and to its link with interleaving.

1 Introduction

Agent programming languages have proved useful for formally modelling implemented systems such as Jason [3], PRS [17], and JACK [8], and for reasoning about their behaviour. Over the past decades, many agent programming languages and extensions have emerged, e.g. [25, 19, 10, 28, 22, 26, 9, 12, 11]. A key feature in some of them is their support for the specification of 'concurrent' actions and programs. However, while their notion of concurrency is useful in some applications, it is still limited as it amounts to a nondeterministic choice between (sequential) action interleavings. Thus, the notion does not represent 'true concurrency', which can more naturally exploit multi-core computers and multi-robot manufacturing cells, e.g. a cell comprising two robot arms that work on a part simultaneously. This paper provides a true concurrency operational semantics for a BDI agent programming language, allowing actions to overlap in execution.

From the implemented BDI agent systems, there are some that support some form of true concurrency, e.g. SPARK [22], JAM [20], BDI4JADE [23], PRS, and JACK.[1] JACK gives four (programmer selectable) definitions for the success of a concurrent program: *(1)* it immediately succeeds on the successful termination of a branch; *(2)* it allows all branches to terminate but only succeeds if at least one of them succeeds; *(3)* it succeeds iff all branches terminate successfully, and immediately fails on the failure of a branch; or *(4)* it allows all branches to terminate but only succeeds if all branches succeed. We follow this last definition.

There is also related work in concurrent distributed systems, particularly three strands of work that define behaviour in terms of transitions between configurations, using a structural operational semantics [24]. In the first strand [6, 7], every transition is labelled with a representation of its 'proof', i.e., information comprising the inference rules that were used when deriving the transition. This information enables the extraction of a transition labelled with a partially ordered multiset (pomset) of actions, representing the sequential and concurrent actions performed. A similar transition is extracted in [14, 15], but from information stored in configurations rather than transition labels. In the third strand [4, 5], transitions are labelled with 'composite actions', which represent pomsets. This avoids the need to extract a pomset-labelled transition 'a posteriori' from a sequence of transitions, as done above.

Our work was inspired by the last two strands: our transitions represent composite actions, which are stored in configurations. However, unlike the above strands, we account for concerns specific to agent programming languages, e.g. goal refinement, plan failure, and constructs beyond actions. Our proposal, called Concurrent CAN (CCAN), is based on the work in [27], which refines and extends the CAN agent programming language [28]. We discuss the syntax of CCAN (sec. 2) and its semantics (secs. 3 and 4), and we prove key properties of CCAN, in relation to true concurrency and to its link with interleaving (sec. 5); e.g., we show that if a concurrent program's branches are interleaved as in CAN, Jason, etc., any resulting behaviour can also be produced by the concurrent program. We then conclude and discuss future work (sec. 6).

[1] There are also implemented BDI systems that do not support true concurrency but support action interleaving, e.g. Jason.

2 CCAN Syntax

In this paper, we use a first-order language with a vocabulary comprising mutually disjoint and infinite sets of variable, function, predicate, event-goal, and action symbols.

Like [27], we define a CCAN agent by a plan-library Π, an action-library Λ, and a belief base $\mathcal{B}$. A *belief base* is a set of ground atoms, and an *action-library* is a set of *action-rules* representing actions the agent can perform. An *action*, denoted by a, is of the form $act(\vec{t})$, where act is an n-ary action symbol representing a function that may affect the external environment, and $\vec{t} = t_1, \ldots, t_n$ is a list of (possibly ground) terms. An *action-rule*, as in STRIPS, is of the form $act(\vec{v}){:}\psi \leftarrow \Phi^+; \Phi^-$, where $\vec{v} = v_1, \ldots, v_n$ is a list of distinct variables; ψ, the *pre-condition*, is a formula; and Φ^+ and Φ^-, respectively the *add-list* and *delete-list*, are each a set of atoms representing the action's effects. Any variable occurring in ψ, Φ^+, or Φ^- also occurs in $\vec{v}$. For any action a that occurs in Π, there is exactly one action-rule $a' : \psi \leftarrow \Phi^+; \Phi^- \in \Lambda$ such that $a = a'\theta$ for some substitution θ; we define $pre(a, \Lambda) = \psi\theta$ and $\mathit{eff}(a, \Lambda) = \langle \Phi^+\theta, \Phi^-\theta \rangle$.

A *plan-library* Π is a set of *plan-rules* of the form $ev(\vec{t}){:}\psi \leftarrow P_b$, where $ev(\vec{t})$, denoted by e, is an *event-goal* and ev is an n-ary event-goal symbol; ψ, the *context condition*, is a formula; and P_b, the *plan-body* is a 'standard operating procedure' for achieving e when ψ holds in $\mathcal{B}$. Formally, a *plan-body* is a formula in the language defined by the grammar $P_b ::=$

$$a \mid +b \mid -b \mid ?\phi \mid !e \mid P_b^1 \,;P_b^2 \mid P_b^1 \text{“|”} P_b^2 \mid P_b^1 \parallel P_b^2$$

where $+b$ is a *belief addition*, which adds the atom b to $\mathcal{B}$; $-b$ is a *belief removal*, which removes b from $\mathcal{B}$; $?\phi$ is a *test condition*, which tests whether formula ϕ holds in $\mathcal{B}$; $!e$ is an *event-goal program*, which states that e needs to be achieved; and $P_b^1 \,;P_b^2$ is a *sequential program*, which states that P_b^1 must be executed before P_b^2. Finally, $P_b^1 | P_b^2$ is an *interleaved program*, which allows the resulting actions to be interleaved (but not overlapped);[2] and $P_b^1 \parallel P_b^2$ is a (truly) *concurrent program*, which allows the resulting actions to be interleaved and/or overlapped, as described in [1]. In the sequel, we use the terms 'concurrency' and 'concurrent' only when referring to the latter type of program or its execution.

We impose two constraints relating to concurrency. First, we limit how a concurrent program $P_b^1 \parallel P_b^2$ is interleaved with another program: no other (non-concurrent) step is executed during $P_b^1 \parallel P_b^2$; for example, executing interleaved program $(a_1 \parallel a_2) \mid a_3$ will result in action a_3 happening either before or after both a_1 and a_2. Second, no two branches of a concurrent program are 'related'. Two branches are related if *(i)* a variable appearing in one branch can be bound by the other, or *(ii)* the same atom can be both asserted by one branch and checked or asserted by the other.

