A BDI agent programming language with failure handling, declarative goals, and planning

Sardina, Sebastian; Padgham, Lin

Sardina, S., & Padgham, L. (2011). A BDI agent programming language with failure handling, declarative goals, and planning. Autonomous Agents and Multi-Agent Systems, 23(1), 18–70.
https://doi.org/10.1007/s10458-010-9130-9
Document Version: Accepted Manuscript

Published Version: https://doi.org/10.1007/s10458-010-9130-9

Citation:
Sardina, S and Padgham, L 2011, 'A BDI agent programming language with failure handling, declarative goals, and planning', Autonomous Agents and Multi-Agent Systems, vol. 23, no. 1, pp. 18-70.

Noname manuscript No.
(will be inserted by the editor)

A BDI Agent Programming Language with Failure Handling, Declarative Goals, and Planning

Sebastian Sardina · Lin Padgham

Received: date / Accepted: date

Abstract Agents are an important technology that have the potential to take over contemporary methods for analysing, designing, and implementing complex software. The **B**elief-**D**esire-**I**ntention (BDI) agent paradigm has proven to be one of the major approaches to intelligent agent systems, both in academia and in industry. Typical BDI agent-oriented programming languages rely on user-provided "plan libraries" to achieve goals, and *online* context sensitive subgoal selection and expansion. These allow for the development of systems that are extremely flexible and responsive to the environment, and as a result, well suited for complex applications with (soft) real-time reasoning and control requirements. Nonetheless, complex decision making that goes beyond, but is compatible with, run-time context-dependent plan selection is one of the most natural and important next steps within this technology. In this paper we develop a typical BDI-style agent-oriented programming language that enhances usual BDI programming style with three distinguished features: *declarative goals*, *look-ahead planning*, and *failure handling*. First, an account that mixes both procedural and declarative aspects of goals is necessary in order to reason about important properties of goals and to decouple plans from what these plans are meant to achieve. Second, lookahead deliberation about the effects of one choice of expansion over another is clearly desirable or even mandatory in many circumstances so as to guarantee goal achievability and to avoid undesired situations. Finally, a failure handling mechanism, suitably integrated with both declarative goals and planning, is required in order to model an adequate level of commitment to goals, as well as to be consistent with most real BDI *implemented* systems.

Keywords BDI agent-oriented programming · Goal reasoning · HTN planning

The original publication is available at www.springerlink.com (DOI: 10.1007/s10458-010-9130-9).
This is an errata author-produced version (differences are marked with ✓ in the margin).

Sebastian Sardina
Department of Computer Science & Information Technology
RMIT University
Melbourne, AUSTRALIA
E-mail: sebastian.sardina@rmit.edu.au

Lin Padgham
Department of Computer Science & Information Technology
RMIT University
Melbourne, AUSTRALIA
E-mail: lin.padgham@rmit.edu.au

1 Introduction

Agents are an important technology that have the potential to take over contemporary methods for analysing, designing, and implementing complex software systems suitable for domains such as telecommunications, industrial control, business process management, transportation, logistics, and aeronautics [3, 34]. A recent industry study [4] analysing several applications claimed that the use of BDI (Belief-Desire-Intention) agent technology in complex business settings can improve overall project productivity by an average 350% to 500%.

This paper reports, in detail, on the current state of the CANPlan BDI agent-oriented programming language. CANPlan has been developed both to provide a formal specification that more closely matches powerful implemented BDI systems than existing formal specifications, and to extend the reasoning capabilities of current BDI languages in a way that can be readily incorporated into implemented systems. We focus primarily on goal-based reasoning and on integrating lookahead planning into BDI languages and systems.

Generally speaking, BDI agent-oriented programming languages are built around an explicit representation of beliefs, desires, and intentions. A BDI architecture addresses how these components are represented, updated, and processed to determine the agent's actions. There are a number of agent programming languages and development platforms in the BDI tradition, such as PRS [33] and dMARS [23], AgentSpeak and Jason [6, 53], JADEX [52], 3APL [14, 30] and 2APL [12], GOAL [16], Jack [9], SRI's SPARK [45], and JAM [31].

The concept of a *goal* is central to both agent theory and agent-oriented programming. Rational agents behave because they try to satisfy and bring about their goals—goals explain and specify the agent's proactive behaviour. In agent theory [11, 54] and planning [28], goals are interpreted in a *declarative* "goal-to-be" manner, as states of affairs to bring about (e.g., not being thirsty). In contrast, mainly due to practical concerns, agent programming languages have taken a *procedural* "goal-to-do" perspective, by seeing goals as *tasks* or *processes* that are to be completely carried out (e.g., quench thirst). As a consequence, the level of support for representing and reasoning about goals has not been commensurate with their importance in these languages. Said so, the need to conveniently accommodate declarative aspects of goals in these languages has recently attracted much attention. As argued by van Riemsdijk et al. [68], even a limited account of declarative goals can help decouple plan execution and goal achievement [16, 57, 73], facilitate goal dynamics [60, 67] and sophisticated plan failure handling [32, 73], enable reasoning about goal and plan interaction [62], and enhance goal and plan communication [44].

In this paper, we show how to provide an account of goals that goes beyond the purely procedural view by accommodating some *declarative* aspects of goals, without compromising the effectiveness of the overall BDI system. In addition, we describe a built-in goal-failure recovery mechanism that *(i)* captures the failure handling typical of most implemented BDI systems; *(ii)* is compatible with our account of declarative goals; and *(iii)* provides a commitment to goals, which we call a "*flexible*" strategy, that is conceptually between Rao and Georgeff [54]'s well-known single-minded and open-minded strategies.

When it comes to means-end analysis, that is *how* goals are achieved, BDI frameworks rely entirely on online context sensitive subgoal expansion, *acting as they go*. In fact, BDI systems execute by incrementally "expanding" goals, by using libraries of hierarchical and predefined plans indexed by goals that are meant to encode the typical tasks within the domain. While this execution mechanism facilitates the development of systems that are reactive and responsive to (changes in) the environment, there are times when lookahead deliberation (i.e., hypothetical reasoning) about the effects of one choice of expansion over another is clearly desirable, or even mandatory in order to guarantee goal achievability (e.g., when

important resources may be used). Currently, applications requiring this kind of lookahead *explicitly* encode the necessary reasoning, prior to actually acting, in an ad-hoc manner. A built-in planning mechanism, usable by the agent programmer and fully integrated within the BDI architecture, would provide such functionality in a generic and principled manner. Based on the recent advances in the field of *automated planning* [27, 28, 42, 46], judicious use of planning within a BDI language could be expected to improve the usefulness of the language for developing complex systems, as recently argued by Nau [46].

Considering the many similarities between BDI programming languages and Hierarchical Task Network (HTN) planning [19, 24, 71], we *formally* define how HTN planners can be integrated into a BDI architecture. Specifically, we show that the HTN process of systematically substituting higher-level *tasks* until concrete actions are derived is analogous to the way in which a BDI-based interpreter pushes new plans onto an intention structure, replacing an achievement *goal* with an instantiated *plan*. By providing a built-in HTN lookahead mechanism, an agent can thus verify, in advance, a series of plan selections which can reasonably be expected to succeed in achieving the (sub)goal. In this way, the CANPlan language seamlessly combines the BDI online execution cycle and the HTN offline mechanism into a single *uniform* and *formal* framework. The HTN-style approach to planning is appealing in our context because of its solid formal foundations [26], its several competitive implementations (e.g., SHOP2, JSHOP, UMCP, SIPE-2, etc.), and its well-known similarities with reactive-type execution systems [71]. Furthermore, the formal specification of HTN planning within CANPlan formally justifies the "interfacing" of available HTN planning systems to existing agent platforms, as was done in the practical work reported in [20].

The work reported here builds and is based on previously published one within our group [20, 56, 59, 73]. In particular, the BDI failure handling mechanism and that of declarative goals were first proposed in [73]; whereas the formal incorporation of HTN planning into a BDI language was first done in [59]. Still, this paper refines and extends such works, including the details for handling a language with variables (rather than restricting to a propositional language), providing an account of external events, and a more powerful mechanism for dropping goals. What is more, this paper provides more details about the motivations for definitions, the analysis of the properties, and the related work.

In the following, we present CANPlan in an incremental fashion by gradually building from an account which is conceptually approximately equivalent to AgentSpeak with failure handling (Section 2); then adding a more nuanced account of goals (Section 3); and finally incorporating HTN planning (Section 4). We believe this presentation is sufficiently detailed that it facilitates modification of existing BDI platforms to incorporate the aspects specified. We have in fact largely done this by extending Jack. Implementation issues regarding the particular features of CANPlan are addressed in Section 6. We conclude the paper by discussing related work (Section 7) and future extensions to the language (Section 8).

2 The Core BDI Language

We start by defining the basic agent language that shall be used throughout the paper. This language is based on that introduced in [73] and it accommodates the core features of BDI programming, therefore resembling AgentSpeak [6, 53], probably the best known BDI programming language. We shall call this language CAN$^{\mathcal{A}}$.[1]

[1] CAN stands for **C**onceptual **A**gent **N**otation; the superscript $\mathcal{A}$ refers to AgentSpeak.

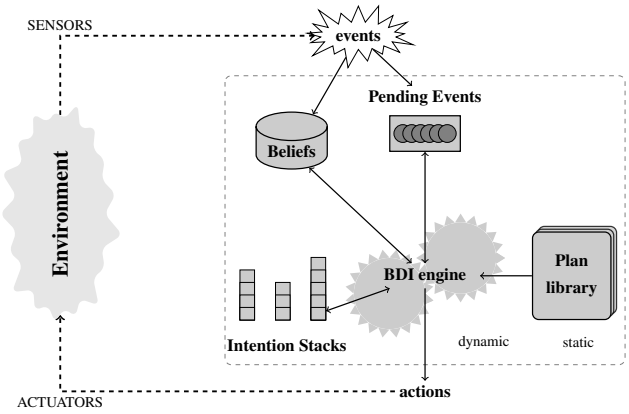

Fig. 1 A typical BDI Agent System Architecture.

In a nutshell, a BDI system—see Figure 1—responds to *events*, the inputs to the system, by selecting a plan from the *plan library*, and placing it into the *intention base*, thus committing to the plan-strategy for responding to the event-goal in question. The execution of the chosen plan may, in turn, post new subgoal events to be achieved. The plan library stands for a collection of pre-defined *hierarchical plans* indexed by goals (i.e., events) and representing the standard operations of the domain. Because an event goal may be resolved in different ways at runtime, BDI programming has often been regarded as "*implicit programming*" [4]. Flexibility is obtained from the fact that different (plan) *choices* could be made at various stages of execution based on the current environment state. Robustness is achieved by trying all available (applicable) plan options to achieve unresolved events; if there is no successful way to achieve a step, then different options are tried at more abstract levels. A crucial point in BDI systems is that execution happens at each step. The assumption is that the use of plans' preconditions to make choices as late as possible, together with the built-in mechanism for retrying alternative options upon failure, will usually ensure that a successful execution eventually ensues, even in the context of changes in the environment.

Besides capturing the standard features of BDI architectures, the formal BDI language that we describe in this section has a few unique characteristics. First, like 3APL and unlike AgentSpeak, CAN$^{\mathcal{A}}$ has a modular operational semantics that separates the execution of a single intention from that of the whole agent. Technically, this is achieved by using two different type of transition systems—one for capturing the evolution of an agent and one for capturing the evolution of an intention—rather than a single transition system, as it is the case with AgentSpeak. This facilitates the incremental extension or modification of the language, as later done in Sections 3 and 4. For instance, one can alter the top-level execution cycle without modifying the semantics of the basic language constructs.

More importantly, CAN$^{\mathcal{A}}$ includes a built-in failure handling mechanism that is consistent with most real BDI implemented platforms, such as Jack [9], dMARS [23], and even 3APL [14], and was first defined in [73]. Informally, when a (sub)goal cannot be achieved by a certain means, alternative means may be tried. A (sub)goal *fails* when all possible strategies are attempted with no success, in which case failure is propagated to higher-level motivating goals. As a consequence, achievement event-goals enjoy, by default, a certain degree of commitment, in that the agent will try as much as possible to resolve the goal successfully. In contrast with the language given in [73], CAN$^{\mathcal{A}}$ excludes declarative goals, which shall be

introduced in Section 3 as a modular extension, handles a first-order language with variables, and accommodates for *external* events (belief update or event goals).

2.1 Syntax

An agent is simply specified by its name $\mathcal{N}$, its initial belief base $\mathcal{B}$, its plan library Π, and its action description library Λ. Generally speaking, an agent is built around three type of atoms, namely, events e, basic beliefs b, and actions act. Belief formulas are built from basic beliefs using the usual logical connectors and are denoted ϕ, ψ, γ, etc. Similarly, programs are built from actions and complex constructs (see below). We write $\phi(\vec{x})$, $e(\vec{x})$, $b(\vec{x})$, $act(\vec{x})$, and $P(\vec{x})$ to state that all the free variables in the formula ϕ, event e, belief b, action act, and program P, respectively, are among vector of variables $\vec{x}$. Term and vector of terms are denoted t and $\vec{t}$, respectively. We write $\phi(\vec{t})$ to denote the formula $\phi(\vec{x})$ with variables $\vec{x}$ instantiated with terms $\vec{t}$ (similar notation applies for events, beliefs, actions, and programs), All this notation will also be used with annotations.

The *belief base* $\mathcal{B}$ of an agent—encoding what the agent believes about the world—is a set of ground atoms *facts* (e.g., *At*(*Home*)). Operations exist to check whether a condition ϕ, a logical formula over the agent's beliefs, follows from a belief set (i.e., $\mathcal{B} \models \phi$), and to add and delete a ground basic belief b to and from a belief base (i.e., $\mathcal{B} \cup \{b\}$ and $\mathcal{B} \setminus \{b\}$, resp.).[2]

The agent's *plan library* Π—encoding the typical operational procedures of the domain—consists of a collection of plan rules of the form

$$e(\vec{t}) : \psi(\vec{x_t}, \vec{y}) \leftarrow P(\vec{x_t}, \vec{y}, \vec{z}).$$

Here, $e(\vec{t})$ is the plan rule's *triggering event*, $\psi(\vec{x_t}, \vec{y})$ its *context condition*, with $\vec{x_t}$ denoting all free variables in terms $\vec{t}$, and $P(\vec{x_t}, \vec{y}, \vec{z})$ its *plan-body program*—P is a reasonable strategy to follow when ψ is believed true in order to resolve/achieve event e. Variables $\vec{y}$ are those free variables not appearing in the triggering event but introduced in the context condition, generally to bind objects that are to be used in the plan-body program P. Similarly, free variables $\vec{z}$ are those appearing in the program P, but not in the context condition or the triggering event (usually introduced in tests or event postings; see below).

The programs in plan rules belong to the following so-called *user program language*:

act	primitive action
$+b$, $-b$	add/delete belief atom
$?\phi$	tests for condition
$!e$	event goal
$P_1; P_2$	sequence
$P_1 \| P_2$	interleaved concurrency

In the *full program language*, there are also a number of auxiliary program forms that are used internally when assigning semantics to constructs, namely:

nil	basic (terminating) program
$P_1 \rhd P_2$	try P_1 with P_2 as backup
$e : (\!\{\psi_1 : P_1, \ldots, \psi_n : P_n\}\!)$	choice of plan / relevant plans for e

[2] Although most practical BDI systems take this type of database-like approach to beliefsets, one could in principle use more expressive knowledge representation formalisms, as long as operations are provided for checking conditions and updating belief bases. For example, Alechina et al. [1] explores more general but still tractable belief revision approaches for BDI agents.

Program *nil* is the empty program stating that nothing is left to execute; program $P_1 \rhd P_2$ states to execute P_1, falling back to executing P_2 if P_1 cannot execute further; and lastly program $e{:}(\!|\{\psi_1 : P_1, \ldots, \psi_n : P_n\}|\!)$ is used to encode a set of guarded plans for event e.

Finally, the *action description library* Λ contains STRIPS-style operators of the form $act(\vec{x}) : \psi(\vec{x}) \leftarrow \Phi^+(\vec{x}); \Phi^-(\vec{x})$, one for each action type in the domain. Formula $\psi(\vec{x})$ corresponds to the action's precondition (i.e., conjunction of literals), and $\Phi^+(\vec{x})$ and $\Phi^-(\vec{x})$ stand for the add and delete lists of atoms, respectively.

2.2 Semantics

The semantics of the language states what it means to execute an agent, that is, it specifies what are the legal executions of an agent. A standard notation for semantics of programming language is Plotkin's structural single-step operational semantics [51].

A *transition relation* $\longrightarrow$ on so-called *configurations* is defined by a set of derivation rules. A transition $C \longrightarrow C'$ specifies that evolving configuration C a *single step* yields configuration C'. We write $C \longrightarrow$ to state that there exists C' such that $C \longrightarrow C'$, $C \not\longrightarrow$ to state that there is no such C', and $\xrightarrow{*}$ to denote the reflexive transitive closure of $\longrightarrow$. A labelled transition is written as $C \xrightarrow{\ell} C'$, where ℓ is the transition label. When no label is stated, we assume that all labels apply. A *derivation rule* consists of a, possibly empty, set of premises, which are transitions together with some auxiliary conditions, and a single transition conclusion derivable from these premises. (See [29, 51] for details on operational semantics for programming languages.)

Before we continue, we shall point out that even though the BDI languages that we shall discuss in this paper do allow variables, as does any practical language, we will first present the semantics for their non-variable fragments. The reason for this is legibility: the treatment of variables requires substantial technical notation and additional complexity that makes the material much more cumbersome. We discuss the extensions required to accommodate variables in Section 5.

So, a CAN$^{\mathcal{A}}$ *agent configuration*, or simply an *agent*, is defined by a tuple of the form $\langle \mathcal{N}, \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma \rangle$ consisting of the agent name $\mathcal{N}$, a plan library Π, an action description library Λ, a belief base $\mathcal{B}$, the sequence of actions $\mathcal{A}$ executed so far by the agent, and the intention base Γ. An intention I is a tuple $\langle id, P \rangle$, where $id \in \mathbb{N}$ is the (unique) intention identifier and P is a program term in the full program language. The *intention base* then is the set of active intentions that the agent is currently pursuing. Sometimes we will need to add one or more plan-body programs in the intention base, as new intentions.

Definition 1. Let Γ be an intention base and γ be a set of ground plan-body programs. Intention base $\Gamma \Cup \gamma$ denotes the intention base resulting from incorporating each $P \in \gamma$ into intention base Γ, as a new intention of the form $\langle id, P \rangle$, where id is the intention's *unique* identifier (i.e., no other intention in $\Gamma \Cup \gamma$ shares the same identifier). ■

The semantics of CAN$^{\mathcal{A}}$ is designed in two layers, by means of two types of transitions. The first transition $\longrightarrow$ states what it means to evolve a single intention and is defined in terms of *intention-level configurations* of the form $\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, P \rangle$ consisting of the agent's plan and action libraries Π and Λ, respectively, its belief base $\mathcal{B}$, the sequence of primitive actions $\mathcal{A}$ executed so far, and the plan-body program P being executed (i.e., the intention of interest).[3] Thus, derivation rules for $\longrightarrow$ characterize the *intention-level* execution semantics.

[3] For legibility, we will omit both Π and Λ when not explicitly required and just write $\langle \mathcal{B}, \mathcal{A}, P \rangle$.

The second type of transition $\Longrightarrow$ is between (full) agent configurations and defines the *agent-level* execution. Agent transitions are stated in terms of intention-level transitions and defines what it means to execute a complete agent.

2.2.1 Intention-Level Execution

Let us now provide the derivation rules, grouped in clusters for legibility, characterizing the legal intention-level transitions of the form $\langle \mathcal{B}, \mathcal{A}, P\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P'\rangle$.

Basic Programs Derivation rule ? deals with tests by checking that the condition follows from the current belief base, with adequate bindings if the condition is open. Rule do handles the case of primitive actions by using the domain action description library Λ. Finally, rules $+b$ and $-b$ account for the execution of belief update operations.

$$\frac{\mathcal{B} \models \phi}{\langle \mathcal{B}, \mathcal{A}, ?\phi\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \mathit{nil}\rangle}\ ? \qquad \frac{a : \psi \leftarrow \Phi^-; \Phi^+ \in \Lambda \quad \mathcal{B} \models \psi}{\langle \Lambda, \mathcal{B}, \mathcal{A}, a\rangle \longrightarrow \langle \Lambda, (\mathcal{B} \setminus \Phi^-) \cup \Phi^+, \mathcal{A} \cdot a, \mathit{nil}\rangle}\ do$$

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, +b\rangle \longrightarrow \langle \mathcal{B} \cup \{b\}, \mathcal{A}, \mathit{nil}\rangle}\ +b \qquad \frac{}{\langle \mathcal{B}, \mathcal{A}, -b\rangle \longrightarrow \langle \mathcal{B} \setminus \{b\}, \mathcal{A}, \mathit{nil}\rangle}\ -b$$

Complex Programs The following derivation rules define what it means to execute a sequential program and two interleaved concurrent programs:

$$\frac{\langle \mathcal{B}, \mathcal{A}, P_1\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P_1'\rangle}{\langle \mathcal{B}, \mathcal{A}, P_1; P_2\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P_1'; P_2\rangle}\ Seq_1 \qquad \frac{\langle \mathcal{B}, \mathcal{A}, P\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \mathcal{B}, \mathcal{A}, \mathit{nil}; P\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P'\rangle}\ Seq_2$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, P_1\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \mathcal{B}, \mathcal{A}, P_1 \parallel P_2\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P' \parallel P_2\rangle}\ \parallel_1 \qquad \frac{\langle \mathcal{B}, \mathcal{A}, P_2\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \mathcal{B}, \mathcal{A}, P_1 \parallel P_2\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P_1 \parallel P'\rangle}\ \parallel_2$$

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, \mathit{nil} \parallel \mathit{nil}\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \mathit{nil}\rangle}\ \parallel_{end}$$

Rule Seq_1 evolves a sequence by evolving its first part, while Seq_2 does it by evolving the second part of the sequence provided the first part is finished. A concurrent program may be evolved by evolving either parts (rules $\parallel_1$ and $\parallel_2$), and may be terminated if both parts are terminating (rule $\parallel_{end}$). See that a concurrent program can always execute if one of its sub-programs can execute. Thus, one branch can just "wait" by means of a test condition $?\phi$.

Event & Failure Handling The main feature of CAN$^{\mathcal{A}}$ is its detailed operational semantics for the kind of failure handling typical of implemented BDI systems, where if a plan fails, alternative plans for achieving the goal are tried, if possible. This is accomplished by combining constructs $e : (\!|\Delta|\!)$, which maintains a set of (alternative) relevant plans Δ for event e, and construct $P_1 \rhd P_2$, which tries to execute a strategy P_1 while maintaining the set of possible alternative plans to consider in P_2.

Upon an event goal posting $!e$, either internal or external, a three-stage process is started in order to handle the pending event. The first stage involves collecting the set Δ of *relevant* guarded plans of the form $\langle \psi : P\rangle$, that is, those plans from the library Π whose triggering events are able to *match* the pending event. Formally,

$$\frac{\Delta = \{\psi{:}P \mid \ \ e' : \psi \leftarrow P \in \Pi,\ e' = e\} \quad \Delta \neq \emptyset}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, !e\rangle \longrightarrow \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, e{:}(\!|\Delta|\!)\rangle}\ Event$$

As standard, the plan-body program P of a rule is included in the set of relevant plans Δ if its triggering head e' matches the actual event goal e. In that case, its guard condition in the set of alternatives is the rule's context condition ψ.

Example 1. Imagine an agent that, at some point, may need to arrange a trip to a distant destination for various reasons (e.g., attending a conference, vacation, or business). The goal/task of arranging this trip can be decomposed into various subgoals, such as arranging transportation, accommodation, insurance as well as actually traveling and returning. We imagine then the following plan rules used to address the event subgoal *travelTo*$(dest)$ to go from the current location to destination location $dest$:

$$\textit{travelTo}(dest) : \textit{At}(x) \wedge \textit{WalkDist}(x, dest) \leftarrow !\textit{prepareWalk};\ \textit{walk}(dest); ?\textit{At}(dest)$$

$$\textit{travelTo}(dest) : \textit{At}(x) \wedge \exists y(\textit{InCity}(x, y) \wedge \textit{InCity}(dest, y)) \leftarrow P_{city}(x, dest)$$

$$\textit{travelTo}(dest) : \textit{At}(x) \wedge \neg\exists y(\textit{InCity}(x, y) \wedge \textit{InCity}(dest, y)) \leftarrow P_{far}(x, dest)$$

$$\textit{travelTo}(\textit{Home}) : \textit{true} \leftarrow ?\textit{At}(x);\ P_{home}$$

The first plan rule states to walk to destination when this is close to the current location. See that such strategy requires the agent to first prepare for a walk (e.g., bring an umbrella if it is raining), by posting internally the event subgoal !*prepareWalk*. After walking, the strategy verifies that we have actually arrived to the desired destination. The second and third plan rules state that the agent should follow different strategies, represented by programs P_{city} and P_{far}, depending on whether the trip is local (e.g., take a taxi, ride a bus, or arrange for a lift) or the trip is not within the city (e.g., take a flight or a train). Lastly, the agent is equipped with a special strategy P_{home} she can follow if going home.

Suppose next that, at some point, an internal/external event goal of the form !*travelTo*$(\textit{Uni})$ needs to be resolved. In such case, rule $Event$ above would yield the following (sub)program encoding all the *relevant* options available for addressing the event:

$$\textit{travelTo}(\textit{Uni}) : (\!|\{\psi_1 : P_{walk}(\textit{Uni}),\ \psi_2 : P_{city}(x, \textit{Uni}),\ \psi_3 : P_{far}(x, \textit{Uni})\}|\!), \quad (1)$$

where

$$P_{walk}(\textit{Uni}) \stackrel{\text{def}}{=} !\textit{prepareWalk};\ \textit{walk}(\textit{Uni}); ?\textit{At}(\textit{Uni})$$

$$\psi_1 \stackrel{\text{def}}{=} \textit{At}(x) \wedge \textit{WalkDist}(x, \textit{Uni})$$

$$\psi_2 \stackrel{\text{def}}{=} \textit{At}(x) \wedge \exists y(\textit{InCity}(x, y) \wedge \textit{InCity}(\textit{Uni}, y))$$

$$\psi_3 \stackrel{\text{def}}{=} \textit{At}(x) \wedge \neg\exists y(\textit{InCity}(x, y) \wedge \textit{InCity}(\textit{Uni}, y))$$

Finally, we point out two observations. In some scenarios, the agent may be able to both walk to uni as well as take the P_{city} strategy, that is, both ψ_1 and ψ_2 may hold true in some states of affairs. Second, the strategy for going home (i.e., the last plan rule above) is not included as an option by the $Event$ rule, as its triggering event *travelTo*$(\textit{Home})$ is *not* relevant for the actual event goal *travelTo*$(\textit{Uni})$ to be resolved. ■

The second stage in the process of handling an event goal involves *selecting one applicable strategy* P_i from the set of (remaining) relevant guarded options $e : (\!|\{\psi_1 : P_1, \ldots, \psi_n : P_n\}|\!)$. A strategy option is applicable if it is relevant and its guard context condition is believed true. In that case, the rule Sel below builds a program of the form $P \rhd e : (\!|\Delta'|\!)$, where P is the chosen strategy to be tried and Δ' is the new set of remaining strategies.

$$\frac{\psi : P \in \Delta \quad \mathcal{B} \models \psi}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, e : (\!|\Delta|\!)\rangle \longrightarrow \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, P \rhd e : (\!|\Delta \setminus \{\psi : P\}|\!)\rangle}\ Sel$$

Informally, P_i becomes the so-called *current strategy* to be tried in order to address the event in question. The right hand side program in $\rhd$ encodes the *remaining alternative* strategies that could be considered, should the current strategy fail to execute. Observe that in case of failure of the current strategy, only new options not tried before are considered.

Example 2. Continuing with our above example, suppose next that the agent believes she is currently at home, which is walking distance to the university destination. Then, the plan selection rule Sel may legally transform the set of relevant strategies shown in (1) into the following program:

$$P_{walk}(\mathit{Uni}) \rhd \mathit{travelTo}(\mathit{Uni}) : (\!|\psi_2 : P_{city}(x, \mathit{Uni}), \psi_3 : P_{far}(x, \mathit{Uni})|\!). \tag{2}$$

That is, since the context condition of the first strategy is true, the agent decides to try program $P_{walk}(\mathit{Uni})$; however, she still keeps the other non-chosen strategies as "backup" alternatives (see right-hand side program in the $\rhd$ construct).

Next, the agent may execute program $P_{walk}(\mathit{Uni})$, whose first step involves resolving the subgoal of preparing for the walk. This in turn will involve the use of derivation rule $Event$ and then Sel, but now for event *prepareWalk*. The program above could then evolve to the following one:

$$P'_{walk} \rhd \mathit{travelTo}(\mathit{Uni}) : (\!|\psi_2 : P_{city}(x, \mathit{Uni}), \psi_3 : P_{far}(x, \mathit{Uni})|\!), \tag{3}$$

where $P'_{walk} \stackrel{\text{def}}{=} (P_{pw} \rhd \mathit{prepareWalk} : (\!|\Delta_{pw}|\!)); \mathit{walk}(\mathit{Uni}); ?\mathit{At}(\mathit{Uni})$ is the evolution of program $P_{walk}(\mathit{Uni})$, P_{pw} is the current strategy selected to address event *prepareWalk* and Δ_{pw} encodes the alternative, not yet selected, strategies. Note that the agent needs to carry out P_{pw}, or eventually some strategy in Δ_{pw}, before she can perform action $\mathit{walk}(\mathit{Uni})$. ■

Once an applicable strategy has been selected, it must be carried out to completion, if possible. To that end, the following derivation rules are included to execute the current strategy program one step (rule $\rhd_{step}$) and to fully finish its execution (rule $\rhd_{end}$). Recall that the current strategy is the first program in the "try" construct $\rhd$, e.g., programs P_{walk} and P'_{walk} in (2) and (3) above, respectively.[4]

$$\frac{\langle \mathcal{B}, \mathcal{A}, P_1\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}'', P'\rangle}{\langle \mathcal{B}, \mathcal{A}, P_1 \rhd P_2\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}'', P' \rhd P_2\rangle} \rhd_{step} \qquad \frac{}{\langle \mathcal{B}, \mathcal{A}, \mathit{nil} \rhd P'\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \mathit{nil}\rangle} \rhd_{end}$$

Finally, let us focus on the the third stage in handling an event, namely, the *failure recovery* of an event goal whose current strategy is not able to execute further (e.g., the agent was not able to successfully prepare for the walk, as it is raining and there is no umbrella). Technically, this may arise, for instance, when an action's precondition or a test is not met, or a subgoal event has no applicable plans. In such cases, the current strategy program P in a program of the form $P \rhd e : (\!|\Delta|\!)$—e.g., P_{walk} in program (2) above—would have no intention-level transition (i.e., neither rules $\rhd_{step}$ nor $\rhd_{end}$ apply). In that case, the event goal e may be recovered by falling back to some "backup" strategy in Δ, if any available for execution. This mechanism is exactly what the following derivation rule $\rhd_{rec}$ captures:

$$\frac{P_1 \neq \mathit{nil} \qquad \langle \mathcal{B}, \mathcal{A}, P_1\rangle \not\longrightarrow \qquad \langle \mathcal{B}, \mathcal{A}, P_2\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}'', P'_2\rangle}{\langle \mathcal{B}, \mathcal{A}, P_1 \rhd P_2\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}'', P'_2\rangle} \rhd_{rec}$$

[4] The "try" construct $\rhd$ should not be understood as a concurrency constructs. Though it may resemble constructs like ConGolog's prioritized concurrency construct $\rangle\!\rangle$, the intended meaning of $P_1 \rhd P_2$ is not to execute both programs to completion, but only one of them (and preferably P_1).

Notice that, because in our case P_2 was constructed via rules $Event$ and Sel, it ought to be of the form $e : (\!|\Delta|\!)$. Thus, the requirement that P_2 evolves to P_2' implies that there is an alternative strategy P' in Δ that is *applicable at the current situation*—P_2 shall evolve to P_2' due to rule Sel. Otherwise, if Δ contains no applicable option, it makes sense for the agent to simply "wait," rather than drop its current strategy, only to discover *afterwards* that there is currently no better option. Moreover, since rule Sel does not include already tried strategies into the set of remaining strategies Δ, the new selected strategy program P' may not be the same as previous tries.[5]

Example 3. Returning to our example, imagine that, for some reason, the agent was not able to successfully prepare for the walk and was therefore unable to resolve the first step in strategy P_{walk}, namely, it was unable to find a successful plan for resolving event *prepareWalk*.

At that point, program P_{walk} is not able to execute and, as a result, recovery rule $\rhd_{rec}$ can be applied to the program shown in equation 2: program P_1 becomes the current blocked walking strategy, whereas P_2 becomes the right hand side of the program in equation 2. Since the university is indeed within the same city, the guard condition for alternative strategy P_{city} does apply and as a result one application of rule $\rhd rec$ would yield the following program:

$$P_{city}(\mathit{Home}, \mathit{Uni}) \rhd \mathit{travelTo}(\mathit{Uni}) : (\!|\psi_3 : P_{far}(x, \mathit{Uni})|\!). \tag{4}$$

Now the agent will try to go to university by following the P_{city} strategy (e.g., taking a taxi). Observe also how the remaining alternatives are updated to those ones that have not been tried so far. ■

Putting it all together, then, by suitably combining constructs $e : (\!|\Delta|\!)$ and $\rhd$, we are able to model the desired *plan selection* and *failure handling* mechanisms for event goals.

2.2.2 Agent-Level Execution

On top of the above intention-level rules, we characterize the evolution of an agent who is pursuing multiple goals and intentions concurrently. Since we do not discuss multi-agent features in this article, we shall drop the agent name $\mathcal{N}$ and just write agent configuration as $\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma \rangle$. When C is an agent configuration, we use $C[X]$ to refer to component X of C (e.g., $C[\mathcal{B}]$ and $C[\Gamma]$ stand for the belief base and intention base of agent C, respectively).

Top-Level Agent Execution The top-level semantics for our language closely matches Rao and Georgeff's abstract interpreter for intelligent rational agents [55] which, roughly speaking, requires the following three steps: *(i)* select an intention and execute a step; *(ii)* incorporate any pending external events; and finally *(iii)* update the set of goals and intentions.

Technically, in defining the single-step evolution of a $\mathsf{CAN}^{\mathcal{A}}$ agent, three auxiliary agent-level transition types capturing the above three steps, namely int, event, and goal, are used: (Note that goal-type transitions are defined in terms of *pairs* of agent configurations.)

$$\frac{C \overset{\mathsf{int}}{\Longrightarrow} C_1 \quad C_1 \overset{\mathsf{event}}{\Longrightarrow} C_2 \quad \langle C, C_2 \rangle \overset{\mathsf{goal}\,*}{\Longrightarrow} \langle C, C' \rangle \quad \langle C, C' \rangle \overset{\mathsf{goal}}{\not\Longrightarrow}}{C \overset{\mathsf{CAN}^{\mathcal{A}}}{\Longrightarrow} C'} \; A_{\mathsf{CAN}^{\mathcal{A}}}$$

[5] It is straightforward to modify rule Sel so as to keep the chosen strategy in the set of remaining alternatives, thus allowing the agent to try previously failed programs. One can even let the programmer decide which mechanism should be used for a particular case, by having different event posting constructs. Our particular choice here is mostly motivated by the fact that real BDI platforms do discard previously failed strategies, since, by doing so, they avoid cases in which the agent is stuck failing the same plans over and over.

That is, an agent step involves a step on some active intention, followed by the assimilation of all external events that have occurred during the execution cycle (including information from sensors), and finally followed by a complete update of its goals. Informally, the last two requirements state that the agent should perform as many goal update transitions as possible, i.e., until no more goal updates can be done (in agent configuration C'). Observe that the whole goal update process is broken into a sequence of atomic goal update steps. In order to have access to the *original* agent configuration C at the start of the agent execution cycle, goal update transitions are defined in terms of *pairs* of configurations, where the first component (i.e., the original agent configuration) is meant to be propagated without any change.

Below, we develop in detail the three kinds of transitions used in the main agent rule.

Intention Selection & Execution The first step in an agent cycle involves selecting an active intention from the intention base and evolving it one step. An intention can evolve/execute by making a legal intention-level step, as defined by the derivation rules explained before. This may result in some internal reasoning (e.g., an application of derivation rule Sel) and even the execution of an action. Formally,

$$\frac{\langle id, P\rangle \in \Gamma \quad \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, P\rangle \longrightarrow \langle \Pi, \Lambda, \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma\rangle \overset{\mathsf{int}}{\Longrightarrow} \langle \Pi, \Lambda, \mathcal{B}', \mathcal{A}', (\Gamma \setminus \{\langle id, P\rangle\}) \cup \{\langle id, P'\rangle\}\rangle} \; A_{\mathsf{int}}$$

In our example, we could imagine the agent having, at some point, an intention of the form $\langle \#4, !\mathit{travelTo}(\mathit{Uni})\rangle$ in order to go to the university. If such intention is chosen and rule A_{int} is applied, the new intention base will contain an intention of the form $\langle \#4, P_1\rangle$, where P_1 is the program shown in (1) above. Yet another step on such transition would yield intention $\langle \#4, P_2\rangle$, where P_2 is the program shown in (2), and so on.

It could be claimed that a blocked intention may eventually become unblocked and, hence, the agent should just "wait" rather than abandon it. We argue, though, that for this to be a general principled approach, the programmer should made this "waiting" explicit (e.g., wait for some change in the environment that is expected to happen). It is straightforward to include a variation of the test construct $?\phi$ to that end, namely a construct $??\phi$ that would cause an intention to wait for a condition without being blocked.

Incorporating External Events Events originated from the environment have to be assimilated by the agent at every cycle. Their rather informal treatment in many existing formal programming languages (e.g., [5, 23, 53, 73]) makes it difficult to prove properties relative to the external world.

External events may account for sensor information, which changes the agent's beliefs about the world,[6] or external achievement event goals that the agent must react to (e.g., a request from another agent). We shall distinguish three types of events: *(i)* $!e$ stands for an external (achievement) event; *(ii)* $+b$ stands for the sensor input that b is true; and *(iii)* $-b$ stands for sensor input that b is false. The set of all possible ground events in the domain is denoted Events, whereas the set of all possible agent configurations is denoted Confs. An (external) environment is defined as follows.

Definition 2 (Environment). An *environment* is a total function $\mathcal{E} : \mathit{Confs} \mapsto 2^{\mathit{Events}}$ such that for every $C \in \mathit{Confs}$ and ground atom b, if $+b \in \mathcal{E}(C)$, then $-b \notin \mathcal{E}(C)$. ■

[6] Original versions of CAN/CANPlan [59, 73] did not deal with belief updates from sensor inputs; only plans could update the agent's beliefs.

First, note that environments (or their models) are assumed, at the cognitive level at least, to be always *consistent*. Second, observe that each function $\mathcal{E}$, being deterministic, stands for *one possible* environment. Therefore, the semantics obtained for our language is predicated on such environment, as is the case in other works (e.g., [40, 58]). To reason about executions under incomplete information on the environment, one needs then to quantify over the relevant set of environment functions $\mathcal{E}$. Finally, environments are defined in terms of agent configurations, so that external events are allowed to occur while the agent is performing *internal* practical reasoning. We are in fact interested in studying the behaviour of the agent system if, for example, an unexpected event arises while the agent is building the set of relevant plans for some goal or updating its beliefs.[7] For example, $\{+At(Uni), -At(c)\} \subseteq \mathcal{E}(C)$, when $C[\mathcal{B}] \models At(c)$, may apply due to the agent receiving location information from its GPS.

Assimilating all external information amounts to updating the belief base with the new information received as well as updating the intention base to accommodate all new events. So, relative to a (consistent) environment, we define the following—always applicable—derivation rule to handle external events.[8]

$$\frac{\mathcal{B}' = (\mathcal{B} \setminus \{b \mid -b \in \mathcal{E}(C)\}) \cup \{b \mid +b \in \mathcal{E}(C)\} \quad \gamma^! = \{!e \mid !e \in \mathcal{E}(C)\}}{C = \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma\rangle \overset{\mathsf{event}}{\Longrightarrow} \langle \Pi, \Lambda, \mathcal{B}', \mathcal{A}, \Gamma \Cup \gamma^!\rangle} \; A_{\mathsf{ev}}$$

(Recall operation $\Gamma \Cup \gamma$ incorporates all programs in γ into intention base Γ; Definition 1.)

Thus, the external events corresponding to sensing information are used to update the beliefs of the agent, whereas the set of *achievement* event goals $\gamma^!$ from the environment (e.g., messages from other agents) are incorporated as new intentions.

Goal Updates Finally, let us develop the derivation rules for transitions $\overset{\mathsf{goal}}{\Longrightarrow}$ in charge of performing goal updates. Recall that in order to carry the original agent configuration C_{init} at the start of the agent execution cycle, this transition is defined in terms of *pairs* of configurations $\langle C_{init}, C\rangle$, where configuration C_{init} is always kept intact.

We consider two type of goal updates. Recall that in languages like CAN$^{\mathcal{A}}$, goals are represented with events, i.e., tasks the agent ought to perform. The first type of update involves terminating those top-level goals, that is, intentions, that cannot execute further, either because they are totally blocked or because they have actually executed successfully to completion (i.e., $P = nil$). In such case, the intention is simply dropped from the intention base:

$$\frac{\langle id, P\rangle \in \Gamma \quad \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, P\rangle \not\longrightarrow}{\langle C_{init}, \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma\rangle\rangle \overset{\mathsf{goal}}{\Longrightarrow} \langle C_{init}, \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma \setminus \{\langle id, P\rangle\}\rangle\rangle} \; A^1_{\mathsf{goal}}$$

Notice that for this kind of update, the original agent configuration C_{init} is *not* used.

The second type of update involves generating new event goals due to changes in the beliefs of the agent. Like AgentSpeak [5, 53] and many BDI platforms, new distinguished achievement event goals—here of the form $!{+}b$ and $!{-}b$—are created for every *actual* belief update that has happened in the current agent cycle. [9]

[7] Since the history of world altering actions is part of the agent configuration, an environment defined only in terms of (observable) external world, as done in [40] for instance, can easily be represented.

[8] For simplicity, we keep the environment $\mathcal{E}$ global and implicit. Technically, $\mathcal{E}$ should be part of the agent configuration. In fact, one could easily follow [5] and define transitions between pairs of *system configurations* $\langle \mathcal{E}, C\rangle$, where $\mathcal{E}$ is an environment "circumstance" and C is an agent configuration.

[9] Though we kept notation as similar as possible to AgentSpeak, there are still a few syntactic differences when it comes to events. Since our language handles failure at the semantic level, we need only three kinds of *triggering* events: e (for simple achievement goals; equivalent to $+!e$ in AgentSpeak), and $+b$ and $-b$ (for belief update goals; as in AgentSpeak). AgentSpeak has also $-!e$ and $-?b$ for handling failure.

This allows agents, if necessary, to react to such belief changes by including the corresponding plans in the plan library. In the Jack [9] BDI platform, for instance, this mechanism is referred to as *automatic events*. Observe that the rules below create a new intention of the form $!{+}b$ or $!{-}b$ provided there are relevant plans for the belief update event (last constraint) and that the agent is not already working on such event goal (fourth constraint).

$$\frac{C_{init}[\mathcal{B}] \not\models b \quad \mathcal{B} \models b \quad e = +b \quad e \notin \mathcal{EG}(\Gamma) \quad \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, !e\rangle \longrightarrow}{\langle C_{init}, \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma\rangle\rangle \overset{\mathsf{goal}}{\Longrightarrow} \langle C_{init}, \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma \Cup \{!e\}\rangle\rangle} A^2_{\mathsf{goal}}$$

$$\frac{C_{init}[\mathcal{B}] \models b \quad \mathcal{B} \not\models b \quad e = -b \quad e \notin \mathcal{EG}(\Gamma) \quad \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, !e\rangle \longrightarrow}{\langle C_{init}, \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma\rangle\rangle \overset{\mathsf{goal}}{\Longrightarrow} \langle C_{init}, \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma \Cup \{!e\}\rangle\rangle} A^3_{\mathsf{goal}}$$

where $\mathcal{EG}(\Gamma) = \{e \mid$ there exists $\langle id, P\rangle \in \Gamma$ such that $P \in \{!e, e{:}(\!|\Delta|\!),\ P' \rhd e{:}(\!|\Delta|\!)\}\}$ is the set of top-level event goals in the intention base Γ. Observe how agent configuration C_{init} is used above to check what was believed originally (i.e., at the start of the agent cycle). Note also that the process of generating a belief update event is completely independent from the actual source of the corresponding belief change. Such belief change may have been directly linked to sensing information or the consequence of a complex belief revision in rule A_{ev}, when considering operations $\cup$ and $\setminus$ on belief bases as general belief change operators.

With the set of derivation rules defined, we can define the meaning of an agent execution.

Definition 3 (BDI Agent Execution). A *t-BDI execution* E of an agent C in language t (relative to an environment $\mathcal{E}$) is a, possibly infinite, sequence of agent configurations $C_0 \cdot C_1 \cdot \ldots$ such that $C_i \overset{\mathsf{t}}{\Longrightarrow} C_{i+1}$, for every $i \geq 0$. A *terminating* execution is a finite execution $C_0 \cdot \ldots \cdot C_n$ where $C_n[\Gamma] = \{\}$, that is, all intentions have been successfully completed. ■

For example, we take $\mathsf{t} = \mathsf{CAN}^{\mathcal{A}}$ to define the possible executions for the core BDI language $\mathsf{CAN}^{\mathcal{A}}$ arising from the intention-level and agent-level derivation rules presented above.

An intention is blocked when it is not possible to evolve it one step further.

Definition 4 (Blocked Program/Intention). A program P is *blocked* in an agent configuration $\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma\rangle$ iff $\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, P\rangle \not\longrightarrow$. An intention $I = \langle id, P\rangle$ is *blocked* in an agent configuration if program P is blocked in such configuration. ■

Finally, we define different ways an intention can execute throughout a BDI agent execution. As with configurations, $I[X]$ denotes component X in intention I: $I[id]$ stands for I's identifier and $I[P]$ for I's program.