To formalise the assumption that the branches of a concurrent program are unrelated, we define some auxiliary notions. Let Π and Λ be a plan- and an action-library, respectively. First, given any expression E, we use $\text{ATS}(E)$ to denote the set of atoms occurring in E. Second, given an event-goal e, we use $\text{REL}(e, \Pi) = \{\psi\theta : P_b\theta \mid e' : \psi \leftarrow P_b \in \Pi,\ \theta = \mathsf{mgu}(e, e')\}$ to denote the relevant plan-rules for e, i.e., rules with 'heads' e' that match e via a most general unifier (mgu). Third, given a plan-body P_b, we recursively define the set of atoms that are possibly *checked* by P_b as follows: $\text{CHK}(P_b, \Pi, \Lambda) =$

$$\begin{cases} \emptyset & \text{if } P_b \in \{+b, -b\}, \\ \text{ATS}(\phi) & \text{if } P_b = ?\phi, \\ \text{ATS}(pre(a, \Lambda)) & \text{if } P_b = a, \\ \bigcup_{\psi : P_b' \in \text{REL}(e,\Pi)} \text{ATS}(\psi) \cup \text{CHK}(P_b', \Pi, \Lambda) & \text{if } P_b = !e, \\ \text{CHK}(P_b^1, \Pi, \Lambda) \cup \text{CHK}(P_b^2, \Pi, \Lambda) & \text{if } P_b \in \{P_b^1 \mid P_b^2, \\ & \quad P_b^1 \parallel P_b^2, P_b^1 ; P_b^2\}. \end{cases}$$

Similarly, we define the set of atoms that are possibly *asserted* by P_b as follows: $\text{ASS}(P_b, \Pi, \Lambda) =$

$$\begin{cases} \{b\} & \text{if } P_b \in \{+b, -b\}, \\ \emptyset & \text{if } P_b = ?\phi, \\ \text{ATS}(\mathit{eff}(a, \Lambda)) & \text{if } P_b = a, \\ \bigcup_{\psi : P_b' \in \text{REL}(e,\Pi)} \text{ASS}(P_b', \Pi, \Lambda) & \text{if } P_b = !e, \\ \text{ASS}(P_b^1, \Pi, \Lambda) \cup \text{ASS}(P_b^2, \Pi, \Lambda) & \text{if } P_b \in \{P_b^1 \mid P_b^2, \\ & \quad P_b^1 \parallel P_b^2, P_b^1 ; P_b^2\}. \end{cases}$$

Finally, for any concurrent program $P_b^1 \parallel \ldots \parallel P_b^n$ occurring in Π, we assume that for any $i, j \in [1, n]$, with $i \neq j$, there does not exist a *(i)* variable that occurs in both P_b^i and P_b^j, and *(ii)* unifier for any pair of atoms $\{l, l'\}$, where $l \in \text{CHK}(P_b^i, \Pi, \Lambda) \cup \text{ASS}(P_b^i, \Pi, \Lambda)$ and $l' \in \text{ASS}(P_b^j, \Pi, \Lambda)$.[3]

3 Single-Intention CCAN Semantics

We define an *agent configuration* as a tuple $[\Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma]$, where Π is a plan-library; Λ is an action-library; $\mathcal{B}$ is a belief base; and $\mathcal{A}$ is an *action history* representing the sequential and concurrent actions executed so far. Formally, an *action history* is a formula in the language defined by the grammar

$$\mathcal{A} ::= a \mid \mathcal{A}_1 \,;\mathcal{A}_2 \mid \mathcal{A}_1 \parallel \mathcal{A}_2$$

where $\mathcal{A}_1 \parallel \mathcal{A}_2$ is an abstract representation for *all* the associated 'schedules', each comprising interleaved and/or overlapping actions. Finally, Γ is a set of *programs*, or 'intentions', each of which is the current evolution in the execution of a plan-body that is being pursued in order to achieve a top-level event-goal. As it is usual with small-step operational semantics of programming languages, the syntax of plan-bodies has to be extended with new constructs to represent these current evolutions. Formally,

[2] The exact schedule that emerges from the resulting actions will be based on runtime choices.

[3] We refer the reader to [13] for insights into algorithms for checking these conditions.

$$\frac{\Delta = \text{REL}(e) \neq \emptyset}{[\mathcal{B},\mathcal{A},!e] \xrightarrow{\text{ONE}} [\mathcal{B},\mathcal{A},e:(\!|\Delta|\!)]}\ Ev \qquad \frac{\psi : P \in \Delta \quad \mathcal{B} \models \psi\theta}{[\mathcal{B},\mathcal{A},e:(\!|\Delta|\!)] \xrightarrow{\text{ONE}} [\mathcal{B},\mathcal{A},P\theta \rhd e:(\!|\Delta \setminus \{\psi:P\}|\!)]}\ Sel$$

$$\frac{[\mathcal{B},\mathcal{A},P_1] \xrightarrow{\text{ONE}} [\mathcal{B}',\mathcal{A}',P_1']}{[\mathcal{B},\mathcal{A},P_1;P_2] \xrightarrow{\text{ONE}} [\mathcal{B}',\mathcal{A}',P_1';P_2]}\ Seq_1 \qquad \frac{\text{FIN}(P_1)}{[\mathcal{B},\mathcal{A},P_1;P_2] \xrightarrow{\text{ONE}} [\mathcal{B},\mathcal{A},P_2]}\ Seq_2$$

$$\frac{[\mathcal{B},\mathcal{A},P_1] \xrightarrow{\text{ONE}} [\mathcal{B}',\mathcal{A}',P_1']}{[\mathcal{B},\mathcal{A},P_1 \mid P_2] \xrightarrow{\text{ONE}} [\mathcal{B}',\mathcal{A}',P_1' \mid P_2]}\ |_1 \qquad \frac{[\mathcal{B},\mathcal{A},P_2] \xrightarrow{\text{ONE}} [\mathcal{B}',\mathcal{A}',P_2']}{[\mathcal{B},\mathcal{A},P_1 \mid P_2] \xrightarrow{\text{ONE}} [\mathcal{B}',\mathcal{A}',P_1 \mid P_2']}\ |_2$$

$$\frac{[\mathcal{B},\mathcal{A},P_1] \xrightarrow{\text{ONE}} [\mathcal{B}',\mathcal{A}',P_1']}{[\mathcal{B},\mathcal{A},P_1 \rhd P_2] \xrightarrow{\text{ONE}} [\mathcal{B}',\mathcal{A}',P_1' \rhd P_2]}\ \rhd_{step} \qquad \frac{[\mathcal{B},\mathcal{A},P_1] \not\xrightarrow{\text{ONE}} \quad \neg\text{FIN}(P_1) \quad [\mathcal{B},\mathcal{A},P_2] \xrightarrow{\text{ONE}} [\mathcal{B},\mathcal{A},P_2']}{[\mathcal{B},\mathcal{A},P_1 \rhd P_2] \xrightarrow{\text{ONE}} [\mathcal{B},\mathcal{A},P_2']}\ \rhd_{fail}$$

$$\frac{\mathcal{B} \models \phi}{[\mathcal{B},\mathcal{A},?\phi] \xrightarrow{\text{X}} [\mathcal{B},\mathcal{A},\eta]}\ Test \qquad \frac{a' : \psi \leftarrow \Phi^+;\Phi^- \in \Lambda \qquad a'\theta = a \qquad \mathcal{B} \models \psi\theta}{[\mathcal{B},\mathcal{A},a] \xrightarrow{\text{X}} [(\mathcal{B} \setminus \Phi^-\theta) \cup \Phi^+\theta, \mathcal{A};a,\eta]}\ Act$$

$$\frac{[\mathcal{B},\mathcal{A},P_1] \xrightarrow{\text{ALL}} [\mathcal{B}_1,\mathcal{A}_1,P_1'] \quad [\mathcal{B},\mathcal{A},P_2] \xrightarrow{\text{ALL}} [\mathcal{B}_2,\mathcal{A}_2,P_2']}{[\mathcal{B},\mathcal{A},P_1 \parallel P_2] \xrightarrow{\text{X}} [\text{MERGE}(\mathcal{B}_1,\mathcal{B}_2,\mathcal{B}),\text{APPEND}(\mathcal{A}_1,\mathcal{A}_2,\mathcal{A}),P_1' \not\parallel P_2']}\ \parallel$$