Definition 5 (Intention Execution). Let $E = C_0 \cdot C_1 \cdot \ldots \cdot C_n$ be a BDI execution for an agent C_0. Intention $I \in C_0[\Gamma]$ in C_0 has been *fully executed* in E if there is no $I' \in C_n[\Gamma]$ such that $I'[id] = I[id]$; otherwise I is said to be *executing* in E. In addition, we say that intention I has been *successfully* executed in E if $\langle I[id], \mathit{nil}\rangle \in C_i$, for some $i \leq n$; and I has *failed* in E if it has been fully but not successfully executed in E. ■

So, a fully executed intention is one that has been removed from the intention base; and an intention has been successfully executed when it has reached the empty *nil* program. Consequently, for an agent to *fail* an intention by dropping it before it is finished, the intention must have been unable to execute (i.e., blocked) somewhere along the execution.

2.3 On the commitment of CAN$^{\mathcal{A}}$ to goals

As the reader has already noticed, CAN$^{\mathcal{A}}$ is basically a typical BDI-style agent programming language, close to languages like AgentSpeak and Jason, PRS, and even 3APL and its variants. Probably the characteristic feature of CAN$^{\mathcal{A}}$ is its *built-in failure handling mechanism* consistent with most real BDI implemented systems such as dMARS [23] and Jack [9]. When a subgoal cannot be achieved by a certain mean, alternative applicable means are considered and tried, via derivation rules Sel and $\rhd_{rec}$. A (sub)goal *fails* when all possible strategies are attempted with no success; and thus failure is propagated to higher-level motivating goals. As a consequence, achievement event goals $!e$ enjoy, by default, a certain degree of commitment: the agent will try as much as possible to resolve them successfully. In what follows, we make all this precise and set the technical stage for the next two sections.

We start by identifying what an "active" goal is within an CAN$^{\mathcal{A}}$ agent, that is, one that the agent is pursuing with some commitment. An active goal arises when an event goal $!e$ is "adopted" by means of derivation rule $Event$. As typical of most BDI programming languages, goals in CAN$^{\mathcal{A}}$ have a strong *procedural* flavor based around the concept of *events*.

Definition 6 (Active Event Goal). An *active event-goal* is a program G of the form $e:(\!|\Delta|\!)$ or $P \rhd e:(\!|\Delta|\!)$. The event of G is denoted $G[e]$. Program P, if any, is G's *current strategy* and $e:(\!|\Delta|\!)$ is G's set of *alternative strategies*. A goal G *has an alternative applicable strategy* in an agent C if $e:(\!|\Delta|\!)$ is not blocked in C. Goal G is *fully blocked* in C if G's current strategy, if any, is blocked in C and G has no alternative applicable strategy in C. ■

Observe that a program of the form $!e$ is *not* considered an active goal, as it has not yet been "adopted," that is, started, by the agent.

To resolve a (top-level) goal, an intention often needs to work on several subgoals in a *hierarchical* manner: some goals are pursued as mere *instruments* for other higher-level goals. The following concept captures this formally.

Definition 7 (Active Goal Trace, for CAN$^{\mathcal{A}}$ agents). An *active goal trace* λ is a sequence of active goals $G_1 \cdot \ldots \cdot G_n$. The multiset of all active goal traces in a program P, denoted $\mathcal{GT}race(P)$, is inductively defined as follows:[10]

$$\mathcal{GT}race(P) = \begin{cases} \emptyset & \text{if} \quad P = \mathit{nil} \mid \mathit{act} \mid ?\phi \mid +b \mid -b \\ \{P \cdot \lambda \mid \lambda \in \mathcal{GT}race(P_1)\} & \text{if} \quad P = P_1 \rhd e:(\!|\Delta|\!), \mathcal{GT}race(P_1) \neq \emptyset \\ \{P\} & \text{if} \quad P = P_1 \rhd e:(\!|\Delta|\!), \mathcal{GT}race(P_1) = \emptyset \\ \{P\} & \text{if} \quad P = e:(\!|\Delta|\!) \\ \mathcal{GT}race(P_1) & \text{if} \quad P = P_1; P_2 \\ \mathcal{GT}race(P_1) \uplus \mathcal{GT}race(P_2) & \text{if} \quad P = P_1 \| P_2 \end{cases}$$

The set trivially extends to intentions as $\mathcal{GT}race(\langle id, P\rangle) = \mathcal{GT}race(P)$. ■

Informally, an active goal trace represents a chain of subgoals that are active in an intention. The *k-th subgoal* in an active goal trace λ is the k-th element in λ, and is denoted with $\lambda[k]$ (where $\lambda[k] = \epsilon$ when $k > |\lambda|$). So, we say that the $(n+1)$-th subgoal in λ is a *subsidiary* goal for the *motivating* n-th subgoal in λ. The *set of all active goals* in a program P (intention $I = \langle id, P\rangle$) is defined as $\mathcal{G}(P) = \{\lambda[k] \mid \lambda \in \mathcal{GT}race(P), k \geq 1\}$ ($\mathcal{G}(I) = \mathcal{G}(P)$).

[10] Observe we need multisets, rather than sets, because the agent may be pursuing the same hierarchy of goals more than once in different concurrent programs. Here, operation $\uplus$ is the the multiset union. For details on multisets and their operations we refer to [38, page 483].

Observe that, because for an event goal to be active it must have been previously *adopted*—by means of intention-level derivation rule $Event$—there can be no active goal within the second part of a sequence $P_1; P_2$, as P_2 has not yet been started (fourth case above). Note also that, due to potential concurrent execution of programs (fifth case above), an intention may give rise to *several* active goal traces—an intention free of concurrency, though, always has (at most) one active goal trace. Indeed, an active goal may have multiple subsidiary active sub-goals; a single trace corresponds to *one* hierarchical chain of goals and sub-goals.

Example 4. Following our example, the program (3) above (pp. 9) has only one active goal trace $\lambda = G_1 \cdot G_2$, where

$$\begin{aligned} G_1 &= P'_{walk} \rhd \mathit{travelTo}(\mathit{Uni}) : (\!|\{\psi_2 : P_{city}(x, \mathit{Uni}), \psi_3 : P_{far}(x, \mathit{Uni})\}|\!); \\ G_2 &= P_{pw} \rhd \mathit{prepareWalk} : (\!|\Delta_{pw}|\!). \end{aligned}$$

The first active goal G_1 in trace λ is indeed the whole program shown in (3), which stands for how the agent is handling event-goal *travelTo*(*Uni*). The second active goal G_2, in turn, stands for how the agent is handling goal event *prepareWalk*, which is instrumental to the higher-level goal *travelTo*(*Uni*). See that goal G_2 only accounts for those subgoals in program P'_{walk} that are already *active*.

Next, imagine that program P_{pw} involves the concurrent execution of two subgoal events followed by another subgoal event, that is, $P_{pw} = (!\mathit{ev\_subgoal}_1 \,\|\, !\mathit{ev\_subgoal}_2); !\mathit{ev\_subgoal}_3$, and suppose that both concurrent events have just been handled via rule $Event$, thus yielding the following program:

$$P''_{walk} \rhd \mathit{travelTo}(\mathit{Uni}) : (\!|\{\psi_2 : P_{city}(x, \mathit{Uni}), \psi_3 : P_{far}(x, \mathit{Uni})\}|\!), \tag{5}$$

where

$$\begin{aligned} P''_{walk} &= [((P^1_{pw} \| P^2_{pw});\ !\mathit{ev\_subgoal}_3) \rhd \mathit{prepareWalk} : (\!|\Delta_{pw}|\!)];\ \mathit{walk}(\mathit{Uni}); ?\mathit{At}(\mathit{Uni}); \\ P^i_{pw} &= \mathit{ev\_subgoal}_i : (\!|\Delta^i_{pw}|\!),\ \text{for } i \in \{1, 2\}. \end{aligned}$$

In other words, both concurrent events in P_{pw} have started being executed and hence they have become active goals. Then, program (5) above yields now *two* active goal traces, namely, $\lambda_i = G_1 \cdot G_2 \cdot G^i_3$, for $i \in \{1, 2\}$, such that:

$$\begin{aligned} G_1 &= P''_{walk} \rhd \mathit{travelTo}(\mathit{Uni}) : (\!|\{\psi_2 : P_{city}(x, \mathit{Uni}), \psi_3 : P_{far}(x, \mathit{Uni})\}|\!); \\ G_2 &= ((P^1_{pw} \| P^2_{pw});\ !\mathit{ev\_subgoal}_3) \rhd \mathit{prepareWalk} : (\!|\Delta_{pw}|\!); \\ G^i_3 &= \mathit{ev\_subgoal}_i : (\!|\Delta^i_{pw}|\!). \end{aligned}$$

First, see that whereas traces λ_1 and λ_2 share the first two subgoals, they differ on the third one depending on which of the two concurrent programs P^1_{pw} or P^2_{pw} are considered, respectively. Second, observe that from the active traces, one can easily obtain which goal *events* are active, by looking at the event mentioned in each goal. In our case, the events being handled are *travelTo*(*Uni*), *prepareWalk*, $\mathit{ev\_subgoal}_1$, and $\mathit{ev\_subgoal}_2$. Lastly, we recall that only *active* goals are considered in traces, that is, events that the agent has already started working on. Thus, for example, event-goal $!\mathit{ev\_subgoal}_3$ has not yet been started, is not active, and is therefore not an active goal in any trace (yet). ■

We now have all the machinery required to state the two main results for CAN$^{\mathcal{A}}$. The first result is related to the *failure-handling* or *goal-recovery* mechanism. Roughly speaking, it states that the built-in failure handling mechanism respects the hierarchical structure of goals, by preserving what has already been executed as much as possible. To understand the claim better, let us recall how goal-recovery works. At any point in time, an agent may be pursuing a particular (current) strategy P to resolve an event-goal $!e$. If at some point such strategy cannot be continued further (i.e., P is blocked), then the agent may resort to alternative courses of actions for event e. Technically, the agent may apply intention-level derivation rule $\rhd_{rec}$ so as to abandon the current strategy P in an active goal $P \rhd e : (\!|\Delta|\!)$, and adopt an alternative applicable strategy within set Δ.

✓ **Theorem 1.** *Let C be a CAN$^{\mathcal{A}}$ agent and $I \in C[\Gamma]$ be an active intention in C. Furthermore, let $G_k = \lambda[k]$, with $k \geq 1$, be the k-th active goal in some active goal trace $\lambda \in \mathcal{GT}race(I)$. If G_k's current strategy is blocked, then for every $k' > k$, subgoal $G_{k'} = \lambda[k']$ is fully blocked.*

Proof. If G_k is of the form $e_k : (\!|\Delta_k|\!)$, then there are no lower-level goals than G_k in the trace, G_k is the last goal. Suppose now that $G_k = P_k \rhd e_k : (\!|\Delta_k|\!)$. We perform induction on k'. If $k' = k+1$ (base case), then $G_{k'} = P_k$, that is, goal $G_{k'}$ is in fact G_k's current strategy. Since G_k's current strategy is in fact P_k, then $G_{k'}$ ought to be fully blocked (see Definition 6).

Next, suppose the claim holds for all goals in the trace up to some $k' \leq \hat{k} < |\lambda|$ and let us consider goal $\hat{k}+1$. By induction, active goal $\lambda[\hat{k}]$ is fully blocked. Since $\lambda[\hat{k}+1] \leq |\lambda|$, goal $\lambda[\hat{k}]$ is of the form $P_k \rhd e_k : (\!|\Delta_k|\!)$ and, moreover, $\lambda[\hat{k}+1] = P_k$. In turn, P_k itself must be of the form $e_{k+1} : (\!|\Delta_{k+1}|\!)$ or of the form $P_{k+1} \rhd e_{k+1} : (\!|\Delta_{k+1}|\!)$. In both cases, because program $\lambda[\hat{k}]$ is fully blocked in C, programs P_{k+1} and $e_{k+1} : (\!|\Delta_{k+1}|\!)$ must be blocked in C. Hence, goal $\lambda[\hat{k}+1]$'s current strategy (P_{k+1}) is blocked and it has no alternative applicable strategies in C. □

As a consequence, the current strategy for a goal G_k, or even the whole goal itself, may be reconsidered by the agent *only if* all its active lower-level subsidiary subgoals have their current strategy blocked and no alternative strategy to try. Thus, the goal failure recovery works hierarchically on the set of goals being pursued.[11]

The second result is important in that it *characterizes* how active goals may change within an intention after an agent execution cycle. The first case states that every existing goal G *before* the agent step is either *(1a)* preserved intact; *(1b)* updated or fully terminated due to a step performed on it; or *(1c)* removed due to goal-failure recovery of some higher-level goal. The second case states that every goal G' *after* the agent step is either an existing one or the evolution of an existing one (cases *(2a) and (2b))*, or a newly created one by some existing goal (case *(2c)*). No other dynamics for goals can apply.

Theorem 2. *Let C and C' be two CAN$^{\mathcal{A}}$ agent configurations such that $C \overset{\text{agent}}{\Longrightarrow} C'$. Let $\langle id, P\rangle \in \Gamma$ and $\langle id, P'\rangle \in \Gamma'$. Then (here, $\mathcal{B}' = C'[\mathcal{B}]$, and $\mathcal{A}' = C'[\mathcal{A}]$):*

1. *For every $G \in \mathcal{G}(P)$ (i.e., G is an active goal in P), one of the following three cases must apply (see that $\mathcal{B}''$ is assumed to be existentially quantified):*
   - *(a) $G \in \mathcal{G}(P')$, that is, G has remained unchanged;*
   - *(b) $\langle \mathcal{B}, \mathcal{A}, G\rangle \longrightarrow \langle \mathcal{B}'', \mathcal{A}', G'\rangle$ and $G' \in \mathcal{G}(P') \cup \{nil\}$, that is, G has evolved to G' when P was evolved to P'; or*

[11] In some cases, though, an agent may want to drop a goal even when some of its subgoals are not failed, for instance, when the goal is not "desired" anymore. This however should be considered as goal "*abortion*" rather than goal failure; see [64] for treatment of this issue.

(c) for some $\lambda \in \mathcal{GT}race(P)$, and $1 \leq k_1 < k_2$, goal $G = \lambda[k_2]$ is fully blocked in C and goal $G_1 = \lambda[k_1]$ has its current strategy blocked, but has an alternative applicable strategy in C, that is, a higher-level goal G_1 has been recovered.

2. *For every $G' \in \mathcal{G}(P')$ (i.e. G' is an active goal in P'), there exists $G \in \mathcal{G}(P)$ such that one of the following cases must apply (again, $\mathcal{B}''$ is assumed to be existentially quantified):*
   (a) $G' = G$, that is, G' is an already active goal that has remained unchanged;
   (b) $\langle \mathcal{B}, \mathcal{A}, G\rangle \longrightarrow \langle \mathcal{B}'', \mathcal{A}', G'\rangle$, that is, G' is an evolution of some active goal; or
   (c) $\langle \mathcal{B}, \mathcal{A}, G\rangle \longrightarrow \langle \mathcal{B}'', \mathcal{A}', G''\rangle$ and $\mathcal{G}(G'') = (\mathcal{G}(G) \setminus \{G\}) \cup \{G'', G'\}$, that is, goal G' has just been adopted by the current strategy of some active goal G.

Proof. If $I' = I$, then the intention was not selected for execution in the agent cycle. Then, $\mathcal{GT}race(P) = \mathcal{GT}race(P')$ and cases (1a) and (2a) apply trivially.

Suppose that $I \neq I'$. Then, $\langle \mathcal{B}, \mathcal{A}, P\rangle \longrightarrow \langle \mathcal{B}'', \mathcal{A}', P'\rangle$, for some $\mathcal{B}''$. It is not hard to see that, because every pair of active goal traces of a program must share a common prefix (at least their first top-level goal), the set $\mathcal{GT}race(P)$ induces a unique (up to isomorphism) unordered *tree of active goals* $\mathcal{T}_P$, where each node v in $\mathcal{T}_P$ is labelled with an active goal G_v and where each branch corresponds one-to-one to an active goal trace in $\mathcal{GT}race(P)$. So, let us see how the tree of active goals $\mathcal{T}_{P'}$ is related to the original tree of active goals $\mathcal{T}_P$.

First, the above transition must be due to a transition $\langle \mathcal{B}, \mathcal{A}, G_v\rangle \longrightarrow \langle \mathcal{B}'', \mathcal{A}', G'_v\rangle$ of an active goal G_v in some node v of $\mathcal{T}_P$, that is, one of the following cases applies: *(i)* G_v's current strategy performs a step on a primitive action, test, belief update operation, or parallel terminating programs (basic rule act, ?, $+b$, $-b$, or $\|_{end}$); *(ii)* G_v's performs a terminating step since its current strategy has successfully completed (rule $_{\triangleright}t$); *(iii)* G_v's current strategy performs a step on an internal event goal $!e$ (rule $Event$); or *(iv)* G_v's current strategy is blocked but performs a recovery step on an existing alternative strategy (basic rule $\triangleright_{rec}$).

Second, we observe that given the basic transition on goal node G_v, every ancestor goal G_w of G_v makes a non-basic transition of the form $\langle \mathcal{B}, \mathcal{A}, G_w\rangle \longrightarrow \langle \mathcal{B}'', \mathcal{A}', G'_w\rangle$, for some G'_w. This is the case because every ancestor goal of G_v is in fact a program that mentions, nested in its current strategy, goal G_v. Thus, G'_w is G_w with G_v replaced with G'_v.

So, the new tree $\mathcal{T}_{P'}$ is obtained from tree $\mathcal{T}_P$ by suitably changing node G_v, every ancestor of G_v and, possibly, adding a new descendant of G_v (for case *(iii)*) or dropping all its descendants (for case *(iv)*). More concretely:

- In case *(i)*, node G_v is updated to G'_v and every ancestor G_w is updated to G'_w.
- In case *(ii)*, node G_v is a leaf in $\mathcal{T}_P$, which is removed completely and every ancestor G_w updated to G'_w.
- In case *(iii)*, node G_v is updated to G'_v, every ancestor G_w is updated to G'_w, and a new node G_r, representing the just adopted goal, is created as a child node of G'_v.
- Finally, in case *(iv)*, node G_v is updated to G'_v, every ancestor G_w is updated to G'_w, and all descendants of G_v are removed.

In all cases, the remaining nodes in $\mathcal{T}_P$ remain unchanged.

With this understanding of how $\mathcal{T}_{P'}$ is obtained from $\mathcal{T}_P$, one can easily verify that the two claims of the theorem hold. For the first part, see that every goal G in $\mathcal{T}_P$ is either kept unchanged in $\mathcal{T}_{P'}$, updated to its intention-level evolution G' in $\mathcal{T}_{P'}$, or completely removed in $\mathcal{T}_{P'}$ due to its own termination or to a goal recovery transition at a higher-level goal (in which case G is fully blocked by Theorem 1). For the second part of the theorem, observe that every goal G' in the evolved tree $\mathcal{T}_{P'}$ must be an unchanged active goal, the evolution of some previous active goal, or a new goal adopted by an existing active goal (case *(iii)*). □

Putting it all together, Theorem 1 constrains the way the built-in goal recovery mechanism works, whereas Theorem 2 identifies the relation between the goals being pursued before and after an agent evolution in each intention.

3 CAN: Declarative Goals in BDI Programming

One of the main drawbacks of typical BDI agent programming languages like CAN$^{\mathcal{A}}$ is their extreme *procedural* view of goals as mere tasks or processes that ought to be executed to completion. Even though a procedural view of goals in the form of *know-how* information is often highly desirable to ensure that these can be achieved efficiently in dynamic environments [46], a *declarative* perspective, as common in agent theory [11, 54] and automated planning [28, 46], opens the door to more sophisticated reasoning. Indeed, with an explicit notion at hand of what state of affairs a goal stands for, one may be able to check whether the goal in question has been achieved, whether it has become impossible, or whether it may interfere with other goals [62, 63]. The fact that most BDI agent programming languages only deal with procedural aspects of goals shows the existing gap between BDI theory and implementation. Nonetheless, the need for richer accounts of goals in these languages has recently been recognized in the literature (e.g., [16, 32, 57, 67, 73]).

In this section, we show how the language presented in the previous section can be incrementally and modularly extended to accommodate an account of goals with both procedural and declarative aspects. The idea is to enrich the BDI event goals with some *declarative information* so as to decouple plan failure/completion from goal failure/achievability.

Following Winikoff et al. [73], we first enhance the simple event-goal program $!e$ in CAN$^{\mathcal{A}}$ with a new type of program that accommodates declarative information about the goal to be achieved. More concretely, we extend the full program language from Section 2.1 with a so-called goal-program construct, though we shall restrict the *user* program language—the languages available for programming plan libraries Π—to *user* goal-programs only.

Definition 8 (Goal-programs and Declarative Goals). A *goal-program* is a program of the form $\mathsf{Goal}(\phi_s, P, \phi_f)$, where ϕ_s and ϕ_f are belief formulas and P is a program. A *user* goal-program is one where P is of the form $!e$. When an agent is executing a goal-program, we say that it is pursuing the *declarative goal* $\langle \phi_s, \phi_f \rangle$. ■

The intended meaning of a program $\mathsf{Goal}(\phi_s, P, \phi_f)$ is that "*the (declarative) goal state ϕ_s should be achieved by using the (procedural) program P; failing if ϕ_f becomes true (e.g., the goal is impossible, not required anymore, etc.)*" By "declarative" here we mean that its desired result is specified as a *state of affairs*, by means of formula ϕ_s.

Generally speaking, the execution of a goal-program is expected to be consistent with some desired properties of goals, namely:

Persistent A rational agent should not abandon a goal without good reasons. A goal-program will insist on resolving event $!e$ as long as it has not been achieved or deemed impossible due to its success and failure conditions, respectively.

Unachieved A rational agent should not be pursuing goals that are already true. A goal-program is successfully dropped as soon as its success condition ϕ_s holds, that is, when the goal has been achieved.[12]

[12] We note that dropping a goal when its success condition ϕ_s holds true may, in some cases, violate the implicit non-functional requirements encoded in the procedural knowledge (e.g., always leave the safe locked after using it). Though beyond the scope of this paper, ways to address this could be inclusion of such require-

Possible A rational agent should only pursue goals that are eventually possible to achieve. At any point in the execution of a goal-program, if its failure condition ϕ_f becomes true, then the goal is deemed impossible and dropped with failure.

The importance of the success ϕ_s and the failure ϕ_f condition is that, together, they decouple the success and failure of the goal from the success or failure of its plans—a goal should not be dropped merely because a plan to achieve the goal has failed, and a goal cannot be assumed achieved just because the plan has executed fully.

Example 5. Let us come back to event goal *travelTo*$(dest)$ from Example 1. The agent could use that event-goal when she needs to go to a place as part of a larger goal, such as business or holiday trip:

$$\begin{aligned} &\mathit{doTrip}(dest, reason) : \mathit{Work}(reason) \leftarrow \\ &\qquad (!\mathit{arrangeTransp}(dest) \,\|\, !\mathit{arrangeHotel}(dest) \,\|\, !\mathit{arrangeLocalCar}(dest)); \\ &\qquad !\mathit{travelTo}(dest);\ !\mathit{doWork};\ ?\mathit{Address}(\mathit{Home}, addr);\ !\mathit{travelTo}(addr). \end{aligned}$$

Informally, the agent first arranges the trip—by concurrently booking the transportation, accommodation, and local transportation at destination. Then, she does the actual traveling and fulfills the work duties, and finally returns home.

In this case, we could enhance the event-goal $\mathit{postravelTo}(dest)$ to travel to the destination with the following declarative goal-program:

$$\mathsf{Goal}(\mathit{At}(dest), !\mathit{travelTo}(dest), \mathit{Unreachable}(dest) \vee \mathit{Cancelled}(reason))).$$

This program will not just succeed when the corresponding event goal completes execution, but when *its execution actually achieves the goal*, that is, when $\mathit{At}(dest)$ is believed true. Furthermore, the subgoal will be dropped with failure if the agent comes to believe the destination is unreachable or that the reason for the trip does not apply anymore (e.g., the conference has been cancelled). ■

As with all the other constructs, we need to provide the semantics for the new construct. To that end, five new intention-level derivation rules are introduced. The first rule is meant to "initialize" the execution of a goal-program $\mathsf{Goal}(\phi_s, !e, \phi_f)$ when this is first encountered at execution time, thus "adopting" the declarative goal $\langle \phi_s, \phi_f \rangle$. A successful adoption of a goal requires that the goal is not already true or deemed impossible/failed, and that the agent does have some relevant plan to eventually handle the goal—agents should not adopt goals for which there are no capabilities. Formally,[13]

$$\frac{\mathcal{B} \not\models (\phi_s \vee \phi_f) \quad \langle \mathcal{B}, \mathcal{A}, !e \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P \rangle}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Goal}(\phi_s, !e, \phi_f) \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \mathsf{Goal}(\phi_s, P \rhd P, \phi_f) \rangle}\ \mathsf{G}_{adopt}$$

Roughly speaking, adopting a declarative goal involves setting its procedural program to $P \rhd P$. For instance, if $e = \mathit{travelTo}(\mathit{Uni})$ as in Example 1, then P would be the program shown in (1); see pp. 8. Note that program P in the rule above would always stand, in our language, for the set $e : (\!|\Delta|\!)$ of relevant program-strategies for addressing the event—the second requirement of rule G_{adopt} holds only if derivation rule Event (see pp. 7) applies. So, the program built by the rule is of the form $\mathsf{Goal}(\phi_s, e : (\!|\Delta|\!) \rhd e : (\!|\Delta|\!), \phi_f)$.

ments in the goal's success condition itself, execution of "clean-up" procedures following plan termination (as exists in, e.g., Jack), or use of transaction-like mechanisms to disallow interruption within a specified block.

[13] Rule G_{adopt} could be further developed to capture extra constraints, such as the goal not being in conflict with a goal already committed to (see discussion section).

Next, the agent carries out the current strategy by basically "consuming" the first left program in the already adopted goal. More concretely, the agent shall execute the goal-program current strategy P_1 in the rule below, while keeping (backup) program P_2 "intact." This, of course, provided the success or failure condition do not hold true.

$$\frac{\langle \mathcal{B}, \mathcal{A}, P_1\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P'\rangle \quad \mathcal{B} \not\models \phi_s \quad \mathcal{B} \not\models \phi_f}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Goal}(\phi_s, P_1 \rhd P_2, \phi_f)\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \mathsf{Goal}(\phi_s, P' \rhd P_2, \phi_f)\rangle}\ \mathsf{G}_{step}$$

Once again, program P_2 is, in fact, the original set of relevant strategies $e : (\!|\Delta|\!)$ obtained when the goal was adopted via rule G_{adopt}. The idea behind carrying along the original set of strategies is that should the agent run out of current options, she may consider re-instantiating those original set of relevant strategies. Informally, the agent shall "insist" on the available strategies as much as possible until the the goal is realized or deemed impossible (see below).

In contrast with standard events, the agent is not concerned only with the (total) execution of the programs, but also with conforming to the declarative aspects of the goal. As a result, the following two rules allow the goal to be dropped if it becomes achieved or failed:

$$\frac{\mathcal{B} \models \phi_s}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Goal}(\phi_s, P, \phi_f)\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \mathit{nil}\rangle}\mathsf{G}_{succ} \qquad \frac{\mathcal{B} \models \phi_f}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Goal}(\phi_s, P, \phi_f)\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, ?\mathit{false}\rangle}\mathsf{G}_{fail}$$

Thus, succeeding a goal means that there is nothing else to be done for it—the remaining program is the empty program *nil*. Failing the goal is captured by evolving to a program that is always impossible (i.e., always blocked) for the agent, namely, test program ?*false*.

Finally, we consider the case in which the goal has not yet been achieved but its procedural program cannot continue further, either because it has executed fully or because it has reached a dead-end and is blocked. To capture the expected persistence of (declarative) goals, the very *original* strategies for the goal (carried along as program P_2 below) are re-instantiated as the current strategy, in the hope that an applicable one can be found. Formally,

$$\frac{\langle \mathcal{B}, \mathcal{A}, P_1\rangle \not\longrightarrow \quad \langle \mathcal{B}, \mathcal{A}, P_2\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P_2'\rangle}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Goal}(\phi_s, P_1 \rhd P_2, \phi_f)\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \mathsf{Goal}(\phi_s, P_2' \rhd P_2, \phi_f)\rangle}\ \mathsf{G}_{restart}$$

Observe that for a goal to be re-instantiated, the current strategy P_1 must be blocked (first requirement) and P_2, which stands for the original set of relevant plans $e : (\!|\Delta|\!)$, must contain a backup alternative strategy (second requirement). Otherwise, if there is *no applicable* strategy in $P_2 = e : (\!|\Delta|\!)$, then the whole goal-program should become *blocked*. This is important because it shall enable the agent—via rule $\rhd_{rec}$ (page 9)—to actually *drop* a (blocked) declarative goal-program (e.g., the goal to travel to the bookstore) if there is an alternative way (e.g., buying books online) of achieving a higher-level motivating goal (e.g., buying a particular book).[14] Note also the difference between a goal-program being blocked and its failure condition being true. In the latter case, the goal-program may indeed perform a single step via rule G_{fail} to the always failing program ?*false*.

Example 6. Suppose that instead of addressing the event-goal *travelTo*(*Uni*), as in Example 1, the agent is meant to execute the following declarative version:

$$\mathsf{Goal}(\mathit{At}(\mathit{Uni}), !\mathit{travelTo}(\mathit{Uni}), \mathit{Cancelled}(\mathit{Exam})).$$

[14] This is one of the main differences with the goal-programs in [73], where declarative goals may never be dropped for the sake of higher-level goals.

After the application of rule G_{adopt} this program will evolve to:

$$\mathsf{Goal}(\mathit{At}(\mathit{Uni}), P \rhd P, \mathit{Cancelled}(\mathit{Exam})).$$

where $P = \mathit{travelTo}(\mathit{Uni}) : \{\!\{\psi_1 : P_{walk}(\mathit{Uni}),\ \psi_2 : P_{city}(x, \mathit{Uni}),\ \psi_3 : P_{far}(x, \mathit{Uni})\}\!\}$ is the program shown in equation (1), pp. 8, encoding the set of all relevant program-strategies.

Next, we imagine the agent executing the first (left) copy of P in a similar manner as discussed in Examples 2 and 3, though this time by relying on derivation rule $\mathsf{G}step$:

$$\mathsf{Goal}(\mathit{At}(\mathit{Uni}), P' \rhd P, \mathit{Cancelled}(\mathit{Exam})), \tag{6}$$

where $P' = P_{city}(\mathit{Home}, \mathit{Uni}) \rhd \mathit{travelTo}(\mathit{Uni}) : \{\!\{\langle \psi_3 : P_{far}(x, \mathit{Uni})\rangle\}\!\}$. In other words, the current strategy of the goal-program—the left copy of P–has basically executed as a standard event does: P' is exactly the program shown in equation (4); pp. 4.

Next, suppose that current strategy of P_{city} cannot execute further, e.g., taxis are fully booked and the buses have been cancelled in the city. Under a standard event-goal, the whole program P' would simply *fail*, as the remaining alternative strategy P_{far} in P' is *not* applicable for short distances. Nonetheless, while the strategy to *walk* is not included in P' anymore, as it has been tried without success (see Example 3), it is still accounted in the original set of relevant strategies P. Thus, because the procedural event-goal is running *within* a goal-program, rule $\mathsf{G}_{restart}$ could be applied to come back to the original set of strategies P. If walking is still a feasible option, the rule yields the following next program:

$$\mathsf{Goal}(\mathit{At}(\mathit{Uni}), P'' \rhd P, \mathit{Cancelled}(\mathit{Exam})),$$

where $P'' = P_{walk}(\mathit{Uni}) \rhd \mathit{travelTo}(\mathit{Uni}) : \{\!\{\psi_2 : P_{city}(x, \mathit{Uni}), \psi_3 : P_{far}(x, \mathit{Uni})\}\!\}$ is the program shown in (2) stating to walk to destination—the agent insists on the goal as there are still "reasonable" strategies to be tried.

On the other hand, if no applicable strategy is found in P (e.g., GPS signal was lost, the agent does not know its current location and cannot therefore evaluate atom $\mathit{At}(x)$), then $\mathsf{G}_{restart}$ cannot be used and the whole goal-program (6) above becomes *blocked*.

Finally, we point out that if, at any time, the agent happens to be at the university or learns the exam has been cancelled, the whole goal-program is terminated via rules G_{succ} and G_{fail} even if the procedural part of it has not executed to completion. ■

So, by re-instantiating the original strategies when the current one has not been able to actually achieve the corresponding goal, the language provides a *persistence* on declarative goals that standard BDI events lack. Failure of the program should not be considered, in principle, equal to the failure of the goal [15, 67]: as long as there are applicable plans, there are reasons to believe that the goal is still achievable. Nonetheless, if no "recovery" alternative plan can be found, then the whole goal may be re-considered for the sake of higher-level motivating goals. Observe that when recovering, the original strategies in P_2 are still kept as backup: the agent may need to come back to them (again) if executing P_2' still fails to realize the goal.

This concludes the set of intention-level derivation rules for goal-programs.

3.1 Generating Goals Proactively

Besides allowing plans in the plan library to make use of declarative goals, we shall also allow agents to generate top-level declarative goals in a proactive manner. To that end, we

equip our agents with a special *motivation library* $\mathcal{M}$. Intuitively, library $\mathcal{M}$ stands for the agent's intrinsic motivations or desires. At this stage, we consider motivation libraries that only account for what has been elsewhere called *desires* (as conditional goals) [67, 68] or *automatic events* [9]: goals that are conditionalised by beliefs. Hence, an agent may adopt a new goal on the basis of recognising a particular world state.

So, a motivation library $\mathcal{M}$ consists of rules of the form

$$\psi \rightsquigarrow \mathsf{Goal}(\phi_s, !e, \phi_f),$$

Informally, if the agent comes to believe ψ, she should *consider adopting* the declarative event goal $\mathsf{Goal}(\phi_s, !e, \phi_f)$.

Example 7. The following motivational rule states that when the agent happens to learn her paper was accepted, she should start working towards producing the paper camera ready version for final submission:

$$\mathit{PaperAccepted} \rightsquigarrow \mathsf{Goal}(\mathit{CameraReady}, !\mathit{prepareCameraReady}, \mathit{PaperWithdrawn}).$$

The adopted goal may be dropped with success when the camera ready version is produced (i.e., *CameraReady* holds true) or with failure if the paper has been withdrawn from the conference (e.g., a technical error has been found or the agent cannot attend the venue). ■

By means of her motivation library, the agent may now create intentions—new focus of attention—not only for responding to external events, but for satisfying her own internal desires as well.[15] Observe that we have not imposed any semantic constraints on the new library and it is hence conceivable for an agent to hold "contradictory" motivations.

3.2 Agent Level Execution with Goals

We now explain how the agent-level semantics from Section 2.2.2 need to be extended to accommodate the extended language. The main top-level rule implementing the abstract BDI execution cycle (see pp. 10) remains exactly the same, now labelled CAN instead (i.e., derivation rule A_{CAN}). To account for the agent's motivation library, agent configurations are extended to tuples of the form $\langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \Gamma \rangle$, where $\mathcal{M}$ is a motivation library as above.

The remaining changes involve two new goal-update rules for characterizing the $\overset{\mathsf{goal}}{\Longrightarrow}$ agent transitions, and a slight adaptation of derivation rule A_{int} (pp. 11) for executing one selected intention. To show these three changes, we first need to extend and introduce a few concepts. First of all, besides event goals (Definition 6; pp. 14), an intention may now also be working on *declarative* goals.

Definition 9 (Active Declarative Goal). An *active declarative goal* is a program of the form $G = \mathsf{Goal}(\phi_s, P \rhd e : (\!|\Delta|\!), \phi_f)$. Program P is the goal's *current strategy* and $e : (\!|\Delta|\!)$ encodes the *alternative strategies* for the goal. We say that goal G *has an alternative applicable strategy* in an agent C if program $e : (\!|\Delta|\!)$ is not blocked in C. ■

[15] Of course, rational agents may adopt goals for other reasons besides these two. For example, agent communication [44] and social norms and obligations [8, 39] are also typical sources of motivations for agents. Interestingly, for instance, the norm specification in the NoA architecture [39] is very close to our motivational rules; besides having an "activation" condition they also include an "expiration" condition.

Again, due to its syntactic form, for a declarative goal to be active, it must have been previously *adopted*, by means of intention-level derivation rule G_{adopt}.

Next, we extend the notion of active goal traces (Definition 7; pp. 14) in order to account for active declarative goals (see second case).

Definition 10 (Active Goal Trace, for CAN agents). An *active goal trace* λ is a sequence of active goals $G_1 \cdot \ldots \cdot G_n$. The multiset of all active goal traces in a program P, denoted $\mathcal{GT}race(P)$, is inductively defined as follows:

$$\mathcal{GT}race(P) = \begin{cases} \emptyset & \text{if} \quad P = \mathit{nil} \mid \mathit{act} \mid ?\phi \mid +b \mid -b \\ \{P \cdot \lambda \mid \lambda \in \mathcal{GT}race(P_1)\} & \text{if} \quad P = P_1 \rhd e{:}(\!|\Delta|\!) \mid \mathsf{Goal}(\phi_s, P_1 \rhd P_2, \phi_f) \\ & \quad \text{and } \mathcal{GT}race(P_1) \neq \emptyset \\ \{P\} & \text{if} \quad P = P_1 \rhd e{:}(\!|\Delta|\!) \mid \mathsf{Goal}(\phi_s, P_1 \rhd P_2, \phi_f) \\ & \quad \text{and } \mathcal{GT}race(P_1) = \emptyset \\ \{P\} & \text{if} \quad P = e{:}(\!|\Delta|\!) \\ \mathcal{GT}race(P_1) & \text{if} \quad P = P_1; P_2 \\ \mathcal{GT}race(P_1) \uplus \mathcal{GT}race(P_2) & \text{if} \quad P = P_1 \| P_2 \end{cases}$$

✓ ■

Hence, an active goal trace encodes a hierarchical chain of both active event goals and active declarative goals.

When it comes to declarative goals, we need to "extract" those that the agent is currently pursuing. To that end, we define $\mathcal{DG}(P)$ to be the set of *declarative* goals of the form $\langle \phi_s, \phi_f \rangle$ that the agent has already adopted, and is executing, within intention program P.[16]

Definition 11. The multiset of all active declarative goals in a program P, denoted $\mathcal{DG}(P)$, is defined as $\mathcal{DG}(P) = \biguplus_{n \in \mathbb{N}} \mathcal{DG}(P, n)$, where $\mathcal{DG}(P, n)$ stands for the multiset of all active declarative goals at level n (the top-level goal being at level 1) and is defined as follows:

$$\mathcal{DG}(P, n) = \{\langle \phi_s, \phi_f \rangle \mid (\exists \lambda).\lambda \in \mathcal{G}(P) \wedge \lambda[n] = \mathsf{Goal}(\phi_s, P', \phi_f)\}.$$

The multiset of achieved/failed goals in a program P at beliefset $\mathcal{B}$ is defined as follows:

$$\mathcal{DG}^{end}(\mathcal{B}, P) = \{\langle \phi_s, \phi_f \rangle \mid \ \langle \phi_s, \phi_f \rangle \in \mathcal{DG}(P),\ \mathcal{B} \models \phi_s \text{ or } \mathcal{B} \models \phi_f\}.$$

Lastly, all these notions extend to intentions and intention bases in a straightforward way, e.g., $\mathcal{DG}(\langle id, P \rangle) = \mathcal{DG}(P)$ and $\mathcal{DG}(\Gamma) = \biguplus_{I \in \Gamma} \mathcal{DG}(I)$. ■

Again, we appeal to multisets because the same declarative goal may be pursued many times in one intention (e.g., in different parallel sub-programs), and we want this to be captured in the above notions. So, for example, if our student agent is working on the goal-program shown in equation (6), pp. 21, as the fourth subgoal for an (original) top-level intention of the form $I = \langle \#2, \mathsf{Goal}(\mathit{ExamDone}, !\mathit{writeExam}, \mathit{Cancelled}(\mathit{Exam})) \rangle$, then $\mathcal{DG}(P, 1) = \{\langle \mathit{ExamDone}, \mathit{Cancelled}(\mathit{Exam}) \rangle\}$ and $\mathcal{DG}(P, 4) = \{\langle \mathit{At}(\mathit{Uni}), \mathit{Cancelled}(\mathit{Exam}) \rangle\}$. Also, if $\mathcal{B} \models \mathit{Cancelled}(\mathit{Exam})$, for some belief base $\mathcal{B}$ (i.e., the exam has been cancelled), then $\{\langle \mathit{At}(\mathit{Uni}), \mathit{Cancelled}(\mathit{Exam}) \rangle, \langle \mathit{ExamDone}, \mathit{Cancelled}(\mathit{Exam}) \rangle\} \subseteq \mathcal{DG}^{end}(\hat{\mathcal{B}}, I)$.

At this point, we have all the technical machinery to extend the agent-level semantics of the core language (Section 2.2.2) to the CAN language. The first new rule accommodates a

[16] Unlike the original semantics of CAN [73], we do not keep and update an explicit goal base $\mathcal{G}$, as this is already *implicitly* represented in the intention base Γ.

proactive mechanism for generating new top-level intentions from the motivational library: (Recall operation $\Gamma \Cup \gamma$ from Definition 1;pp. 6)

$$\frac{\psi \rightsquigarrow P \in \mathcal{M} \quad C_{init}[\mathcal{B}] \not\models \psi \quad \mathcal{B} \models \psi \quad \langle \mathcal{B}, \mathcal{A}, P\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, P'\rangle \quad \nexists \langle id, P'\rangle \in \Gamma}{\langle C_{init}, \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \Gamma\rangle\rangle \overset{\mathsf{goal}}{\Longrightarrow} \langle C_{init}, \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \Gamma \Cup \{P'\}\rangle\rangle} \; A^4_{\mathsf{goal}}$$

That is, when the agent comes to believe that ψ holds (second and third constraints), she shall "adopt" program P as a new intention, provided P can execute (fourth constraint) and is not already an active intention (fifth constraint). Note that since we have so far restricted P to be a declarative goal-program of the form $\mathsf{Goal}(\phi_s, P, \phi_f)$, P can execute *only if* rule G_{adopt} (pp. 19) applies—that is, the goal is fully specified, unachieved, deemed possible, and with capabilities (i.e., relevant plans) available. Taking our Example 7, if the agent happens to receive an email confirming the acceptance of her paper, the above rule may yield the following new intention in her intention base:

$$\langle \#12, \mathsf{Goal}(\mathit{CameraReady}, \mathit{prepareCameraReady} : (\!|\Delta|\!), \mathit{PaperWithdrawn})\rangle.$$

The second rule allows the agent to update its goal base by *legally dropping* a current goal. As discussed, a goal ought to be abandoned if it has been achieved or is deemed failed. Hence, a goal update may ensue whenever the agent is able to remove a goal that is in the set $\mathcal{DG}^{end}(\mathcal{B}, P)$, thus making this set smaller:

$$\frac{\langle id, P\rangle \in \Gamma \quad \langle \mathcal{B}, \mathcal{A}, P\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, P'\rangle \quad |\mathcal{DG}^{end}(\mathcal{B}, P)| < |\mathcal{DG}^{end}(\mathcal{B}, P')|}{\langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \Gamma\rangle \overset{\mathsf{goal}}{\Longrightarrow} \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, (\Gamma \setminus \{\langle id, P\rangle\}) \cup \{\langle id, P'\rangle\}\rangle} \; A^5_{\mathsf{goal}}$$

In other words, an active intention P is legally evolved to P'—basically, due to rule G_{succ} or rule G_{fail} (see pp. 20)—such that P' has less achieved/failed goals.

It is worth mentioning that, as is the case with rules A^{1-4}_{goal} from CAN$^{\mathcal{A}}$, the two new goal update rules also handle *single* updates, that is, the update of *one* goal only. Multiple applications of these rules will be required for the agent to fully update its goal base.

Lastly, we need to modify the agent-level rule in charge of terminating an intention, namely, rule A^1_{goal} (pp. 12). In CAN$^{\mathcal{A}}$, any intention that is blocked may be completely dropped. In the presence of declarative goals, however, only purely *reactive* intentions—intentions not pursuing any declarative goal—may be abandoned when blocked. There must be better reasons, though, to drop a declarative goal, besides being blocked (see below).

$$\frac{\langle id, P\rangle \in \Gamma \quad \mathcal{DG}(P) = \emptyset \quad \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, P\rangle \not\longrightarrow}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma\rangle \overset{\mathsf{int}}{\Longrightarrow} \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma \setminus \{\langle id, P\rangle\}\rangle} \; A^1_{\mathsf{goal}}$$

By including the constraint $\mathcal{DG}(P) = \emptyset$, we ensure that this rule does not allow dropping of any intention working on a declarative goal.

This concludes the addition of declarative goals into the core language of Section 2, thus yielding the extended language CAN. Notably, the new language is an *incremental* extension of the core language, or in other words, CAN$^{\mathcal{A}}$ is a *fragment* of CAN. In fact, it is not hard to prove that when it comes to agents not using the goal-program construct Goal (i.e., Π and Γ do not mention construct Goal and $\mathcal{M} = \emptyset$), the CAN and CAN$^{\mathcal{A}}$ BDI execution coincide.

As in CAN$^{\mathcal{A}}$, the failure handling mechanism does respect the hierarchical structure of the active goals, where these may be either standard event-goals or declarative ones.

Theorem 3. *Theorem 1 (pp. 16) holds for CAN agents and goal traces as in Definition 10.*

Proof. Exactly as the proof for Theorem 1 except that now G_k can be a declarative goal of the form $\mathsf{Goal}(\phi_s, P_k \rhd e_k : (\!|\Delta_k|\!), \phi_f)$. In such case, since the current and alternative strategies for a declarative goal are independent of the goal success and failure conditions, the same argument still follows through. That is, program P_k is indeed blocked in C which shall imply that every active goal in P_k will be fully blocked. □

Hence, the alternative strategies for a goal, either event or declarative, may be considered (by means of rules $\rhd_{rec}$ or $\mathsf{G}_{restart}$) *only if* no working alternative strategies can be found for *all* the subgoals that are instrumental to it. As a consequence, failure is handled bottom-up by trying to recover the lowest active goal possible.

3.3 On the Commitment of CAN: Between Single and Open Minded Agents

Let us focus now specifically on (the commitment to) declarative goals. As already discussed, it is generally accepted that a rational agent should not insist on goals that are deemed achieved or impossible. A *single-minded* agent maintains her commitment to a goal until she believes she has achieved it or that there are no options available to bring the goal about [54]. CAN agents go a step further by providing what we shall call a "flexible" single-minded type of commitment.

Informally, a *flexible single-minded* agent behaves like a single-minded agent, except that she may reconsider a subgoal of a motivating goal under certain circumstances. In that way, the unachieved and possible properties of goals are understood as sufficient conditions but not necessary conditions for an agent to drop her goals.