$$\frac{[\mathcal{B},\mathcal{A},P_1] \xrightarrow{\text{ONE}} [\mathcal{B}_1,\mathcal{A}_1,P_1'] \quad [\mathcal{B}_1,\mathcal{A}_1,P_1';P_2] \xrightarrow{\text{ALL}} [\mathcal{B}_2,\mathcal{A}_2,P_3] \quad [\mathcal{B}_2,\mathcal{A}_2,P_3] \not\xrightarrow{\text{ALL}}}{[\mathcal{B},\mathcal{A},P_1;P_2] \xrightarrow{\text{ALL}} [\mathcal{B}_2,\mathcal{A}_2,P_3]}\ Seq_1^{\parallel}$$

$$\frac{\text{FIN}(P_1) \quad [\mathcal{B},\mathcal{A},P_2] \xrightarrow{\text{ALL}} [\mathcal{B}_1,\mathcal{A}_1,P_3]}{[\mathcal{B},\mathcal{A},P_1;P_2] \xrightarrow{\text{ALL}} [\mathcal{B}_1,\mathcal{A}_1,P_3]}\ Seq_2^{\parallel} \qquad \frac{[\mathcal{B},\mathcal{A},P_1] \xrightarrow{\text{ONE}} [\mathcal{B}_1,\mathcal{A}_1,P_1'] \quad [\mathcal{B}_1,\mathcal{A}_1,P_1';P_2] \not\xrightarrow{\text{ALL}}}{[\mathcal{B},\mathcal{A},P_1;P_2] \xrightarrow{\text{ALL}} [\mathcal{B}_1,\mathcal{A}_1,P_1';P_2]}\ Seq_3^{\parallel}$$

$$\frac{P \in \{!e, P_1 \mid P_2\} \quad [\mathcal{B},\mathcal{A},P\,;?\top] \xrightarrow{\text{ALL}} [\mathcal{B}',\mathcal{A}',P']}{[\mathcal{B},\mathcal{A},P] \xrightarrow{\text{ALL}} [\mathcal{B}',\mathcal{A}',P']}\ Rewrite^{\parallel}$$

Figure 1: CCAN derivation rules for configurations with single intentions.

a ***program*** is a formula in the language defined by the grammar $P ::=$

$$a \mid +b \mid -b \mid ?\phi \mid !e \mid P_1\,;P_2 \mid P_1\text{“|”}P_2 \mid P_1 \parallel P_2 \mid$$
$$\eta \mid e : (\!|\{\psi_1 : P_1, \ldots, \psi_n : P_n\}|\!) \mid P_1 \rhd P_2 \mid P_1 \not\parallel P_2$$

where η (or 'nil') indicates that a program has finished, i.e., successfully terminated; $P_1 \not\parallel P_2$ indicates that a concurrent program has terminated (not necessarily successfully); $e : (\!|\{\psi_1 : P_1, \ldots, \psi_n : P_n\}|\!)$ represents the set of plan-rules that are relevant for achieving event-goal e; and 'failure handling' program $P \rhd P'$, with $P' = e : (\!|\{\psi_1 : P_1, \ldots, \psi_n : P_n\}|\!)$, executes program P in order to achieve event-goal e, and if P fails, an alternative program (plan-body) P_i is tried if it is applicable. Note that a program is more general than those generated by our semantics.

Like [27], we define a *transition relation* on configurations in terms of a set of derivation rules [24]; we omit the elements Π and Λ from configurations in our transitions as those elements do not change between transitions. A derivation rule has an antecedent and a conclusion: the latter is a single transition, and the former is either empty or a conjunction of auxiliary conditions and/or transitions representing 'internal' execution steps. In this paper we only use labelled transitions. A transition $C \xrightarrow{\text{ONE}} C'$ indicates that doing one execution step on configuration C, which may involve multiple internal execution steps, yields configuration C'. A transition $C \xrightarrow{\text{ALL}} C'$ indicates that C' is a result of doing all possible internal execution steps from C. Intuitively, ONE-type transitions model behaviour in the context of standard execution, and ALL-type ones model behaviour in the context of concurrent execution.

We first give our semantics for single-intention configurations of the form $[\Pi, \Lambda, \mathcal{B}, \mathcal{A}, P]$, where P is a program. In sec. 3.1 we give derivation rules for standard programs, and in sec. 3.2 we give rules that relate to concurrent programs.

3.1 Derivation Rules for Standard Programs

Fig. 1 (top half) shows the derivation rules for standard CCAN programs, including interleaved programs.

Rule Ev creates the set Δ of relevant plan-rules for a given event-goal program $!e$. Rule Sel selects an applicable plan-rule for an event-goal e from its relevant plan-rules Δ, and schedules the corresponding plan-body for execution.

Rules Seq_1 and Seq_2 give semantics for sequential execution: Seq_1 executes one step on a sequential program $P_1;P_2$ by executing a step on its first program P_1, and Seq_2 removes P_1 if it has *finished* (as we define in sec. 3.2), e.g., if $P_1 = \eta$. Rules $|_1$ and $|_2$ give semantics for interleaved execution: given an interleaved program $P_1 \mid P_2$, one step is executed either on P_1 (using rule $|_1$) or on P_2 (using $|_2$).

Rules $\rhd_{step}$ and $\rhd_{fail}$ give semantics for executing a previously selected plan-body and for failure handling, respectively: rule $\rhd_{step}$ executes one step on a program $P_1 \rhd P_2$ by executing a step on P_1, provided it has neither failed nor finished, and rule $\rhd_{fail}$ removes P_1 if it

has failed, and executes a step on program $P_2 = e : (\!|\Delta|\!)$. A program has *failed* if it has not finished (as in the second condition in the antecedent of $\rhd_{fail}$) but it is 'stuck' (as in the first condition in the antecedent of $\rhd_{fail}$), i.e., it is not possible to execute a step on the program (e.g., an event-goal program $!e$ when there are no relevant plan-rules for e). Given a configuration C, we use $C \xrightarrow{\text{ONE}}$ as an abbreviation for $\exists C', C \xrightarrow{\text{ONE}} C'$ (and use $C \xrightarrow{\text{ALL}}, C \not\xrightarrow{\text{ONE}}$, and $C \not\xrightarrow{\text{ALL}}$ similarly).

Finally, rule $Test$ executes one step on a test program $?\phi$ if condition ϕ holds in the belief base, and rule Act gives semantics for actions. This rule's antecedent checks whether the relevant action-rule of a given action a is applicable, and the conclusion applies the action's effects to the belief base, and appends a to action history $\mathcal{A}$. When the transition label X occurs in a derivation rule, the associated transition represents both transition types (ONE and ALL).

3.2 Derivation Rules for Concurrent Programs

Fig. 1 (bottom half) shows the derivation rules that relate to concurrent programs. The main rule $\|$ executes one step on a program $P_1 \parallel P_2$, which amounts to independently doing all possible internal execution steps on each branch P_1 and P_2. The rule applies when at least one step is possible on each branch, and in the context of both standard and concurrent execution (which enables 'nested' concurrency).