The following result states that CAN agents always drop their commitments to goals that are believed achieved or deemed impossible/unnecessary.

Theorem 4. *Let C_0 be a CAN agent configuration such that for every $\langle \phi_s, \phi_f \rangle \in \mathcal{DG}(C_0[\Gamma])$, $\mathcal{B} \not\models \phi_s$ and $\mathcal{B} \not\models \phi_f$. Let $C_0 \cdot \ldots \cdot C_n$ be a BDI execution of C_0 (relative to an environment $\mathcal{E}$). Then, for every declarative goal $\langle \phi_s, \phi_f \rangle \in \mathcal{DG}(C_n[\Gamma])$, $\mathcal{B} \not\models \phi_s$ and $\mathcal{B} \not\models \phi_f$ apply.*

Proof. Follows from the fact that, for every C, if $C \overset{\mathsf{CAN}}{\Longrightarrow} C'$, then C' may not contain any intention with an achieved or failed goal, that is, there is no $\langle \phi_s, \phi_f \rangle \in \mathcal{DG}(C'[\Gamma])$ such that $\mathcal{B} \not\models \phi_s$ or $\mathcal{B} \not\models \phi_f$. If, on the contrary, there is such a goal, then $C' \overset{\mathsf{goal}}{\Longrightarrow} C''$, for some C'', due to derivation rule A^2_{goal}, which will allow the dropping of the corresponding goal-program. This means $C' \overset{\mathsf{goal}}{\not\Longrightarrow}$ does not hold and neither does $C \overset{\mathsf{CAN}}{\Longrightarrow} C'$. □

Thus, no matter how an agent evolves relative to the environment, her goal base is correctly updated—she would never desire goals that are currently true or deemed failed.

The second result identifies all the *reasons* why a CAN agent may drop a declarative goal. More specifically, it provides the necessary conditions for our flexible single-minded type of commitment. Although Theorem 4 suggests that, at the very minimum, a declarative goal ought to be dropped if it has been achieved or deemed failed, there are also other reasons why a goal may be abandoned. A subgoal, for instance, ought to be dropped if it is a subsidiary goal for a higher-level motivating goal which is considered achieved or unachievable. More interestingly, a *subsidiary* declarative goal might be abandoned by the agent if she does not have any current way of acting upon it, but an alternative plan is found for a higher-level motivating goal. In that case, the agent may consider dropping the lower-level instrumental goal to pursue the alternative plan for the higher-level goal.

Theorem 5. *Let C and C' be two agent configurations such that $C \overset{\mathsf{agent}}{\Longrightarrow} C'$ and $\langle \phi_s, \phi_f \rangle \in \mathcal{DG}(C[\Gamma])$, but $\langle \phi_s, \phi_f \rangle \notin \mathcal{DG}(C'[\Gamma])$. Then, one of the following cases applies (below, $\mathcal{B}'$ is the belief base of configuration C', that is, $\mathcal{B}' = C'[\mathcal{B}]$):*

1. *$\mathcal{B}' \models \phi_s$, i.e., the goal has been achieved.*
2. *$\mathcal{B}' \models \phi_f$, i.e., the goal is believed to be impossible.*
3. *For every $I \in C[\Gamma]$, if $\lambda \in \mathcal{GT}race(I)$ and $\lambda[k] = \mathsf{Goal}(\phi_s, P, \phi_f)$, then there exists $1 \leq k' < k$ such that either*
   (a) $\lambda[k'] = \mathsf{Goal}(\phi'_s, P', \phi'_f)$ and $\mathcal{B}' \models \phi'_s \vee \phi'_f$, for some ϕ'_s, ϕ'_f and P'; or
   (b) goal $\lambda[k]$ is fully blocked and goal $\lambda[k']$ has its current strategy blocked but an alternative applicable strategy in C.

Proof. Suppose that $\mathcal{B}' \not\models \phi_s$ and $\mathcal{B}' \not\models \phi_f$, that is the goal is neither achieved nor deemed unfeasible by the agent. Observe first that there exist intermediate configurations C_1 and C_2 such that $C \overset{\mathsf{int}}{\Longrightarrow} C_1$, $C_1 \overset{\mathsf{event}}{\Longrightarrow} C_2$, $C_2 \overset{\mathsf{goal}*}{\Longrightarrow} C'$ and $C' \overset{\mathsf{goal}}{\not\Longrightarrow}$. Take now any $I = \langle id, P \rangle \in \Gamma$ such that $\langle \phi_s, \phi_f \rangle \in \mathcal{DG}(I)$, that is, for some $\lambda \in \mathcal{GT}race(P)$ and $k \geq 1$, $\lambda[k] = \mathsf{Goal}(\phi_s, P_k, \phi_f)$.

By assumption, if $I' = \langle id, P' \rangle \in \Gamma'$, then $\langle \phi_s, \phi_f \rangle \notin \mathcal{DG}(P')$. Suppose next that, for all $k' < k$ such that $\lambda[k'] = \mathsf{Goal}(\phi'_s, P_{k'}, \phi'_s)$, neither $\mathcal{B}' \models \phi'_s$ nor $\mathcal{B}' \models \phi'_f$ hold. That, together with the fact that $\mathcal{B}' \not\models \phi_s$ and $\mathcal{B}' \not\models \phi_f$, goal $\lambda[k]$ was *not* dropped during the (last) goal update transition. Thus, it must have been dropped during the $C \overset{\mathsf{int}}{\Longrightarrow} C_1$ transition. The only way this could happen is if $\lambda[k]$ is fully blocked in C, and a goal-recovery step is performed on a higher-level motivating goal in λ for which an applicable alternative strategy exists in C. Thus, case (3b) applies and the theorem follows. □

The first two cases account for the situations where the declarative goal is abandoned because it has been achieved or deemed failed, respectively. The third case covers the situations in which the dropped goal is not required anymore as a subsidiary goal for a higher-level motivating goal $G_{k'}$. This could happen either because the higher-level goal in question has been considered achieved or failed (case 3a), or because an alternative way of addressing it has been selected (case 3b).

It is worth noting a few interesting points. First, it is conceivable that a goal G could be fully blocked (i.e., G's current strategy is blocked and G has no alternative applicable strategy), while the current strategy of a higher-level motivating goal G' is not blocked. Such situation could arise when the current strategy for G' involves the *concurrent* execution of two programs (i.e., $G = P_1 \| P_2$), goal G belongs to just one of those programs, say P_1, and the other parallel program P_2 is able to evolve. Second, the above theorem refers to active *declarative goals* only. Nonetheless, the theorem can easily be rephrased for active *event goals* of the form $P \rhd e : \langle\!|\Delta|\!\rangle$, by merely replacing the first two cases with a single case accounting for the fact that the goal's current strategy is the terminating program *nil*. Third, the theorem is only concerned with situations in which the declarative goal is *completely* abandoned by the agent. One could imagine, however, cases where the very same goal $\langle \phi_s, \phi_f \rangle$ is being simultaneously pursued by different intentions or even multiple times within the same intention. In those cases, a particular instance of such goal may be (locally) dropped (due to case 3 above) without necessarily abandoning all its other instantiations—goal $\langle \phi_s, \phi_f \rangle$ would still show up in configuration C'. It is not hard to see that a *local*, though more cumbersome, version of the above theorem can be devised.

So, by putting together Theorems 4 and 5, we claim that CAN agents are indeed *flexible single-minded*, in the sense explained above. Their commitment strategy goes beyond

the well-established single-minded type, in that a (problematic) goal may be reconsidered as an appropriate instrument for some motivating goal. Observe such goal reconsideration is *optional*, as the agent may consider devoting its attention somewhere else in the hope that the problematic sub-goal becomes enabled again. The actual decision could be domain dependent or may even depend on implementation or runtime properties (e.g., how much time is devoted to re-consideration or how dynamic is the environment; see [36]). Thus, as with other features like plan selection, the semantics provided here is *skeptical* and the details are left to the actual BDI interpreter implementation.

We close by pointing out that the success and failure conditions in goal-programs can be used to capture other reasons besides the actual achievability or impossibility of the goal. The programmer, for instance, can use such conditions to design an agent that drops a goal when its original motivation is not present anymore (e.g., a canceled request from another agent), or when the goal, though achievable, has a high cost. It follows then that the flexible single-minded type of commitment that CAN agents enjoy lies between the simple single-minded strategy and the sophisticated, though underspecified, open-minded strategy [54].

4 CANPlan: Integrating Hierarchical Planning in BDI Languages

Since typical BDI systems are extremely flexible and responsive to the environment, they are well suited for complex applications with (soft) real-time reasoning and control requirements. Nonetheless, a limitation of such systems is that they normally do no *lookahead* reasoning: means-end analysis is entirely based on context sensitive (reactive) subgoal expansion, acting as they go. In some circumstances, however, lookahead deliberation (i.e., hypothetical reasoning) about the effects of one choice of expansion over another is clearly desirable, or even mandatory in order to avoid undesired situations. This is the case, for instance, when *(a)* important resources may be used in taking actions that do not lead to a successful outcome; *(b)* actions are not always reversible and may lead to states from which there is no successful outcome; *(c)* execution of actions take substantially longer than "thinking" (or planning); and *(d)* actions have side effects which are undesirable if they turn out not to be useful.

Another approach to means-end analysis is that of *automated planning*, a field that has experienced an outstanding progress in the last decade [27, 28, 42, 70]. Over the years, planning systems have been developed that are capable of solving large and complex problems, using richly expressive domain models and meeting advanced demands on the structure and quality of solutions [28]. Among others, heuristic search, forward search, graphplan-based mechanisms, and control knowledge are some of the techniques successfully used by current state-of-the art planners. In particular, we shall be interested here in *control knowledge* techniques, where domain independent planners are able to exploit user-provided domain information in order to guide the planning process. One such popular approach is that of hierarchical HTN-style planning [26, 28], in which domain knowledge is provided in the form of hierarchical information for decomposing complex tasks into simpler processes. As it will become evident below, HTN planners and BDI agent systems share many similarities, and therefore are suitable candidates for a principled integration.

So, in this section, we develop the full language CANPlan (CAN + planning), an extended version of CAN that incorporates an account of *(offline) lookahead* in the form of hierarchical HTN-style planning. Such a built-in planning mechanism will allow for a careful analysis, when necessary, on how to expand different plans. One could argue, of course, that it is always possible, in critical situations, to explicitly program lookahead within existing BDI systems. However, such code would generally be domain dependent, can be fairly complex,

and would lie outside the infrastructure support provided by the BDI agent platform. Alternatively, there are many frameworks that attempt to interleave BDI-type execution with offline planning (e.g., [2, 22, 37, 50, 72]). Still, these are mostly implemented systems with no precise semantics and with little programmer control over when to plan. The approach presented here, instead, is aimed to provide a formal specification of planning as a *built-in feature* of the BDI infrastructure that the programmer can use as appropriate. We have in fact implemented this as a distinguished new construct within Jack (see Section 6).

Before going into the details of CANPlan, let us first provide a brief overview of HTN-style planning planning.

4.1 HTN Planning

Hierarchical Task Network (HTN) planning is a well-known approach to automated planning based on the decomposition of (high-level) *tasks* into *subtasks* by applying HTN *methods*. Examples of HTN (implemented) systems include SHOP [47] and its successor SHOP2 [48]. Such systems have been applied in several domains and have a significant user base that includes government laboratories, industries, and universities [49]. SHOP2, in particular, excelled in the 2002 International Planning Competition [42].

From now on, we shall mostly follow the definitions of HTN-planning from [26]. The central concept in HTN planning is that of a *task*. There are two kinds of tasks. A *primitive task* is an action $act(\vec{x})$ which can be directly executed by the agent in the environment (e.g., $drive(x_1, x_2)$). A (high-level) *compound task* $e(\vec{x})$ is one that cannot be executed directly (e.g., $travel(origin, dest)$). A *task network* $d = [s, \phi]$ is a collection of tasks s that need to be accomplished and a boolean formula of constraints ϕ. Constraints impose restrictions on the ordering of the tasks $(e \prec e')$, on the binding of variables $(x = x')$ and $(x = c)$ (c is a constant), and on what literals must be true before or after a task $(l, e), (e, l)$, or during two tasks (e, l, e'). A *method* (e, d) encodes a way of decomposing a high-level compound task e into lower-level tasks using task network d. HTN methods thus provide the procedural knowledge of the domain.

Example 8. The method m_{travel} encodes one way of travelling to a close-by destination:

$$m_{travel} = \langle \textit{travelTo}(x),\ d_{taxi} \rangle;$$
$$d_{taxi} = [\{t_1 : \textit{getTaxi},\ t_2 : \textit{ride}(x, y),\ t_3 : \textit{payDriver}\}, \phi];$$
$$\phi = t_1 \prec t_2 \wedge t_1 \prec t_3 \wedge ((t_1, \textit{FlatTariff}) \vee t_2 \prec t_3) \wedge (\textit{At}(x) \wedge \textit{Close}(x, y), t_1).$$

Notice that, when traveling by taxi, one should always pay at the end of the trip, unless the tariff found after booking the taxi is *flat*. ■

An HTN *planning domain* $\mathcal{D} = \langle \Pi, \Lambda \rangle$ consists of a library Π of methods and a library Λ of primitive tasks. Each primitive task in Λ is a STRIPS-style operator with corresponding preconditions and effects in the form of *add* and *delete* lists. It is convenient to assume the existence of a dummy $noOp$ operator in Λ with empty precondition and no effects, used to decompose a task trivially. An HTN *planning problem* **P** is a triple $\langle d, \mathcal{B}, \mathcal{D} \rangle$ where d is the task network to be accomplished, $\mathcal{B}$ is the initial state (i.e., a set of all ground atoms that are true in $\mathcal{B}$), and $\mathcal{D}$ is a planning domain. A *plan* σ is a sequence $act_1 \cdot \ldots \cdot act_n$ of ground actions (that is, ground primitive tasks).

Given a planning problem **P**, the planning process involves selecting and applying an applicable reduction method from $\mathcal{D}$ to some compound task in d. This results in a new,

and typically more "primitive," task network d'. This reduction process is repeated until only primitive tasks (i.e., actions) are left. If no applicable reduction can be found for a compound task at any stage, the planner "backtracks" and tries an alternative reduction for a compound task previously reduced. If all compound tasks can eventually be reduced, a plan solution σ is obtained. The set of all plans that solves a planning problem $\mathbf{P} = \langle d, \mathcal{B}, \mathcal{D} \rangle$ is denoted $sol(d, \mathcal{B}, \mathcal{D})$; its definition provides a clear operational semantics for HTN planning.

We refer to [26, 28] for more details on HTN and its formal semantics.

4.1.1 BDI and HTN Systems: Similarities

BDI agent systems and HTN planners come from different communities and differ in many important ways. The former focus on the *execution* of plans, whereas the latter is concerned with the actual *generation* of such plans. The former are generally designed to *respond* to goals and information; the latter are designed to *bring about* goals. In addition, BDI systems are meant to be *embedded* in the real world and therefore take decisions based on a particular (current) state. Planners, on the other hand, perform *hypothetical reasoning* about actions and their interactions in multiple potential states. Thus, failure has very different meaning for these two types of systems. In the context of planning, failure means that a plan or potential plan is not suitable; within BDI agent systems failure typically means that an active (sub)plan ought to be aborted. Whereas backtracking upon failure is an option for planning systems, it is generally not for BDI systems, as actions are taken in the real world.

In spite of all the above differences, BDI agent-oriented programming languages and HTN planners share many similarities [19, 24, 71], both in terms of the type of knowledge they use as well as on how such knowledge is manipulated to create solutions. First of all, HTN systems and BDI languages assume an explicit representation of the agent's knowledge (i.e., the state or belief base) and a set of primitive tasks or actions that the agent can directly execute in the world. Secondly, and most importantly, procedural knowledge about the domain is available in both HTN and BDI systems in the form of reduction methods and plan rules, respectively. HTN methods and BDI plan rules are meant to describe the "standard operating procedures" that are normally used to carry on common tasks in some domain, thus corresponding well to the way that users/experts think about problems. Thirdly, both systems create solutions by reducing higher-level entities into lower-level ones using a given set of reduction "recipes." Whereas a BDI system "reduces" an event into a plan-body program using a plan from the plan library, an HTN planner reduces a compound task into a task network using a reduction method from the method library. Figure 2 gives an indication of the mapping between HTN and BDI entities and notions.[17]

Example 9. The corresponding plan-rule for the traveling method m_{travel} described in Example 8 is as follows:

$$travelTo(y) : At(x) \wedge Close(x, y) \leftarrow !getTaxi; ?(FlatTariff); (ride(x, y) \parallel !payDriver).$$

Hence, the network d_{taxi} corresponds to the rule's plan-body. ■

Special consideration has to be taken when considering goal-programs within a planning context. The goal-construct as formulated here is not available in most BDI agent systems and no direct construct exists within HTN planners either. Nonetheless, a program $\mathsf{Goal}(\phi_s, P, \phi_f)$ within the context of a planning construct can be understood as "*searching*

[17] The table is not complete; whereas some entities have a straightforward mapping, some others require a more elaborate translation; see [19, 71] for a more detailed mapping.

BDI SYSTEMS	HTN SYSTEMS
belief base	state
plan library	method library
event	compound task
action	primitive task
plan-body/program	network task
plan rule	method
plan rule context	method precondition
test $?\phi$ in plan-body	state constraints
sequence ; in plan-body	ordering constraint $\prec$
parallelism $\|\|$ in plan-body	no ordering constraint
goal-programs $\mathsf{Goal}(\phi_s, P, \phi_f)$	task P with a constraint (P, ϕ_s)
relevant plans for an event	matching methods for a task
plan selection	task reduction
successful execution of plan	task-network solution

Fig. 2 Comparison between BDI and HTN systems.

for a solution of P that would bring about a state of affairs where ϕ_s holds." Because of that, we shall see goal-programs of the form $\mathsf{Goal}(\phi_s, P, \phi_f)$ within a planning context as (if they were) programs of the form $(P; ?\phi_s)$.[18]

4.2 A Local lookahead planner for CAN

In incorporating planning into BDI programming languages, several issues need to be addressed. First, we want to keep the language as uniform as possible. Second, we want to allow control over when and on what planning is to be performed within the BDI architecture. Third, we need to decide what domain information the planner will use—we want the planner to re-use as much information as possible from an existing BDI specification. Lastly, the result of the planning process ought to be carried on, and possibly monitored, within the BDI execution cycle in a uniform manner.

In a nutshell, we shall enhance CAN with a form of *on-demand planning* by adding a new construct $\mathsf{Plan}(P)$ to the language: *plan for P offline (i.e., without actually executing P), searching for a complete hierarchical decomposition*. Thus, on program $\mathsf{Plan}(P)$, the agent is meant to *deliberate* about how to perform P before committing to even its first step. The obtained extended language will be called CANPlan (CAN + planning).

As with other constructs in the language, we need to provide the operational rules for the Plan construct. To do this, we distinguish, from now on, between two types of intention-level transitions, namely, "bdi" and "plan" (labelled) transitions. Intuitively, bdi-type steps will be used to model the normal BDI execution cycle, whereas plan-type transitions will represent (internal) deliberation steps within a *planning* context. When no label is stated, both apply.

Following the semantics of the so-called "search operator" in the IndiGolog logic-based agent programming language [17, 58], the main operational rule states that a basic configuration $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle$ can evolve to configuration $\langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P')\rangle$ provided that configuration $\langle \mathcal{B}, \mathcal{A}, P\rangle$ can evolve to configuration $\langle \mathcal{B}', \mathcal{A}', P'\rangle$ from where it is possible to reach a

[18] One can easily get closer approximations to goal-programs which take the success and failure conditions into account *before* trying P. This would amount to understanding goal-programs within planning as programs of the form **if** ϕ_s **then** $?\mathit{true}$ **else** $(?\phi_f; P; ?\phi_s)$. Such a translation would instruct the planner whether it is worth decomposing P for ϕ_s, as a kind of control knowledge at the outset of the planning process. However, the translation required is notationally more involved and we therefore avoid it here for legibility purposes.

final configuration in a finite number of *planning* steps (recall from Section 2.2, pp. 6, that $\xrightarrow{\mathsf{plan}}{}^{*}$ stands for the reflexive transitive closure of transition relation $\xrightarrow{\mathsf{plan}}$):

$$\frac{\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}', \mathcal{A}', P'\rangle \quad \langle \mathcal{B}', \mathcal{A}', P'\rangle \xrightarrow{\mathsf{plan}}{}^{*} \langle \mathcal{B}'', \mathcal{A}'', \mathit{nil}\rangle}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P')\rangle}\ \mathsf{P}$$

Intuitively, the Plan construct guarantees that a "*safe*" step is chosen, that is, a step that is on an execution path that is guaranteed to eventually succeed. In addition, by propagating the Plan construct to the remaining program, only such safe evolutions will be selected throughout the whole execution of the program.

Also, two simpler rules are used to terminate empty planning problems, and to handle nested planning problems (i.e., a planning step already within a planning plan context):

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(\mathit{nil})\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \mathit{nil}\rangle}\ \mathsf{P}_{end} \qquad \frac{\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P')\rangle}\ \mathsf{P}_{\mathsf{P}}$$

In addition to the rules for the Plan construct, we need to restrict the applicability of some existing rules in order to adequately integrate the new construct within the BDI execution cycle and the Goal construct—planning is *not* merely lookahead on the BDI execution cycle.

First of all, we shall distinguish failure during BDI execution (usually triggering the failure recovery mechanism) from failure during planning (usually dealt with backtracking). To that end, we confine the *failure handling* mechanism only to the online execution, by restricting derivation rule $\rhd_{restart}$ (see Section 2, pp. 3) to the bdi context, that is:

$$\frac{P_1 \neq \mathit{nil} \quad \langle \mathcal{B}, \mathcal{A}, P_1\rangle \xrightarrow{\mathsf{bdi}} \not\quad \langle \mathcal{B}, \mathcal{A}, P_2\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', P_2'\rangle}{\langle \mathcal{B}, \mathcal{A}, P_1 \rhd P_2\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', P_2'\rangle}\ \rhd_f^{\mathsf{bdi}}$$

By using this variant of rule $\rhd_{rec}$, only the BDI execution cycle would be allowed to re-try alternative plans for an event upon the failure of some strategy. So, for example, a program of the form $(?\mathit{false} \rhd e : (\!|\Delta|\!))$ would have no transition within a plan context, whereas alternatives in Δ would be tried within a bdi context.

Another distinction that needs to be made is between goal adoption during online execution (as in CAN) from goal adoption during planning reasoning. To do so, we restrict the previous rule G_{adopt} (see Section 3, pp. 19) to bdi type of transitions, and provide the following alternative rule for adopting declarative goals when planning:

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Goal}(\phi_s, !e, \phi_f)\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}, \mathcal{A}, (!e; ?\phi_s)\rangle}\ \mathsf{G}^{\mathsf{plan}}_{adopt}$$

That is, during planning, the intended meaning of a goal $\mathsf{Goal}(\phi_s, !e, \phi_f)$ is "*plan for a (total) execution of* $!e$ *that will bring about* ϕ_s." Notice that since, in principle, all possible ways of resolving $!e$ will be considered at planning time, there is no need to check for feasibility via failure condition ϕ_f. In this way, the Goal construct becomes relevant only when executed in an *online* fashion, which is compatible with its original motivations in [73]. (Again, more involved adoption mechanisms are conceivable in the context of particular planning techniques, such as using ϕ_f as declarative control knowledge information [28].)

Finally, agent-level transitions should now rely on *online* intention-level transitions. Technically, the intention-level transitions used in rules A_{int} (pp. 11) and A^1_{goal} (pp. 24) should now be based on bdi-type intention-level transitions.

This concludes the formal integration of a local planning mechanism into the BDI language, yielding thus the language CANPlan. It involved four new intention-level rules, and the confinement of two intention-level and two agent-level rules to the bdi context only—all remaining rules for CAN are therefore available within both bdi and plan contexts.

The first main result of this section states that construct $\mathsf{Plan}(P)$ guarantees, under plausible assumptions, the successful execution of program P. Intuitively, this means that the planning construct provides a careful plan selection analysis that will yield non-failing executions: as long as there are no negative changes in the environment and the agent does not itself pursue an interfering intention, the execution of P, which the agent has planned for, will not get stuck at any point. Informally, the reason why this theorem applies is that each step performed on a $\mathsf{Plan}(P)$ program is "safe," in that a successful continuation of it does exist, together with the fact that the Plan construct itself is propagated (see main rule P).

Theorem 6. *Let C be a an agent configuration such that intention $I = \langle id, \mathsf{Plan}(P)\rangle \in C[\Gamma]$ is not blocked in C. Let $E = C_0 \cdot \ldots \cdot C_n$ be a BDI execution of $C_0 = C$, where $I_i = \langle id, \mathsf{Plan}(P_i)\rangle \in C_i[\Gamma]$ is the evolution of I throughout E, such that for every $i < n$:*

(a) $I_{i+1} = I_i$ *and* $C_{i+1}[\mathcal{B}] = C_i[\mathcal{B}]$*; or*
(b) $\langle C_i[\mathcal{B}], C_i[\mathcal{A}], \mathsf{Plan}(P_i)\rangle \xrightarrow{\mathsf{bdi}} \langle C_{i+1}[\mathcal{B}], C_{i+1}[\mathcal{A}], \mathsf{Plan}(P_{i+1})\rangle$.

Then, intention I_i is not blocked in C_i, for every $i \leq n$.

Proof. We prove this by induction on n. The base case, when $n = 0$, is trivial by assumption: I_0 is not blocked in C_0. Suppose the result holds for $n = k$ and take an execution $E = C_0 \cdot \ldots \cdot C_k \cdot C_{k+1}$ under the above assumptions, and thus, either *(a)* or *(b)* applies between C_k and C_{k+1}. If case *(a)* applies in E, then $I_k = \langle id, \mathsf{Plan}(P_k)\rangle$ would also be blocked at configuration C_k, since $P_{k-1} = P_k$ and $\mathcal{B}_k = \mathcal{B}_{k+1}$, thus reaching a contradiction with the induction hypothesis. So, it has to be the case that $\langle \mathcal{B}_k, \mathcal{A}_k, \mathsf{Plan}(P_k)\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}_{k+1}, \mathcal{A}_{k+1}, \mathsf{Plan}(P_{k+1})\rangle$, that is, the last step in E actually involves the execution of intention I. This can only hold due to derivation rule P and hence $\langle \mathcal{B}_k, \mathcal{A}_k, \mathsf{Plan}(P_k)\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}_{k+1}, \mathcal{A}_{k+1}, \mathsf{Plan}(P_{k+1})\rangle$ and $\langle \mathcal{B}_{k+1}, \mathcal{A}_{k+1}, \mathsf{Plan}(P_{k+1})\rangle \xrightarrow{\mathsf{plan}^*} \langle \mathcal{B}', \mathcal{A}', \mathit{nil}\rangle$, for some $\mathcal{B}'$ and $\mathcal{A}'$. This, in turn, implies that there exists $\ell > 0$ such that $\langle \mathcal{B}_{k+1}, \mathcal{A}_{k+1}, \mathsf{Plan}(P_{k+1})\rangle \xrightarrow{\mathsf{plan}^\ell} \langle \mathcal{B}', \mathcal{A}', \mathit{nil}\rangle$ ($\ell \neq 0$ since $\mathsf{Plan}(P_{k+1}) \neq \mathit{nil}$). If $\ell = 1$, then $P_{k+1} = \mathit{nil}$ and transition $\langle \mathcal{B}_{k+1}, \mathcal{A}_{k+1}, \mathsf{Plan}(P_{k+1})\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}_{k+1}, \mathcal{A}_{k+1}, \mathit{nil}\rangle$ applies due to rule P_{end}. Otherwise, if $\ell > 1$, then rule P can be applied and $\langle \mathcal{B}_{k+1}, \mathcal{A}_{k+1}, \mathsf{Plan}(P_{k+1})\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}_{k+2}, \mathcal{A}_{k+2}, \mathsf{Plan}(P_{k+2})\rangle$, for some $\mathcal{B}_{k+2}$, $\mathcal{A}_{k+2}$, and P_{k+2}. In both cases, intention I_{k+1} cannot be blocked in C_{k+1} and the thesis follows. □

Thus, by using the new lookahead construct $\mathsf{Plan}(P)$, the programmer can make sure—to some extent—that *failing* executions of program P will be avoided. This contrasts with the usual (default) BDI execution of P which may potentially *fail* program P due to wrong decisions at choice points. Nonetheless, it should be clear that the proposed deliberation module is *local*, in the sense that it does not take into account the potential interactions with the external environment and other concurrent intentions. Still, notice that the above theorem does account for some (limited) interleaved execution of other intentions, as long as these do not produce world-changing actions.[19]

[19] Such constraints could be lifted, for instance, if a meta-level module is able to interleave intentions that do not interact negatively [63].

Example 10. Let us come back to the plan rule in Example 5. As discussed, the first step in the strategy is to arrange transportation to the destination (e.g., arrange flight), accommodation (e.g., book hotel), and local transportation (e.g., rent a car). Because these three tasks may depend on shared resources (e.g., money), it could be the case that choices performed in one of the these three sub-tasks impact negatively, and preclude, the successful completion of the other tasks. Indeed, the system may book successfully an expensive flight, only to realize that no hotel can be booked with the remaining funds.

To address this problem, the programmer can rely on local planning and write the following alternative strategy:

$$\begin{aligned} &\mathit{doTrip}(dest, reason) : \mathit{Work}(reason) \leftarrow \\ &\qquad \mathsf{Plan}[(!\mathit{arrangeTransp}(dest) \| \; !\mathit{arrangeHotel}(dest) \| \; !\mathit{arrangeLocalCar}(dest))]; \\ &\qquad !\mathit{travelTo}(dest); \; !\mathit{doWork}; \; ?\mathit{Address}(\mathit{Home}, addr); \; !\mathit{travelTo}(addr). \end{aligned}$$

Hence, it is now left to the lookahead reasoning module to make sure that, in resolving the three subgoal events, the (right) choices are made in such a way that all three can be successfully completed. ■

Before, we argued that the CAN language was an incremental extensions over the CAN$^{\mathcal{A}}$ one. Here, we show that CANPlan is an *incremental* extension over the CAN language—CAN agents are CANPlan agents with no planning.

Theorem 7. *Let C be a planning-free CANPlan agent (i.e., one where library Π and intention base Γ do not mention construct* Plan*). Then, E is a CANPlan BDI execution of C relative to $\mathcal{E}$ iff E is a CAN BDI execution of C relative to $\mathcal{E}$, for any environment $\mathcal{E}$.*

Proof. Direct from the fact that the top-level agent rules are the same for both languages and none of the three derivation rules for Plan ever apply for planning-free agents. □

4.3 HTN-style planning via the Plan construct

In Section 4.1.1, we informally reviewed the relation between BDI agent systems and HTN planners. Here, we focus on the formal relationship between our planning construct Plan and HTN-style planning. To that end, we first define *bounded* agents as those CANPlan agents whose belief base and belief conditions are defined in a language which follows the same constraints as those imposed by HTN planners [26] (e.g., first-order atoms, finite domains, closed world assumption). It is worth pointing out that, in practice, most existing BDI programming language implementations do actualise such constraints. Furthermore, we assume, without loss of generality, that bounded agents do not make use of belief update statements $+b$ and $-b$ in their plans—only primitive actions can change the belief base.[20]

The second main result of this section establishes, formally, the link between the Plan construct and HTN planning. Intuitively, any total execution of a planning problem in our agents corresponds one-to-one to an HTN solution. More concretely, if a full intention-level execution resolving an event e yields a sequence of primitive actions σ, then σ is indeed a valid HTN solution for the corresponding abstract task e, and vice-versa. (Recall from Section 4.1 that $sol(e, \mathcal{B}, \langle \Pi, \Lambda \rangle)$ is the set of all HTN solutions for task e.)

[20] Belief update statements can be easily modelled using special actions with empty preconditions.

Theorem 8. *For any libraries Π and Λ, and belief bases $\mathcal{B}$ and $\mathcal{B}'$ in a bounded agent, and for any action sequences $\mathcal{A}$ and σ, and event e:*[21]

$$\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \mathsf{Plan}(!e)\rangle \xrightarrow{\mathsf{bdi}}{}^{*} \langle \Pi, \Lambda, \mathcal{B}', \mathcal{A}\cdot\sigma, nil\rangle \ \textit{ iff } \ \sigma \in sol(e, \mathcal{B}, \langle \Pi, \Lambda\rangle).$$

Proof. See Appendix A. □

As a direct consequence, a Plan step within the BDI execution may evolve if and only if the corresponding HTN problem has a solution; formally, $\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \mathsf{Plan}(!e)\rangle \xrightarrow{\mathsf{bdi}}$ iff $sol(e, \mathcal{B}, \langle \Pi, \Lambda\rangle) \neq \emptyset$, for any bounded agent. Thus, one could indeed see construct Plan as an HTN planner over the same domain knowledge as the BDI agent. In other words, provided we restrict to the language of HTN, our deliberator construct Plan provides a built-in HTN planner within the whole BDI framework.

The importance of the above result is twofold. Theoretically, it shows that the BDI execution model and the HTN planning framework are strongly related. More concretely, by a suitable HTN understanding of the BDI structures (see Figure 2) and by merely "turning off" the BDI failure handling mechanism, one obtains the essence of an HTN planner. The fact that the changes required to the BDI architecture for modelling HTN-style planning are minimal does not diminish the mixed account of execution and planning. On the contrary, it demonstrates that one could get both types of systems integrated in a parsimonious and uniform manner. Practically, the result above supports the use of existing HTN planning systems, such as SHOP or SHOP2, within current BDI platforms, such as Jason or Jack. As a matter of fact, one would not expect the BDI system itself to do the lookahead planning reasoning, that is, implement the derivation rule P, but an external HTN planner to do so, whose output plan shall be used in the BDI language.

Another important issue to investigate is how the executions obtained via planning relate to classical BDI executions. In particular, we want to know the impact of doing planning on a basic agent, that is, one corresponding basically to classical BDI agent programming languages like AgentSpeak or PRS. Roughly speaking, we show that doing planning within the classical BDI execution cycle reduces to *intelligent* plan selection. To show that, we first prove the following intermediate result, stating that in the context of classical BDI agents, every legal planning step can be *mimicked* with a corresponding non-planning step.

Lemma 1 *For every program P not mentioning construct* Goal *or construct* Plan*, if* $\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle \xrightarrow{\mathsf{bdi}} \langle \Pi, \Lambda, \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P')\rangle$*, then* $\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{bdi}} \langle \Pi, \Lambda, \mathcal{B}', \mathcal{A}', P'\rangle$.

Proof. By assumption, $\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}} \langle \Pi, \Lambda, \mathcal{B}', \mathcal{A}', P'\rangle$ due to the application of some set χ_{plan} of plan-type intention-level derivation rules. Since P does not mention any Plan or Goal construct, then $\mathsf{P_P} \notin \chi_{\mathsf{plan}}$ and $\mathsf{G}^{\mathsf{plan}}_{adopt} \notin \chi_{\mathsf{plan}}$, and thus, all the rules in χ_{plan} have their counterpart as bdi-type rules. It follows then that $\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{bdi}} \langle \Pi, \Lambda, \mathcal{B}', \mathcal{A}', P'\rangle$. □

With this result at hand, one can then show that any agent execution that successfully terminates a planning intention can be *simulated* by the classical BDI execution engine.

Theorem 9. *Let C be a CANPlan agent and $I = \langle id, \mathsf{Plan}(P)\rangle \in C[\Gamma]$ an active intention in C, where program P and library $C[\Pi]$ do not mention any* Plan *or* Goal *construct. Let agent*

[21] For legibility, we keep the translation between the BDI domain knowledge (i.e., libraries Π and Λ, and programs) and the HTN domain knowledge (i.e., planning domain and task networks) implicit.

C' be like C but with intention base $C'[\Gamma] = (C[\Gamma] \setminus \{I\}) \cup \{\langle id, P\rangle\}$. Suppose next that $C_0 \cdot \ldots \cdot C_n$ is a CANPlan-BDI execution of $C_0 = C$ such that $\langle id, \mathsf{Plan}(nil)\rangle \in C_n[\Gamma]$. Then, there is a CANPlan-BDI execution $C'_0 \cdot \ldots \cdot C'_n$, of agent $C'_0 = C'$ such that C'_i is like C_i but with intention base $C'_i[\Gamma] = (C_i[\Gamma] \setminus \{\langle id, \mathsf{Plan}(P_i)\rangle\}) \cup \{\langle id, P_i\rangle\}$, for every $i \in \{0, \ldots, n\}$.

Proof. It is easy to see that for each $i \in \{1, \ldots, n\}$, $\langle id, \mathsf{Plan}(P_i)\rangle \in C_i[\Gamma]$, for some P_i. The theorem then follows directly from Lemma 1 and the fact that, because P and Π are Goal and Plan free, so are each of the programs P_i along the execution. □

So, the execution cycle itself can obtain the same outcomes as the planning module does, provided the correct plan choices are made throughout the execution.

On the other hand, when the full CANPlan language is considered, the interaction between planning, concurrency, and goal-programs, makes planning more than just lookahead on the BDI execution cycle. In fact, program $\mathsf{Plan}(P_1)\|P_2$ may not be able to imitate program $\mathsf{Plan}(\mathsf{Plan}(P_1)\|P_2)$, as the latter is equivalent to executing $\mathsf{Plan}(P_1\|P_2)$—a Plan construct is just *redundant* within the context of another Plan construct due to rule $\mathsf{P_P}$.[22] Similarly, program $\mathsf{Goal}(\phi_s, P, \phi_f)$ may not be able to simulate all executions of $\mathsf{Plan}(\mathsf{Goal}(\phi_s, P, \phi_f))$, as the offline and online interpretations of goal-programs differ.

Surprisingly, also, the BDI execution engine may obtain (other) successful executions that the planner cannot produce. More concretely, due to the unavailability of failure recovery at planning time, the built-in planner cannot always imitate the behavior of an intention totally executed online, i.e., with no lookahead.

Example 11. Consider an agent configuration where all actions are possible, propositions p and q are both false initially (i.e., $\mathcal{B} \models \neg p \wedge \neg q$), and action act_1's effect is to make p true (i.e., $act_1 : \mathit{true} \leftarrow \{p\}; \{\} \in \Lambda$). Next suppose the agent's plan library Π contains only two plan rules for handling an event e, namely, $e : \mathit{true} \leftarrow act_1; ?q; act_2$; and $e : p \leftarrow act_3; act_2$.

First, there is no solution for program $\mathsf{Plan}(!e)$, that is, $\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \mathsf{Plan}(!e)\rangle \overset{\mathsf{plan}}{\not\longrightarrow}$: the first plan rule would fail when testing for q, and the second one is simply not applicable. On the other hand, a successful BDI execution of program $!e$ can be obtained, by partially executing the plan-body of the first plan rule (i.e., executing action act_1) and then—upon failure on test $?q$—fully executing the plan-body of the second plan rule, whose context would now hold true due to the execution of action act_1. ■

As one can observe, the above counter-example relies on both the plan failure handling mechanism built into the BDI execution cycle and the programmer not having provided a full set of plans. In fact, if the plan library had also included a third rule of the form $e : \mathit{true} \leftarrow act_1; ?p; act_3; act_2$, then the planner would have found a full execution. Still, as agent's plan libraries are often developed incrementally and in modules, the above situation could very well arise. Notice also that the agent used is basically a $\mathsf{CAN}^{\mathcal{A}}$ agent, so the result is independent of both the planning and goal-program constructs.

Summarizing, the combined framework of (default) BDI execution plus local hierarchical planning is strictly more general than hierarchical planning alone. Furthermore, as discussed after Theorem 6, by using the new local planning mechanism the programmer can rule out—to some extent—BDI executions that are bound to fail.

[22] In other words, there is no notion of "nesting" planning. An alternative language where $\mathsf{Plan}(\mathsf{Plan}(P_1)\|P_2)$ is not equivalent to $\mathsf{Plan}(P_1\|P_2)$ can be easily obtained by dropping derivation rule $\mathsf{Plan_P}$ and making rule Plan also available within the plan context. Then, Theorem 9 would also hold when Plan constructs are mentioned in either Π or P. Nonetheless, such alternative language would require an account of HTN planning within an HTN planner. We stick here to the standard non-nested version of HTN planning.

4.4 Active goals in CANPlan

We finish by generalizing some of the definitions and results given for CAN agents regarding goals, now in the potential presence of planning programs. Besides event and declarative goals (Definitions 6 and 11), we also distinguish a third kind of active goals in CANPlan.

Definition 12 (Active Planning Goal). An *active planning goal* is a program of the form $G = \mathsf{Plan}(P)$, where $\mathsf{Plan}(P)$ itself is said to be the goal's *current strategy*. ■

Note that planning goals do not have alternative strategies—alternatives are only meaningful during online execution. With this new type of active goal, we further extend the notion of active goal traces (Definition 10) accordingly (see second case).

Definition 13 (Active Goal Trace, for CANPlan agents). An *active goal trace* λ is a sequence of active goals $G_1 \cdot \ldots \cdot G_n$. The multiset of all active goal traces in a program P, denoted $\mathcal{GT}race(P)$, is inductively defined as follows:

✓

$$\mathcal{GT}race(P) = \begin{cases} \emptyset & \text{if} \quad P = \mathit{nil} \mid \mathit{act} \mid ?\phi \mid +b \mid -b \\ \{P \cdot \lambda \mid \lambda \in \mathcal{GT}race(P_1)\} & \text{if} \quad P = \Omega,\ \mathcal{GT}race(P_1) \neq \emptyset \\ \{P\} & \text{if} \quad P = \Omega,\ \mathcal{GT}race(P_1) = \emptyset \\ \{P\} & \text{if} \quad P = e{:}(\!|\Delta|\!) \\ \mathcal{GT}race(P_1) & \text{if} \quad P = P_1; P_2 \\ \mathcal{GT}race(P_1) \uplus \mathcal{GT}race(P_2) & \text{if} \quad P = P_1 \| P_2 \end{cases}$$

where $\Omega = P_1 \rhd e{:}(\!|\Delta|\!) \mid \mathsf{Goal}(\phi_s, P_1 \rhd P_2, \phi_f) \mid \mathsf{Plan}(P_1)$. ■

The full-fledged version of Theorem 1 (for CAN$^{\mathcal{A}}$) and Theorem 3 (for CAN) in the context of planning goals can now be restated as follows. Observe that this result accommodates the fact that an active goal may also be abandoned if it is instrumental to a (higher) motivating *planning goal* for which there is no (total) solution, as the successful achievement of such instrumental goal is meaningless if the motivating planning goal is unsolvable.

Theorem 10. *Let C be a CANPlan agent and $I \in C[\Gamma]$ be an active intention in C. Furthermore, let $G_k = \lambda[k]$, with $k \geq 1$, be the k-th active goal in some active goal trace $\lambda \in \mathcal{GT}race(I)$. If G_k's current strategy is blocked, then for every subgoal $G_{k'} = \lambda[k']$, with $k' > k$, either:*

1. *goal $G_{k'}$ is fully blocked in C, i.e., its current strategy is blocked and it has no feasible (applicable) alternative strategy; or*
2. *there exists $k \leq k'' < k'$, such that goal $\lambda[k'']$ is an active but blocked planning goal.*

Proof. On the contrary, suppose $G_{k'}$ is not part of a higher-level planning goal $G_{k''}$ that is instrumental to G_k. If $G_{k'}$'s current strategy is not blocked or $G_{k'}$ has an alternative applicable strategy, then $G_{k'}$ is not fully blocked and it can evolve a single step. Since there is no planning goal between G_k and $G_{k'}$, then G_k's current strategy is not blocked, as a step on it can be performed by evolving its instrumental subgoal $G_{k'}$. □

So, a planning goal takes precedence over all its subsidiary goals, since all of them ought to be successfully solved in order to solve the planning goal itself.

In addition, Theorems 4 and 5 (pp. 25) apply also for CANPlan agents, and therefore, goal-programs behave the same *when executed online*. This is because, due to the way declarative

goal-programs are adopted at planning time via rule $\mathsf{G}^{\mathsf{plan}}_{adopt}$ (pp. 31), there could never be an active declarative goal instrumental to an active planning goal.[23]

We close by pointing out a particular powerful combination of the Goal and Plan constructs in CANPlan, namely, $\mathsf{Plan}(\phi_s, P, \phi_f) \stackrel{\text{def}}{=} \mathsf{Goal}(\phi_s, \mathsf{Plan}(P; ?\phi_s), \phi_f)$. Under such construct, the benefits of lookahead analysis and online goal monitoring are combined together. In contrast with the simpler version $\mathsf{Plan}(P; ?\phi_s)$, should ϕ_s become true while executing P, either fortuitously or due to P itself, the whole program terminates with success. Similarly, should condition ϕ_f become true while P is being performed, P is terminated with failure.

5 Handling of Variables

Clearly, the use of variables in agent programs is *mandatory* for any practical programming language. However, accounting for variables in the semantics of the language poses several technical difficulties, yielding a formal framework that is considerably more complicated notationally. At an abstract level, when considering variables in programs we need to account for the following issues:

1. Test programs may hold with different bindings and action programs may mention variables that need to be resolved before they can be performed.
2. Variables in plan-body programs need to be propagated along their execution.
3. Variables in plan rule context conditions imply that a strategy may be applicable for different instantiations of such variables. This, in turn, implies that a more involved formalization of failure recovery is required since the agent may want to try the same strategy with different bindings (e.g., with different domain objects).
4. Careful handling of shared variables in parallel programs—e.g., variable x in program $P_1(x) \parallel P_2(x)$—is required to avoid undesired side-effects, where "failed" bindings done in one program are used by the other concurrent programs. For instance, an event within P_1 may bind variable x to some object c_1, only to fail afterwards and be recovered by an alternative strategy that would successfully end up binding x to c_2. In such cases, program P_2 should never make use of the first failed binding $x = c_1$.

Whereas the handling of the first two issues is relatively straightforward and standard in the literature, the last two are more involved and would yield more complex derivation rules than the ones discussed in the paper.

So, in what follows, we shall discuss the major changes to the framework developed above that are required to accommodate variables in agent programs.

5.1 Substitutions

Below, we shall extensively appeal to *substitutions*, also called *variable bindings*, which will be denoted using θ or η, possibly with annotations.

Definition 14 (Lloyd [41]). A *substitution* θ is a finite set of the form $\{x_1/t_1, \ldots, x_n/t_n\}$, where each x_i is a distinct variable and each t_i is a term distinct from x_i. θ is called a *ground substitution* if the t_i are all ground terms, a *variable-pure substitution* if the t_i are all variables, and a *non-variable substitution* if no t_i is a variable. ■

[23] Technically, for every plan-body program P and goal trace $\lambda \in \mathcal{G}(P)$, if the n-th active subgoal in λ is a planning goal, then $\lambda[m]$, for any $m > n$ is not an active declarative goal.