The conclusion of the rule does two things. First, it merges the independent and 'local' updates to (copies of) belief base $\mathcal{B}$ by branches P_1 and P_2. We define $\text{MERGE}(\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}) = (\mathcal{B} \cup \mathcal{B}_1^+ \cup \mathcal{B}_2^+) \setminus (\mathcal{B}_1^- \cup \mathcal{B}_2^-)$, where for both $i \in [1,2]$, $\mathcal{B}_i^+ = \mathcal{B}_i \setminus \mathcal{B}$ and $\mathcal{B}_i^- = \mathcal{B} \setminus \mathcal{B}_i$. Second, the conclusion combines the action histories (if any) yielded by the two branches to form a new history, which is appended to $\mathcal{A}$. This notion of combining and appending is defined as follows. Let $\mathcal{A}, \mathcal{A}_1$, and $\mathcal{A}_2$ be as in rule $\|$. For both $i \in [1,2]$, let $\mathcal{A}_i = \mathcal{A}\,;\mathcal{A}_i'$ for some $\mathcal{A}_i'$, or let $\mathcal{A}_i = \mathcal{A}$, i.e., no actions were yielded by the corresponding branch. Then, $\text{APPEND}(\mathcal{A}_1, \mathcal{A}_2, \mathcal{A})$ is defined as *(i)* $\mathcal{A}\,;(\mathcal{A}_1' \parallel \mathcal{A}_2')$ if $\mathcal{A}_1, \mathcal{A}_2 \neq \mathcal{A}$, *(ii)* $\mathcal{A}_2$ if $\mathcal{A}_1 = \mathcal{A}$, and *(iii)* $\mathcal{A}_1$ if $\mathcal{A}_2 = \mathcal{A}$.

Rules $Seq_1^{\|}, Seq_2^{\|}$ and $Seq_3^{\|}$ give semantics for sequential execution in the context of concurrency. Rule $Seq_1^{\|}$ applies when at least two execution steps are possible on a given sequential program $P_1; P_2$. The antecedent executes one step on P_1, and then recursively performs all possible execution steps on the remainder $P_1'; P_2$. Rule $Seq_3^{\|}$ applies when a given sequential program $P_1; P_2$ (possibly a remainder) can terminate in one step. Rule $Seq_2^{\|}$ is analogous to Seq_2, and applies when P_1 has finished and at least one step is possible on P_2. Formally, given the set of all programs P_{all}, function $\text{FIN} : P_{all} \mapsto \{\top, \bot\}$ indicates whether a given program $P \in P_{all}$ has finished. The function is defined as follows:

$$\text{FIN}(P) = \begin{cases} \text{FIN}(P_1) \wedge \text{FIN}(P_2) & \text{if } P \in \{P_1 \between P_2, P_1 \mid P_2\}, \\ \text{FIN}(P_1) & \text{if } P = P_1 \rhd P_2, \\ \top & \text{if } P = \eta, \\ \bot & \text{otherwise.} \end{cases}$$

Proposition 1. *If* $\text{FIN}(P)$ *holds for some program* P*, then for any belief base* $\mathcal{B}$ *and action history* $\mathcal{A}$*, there is no* $\mathcal{B}', \mathcal{A}'$ *and* P' *such that* $[\mathcal{B}, \mathcal{A}, P] \xrightarrow{\text{ONE}} [\mathcal{B}', \mathcal{A}', P']$.

Proof. The case $P \in \{\eta, P_1 \between P_2\}$ (for some P_1 and P_2) is trivial as no rules can be applied to P. If $P = P_1 \rhd P_2$, we show that neither rule $\rhd_{step}$ nor $\rhd_{fail}$ applies to P. Since $\text{FIN}(P)$ holds, so does $\text{FIN}(P_1)$. Thus $\rhd_{fail}$ cannot be applied to P. Similarly, $\rhd_{step}$ cannot be applied if $P_1 \in \{\eta, P_1^1 \between P_1^2\}$. If $P_1 = P_1^1 \rhd P_1^2$, i.e., construct $\rhd$ is 'nested' in P, the proposition follows because $\text{FIN}(P_1^1)$ holds, and $\rhd$ can only be nested to a finite depth in P. The cases where P_1 and P are interleaved programs are proved similarly. □

Finally, given an event-goal or interleaved program, rule $Rewrite^{\|}$ 'rewrites' it to make it the first program of a simple sequence, which is executed as above. This avoids the need for rules to handle programs $P_1 \mid P_2$ and $!e$ (and evolutions such as $P \rhd P$') in the context of concurrency.

3.3 An Example

We will now illustrate some of the derivation rules in fig. 1 with an example. Consider a manufacturing facility with a robotic station that engraves the surfaces of wooden spheres. The station comprises a camera, a fixture that holds and rotates a sphere while other operations are being performed on it, and a robotic arm with a built-in circular tool changing rack comprising 6 tools (e.g. for milling and drilling).

The fixture can perform action $rX(N)$ (resp. $rZ(M)$), which rotates a wooden sphere, if it is currently in the fixture, N (resp. M) times on the x-axis (resp. z-axis); each complete rotation takes 5 seconds and starts instantly. Both actions have precondition in, which is a proposition that holds only if a sphere is sensed in the fixture. In our scenario, $M = N = 2$; the extra rotation on each axis leaves sufficient time for a concurrent preparatory (tool changing) action to complete. The camera performs action $r(N)$, which records, for N seconds, a video of all the actions that are being performed on the sphere in the fixture; we use $N = 25$.

The capabilities of the robot are as follows. Event-goal e, if it uses plan-rule $e : in \wedge \neg at \leftarrow c\,; m(20)$, first prepares to engrave the sphere by changing the current tool to the milling tool, and then mills the sphere. Event-goal e uses rule $e : in \wedge at \leftarrow m(20)$ if in holds and the milling tool was used last, i.e., proposition at holds. Action c above rotates the tool rack until the milling tool is