Observe that while ground substitutions are non-variable ones, the converse is not true (e.g., $\theta = \{x/cell(12, y)\}$). Informally, non-variable substitutions do not contain any "renaming" of variables.

As usual, $\theta\theta'$ denotes the *composition* of substitutions θ and θ'. When E is an expression, $E\theta$ is the expression obtained from E by simultaneously replacing each occurrence of the variable x_i in E with the term t_i, for all $i \in \{1, \ldots, n\}$. For example, if $\phi = \mathit{Holding}(x, y)$ and $\theta = \{x/\mathit{John}, y/\mathit{box}(x)\}$, $\theta' = \{x/23\}$, then we obtain $\phi\theta = \neg\mathit{Holding}(\mathit{John}, \mathit{box}(x))$, and $\phi\theta\theta' = \mathit{Holding}(\mathit{John}, \mathit{box}(23))$.

In the rest of the paper, we shall make use of following convenient notation. The the set of all *most general unifiers* between expressions E_1 and E_2 is denoted $\mathsf{mgu}(E_1, E_2)$. Set $\mathsf{ren}(X, Y)$ will denote the set of all *renaming* substitutions for variables in X without using variables in Y, that is, the set of all variable-pure substitution of the form $\{x_1/y_1, \ldots, x_n/y_n\}$, where $X = \{x_1, \ldots, x_n\}$ and such that $\{y_1, \ldots, y_n\} \cap (X \cup Y) = \emptyset$. When θ is a variable-pure substitution, θ^{-1} stands for the "inverse" of θ, that is, $\theta^{-1} = \{x/y \mid y/x \in \theta\}$. (Of course, the inverse operation is not well-defined for non variable-pure substitutions.) Finally, we will use $\mathsf{vars}(E)$ to denote the set of all free variables in expression E (e.g., $\mathsf{vars}(\psi)$, $\mathsf{vars}(P)$, or even $\mathsf{vars}(\theta)$); and $\widehat{\theta}$ to denote the (induced) equality formula $\bigwedge_{x/t \in \theta} x = t$.

5.2 Definition Generalizations

When considering variables, some concepts need to be extended to account for variable bindings. An *intention* I is now tuple $\langle id, P, \eta\rangle$, where $id \in \mathbb{N}$ is the (unique) intention identifier, P is a, possibly open, program term in the full program language, and η is the set of (current) variable bindings for P. The *intention insertion operation* $\Gamma \Cup \gamma$ (Definition 1) denotes the intention base resulting from incorporating each $P \in \gamma$ into intention base Γ, as a new intention of the form $\langle id, P, \emptyset\rangle$, where id is the intention's *unique* identifier (i.e., no other intention in $\Gamma \Cup \gamma$ shares the same identifier). Finally, *intention-level configurations* are extended to tuples of the form $\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \eta, P\rangle$, where η stands for the current variable bindings performed so far in the execution of the program. To generalize Definition 4, we say that a program P is *blocked* in an agent configuration $\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma\rangle$ iff $\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \emptyset, P\rangle \not\longrightarrow$. An intention $I = \langle id, P, \eta\rangle$ is *blocked* in an agent configuration if program $P\eta$ is blocked.

As discussed above, when formulas have free variables (e.g., $\mathit{Holding}(x)$), they can hold in a belief base relative to some bindings of such variables—a formula may hold for different domain objects. This will have a substantial impact when an agent executes test programs of the form $?\phi(\vec{x})$ as well as when it evaluates plan rules' context conditions of the form $\psi(\vec{x})$. To handle this, we define the notion of an *answer* for a query. Informally, an answer is a grounding of every variable except those ones constrained only by equality atoms; for such variables, we look for most "general" answers.

Definition 15. Let $\mathcal{B}$ be a belief base and ψ be a test formula, possibly with free variables. A substitution θ is an *answer* to ψ relative to $\mathcal{B}$, written $\mathcal{B} \models\!\!\!\!\cdot\; \psi\theta$, iff

1. $\mathcal{B} \models \psi\theta$;
2. each variable in $\mathsf{vars}(\psi\theta)$ appears in equality atoms in ψ only;
3. θ is a most general substitution satisfying the previous two conditions, i.e., there is no θ' satisfying the first two conditions such that $\mathsf{vars}(\psi\theta) \subset \mathsf{vars}(\psi\theta')$. ■

Informally, the reason why we treat equality atoms differently is because we will use such atoms to encode plan relevance constraints (i.e., an actual event posting *unifying* with the

triggering event of a plan rule). The above definition takes a least-committed approach to such atoms, whereas tests of domain belief atoms (e.g., $?At(x)$) are used by the BDI programmer to obtain *concrete* groundings (e.g., $x = Home$).

So, if $\psi = Holding(x, y) \wedge x = John \wedge w = z$, then $\theta = \{x/John, y/box(23), w/z\}$ is an answer provided that $\mathcal{B} \models Holding(John, box(23))$ holds. However, substitution $\theta' = \{x/John, y/box(23), w/box(1), z/box(1)\}$ is not an answer, as it over-commits to variables w and z. See that any answer for ψ should ground variables x and y since they appear in a belief atom.

Finally, the set of *active goal traces* (Definition 7) for intentions should now be redefined as expected, namely, $\mathcal{GT}race(\langle id, P, \eta\rangle) = \mathcal{GT}race(P\eta)$. Similarly, the *set of all active goals* in a program P is defined as $\mathcal{G}(P) = \{\lambda[k] \mid \lambda \in \mathcal{GT}race(P), k \geq 1\}$, and for intentions as $\mathcal{G}(\langle id, P, \eta\rangle) = \mathcal{G}(P\eta)$. The multiset of active declarative goals in an intention is now redefined as $\mathcal{DG}(\langle id, P, \eta\rangle) = \mathcal{DG}(P\eta)$.

5.3 Derivation Rules Extensions

Here we will go over the derivation rules for CANPlan that would require more than trivial changes. We refer to Appendix B for the complete set of rules for the language with variables.

Event Handling The rule in charge of constructing the set of relevant plans for an event, namely rule Ev, needs to be adapted so that *(i)* fresh variables are used form the plan library to clashing with the variables already being used in the intention; and *(ii)* the "matching" of the actual event posting with the triggering event of plan rules handles event with variables.

$$\frac{\theta_r \in \mathsf{ren}(\mathsf{vars}(\Pi), \mathsf{vars}(\eta)) \quad \Delta = \{\psi \wedge \widehat{\theta} : P \mid \ e' : \psi \leftarrow P \in \Pi\theta_r,\ \theta \in \mathsf{mgu}(e\eta, e')\} \neq \emptyset}{\langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \eta, !e\rangle \longrightarrow \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \eta, e : (\!|\Delta|\!)\rangle} Event$$

That is, the rule uses a *renaming* of the plan-library $\Pi\theta_r$ so that fresh variables, not appearing in the intention, are used. Moreover, the actual event $e\eta$ being resolved needs to unify with the triggering head e' of plan rules. Finally, when a unifying plan rule is found, its guard condition is built from the rule's context condition ψ, together with the equality formula $\widehat{\theta}$ induced by the corresponding unifier (stating how variables should be restricted to make $e\eta$ and e' "match.") In that way, the relevance condition is accounted in the guard condition.

For example, consider the event goal $e = getObj(r, y)$ for collecting object y from room r, and suppose that the bindings η performed so far are such that $y/Watch \in \eta$. Then, the actual pending event that needs to be resolved is $e\eta = getObj(r, Watch)$: get the watch object from some room. Suppose next that the (renamed) plan library includes a rule of the form $getObj(Locker, x_2) : \psi(x_2, x_3) \leftarrow P(x_2, x_3, x_4)$, encoding a strategy P to grab things from the locker room when ψ holds. Such plan is *relevant* for the the actual pending event by taking $\theta = \{x_2/Watch, r/Locker\}$. As a result, the set Δ would include a pair of the form $\langle \psi(x_2, x_3) \wedge (x_2 = Watch \wedge r = Locker) : P(x_2, x_3, x_4)\rangle$.

Plan Selection The rule for selecting a program strategy from the set of available ones, namely, rule Sel, needs to be adapted so that *(i)* the current strategy selected executes using fresh "local" variables only (i.e., variables not visible outside the strategy) in order to avoid undesired side-effects with other concurrent programs that may be running in the same intention; and *(ii)* already tried alternatives are ruled out from the set of alternatives relative to their previous successful bindings.

To achieve the first point, the chosen alternative $\langle \psi : P \rangle$ is *renamed*, using substitution θ_r, to use completely new (local) variables. In addition, the last step in the current strategy P_s, namely test $?(\widehat{\theta_r})$, involves re-instantiating the original variables, thus committing to all such "temporal" bindings and therefore visible to other concurrent programs.

To achieve the second requirement, the just selected alternative $\langle \psi : P \rangle$ is not removed from the set of alternatives, but it is further restricted to rule out the bindings θ that were just used to make ψ true. More specifically, guard ψ is further constrained with an extra conjunct $(\neg\widehat{\theta_r})\theta$ which states that the bindings θ that made the guard true may *not* satisfy the guard formula (again).

Finally, because all used variable used in an intention need to be mentioned in the current bindings η of the intention, a *dummy* substitution $(\theta_{free})^{-1}$—the inverse of variable-only substitution θ_{free}—is added to the current bindings to that end.

$$\frac{\begin{array}{lll}\psi : P \in \Delta & \theta_r \in \mathsf{ren}(\mathsf{vars}(\psi\eta) \cup \mathsf{vars}(P\eta), \mathsf{vars}(\eta)) & \mathcal{B} \models (\psi\eta\theta_r)\theta \\ P_s = P\eta\theta_r; ?(\widehat{\theta_r}) & \theta_{free} \in \mathsf{ren}(\mathsf{vars}(P_s), \mathsf{vars}(\eta\theta)) & \Delta' = \{\langle \psi \wedge (\neg\widehat{\theta_r})\theta : P \rangle\}\end{array}}{\langle \mathcal{B}, \mathcal{A}, \eta, e : (\!|\Delta|\!) \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \eta\theta(\theta_{free}^{-1}), P_s \rhd e : (\!|(\Delta \setminus \{\psi : P\}) \cup \Delta'|\!) \rangle} \; Sel$$

Observe that, due to the renaming of variables to new local variables, the binding θ that makes the context ψ true, as well as any substitutions performed during P_s's execution, could only bind such local variables.

To illustrate how rule Sel works, let us return to the above example. There, the rule would yield $P_s = P(x_2', x_3', x_4'); ?(x_2 = x_2' \wedge x_3 = x_3' \wedge x_4 = x_4' \wedge r = r')$, where $\theta_r = \{x_2/x_2', x_3/x_3', x_4/x_4', r/r'\}$. If, say, $\psi(x_2, x_3) = At(x_3)$ and the agent happens to be at home, then $\theta = \{x_3'/\mathit{Home}, x_2' = \mathit{Watch}, r' = \mathit{Locker}\}$, that is, $\mathcal{B} \models [\mathit{At}(x_3) \wedge (x_2 = \mathit{Watch} \wedge r = \mathit{Locker})]\eta\theta_r\theta$. In turn, the new guard for the strategy would be $\psi' = (\psi \wedge x_3 \neq \mathit{Home})$—strategy P may not be used again when the agent is at home. See that the test for applicability in rule Sel as well as any execution of P_s may only produce bindings for the local variables x_2', x_3', and x_4', and r'. Thus, the original variable r in the actual pending event would not be instantiated to value *Locker* until the very last test step in P_s is performed. Finally, we point out that all variables mentioned in P_s, including x_4 and x_4', are guaranteed to be mentioned in the current set of bindings due to "dummy" substitution θ_{free}^{-1}.

Interaction between Sel and $\rhd_{rec}$ If, at any point, the current strategy selected via rule Sel cannot execute further, that is it is blocked, derivation rule $\rhd_{rec}$ may apply, by replacing the current strategy with an alternative applicable one. The following result formalizes some of the properties we have informally claimed regarding the way strategies are chosen and executed for an event. First, the particular bindings θ in rule Sel supporting a chosen strategy are ruled out from the (new) set of relevant alternative options —a strategy cannot be tried twice under the same "reasons." Second, the execution of the chosen strategy does not cause any free variable in the event to be bound until the strategy is fully executed: the strategy chosen by rule Sel may end up failing and, if so, any bindings done so far would not be meaningful and the new selected applicable plan, if any, may end up binding them differently.

Theorem 11. *Suppose $\langle \mathcal{B}, \mathcal{A}, \eta, e : (\!|\Delta|\!) \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \eta', P_s \rhd e : (\!|\Delta'|\!) \rangle$. Then,*

1. *there exists (a unique) option $\psi : P \in \Delta$ such that $\Delta' = (\Delta \setminus \{\psi : P\}) \cup \{\psi' : P\}$ and $\mathcal{B} \models \psi\eta\theta$, for some θ, and for any belief base $\mathcal{B}'$, $\mathcal{B}' \not\models \psi'\eta\theta$; and*
2. *if for some $n \geq 1$, $\langle \mathcal{B}_i, \mathcal{A}_i, \eta_i, P_i \rangle \longrightarrow \langle \mathcal{B}_i', \mathcal{A}_i', \eta_{i+1}, P_{i+1} \rangle$, for each $i \leq n-1$ and such that $P_0 = P_s$ and $P_n \neq nil$, then $\mathsf{vars}(e\eta) = \mathsf{vars}(e\eta_n)$.*

Proof. The proof follows from inspecting derivation rule Sel (page 40), which is the rule that ought to be used to evolve program $e:(\!|\Delta|\!)$. The first claim is due to the fact that ψ' shall include a conjunct ruling out the bindings that made ψ true, that is $\psi' = \psi \wedge \neg\widehat{\theta}$. For the second part, observe that due to rule Sel, $P_0 = P_s = P'_s; ?\phi$ for some program P'_s which will *not* mention any free variable in $e\eta$. Because each P_i is an evolution of P_s and $P_n \neq nil$, then only (part of) program P'_s has executed at P_n and the thesis follows. □

Declarative Goals The two most important modifications for handling declarative goals in a language with variables involve extending intention-level rule G_{adopt} and agent-level rule A^1_{goal} (see Appendix B). With respect to the first one, like 2APL [12], we require the declarative goal to be *fully defined* in order to be adopted, that is, its corresponding conditions ought to be fully instantiated. With respect to goal adoption from the motivational library, the extended rule now needs to find an substitution that would make the triggering condition ψ of a motivational rule true (and use such binding accordingly when incorporating the new intention). The motivational rules in library $\mathcal{M}$ are now such that $\mathsf{vars}(\phi_s) \cup \mathsf{vars}(\phi_f) \cup \mathsf{vars}(!e) \subseteq \mathsf{vars}(\psi)$—all free variables in the goal-program are mentioned in the triggering condition.

6 Implementation Issues

All languages in the CAN family are in themselves *high-level* plan languages, in the spirit of process algebras such as the π-calculus and agent systems such as Ψ or Golog, rather than a programming language per se. So, the three languages discussed above concentrate on the high-level description of important aspects of BDI programming, such as plan selection, event handing, belief updates, lookahead planning, etc. rather than on cumbersome implementation details such as data structures and mechanisms for passing data around—they are agnostic with respect to such issues.

Nonetheless, because formal languages like CANPlan are also meant to capture and provide semantics to actual implemented systems and architectures, it is important to understand how the different features of the language can be effectively realized. Because many of the features in CANPlan are standard in BDI agent programming (e.g., belief updates, plan selection, etc.), we shall briefly explain the implementation of those features that are *unique* to the language, namely, its failure handling mechanism, declarative event-goals, and planning capability. Although we discuss these with respect to the Java-based Jack agent development platform, it should be clear that similar approaches can be taken for other BDI platforms.

Goal Failure Handling As argued, the goal-failure recovery in CANPlan captures the typical recovery mechanism implemented in many BDI systems, such as Jack and PRS, in which alternatives plans are tried, if possible, when a plan happens to fail. By doing so, the language provides a default strategy to capture the *commitment* of an agent to event-goals. The Jack programming platform, for instance, includes such built-in mechanism for the so-called `BDIGoalEvent` type of events. Whereas other events in Jack (e.g., `MessageEvent` events) represent transient information that the agent reacts to and for which failure recovery is not available, `BDIGoalEvent` events are used to model goal-directed behaviour, rather than plan-directed behaviour: an agent commits to the desired outcome, not the method chosen to achieve it [9]. Thus, events in CANPlan model Jack's `BDIGoalEvent` type of events.

We point out that other formal BDI programming languages either do not provide any account for goal failure recovery or it is left to the BDI user to explicitly program it. For

instance, 3APL [30] provides the so-called *failure* practical rules, which run at high priorities and provide a mechanism for replacing an intention-program (or part of it) with another program. Those rules then accommodate "reflective capabilities," even on the intention structure, something that is not accounted for in our language. However, failure recovery needs to be explicitly programmed by designing suitable rules for recognizing failure cases and recovering from them. Programming the kind of hierarchical goal-failure recovery provided by most real BDI systems would indeed be a cumbersome task, if at all possible. The language AgentSpeak takes another approach to failure. The language itself does not include any default built-in failure handling, and it allows dropping of a whole intention as soon as this cannot evolve, thus providing a low-level of commitment. Nonetheless, when a plan for an achievement event $!e$ fails, a distinguished failure event $-!e$ is posted within the system. As with 3APL, the programmer may decide to design specific plan rules to handle such failure events in order to recover from the failure of e. In some implementations of AgentSpeak (e.g., Jason [6]), internal actions are provided to access the intention stack, so that the user could, in principle, program a hierarchical recovery strategy similar to that of CANPlan.

Declarative Goals Despite the fact that most BDI programming platforms rely mostly on events to represent the agent's current (pending) goals, some systems have lately incorporated programming mechanisms that are close to our Goal construct. This was, in fact, what partly motivated the need for formalizing such mechanisms. The Jack platform, for instance, includes statements like **@achieve**(<cond>,e), **@insist**(<cond>,e), and **@maintain**(<cond>,e) to check for a condition after the execution of an event, insist on an event if the condition is not met, or carry on an event provided some condition is not violated, respectively. The Jason [6] system also includes similar constructs, such as **DG**(<goal>) for testing the actual goal achievement after execution and **BDG**(<goal>) for "backtracking" on plan selection upon plan failure to re-try alternative options.

Interestingly, it turns out that our declarative goal construct $\mathsf{Goal}(\phi_s, !e, \phi_f)$ can easily be expressed in Jack as follows:

```
while (¬φs) {
   if (@maintain(¬φs && ¬φf , !e))
      { } \\ plan finished successfully
   else
      { ¬φf; } \\ maintain failed; failure condition false?
}
```

Under this code, if the execution of event e finishes successfully, then the while-loop would execute again only if the goal has not yet been achieved. If, however, either ϕ_s or ϕ_f becomes true *during the execution* of event e, then the **@maintain** statement would fail immediately and the else-part of the conditional would execute to check whether the failure was caused by the goal failure condition. If so, such test fails and so will the whole execution. Otherwise, the success condition must have become true and the while loop exits successfully. Finally, it is not hard to see that if neither condition become true but the execution of the event terminates or fails, then the while–loop will be repeated.

For example, the goal-program $\mathsf{Goal}(\mathit{At}(\mathit{Uni}), !\mathit{travelTo}(\mathit{Uni}), \mathit{Cancelled}(\mathit{Exam}))$ from Example 6 (pp. 20) can be directly translated into the following Jack code:

```
@maintain(!At.check(uni) && !Cancelled.check(exam),
          travelTo_ev.post(uni))
  { } \\ plan finished successfully
```

```
else
  { At.check(uni); } \\ maintain failed; check goal success
```

This Jack agent has two simple beliefsets—At(loc) and Cancelled(x)—recording the current location of the agent and whether a venue has been cancelled. Observe that the exclamation mark ! in the above code refers to Jack logical negation (i.e., the atoms do not hold true).

The Planning Construct In earlier work [20], we presented an implementation that combined BDI reasoning with HTN planning. We used Jack BDI system [9] and JSHOP HTN planner, a Java version of SHOP [47]. In concrete, a special **@plan** construct within Jack, available to the programmer in exactly the same way as construct **@subtask**, was provided to initiate HTN-planning rather than plain BDI execution. Although the integrated framework does not fully realise the operational semantics presented here, it does incorporate some important concepts from it. In particular, it allows the programmer to specify from within a Jack program the points at which JSHOP should be called, in a manner similar to the Plan construct. Consistent with the semantics of Plan, JSHOP uses the same domain representation as Jack does (i.e., the plan library Π and belief base $\mathcal{B}$). In fact, the framework builds at runtime a JSHOP planning problem representation automatically from the Jack domain knowledge.

Some differences in the implementation arise from the nature of the systems chosen for the implementation. Since JSHOP is a total-order HTN planner, it cannot accommodate the concurrent construct $\|$. However, since parallelism has benefits, the integrated framework converts JSHOP's total-order solutions into partial-order solutions so that Jack can exploit possible parallelism at execution time. Some other differences exist between the implementation and the semantics for the sake of simplicity. For example, we excluded the $\mathsf{Goal}(\phi_s, P, \phi_f)$ construct in our system, as this construct does not have a direct matching concept in Jack or JSHOP. Including this goal construct and using SHOP2 [48] to accommodate parallel execution of sub-goals naively are left for future work.

The main difference, however, is that the implementation *does not* re-plan at every step, as indicated by the Plan derivation rule defined in the semantics. This would clearly be unnecessarily inefficient. Instead, JSHOP was modified to return the relevant methods and bindings (rather than simply the actions); the BDI execution engine then follows step-by-step such suggested decomposition. Relevant environmental changes are detected by virtue of a step in the returned plan no longer being applicable within the BDI cycle. At that point, the planner is then called once again to provide an updated plan, and if none is available failure will occur in the BDI system. A disadvantage of this is that environmental changes leading to failure may be detected later in the implemented version than in the semantic rules. This approach also has the benefit that an intention produced by a call to Plan will, in fact, *terminate*—successfully if there is no environmental interference. This is stronger than what Theorem 6 states, in which we needed to account for the strange, but theoretically possible, situation where the Plan module continually returns a new and different plan prior to termination.

7 Related Work

There are a plethora of agent-oriented programming languages that are related in some way or another to the language CANPlan described here. Rather than discussing all (minor) differences with these frameworks (e.g., the form of plan rules in the library, types of events, or different variations of the BDI execution engine), we focus here on the two distinguished features of CANPlan: the integration of automated planning and the use of declarative goals.

7.1 Planning and Agent Systems

The underlying strong similarities between BDI agent systems and HTN planners is not new. In fact, Wilkins and Myers [71] proposed the ACT formalism as a *uniform language* for supporting the interoperability of reactive plan-executors and hierarchical planners. The ACT language supports the type of mapping we have proposed in Section 4.1.1.

There are several BDI-like agent programming languages that come with solid *formal semantics*. These include PRS [33, 74] and dMARS [23], AgentSpeak [53], 3APL [14, 30] and 2APL [12], and GOAL [16], among others. None of these languages, though, provide at this point an account of lookahead planning; behavior relies entirely on some type of (online) context sensitive sub-goal expansion.