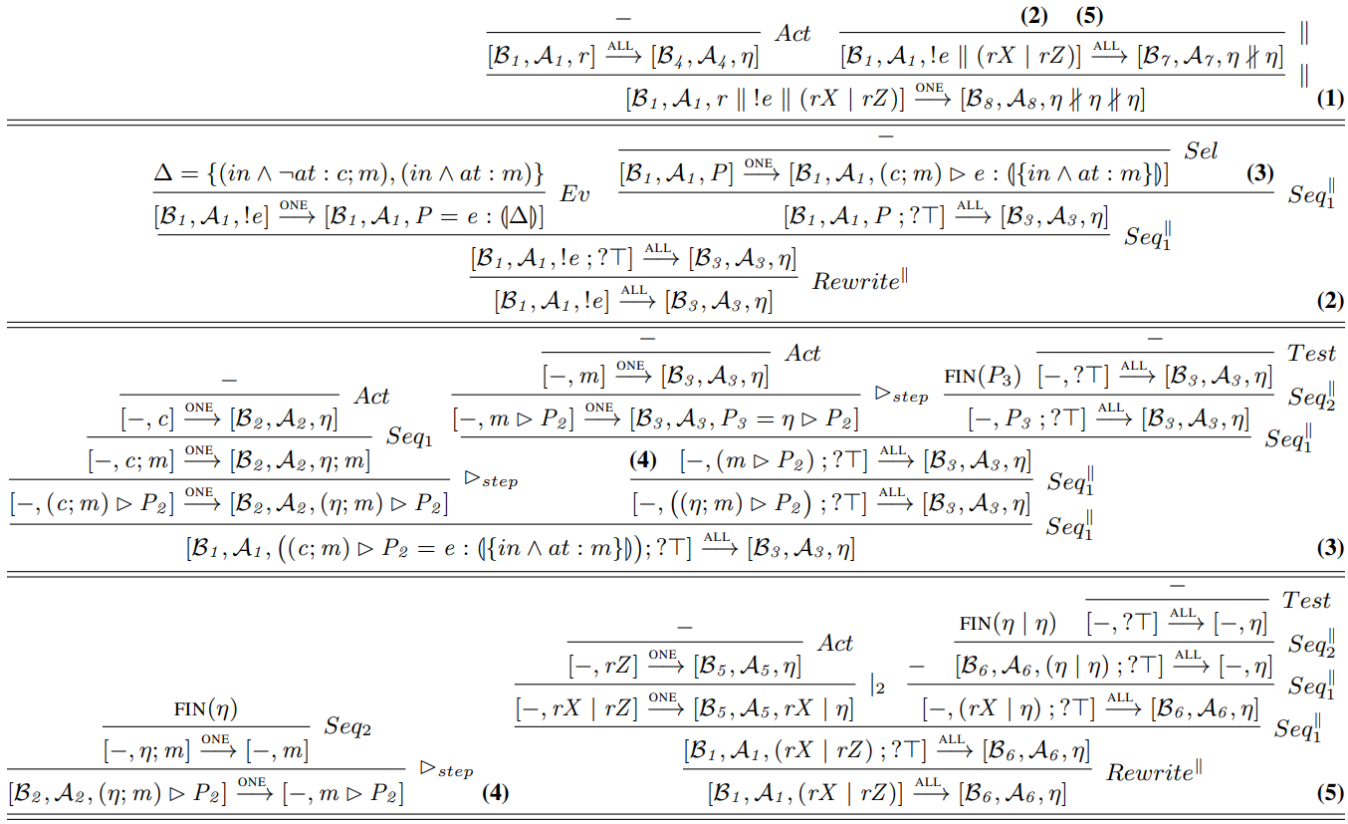

Figure 2: Equation 1 (first row) shows one execution step on concurrent program $r \parallel !e \parallel (rX \mid rZ)$, which involves one step on r, and all the possible steps on both $!e$ (eqs. 2-4 in rows 2-4) and $(rX \mid rZ)$ (eq. 5 in row 4). Action parameters are omitted, as are obvious antecedents, and obvious belief bases and action histories in configurations. The action histories above are: *(i)* $\mathcal{A}_4 = \mathcal{A}_1; r$, *(ii)* $\mathcal{A}_2 = \mathcal{A}_1; c$, *(iii)* $\mathcal{A}_3 = \mathcal{A}_2; m$, *(iv)* $\mathcal{A}_5 = \mathcal{A}_1; rZ$, *(v)* $\mathcal{A}_6 = \mathcal{A}_5; rX$, *(vi)* $\mathcal{A}_7 = \mathcal{A}_1; \big((c\,; m) \parallel (rZ\,; rX)\big)$, and *(vii)* $\mathcal{A}_8 = \mathcal{A}_1; \big(r \parallel (c\,; m) \parallel (rZ\,; rX)\big)$.

reached; it takes a second to rotate to the next tool on the rack (and thus at most 5 seconds). Action $m(N)$, with precondition in, mills for N seconds, which includes starting high speed rotation for the milling tool, moving it into the sphere, moving it out, and ending tool rotation, each of which takes negligible time.[4]

The derivation rule in the top row (eq. 1) of fig. 2 depicts one execution step on the concurrent program $r(25) \parallel !e \parallel (rX(2) \mid rZ(2))$, which specifies that while the sphere is being engraved, it should be (simultaneously) rotated on the x-axis and the z-axis (which can be performed in either order), and that all these activities should be recorded. The program is executed using rule $\parallel$, whose antecedent prescribes the concurrent execution of two programs, each using a step of type ALL. One such step is performed on action $r(25)$, and the other on concurrent 'subprogram' $!e \parallel (rX(2) \mid rZ(2))$ by recursively applying rule $\parallel$, whose antecedent, in turn, prescribes the concurrent execution of event-goal program $!e$ (eq. 2) and interleaved program $rX(2) \mid rZ(2)$ (eq. 5).

Executing the top-level concurrent program yields action history $r(25) \parallel (c\,; m(20)) \parallel (rZ(2)\,; rX(2))$, which *(i)* abstractly represents all the associated action schedules (e.g. where the branches are interleaved, and where $r(25), c$, and $rZ(2)$ start together and overlap),[5] and *(ii)* corresponds to the 'terminated branches' of the concurrent program.

4 Multiple-Intention CCAN Semantics

We now give our semantics for configurations with multiple intentions, i.e., agent configurations, which are of the form $[\Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma]$, where Γ is a set of programs.

A transition between agent configurations is either of type CCAN, EVENT, or INT, and the transition relation on agent configurations is defined by the derivation rules in fig. 3. Rule A_{ccan} is the main rule, which represents the CCAN deliberation cycle. The CCAN-type step in the conclusion of the rule involves two internal steps. In the first internal step, an intention is either *(i)* removed (using rule A_{rem}) if it has failed or finished, *(ii)* progressed (using rule A_{int}) by one step, or *(iii)* progressed by multiple steps, in which case a concurrent program will have been executed. In the second internal step, newly observed event-goals from the (external) environment are processed (using rule A_{ev}), by creating an intention for

[4] For a given axis of rotation, we assume there will be no difference in the engraving on the sphere whether milling is performed for exactly one rotation of the sphere or for longer (e.g. milling starts before the rotation, or ends after).

[5] Since, in systems such as PRS and JACK, branches ('threads') *typically* start execution at roughly the same point in time, we assume the same when writing concurrent branches in CCAN.

$$\frac{[\mathcal{B},\mathcal{A},\Gamma] \overset{\text{INT}}{\Longrightarrow} [\mathcal{B}',\mathcal{A}',\Gamma'] \quad [\mathcal{B}',\mathcal{A}',\Gamma'] \overset{\text{EVENT}}{\Longrightarrow} [\mathcal{B}',\mathcal{A}',\Gamma'']}{[\mathcal{B},\mathcal{A},\Gamma] \overset{\text{CCAN}}{\Longrightarrow} [\mathcal{B}',\mathcal{A}',\Gamma'']}\ A_{ccan} \qquad \frac{P \in \Gamma \quad [\mathcal{B},\mathcal{A},P] \overset{\text{ONE}}{\longrightarrow} [\mathcal{B}',\mathcal{A}',P']}{[\mathcal{B},\mathcal{A},\Gamma] \overset{\text{INT}}{\Longrightarrow} [\mathcal{B}',\mathcal{A}',(\Gamma \setminus \{P\}) \cup \{P'\}]}\ A_{int}$$

$$\frac{P \in \Gamma \quad [\mathcal{B},\mathcal{A},P] \overset{\text{ONE}}{\not\longrightarrow}}{[\mathcal{B},\mathcal{A},\Gamma] \overset{\text{INT}}{\Longrightarrow} [\mathcal{B},\mathcal{A},\Gamma \setminus \{P\}]}\ A_{rem} \qquad \frac{e_1,\ldots,e_n}{[\mathcal{B},\mathcal{A},\Gamma] \overset{\text{EVENT}}{\Longrightarrow} [\mathcal{B},\mathcal{A},\Gamma \cup \{!e_1,\ldots,!e_n\}]}\ A_{ev}$$

Figure 3: CCAN derivation rules for configurations with multiple intentions.

each such event-goal.

5 Properties of CCAN

We now show that our concurrency semantics has three key properties: a concurrent program does not terminate 'prematurely' (before all the branches terminate), as ensured by JACK's fourth definition (sec. 1); the semantics is sound and complete in terms of the action histories that are produced; and if the concurrent program's branches are interleaved, any resulting action history corresponds to a valid action schedule for the concurrent program, i.e., the history is an ordering of a pomset yielded by the concurrent program. We use action histories because we are interested in exploring behavioural equivalence—we thus abstract from things such as belief bases, and 'unobservable steps' such as the creation of a relevant plan set (rule Ev in fig. 1).

We first define the notion of an execution trace, which is a sequence of configurations obtained by performing ONE type execution steps. In the sequel, we assume that all $+b$ and $-b$ programs occurring in the plan-library Π have been replaced by equivalent actions, and given a configuration $C = [\mathcal{B},\mathcal{A},P]$, we define $C_{\mathcal{B}} = \mathcal{B}$, $C_{\mathcal{A}} = \mathcal{A}$, and $C_P = P$.

Definition 1. *An execution trace of a configuration $C = [\mathcal{B},\mathcal{A},P]$ is a finite sequence of configurations $C_1 \cdot \ldots \cdot C_n$ such that $C = C_1$, $n > 1$, and $C_i \overset{\text{ONE}}{\longrightarrow} C_{i+1}$ for all $i \in [1, n-1]$; the trace is said to have terminated if $C_n \overset{\text{ONE}}{\not\longrightarrow}$.*

The first theorem states that after one execution step on a concurrent program (which yields a terminated concurrent program $P'_1 \nparallel \ldots \nparallel P'_n$), each branch will have performed all the possible (internal) execution steps and terminated.

Theorem 1. *Let $\mathcal{B}$ be a belief base, $\mathcal{A}$ an action history, and $P_{\|} = P_1 \parallel \ldots \parallel P_n$ a concurrent program s.t. $P_i \neq P_i^1 \parallel P_i^2$ (for any P_i^1, P_i^2 and $i \in [1,n]$).[6] If $[\mathcal{B},\mathcal{A},P_{\|}] \overset{\text{ONE}}{\longrightarrow} [\mathcal{B}',\mathcal{A}',P'_1 \nparallel \ldots \nparallel P'_n]$, then $[\mathcal{B}',\mathcal{A}',P'_i] \overset{\text{ONE}}{\not\longrightarrow}$ for $i \in [1,n]$.*

Proof Sketch. Consider any P_i above. From the assumption of the theorem and derivation rule $\|$, it follows that $[\mathcal{B},\mathcal{A},P_i] \overset{\text{ALL}}{\longrightarrow} [\mathcal{B}'',\mathcal{A}'',P'_i]$, where $\mathcal{B}''$ and $\mathcal{A}''$ correspond to P'_i and form part of respectively $\mathcal{B}'$ and $\mathcal{A}'$.

If $P_i \in \{?\phi, a\}$, only rule $Test$ or Act can apply, which guarantee that $[\mathcal{B}'',\mathcal{A}'',P'_i] \overset{\text{ONE}}{\not\longrightarrow}$. If $P_i \in \{!e, P \mid P'\}$ (for some P and P'), it is rewritten as a sequential program. Thus, the final case is where P_i is a sequence, or evolves into one. Consider the former (the latter is analogous). Since rule $Seq_2^{\|}$ cannot apply to configuration $[\mathcal{B},\mathcal{A},P_i]$ (because P_i is a plan-body and thus unfinished), either rule $Seq_1^{\|}$ or $Seq_3^{\|}$ must have been applied to $[\mathcal{B},\mathcal{A},P_i]$, both of whose antecedents ensure that $[\mathcal{B}'',\mathcal{A}'',P'_i] \overset{\text{ALL}}{\not\longrightarrow}$ holds. Finally, we prove by contradiction that $[\mathcal{B}'',\mathcal{A}'',P'_i] \overset{\text{ONE}}{\not\longrightarrow}$ also holds.

Let us assume instead that $[\mathcal{B}'',\mathcal{A}'',P'_i] \overset{\text{ONE}}{\longrightarrow}$. Consider the case where $P'_i = P_i^1; P_i^2$ is a sequence such that P_i^1 is not a sequence and $\neg\text{FIN}(P_i^1)$ holds. Since $[\mathcal{B}'',\mathcal{A}'',P_i^1] \overset{\text{ONE}}{\longrightarrow}$ is entailed by our assumption, either the antecedent of rule $Seq_1^{\|}$ or $Seq_3^{\|}$ must hold w.r.t. P'_i. This contradicts the fact that $[\mathcal{B}'',\mathcal{A}'',P'_i] \overset{\text{ALL}}{\not\longrightarrow}$. The cases where $\text{FIN}(P_i^1)$ holds or P'_i is not a sequence also lead to contradictions. □

The theorem can be straightforwardly extended to show that, due to the definition of 'FIN' (sec. 3.2), the concurrent program $P_{\|}$ succeeds, i.e., yields a terminated (concurrent) program that has finished, iff each branch P_i has succeeded.

Theorem 2 concerns soundness and completeness for the derivation rules that relate to concurrent programs. Soundness is due to the fact that any action history yielded by a branch of a concurrent program can also be yielded when the branch is executed separately from the program (perhaps using only rules in the top half of fig. 1). Conversely, completeness is due to the fact that any action history yielded by a non-concurrent program upon its termination can also be yielded when the latter is a branch of a concurrent program.

Theorem 2. *Let $\mathcal{B},\mathcal{A}$ and $P_{\|} = P_1 \parallel \ldots \parallel P_n$ be as above. There exists a transition $[\mathcal{B},\mathcal{A},P_{\|}] \overset{\text{ONE}}{\longrightarrow} [\mathcal{B}',\mathcal{A};\mathcal{A}',P'_{\|}]$, with $\mathcal{A}' = \mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$, iff there exists a terminated execution trace $C_1 = [\mathcal{B},\mathcal{A},P_i] \cdot \ldots \cdot C_m = [\mathcal{B}'',\mathcal{A};\mathcal{A}'_i,P'_i]$ for each $i \in [1,n]$, such that $\mathcal{A}'_i = \mathcal{A}_i$.[7]*

Proof Sketch. We discuss one direction of the proof: consider an execution trace $C_1 \cdot \ldots \cdot C_m$ as above. The proof is involved, requiring induction on the length of the trace and the structure of each C_i. The main part is the inductive case, which takes any pair of configurations C_j and

[6] Any concurrent program can be represented in this 'full' form.

[7] We omit the trivial corollary where some $\mathcal{A}_i$ are 'empty'.

C_{j+1} (for $j \in [1, m-1]$), and the corresponding action history $\mathcal{A}''$ (if $C_{j+1}|_{\mathcal{A}} = C_j|_{\mathcal{A}}; \mathcal{A}''$), and shows that $\mathcal{A}''$ can also be yielded by an ALL-type transition from C_j.

Let us consider the two main cases. In the first, $C_j|_P = P_j^1; P_j^2$ (for some P_j^1 and P_j^2), and P_j^1 is not a sequential program and $\neg\text{FIN}(P_j^1)$ holds. Since the transition $C_j \xrightarrow{\text{ONE}} C_{j+1}$ must have used rule Seq_1, it follows that transition $[C_j|_{\mathcal{B}}, C_j|_{\mathcal{A}}, P_j^1] \xrightarrow{\text{ONE}} [C_{j+1}|_{\mathcal{B}}, C_{j+1}|_{\mathcal{A}}, P_j^{1'}]$ is possible, with $C_{j+1}|_P = P_j^{1'}; P_j^2$. Thus, the first condition in the antecedents of both $Seq_1^{\|}$ and $Seq_3^{\|}$ holds. The interesting subcase is where $\neg\text{FIN}(P_j^{1'})$. Now if $C_{j+1} \not\xrightarrow{\text{ONE}}$ (i.e., $j+1 = m$), it follows that the second condition also holds in the antecedent of $Seq_3^{\|}$. Thus, $\mathcal{A}''$ can be yielded by applying the rule to C_j. If $C_{j+1} \xrightarrow{\text{ONE}}$, the second condition holds in the antecedent of $Seq_1^{\|}$, and consequently also the third. Thus, $\mathcal{A}''$ can be yielded by applying the rule to C_j.

The second main case is where $C_j|_P = !e$ (the case where $C_j|_P = P_j^1 \mid P_j^2$ is analogous). Then, transition $C_j \xrightarrow{\text{ONE}} C_{j+1}$ must have used rule Ev, and $C_{j+1}|_P = e : (\!|\Delta|\!)$. This transition can be simulated by applying rule $Rewrite^{\|}$ to C_j: the rule's antecedent holds because we showed above that either $Seq_1^{\|}$ or $Seq_3^{\|}$ must be applicable to configurations such as $[C_j|_{\mathcal{B}}, C_j|_{\mathcal{A}}, !e\,; ?\top]$. □

The third theorem links concurrency and interleaving, and is based on the standard notions of a pomset and a linear extension of one. We first give some auxiliary definitions to represent action histories produced by a concurrent program in terms of pomsets, which enables comparing their linear extensions with the action histories that are produced when the program's branches are interleaved.

Definition 2. *An action pomset (or simply a pomset) is a 4-tuple $\rho = \langle V, A, \prec, f\rangle$, where the 'vertices' V is a finite set of natural numbers, A is a finite set of actions, the 'ordering relation' $\prec$ is an irreflexive, a transitive, and an asymmetric binary relation on V, and the 'labelling function' f is a surjection from V to A. A sequence of actions $a_1; \ldots; a_n$ is a linear extension of a pomset ρ, denoted $a_1; \ldots; a_n \in \text{LIN}(\rho)$, iff there exists a permutation $v_1 \cdot \ldots \cdot v_n$ of the vertices of ρ such that $(v_i, v_j) \in \prec$ implies $i < j$, and $a_k = f(v_k)$ for all $k \in [1, n]$.*[8]

Next, we define how a pomset is built from a given action history, by the recursive application of sequential and/or parallel composition operators. Let $\rho_1 = \langle V_1, A_1, \prec_1, f_1\rangle$ and $\rho_2 = \langle V_2, A_2, \prec_2, f_2\rangle$ be pomsets, and let r (a 'renaming') be a bijection from V_2 to a set V_3 such that $V_3 \cap V_1 = \emptyset$ and $V_3 \cap V_2 = \emptyset$. Let $\langle V_3, A_3, \prec_3, f_3\rangle$ be the pomset obtained from ρ_2 by renaming it using r, i.e., replacing each occurrence in ρ_2 of each vertex $v \in V_2$ with $r(v)$. Then, we define $\rho_1 \prec^* \rho_2$ as the pomset $\langle V, A, \prec, f\rangle$, where $V = V_1 \cup V_3$, $A = A_1 \cup A_3$, $f = f_1 \cup f_3$, and $\prec = \prec_1 \cup \prec_3 \cup V_1 \times V_3$; similarly, we define $\rho_1 \cup^* \rho_2$ as the pomset $\langle V, A, \prec_1 \cup \prec_3, f\rangle$. Finally, given an action history $\mathcal{A}$, we define the corresponding pomset as

$$\mathbf{P}(\mathcal{A}) = \begin{cases} \mathbf{P}(\mathcal{A}_1) \prec^* \mathbf{P}(\mathcal{A}_2) & \text{if } \mathcal{A} = \mathcal{A}_1; \mathcal{A}_2, \\ \mathbf{P}(\mathcal{A}_1) \cup^* \mathbf{P}(\mathcal{A}_2) & \text{if } \mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2, \\ \langle \{1\}, \{a\}, \emptyset, \{(1, a)\}\rangle & \text{if } \mathcal{A} = a. \end{cases}$$

Proposition 2. *If $\mathcal{A}$ is an action history, $\boldsymbol{P}(\mathcal{A})$ is a pomset.*

Proof. Let a be any action. First, if action history $\mathcal{A} = a$, then $\mathbf{P}(\mathcal{A})$ is the 'atomic' pomset comprising the single action 'a', the empty ordering relation, and the single vertex '1' labelled with the action. Second, each incremental application of either $\prec^*$ or $\cup^*$ to two (initially 'atomic') pomsets ρ_1 and ρ_2 will also yield a pomset because *(i)* the vertices in ρ_2 will be renamed to V_3 (guaranteeing that the resulting labelling function is a surjection), and *(ii)* the *entire* cross product of V_1 and V_3 is added to the resulting ordering relation in the definition of $\rho_1 \prec^* \rho_2$, ensuring transitivity. □

Finally, Theorem 3 states that if the branches of a concurrent program are executed as part of an interleaved program, any linear extension of (the pomset of) a resulting action history will also be a linear extension of some action history that is yielded by the concurrent program. We need to use linear extensions of the former action history (as opposed to using the history directly) to account for concurrent programs that might emerge within the branches as they evolve.

Theorem 3. *Let $\mathcal{B}, \mathcal{A}$ and $P_{\|} = P_1 \parallel \ldots \parallel P_n$ be as above, and suppose $[\mathcal{B}, \mathcal{A}, P_{\|}] \xrightarrow{\text{ONE}}$. Let $[\mathcal{B}, \mathcal{A}, P_1 \mid \ldots \mid P_n] \cdot \ldots \cdot [\mathcal{B}', \mathcal{A}\,; \mathcal{A}_{|}, P']$ be a terminated execution trace. Then, there exists a transition $[\mathcal{B}, \mathcal{A}, P_{\|}] \xrightarrow{\text{ONE}} [\mathcal{B}', \mathcal{A}\,; \mathcal{A}_{\|}, P'_{\|}]$ such that $\text{LIN}(\boldsymbol{P}(\mathcal{A}_{|})) \subseteq \text{LIN}(\boldsymbol{P}(\mathcal{A}_{\|}))$.*

Proof Sketch. Consider the case $n = 2$, i.e., $P_{\|} = P_1 \parallel P_2$, and let $\mathcal{T}$ be the above trace. Let $\mathcal{A}' \in \text{LIN}(\mathbf{P}(\mathcal{A}_{|}))$ be a linear extension corresponding to $\mathcal{T}$. Consider the subcase where the first x actions in $\mathcal{A}'$ are yielded by P_1, and the next y (but not $y+1$) by P_2, with $x, y > 0$. Now consider the prefix of $\mathcal{T}$ that yields the first x actions. The prefix is also an execution trace from $[\mathcal{B}, \mathcal{A}, P_1]$ if we remove program P_2 from the prefix, i.e., we replace program $C|_P = P_1' \mid P_2$ in each configuration C in the prefix by P_1'.

Suppose the prefix ends with $C^1 = [\mathcal{B}^1, \mathcal{A}^1, P_1' \mid P_2]$. Consider the trace $\mathcal{T}'$ that starts from C^1 in $\mathcal{T}$, executes only P_2, and ends immediately on yielding the next y actions. Suppose $\mathcal{T}'$ ends with $C^2 = [\mathcal{B}^2, \mathcal{A}^2, P_1' \mid P_2']$. While belief bases $\mathcal{B}$ and $\mathcal{B}^1$ may differ, we can show that $\mathcal{T}'$ can be simulated by a trace from $[\mathcal{B}, \mathcal{A}, P_2]$ that ends with a configuration C s.t. $C|_P = P_2'$. This follows from our assumption in sec. 2 that P_1 and P_2 are

[8] We treat relations and functions as sets of ordered pairs.

unrelated. A similar reasoning can be applied to the next actions yielded by P_1' and P_2', by the resulting P_1'' and P_2'', and so on, until we build terminated execution traces for $[\mathcal{B}, \mathcal{A}, P_1]$ and $[\mathcal{B}, \mathcal{A}, P_2]$. Using the traces, we can show that the antecedent of rule $\|$ holds for $[\mathcal{B}, \mathcal{A}, P_{\|}]$, from which the theorem also follows. □

The converse of the theorem does not hold due to our constraint in sec. 2 that no other (non-concurrent) step can be executed while a concurrent program is being executed. For example, take concurrent program $P = ((a_1 \parallel a_2) \; ; \; a_3) \parallel a_4$, which also represents its action history $\mathcal{A}$. While the sequence $a_1; a_4; a_2; a_3$ is a linear extension of the pomset of $\mathcal{A}$, the sequence cannot be produced by the interleaved program corresponding to P, namely, $((a_1 \parallel a_2) \; ; \; a_3) \mid a_4$.

6 Discussion

We provide a BDI agent programming language supporting (true) concurrency. This differs from past work on similar languages, which interpret concurrency as interleaving. We support 'nested' concurrency, and a limited form of nesting between concurrency and interleaving. We prove key properties of the semantics in relation to concurrency, e.g. soundness and completeness w.r.t. action histories. Our results can be extended to develop further notions, e.g. what it means for a program to be 'more concurrent' than another.

Concurrency enables branches to be executed on separate processors or machines, yielding smaller makespans than would otherwise be possible. Concurrency also enables particular desirable action schedules (or expressing an aspect of 'user intent' [16, 21]) that would not be possible by interleaving actions, e.g. the schedule in which the recording action $r(25)$ spans over both the rotate actions $rX(2)$ and $rZ(2)$.

There are many interesting avenues to explore in future work. In particular, we could formalise other approaches to concurrency, such as the more sophisticated approach of immediately aborting execution of the remaining branches upon the failure of one branch in a concurrent program [18]. The semantics that we have presented is a crucial step toward formalising such alternative approaches, which would then enable formal comparison. We could also explore how to relax our constraint on how a concurrent program can be interleaved; enable interaction between concurrent branches, e.g. to communicate the binding assigned to a variable that is shared between multiple branches; add concurrency to other systems, e.g. AgentSpeak; model check concurrent (user supplied) JACK plans (cf. [2]); and explore what concurrency means in the presence of advanced agent constructs, e.g. the declarative goal (e.g. [19, 28]) and planning (e.g. [27]) constructs.

References

[1] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[2] Rafael H Bordini, Michael Fisher, Carmen Pardavila, and Michael Wooldridge. Model Checking AgentSpeak. In *Proc. of the International Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 409–416, 2003.

[3] Rafael H. Bordini and Jomi F. Hübner. Semantics for the Jason variant of AgentSpeak:(plan failure and some internal actions). In *Proc. of the European Conf. on Artificial Intelligence (ECAI)*, pages 635–640, 2010.

[4] Gérard Boudol and Ilaria Castellani. On the semantics of concurrency: Partial orders and transition systems. In *Proc. of the International Joint Conf. on Theory and Practice of Software Development (TAPSOFT)*, pages 123–137, 1987.

[5] Gérard Boudol and Ilaria Castellani. Concurrency and atomicity. *Theoretical Computer Science*, 59(1):25–84, 1988.

[6] Gérard Boudol and Ilaria Castellani. A non-interleaving semantics for CCS based on proved transitions. *Fundamenta Informaticae*, XI:433–453, 1988.

[7] Gérard Boudol and Ilaria Castellani. Permutation of transitions: An event structure semantics for CCS and SCCS. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 411–427. Springer Berlin Heidelberg, 1989.

[8] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. Jack intelligent agents - components for intelligent agents in Java. Technical report, Agent Oriented Software, 1999.

[9] Ahmed Chawki Chaouche, A El Fallah Seghrouchni, Jean-Michel Ilié, and Djamel-Eddine Saïdouni. A dynamical plan revising for ambient systems. *Procedia Computer Science*, 32:37–44, 2014.

[10] Mehdi Dastani. 2APL: A practical agent programming language. *Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 16(3):214–248, 2008.

[11] Giuseppe de Giacomo, Yves Lespérance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[12] Lavindra de Silva, Felipe Meneguzzi, and Brian Logan. An operational semantics for a fragment of PRS. In *Proc. of the International Joint Conf. on Artificial Intelligence, (IJCAI)*, pages 195–202, 2018.

[13] Lavindra de Silva, Sebastian Sardina, and Lin Padgham. Summary information for reasoning about hierarchical plans. In *Proc. of the European Conf. on Artificial Intelligence (ECAI)*, pages 1300–1308, 2016.

[14] Pierpaolo Degano, Rocco De Nicola, and Ugo Montanari. Partial ordering derivations for CCS. In *Fundamentals of Computation Theory*, pages 520–533. Springer Berlin Heidelberg, 1985.

[15] Pierpaolo Degano, Rocco De Nicola, and Ugo Montanari. A partial ordering semantics for CCS. *Theoretical Computer Science*, 75(3):223 – 262, 1990.

[16] Maria Fox. Natural hierarchical planning using operator decomposition. In *Proc. of the European Conf. on Planning (ECP)*, pages 195–207, 1997.

[17] Michael P. Georgeff and Francois Felix Ingrand. Decision-making in an embedded reasoning system. In *Proc. of the International Joint Conf. on Artificial Intelligence (IJCAI)*, pages 972–978, 1989.

[18] James Harland, David N Morley, John Thangarajah, and Neil Yorke-Smith. Aborting, suspending, and resuming goals and plans in BDI agents. *Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 31(2):288–331, 2015.

[19] Koen V Hindriks, Frank S de Boer, Wiebe van der Hoek, and John-Jules Ch Meyer. Agent programming with declarative goals. In *Proc. of Intelligent Agents VII. Agent Theories Architectures and Languages, Seventh International Workshop*, pages 228–243. Springer-Verlag, 2001.

[20] M. J. Huber. JAM: a BDI-theoretic mobile agent architecture. In *Proc. of the Annual Conf. on Autonomous Agents*, pages 236–243, 1999.

[21] Subbarao Kambhampati, Amol Mali, and Biplav Srivastava. Hybrid planning for partially hierarchical domains. In *Proc. of the National Conf. on Artificial Intelligence (AAAI)*, pages 882–888, 1998.

[22] D. Morley and K. Myers. The SPARK agent framework. In *Proc. of the International Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 714–721, 2004.

[23] Ingrid Nunes, CJPD Lucena, and Michael Luck. BDI4JADE: a BDI layer on top of JADE. In *Proc. of the Workshop on Programming Multiagent Systems*, pages 88–103, 2011.

[24] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, Denmark, 1981.

[25] Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proc. of the European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 42–55. Springer-Verlag, 1996.

[26] Sebastian Sardina, Lavindra de Silva, and Lin Padgham. Hierarchical planning in BDI agent programming languages: A formal approach. In *Proc. of the International Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1001–1008, 2006.

[27] Sebastian Sardina and Lin Padgham. A BDI agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 23(1):18–70, 2011.

[28] Michael Winikoff, Lin Padgham, James Harland, and John Thangarajah. Declarative & procedural goals in intelligent agent systems. In *Proc. of the International Conf. on Principles of Knowledge Representation and Reasoning (KR)*, pages 470–481, 2002.