There are however a number of implemented platforms which do, in some way or another, mix planning and BDI-style execution. Some of these are *planners*, such as IPEM [2] and Sage [37], that allow for the interleaving of action execution during the planning process. Similarly, A-SHOP [24] is an *agentized* version of the well-known HTN SHOP [47] planner integrated within the IMPACT multi-agent environment [25]. Others are *agent architectures*, such as Retsina [50], SRI's Cypress [72] (based on the mentioned ACT formalism), Propice-Plan [22], and the work on the JADEX [52] agent framework for integrating state-based planning [69]. All these systems are able to do some type of lookahead planning within a typical reactive agent execution. Propice-Plan is perhaps the most similar system to ours, in that it is a typical BDI agent system (PRS-based) that is able to explicitly call a planning module (planner IPP) in order for the agent to anticipate alternative execution paths. Like CANPlan, a unified representation is used by both the planner and the BDI system. On the other hand, it does not exploit the hierarchical nature of BDI plans as it does not appeal to an HTN planner. The work done here differs in, at least, two ways from all the above systems. First, we are particularly concerned with the *formal specification* of a BDI agent with built-in planning capabilities, as well as with the *formal relation* between BDI systems and HTN planners. The above systems focus on *implemented architectures* rather than on the precise semantics of what planning in BDI platforms is. Nonetheless, the formal work that we have presented here was indeed partly motivated by the existence of such systems, in a way analogous to how AgentSpeak was motivated by systems like PRS and dMARS. Second, CANPlan provides a mechanism for local deliberation *on-demand*, as opposed to a fixed integration of planning within the execution engine (e.g., Retsina performs continuous planning coupled with execution and CPEF engages in planning upon goal failure).

Our approach is strongly related to IndiGolog [58], a situation calculus-based high-level programming language with an interleaved account of execution, planning, and sensing. As a matter of fact, our planning construct Plan and its actual semantics was inspired by IndiGolog's "search operator" Σ for local offline deliberation. Nonetheless, IndiGolog takes a more traditional computer science perspective and is not *per se* a BDI programming language; its connections with BDI programming notions and properties is, in general, fairly weak. For instance, there is no notion of goals being pursued besides the overall high-level program being executed, and there is no account of goal retrying upon plan failure as a way of realizing the commitment of the agent. Besides that, IndiGolog's search operator planning module is not related to any automated planning approach. Our account is strongly linked to HTN-planning and has a more practical orientation.

We close by briefly mentioning the recent effort on incorporating classical planning in BDI systems. Meneguzzi and Luck [43] extended AgentSpeak with a mechanism whereby the agent can call an (external) STRIPS classical planner so as to obtain a plan from first-principles that achieves a conjunction of literals. The plan returned, in turn, is then incorpo-

rated into the agent's plan library for future re-use, by suitably defining the corresponding context condition. Though promising, the work is in its very early stages; it does not come with a formal semantics, no proof of correctness is provided, and plan synthesized and learnt are limited to low-level ones (that is, no abstraction is done to non-primitive plans). In some sense, at this point, the approach amounts to an implementation extension of AgentSpeak for performing invocation to an external planner within the plan-body of plan rules. Adding classical planning to BDI systems was also recently studied by de Silva et al. [21]. However, that work was more concerned with generalizing low-level plans returned by the planner to more abstract plans containing abstract steps (i.e., events or compound tasks). By synthesizing new plans at higher level of abstractions, the agent can not only re-use procedural information already available but yield more flexible/robust plans, since higher-level goals/events may be achievable in multiple ways. Nonetheless, both above works are *orthogonal* to the planning approach of CANPlan. Whereas planning from first-principle can be useful in certain situations where no domain "know-how" information is available, the planning account developed in this paper follows the HTN view that in many, if not most, dynamic settings, there is substantial procedural information from the experts that is worth exploiting.

7.2 Goals in Agent-Oriented Programming Languages

Despite the fact that the notion of goals (and that of desires) has been at the core of the BDI model of rational behavior, both at the philosophical and theoretical levels [7, 11, 54, 61], BDI agent-oriented programming languages have historically fallen short on representing and reasoning about them. The reliance on the so-called *events* limits the account of goals to a sophisticated kind of method invocation. Nonetheless, there has recently been a growing effort to account for more sophisticated goals in BDI programming languages.

van Riemsdijk et al. [68] explores the *semantics of declarative goals* in agent programming languages. The work addresses the issue of what it means that a cognitive agent has a certain goal, given the state of the data structures modeling the agent (belief base, goal base, intention base, and rule base). For instance, an agent may pursue two goals that are inconsistent with each other (e.g., p and $\neg p$), but cannot pursue an inconsistent goal itself (e.g., $p \wedge \neg p$). Their approach to goal representation differs from that used in CANPlan in that (active) goals are derived, implicitly, from a *goal base*. Instead, CANPlan does not carry an explicit goal base, as this can be implicitly extracted from the current plans the agent has already committed to, following the usual understanding that goals are desires the agent has committed to realize. We are however interested in exploring the use of an explicit *desire* base that may motivate the agent to potentially adopt new (top-level) goals. The motivation base $\mathcal{M}$ introduced in Section 3 is our starting point for that.

The *dynamics of goals* is also a central issue when it comes to modeling goals and is tightly linked to the notion of commitment. In [67], several motivations and mechanisms are proposed for adopting and dropping declarative goals. The authors formalize what it means for an agent to adopt or drop a goal within an agent transition, by means of different type of adoption/failure rules. Among other differences, the declarative and procedural aspects of goals are not intrinsically related as they are in our CANPlan language. So, for instance, *failure rules* are used to describe the conditions under which the agent should abandon a goal. However, these rules are linked to the declarative aspect of goals only and thus the programmer cannot specify that a *particular strategy* for achieving a goal is bound to fail under some conditions. Similarly, Shapiro et al. [60] develops an account of *goal change* for

situation calculus agents and examined expansion, contraction, and persistence properties for goals, focusing mostly on agents receiving external requests and cancellations.

In [15], three types of goals for agent-oriented programming languages are identified: *perform* goals, *achieve* goals, and *maintain* goals. A goal type is seen as a special agent *attitude* towards goals. Interestingly, the kind of goals we have formally defined in CANPlan can be classified within these three types. While event goals and planning goals are perform-type goals, declarative goals are achieve type of goals. Moreover, some types of motivational rules in $\mathcal{M}$ could be seen as (reactive) maintenance goals.

In [32], *plan patterns* are used in order to indirectly accommodate declarative goals in AgentSpeak. The declarative goals modeled enjoy similar properties to those of our Goal construct (e.g., successful termination upon achievement, re-trying upon failure, etc.). In contrast with the Goal construct, though, the extended AgentSpeak does not model declarative goals with a first-class citizen language construct. Instead, declarative goal statements are "macro expanded" into a more complex set of rules, whose combined effect model the behavior of declarative goals. Although this may appear to be a less involved account of declarative goals than the one proposed in CANPlan, it relies on the so-called "internal actions," special actions that allow for the meta-level manipulation of the intention stack (at the object level). Although the use of internal actions may be convenient for implemented systems like Jason, it is arguable whether their use is desirable for defining the semantic specification of the language—they blur the line between meta-level and object-level concepts.

There has also been work on accommodating declarative goals into 3APL/2APL [12, 14, 66]. In its latest version, 3APL carries a declarative *goal base* and plan-generating rules can be used to select plans on the basis of both belief and goal conditions. Moreover, each active intention is associated with the goal it is meant to achieve, i.e., the "reason" *why* the plan is being pursued. In contrast with CANPlan, a 3APL agent carries declarative information only for the *initial* goal of the intention, no information is carried for any of the active (instrumental) sub-goals. The Goal construct allows for declarative information to be specified at any level of sub-goaling. Also, at this point, 3APL does not make use of the declarative information attached to each intention—intentions are not dropped even when their initial motivating goal is achieved. Still, it is not hard to see how to adapt the deliberation cycle in [13] to account for such information. Although there is no failure condition associated to intentions/goals in 3APL, it is possible to explicitly program the dropping of a goal via a distinguished construct `dropgoal`(ψ).

Perhaps the BDI-style language that takes declarative goals most seriously is de Boer et al. [16]'s GOAL language. Behavior in GOAL arises as a consequence of applying so-called *conditional actions* from a pre-defined library, stating when it is sensible to perform an action given the current beliefs *and the current goals*. Actions include domain actions, as well as belief and goal change operators. One unique feature of GOAL is that it comes with a temporal logic suitable for proving properties of GOAL programs. Also, due to the fact that conditional actions can only have single actions in their plan-bodies and some fairness conditions imposed in the agent execution scheme, many desirable properties of declarative goals are implicitly satisfied (e.g., an agent will not be committed to a plan whose goal is satisfied). Like 3APL, and unlike CANPlan, a GOAL agent maintains a goal base explicitly, which would facilitate goal logical reasoning beyond goal achievability. The way declarative information of goals is attached to its procedural information in CANPlan can make such kind of reasoning more cumbersome. As argued before, the aim in CANPlan is to show how the most practical aspects of declarative information can be incorporated into standard BDI frameworks without major modifications and while maintaining their overall effectiveness. As a result, the declarative goal construct provided sits somewhere in between procedural goals and declar-

ative ones. On the other hand, GOAL does not provide any mechanism for specifying typical (complex) procedural operations of a domain, a feature central to most agent-oriented programming languages, and the framework is currently restricted to propositional languages. Also, the intrinsic relation between goals and subgoals is not captured in the language which, in turn, precludes the specification of generic failure recovery strategies.

8 Conclusion and Future Work

In this paper, we have presented a formal semantics for a powerful BDI-style agent programming language that goes beyond existing accounts in two central aspects of rational agency, namely, goals and means-end analysis. CANPlan includes declarative events (as extensions of the standard events), a goal-failure handling mechanism providing a sophisticated commitment account for goals and plans, and a built-in account of hierarchical lookahead planning.

In particular, the language developed here has the following characteristics:

- A focus on goals and their characteristics including:
  - differentiation between reactive "event-goals" and more persistent goals which include a declarative component;
  - a mechanism for proactively adopting new goals, other than a simple reaction to external events—and similar to the so-called *automatic events* in real BDI platforms;
  - a semantics which ensures agent watchfulness regarding fortuitous goal achievement, thus matching generally accepted definitions of goal-oriented behaviour;
  - a representation and semantics which allows an agent to recognise and respond to situations where a goal has become unachievable (or in some other way undesirable), thus facilitating the realisation of Rao and Georgeff [54]'s condition that goals should be considered possible, as well as commitment strategies that require representation of conditions for dropping goals;
- a failure handling semantics matching most implemented BDI systems, where if a particular approach to achieving a goal fails, an alternative applicable plan is tried;
- a commitment semantics that allows a goal to be dropped not only if it is achieved, or deemed impossible, but also in cases where it is a problematic subgoal of some other motivating goal and there exists some alternative feasible way for achieving the latter;
- a Plan construct equivalent to Hierarchical Task Network (HTN) planning which allows a lookahead on a portion of an agent program, to ensure that choices are made which will result in successful goal achievement if there is no environmental interference;
- a detailed semantics that allows for variables in both formulae and programs.

For legibility and modularity, we developed the full CANPlan language in a incremental manner. We first described the core language CAN$^{\mathcal{A}}$, which is conceptually equivalent to AgentSpeak (hence the superscript) in that it presents the core features of BDI programming languages. Unlike AgentSpeak, though, it captures the failure handling typical of most implemented BDI systems. In Section 3, we extended the core language with declarative event-goals as extensions of the usual events, yielding then the language CAN. We showed that the original failure handling mechanism is compatible with the richer notion of events and that the new language enjoys a commitment strategy that is compatible with, but goes further than, the well-established single-minded strategy [54]. Finally, in Section 4, we further extended CAN to integrate on-demand planning capabilities, yielding the final language CANPlan. The planning mechanism can be used for ensuring *intelligent* plan selection. More importantly, we demonstrated that the account of offline reasoning provided is provably equivalent to the

HTN-style automated planning, thus justifying the integration of established HTN planner systems into existing BDI frameworks.

The BDI language developed here provides a solid foundation for a range of interesting further work and language extensions. One natural extension of CANPlan is the integration of *first-principles* planning, in a way that allows for discovery of new plans while also respecting the "*user-intent*" domain knowledge inherent in the BDI program; see [18, 21, 35]. Another extension we consider is the use of plan monitoring and (intelligent) replanning accounts [58] in order to notice changes that may render a plan useless and to resolve such situation in a manner that is compatible with what has already been committed (and executed). The techniques for plan failure, abortion, and suspension recently developed by Thangarajah et al. [64, 65] are all orthogonal to the issues addressed in CANPlan, and hence, it should be easy to accommodate them into the language. Finally, it would be interesting to further extend the support for reasoning about goals, such as reasoning about conflicts or synergies among current goals within different intentions, as in [10, 63]. For example, one could extend intention-level configurations to include the current agent's goal base $\mathcal{G} = \mathcal{G}(\Gamma)$ and further develop the goal adoption rule G_{adopt} (pp. 19) to avoid adopting conflicting goals or goals already *already implied* by some other already active goals.

We believe the work presented here is a significant step towards obtaining a *formal* BDI agent framework that goes beyond standard reactive execution, and provides firm foundation for exploring additional reasoning mechanisms at both the theoretical and practical levels.

Acknowledgements We acknowledge Michael Winikoff, John Thangarajah, and James Harland for the original work, together with the second author in this paper, on goal-programs reported in [73]. Indeed, this paper has its roots in such work. We particularly thank Michael for detailed discussions on CAN and CAN$^{\mathcal{A}}$. We thank Lavindra de Silva for work on the practical integration of HTN planners and BDI implementations and useful discussions. We are also very grateful to the anonymous reviewers and very many colleagues who have provided us with valuable detailed comments that helped improved the research and paper substantially. We acknowledge Agent Oriented Software and the Australian Research Council for financial support under grants LP0560702 and LP0882234. The first author would also like to acknowledge the National Science and Engineering Research Council of Canada for funding assistance under the PDF program.

References

1. Natasha Alechina, Rafael H. Bordini, Jomi Fred Hübner, Mark Jago, and Brian Logan. Belief revision for AgentSpeak agents. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1288–1290, 2006.
2. Jose Ambros-Ingerson and Sam Steel. Integrating planning, execution and monitoring. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, pages 83–88, St. Paul, MN., 1988.
3. Roxana A. Belecheanu, Steve Munroe, Michael Luck, Terry Payne, Tim Miller, Peter McBurney, and Michal Pechoucek. Commercial applications of agents: Lessons, experiences and challenges. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1549–1555, 2006.
4. Steve S. Benfield, Jim Hendrickson, and Daniel Galanti. Making a strong business case for multiagent technology. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 10–15, 2006.
5. Rafael H. Bordini and Alvaro F. Moreira. Proving BDI properties of agent-oriented programming languages. *Annals of Mathematics and Artificial Intelligence*, 42(1–3): 197–226, 2004.

6. Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley Series in Agent Technology. Wiley, 2007. ISBN 0470029005.
7. Michael E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, 1987.
8. Jan Broersen, Mehdi Dastani, Joris Hulstijn, Zisheng Huang, and Leendert der van Torre. The BOID architecture: Conflicts between beliefs, obligations, intentions and desires. In *Proc. of the Annual Conference on Autonomous Agents (AGENTS)*, pages 9–16, Montreal, Canada, 2001. ACM Press.
9. Paolo Busetta, Ralph Rönnquist, Andew Hodgson, and Andrew Lucas. JACK intelligent agents: Components for intelligent agents in Java. *AgentLink Newsletter*, 2:2–5, January 1999. Agent Oriented Software.
10. Bradley J. Clement, Edmund H. Durfee, and Anthony C. Barrett. Abstract reasoning for planning and coordination. *Journal of Artificial Intelligence Research*, 28:453–515, 2007.
11. Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial Intelligence Journal*, 42:213–261, 1990.
12. Mehdi Dastani. 2APL: A practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, June 2008.
13. Mehdi Dastani, Frank S. de Boer, Frank Dignum, and John-Jules Meyer. Programming agent deliberation: An approach illustrated using the 3APL language. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 97–104, 2003.
14. Mehdi Dastani, Birna van Riemsdijk, and John-Jules Meyer. Programming multi-agent systems in 3APL. In *Multi-Agent Programming*, chapter 2, pages 39–67. Springer, 2005.
15. Mehdi Dastani, Birna van Riemsdijk, and John-Jules Meyer. Goal types in agent programming. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1285–1287, 2006.
16. Frank S. de Boer, Koen V. Hindriks, Wiebe van der Hoek, and John-Jules Meyer. A verification framework for agent programming with declarative goals. *Journal of Applied Logic*, 5(2):277–302, 2007.
17. Giuseppe De Giacomo, Yves Lespérance, Hector J. Levesque, and Sebastian Sardina. IndiGolog: A high-level programming language for embedded reasoning agents. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, chapter 2, pages 31–72. Springer, 2009.
18. Lavindra P. de Silva. *Planning in BDI Agent Systems*. PhD thesis, School of Computer Science and Information Technology, RMIT University, Melbourne, Australia, 2009. Submitted.
19. Lavindra P. de Silva and Lin Padgham. A comparison of BDI based real-time reasoning and HTN based planning. In *Proc. of the Australian Joint Conference on AI (AI)*, pages 1167–1173, 2004.
20. Lavindra P. de Silva and Lin Padgham. Planning on demand in BDI systems. In *Proc. of the International Conference on Automated Planning and Scheduling (ICAPS)*, June 2005. (poster).
21. Lavindra P. de Silva, Sebastian Sardina, and Lin Padgham. First principles planning in BDI systems. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, volume 2, pages 1001–1008, 2009.
22. Olivier Despouys and Francois Felix Ingrand. Propice-Plan: Toward a unified framework for planning and execution. In *Proc. of the European Conference on Planning (ECP)*,

pages 278–293, 1999.
23. Mark d'Inverno, Michael Luck, Michael P. Georgeff, , David Kinny, and Michael Wooldridge. The dMARS architechure: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems*, 9(1–2):5–53, 2004.
24. Jürgen Dix, Héctor Muñoz-Avila, Dana S. Nau, and Lingling Zhang. IMPACTing SHOP: Putting an AI planner into a multi-agent environment. *Annals of Mathematics and Artificial Intelligence*, 37(4):381–407, 2003.
25. Thomas Eiter, Thomas Subrahmanian, and George Pick. Heterogeneous active agents I: Semantics. *Artificial Intelligence Journal*, 108(1-2):179–255, 1999.
26. Kutluhan Erol, James A. Hendler, and Dana S. Nau. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence*, 18(1):69–93, 1996.
27. Alfonso Gerevini, Blai Bonet, and Bob Givan, editors. *Booklet of 4th International Planning Competition*, Lake District, UK, 2006. URL http://www.ldc.usb.ve/~bonet/ipc5/.
28. Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.
29. Matthew Hennessy. *The Semantics of Programming Languages*. Wiley, Chichester, England, 1990.
30. Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
31. Marcus J. Huber. JAM: A BDI-theoretic mobile agent architecture. In *Proc. of the Annual Conference on Autonomous Agents (AGENTS)*, pages 236–243, New York, NY, USA, 1999. ACM Press.
32. Jomi Fred Hübner, Rafael H. Bordini, and Michael Wooldridge. Programming declarative goals using plan patterns. In *Proc. of the International Workshop on Declarative Agent Languages and Technologies (DALT)*, volume 4327 of *LNCS*, pages 123–140. Springer, 2006.
33. Francois Felix Ingrand, Michael P. Georgeff, and Anand S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert: Intelligent Systems and Their Applications*, 7(6):34–44, 1992.
34. Nicholas R. Jennings. An agent-based approch for building complex software systems. *Communications of the ACM*, 44(4):35–41, 2001.
35. Subbarao Kambhampati, Amol Dattatraya Mali, and Biplav Srivastava. Hybrid planning for partially hierarchical domains. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, pages 882–888, 1998.
36. David Kinny and Michael P. Georgeff. Commitment and effectiveness of situated agents. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 82–88, 1991.
37. Craig A. Knoblock. Planning, executing, sensing and replaning for information gathering. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1686–1693, 1995.
38. Donald E. Knuth. *The Art of Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, 1969.
39. Martin J. Kollingbaum and Timothy J. Norman. NoA - A normative agent architecture. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1465–1466, 2003.
40. Fangzhen Lin and Hector J. Levesque. What robots can do: Robot programs and effective achievability. *Artificial Intelligence Journal*, 101:201–226, 1998.

41. John W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 2nd edition, 1987.
42. Derek Long and Maria Fox. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.
43. Felipe Meneguzzi and Michael Luck. Leveraging new plans in AgentSpeak(PL). In *Proc. of the International Workshop on Declarative Agent Languages and Technologies (DALT)*, volume 5397 of *LNCS*, pages 111–127. Springer, 2009.
44. Alvaro F. Moreira, Renata Vieira, and Rafael H. Bordini. Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In *Proc. of the International Workshop on Declarative Agent Languages and Technologies (DALT)*, volume 2990 of *LNCS*, pages 1270–1285. Springer, 2004.
45. David Morley and Karen L. Myers. The SPARK agent framework. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 712–719, 2004.
46. Dana S. Nau. Current trends in automated planning. *AI Magazine*, 28(4):43–58, 2007.
47. Dana S. Nau, Yue Cao, Amnon Lotem, and Héctor Muñoz-Avila. SHOP: Simple hierarchical ordered planner. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 968–973, 1999.
48. Dana S. Nau, Héctor Muñoz-Avila, Yue Cao, Amnon Lotem, and Steven Mitchell. Total-order planning with partially ordered subtasks. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 425–430, 2001.
49. Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, Dan Wu, Fusun Yaman, Héctor Muñoz-Avila, and William Murdock. Applications of SHOP and SHOP2. *IEEE Intelligent Systems*, 20(2):34–41, 2005. Earlier version as Tech. Rep. CS-TR-4604, UMIACS-TR-2004-46.
50. Massimo Paolucci, Dirk Kalp, Anandeep Pannu, Onn Shehory, and Katia Sycara. A planning component for RETSINA agents. In *Proc. of the International Workshop on Agent Theories, Architectures, and Languages (ATAL)*, pages 147–161, UK, 1999. Springer. ISBN 3-540-67200-1.
51. Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, Denmark, 1981.
52. Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. JADEX: Implementing a BDI-infrastructure for JADE agents. *EXP - in search of innovation (Special Issue on JADE)*, 3(3):76–85, 9 2003.
53. Anand S. Rao. Agentspeak(L): BDI agents speak out in a logical computable language. In W. Vander Velde and J. W. Perram, editors, *Proc. of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World. (Agents Breaking Away)*, volume 1038 of *LNCS*, pages 42–55. Springer, 1996.
54. Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In *Proc. of Principles of Knowledge Representation and Reasoning (KR)*, pages 473–484, 1991.
55. Anand S. Rao and Michael P. Georgeff. An abstract architecture for rational agents. In *Proc. of Principles of Knowledge Representation and Reasoning (KR)*, pages 438–449, San Mateo, CA, 1992.
56. Sebastian Sardina and Lin Padgham. Goals in the context of BDI plan failure and planning. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 16–23, Hawaii, USA, 2007.
57. Sebastian Sardina and Steven Shapiro. Rational action in agent programs with prioritized goals. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 417–424, 2003.

58. Sebastian Sardina, Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. On the semantics of deliberation in IndiGolog – From theory to implementation. *Annals of Mathematics and Artificial Intelligence*, 41(2–4):259–299, August 2004.
59. Sebastian Sardina, Lavindra P. de Silva, and Lin Padgham. Hierarchical planning in BDI agent programming languages: A formal approach. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1001–1008, Hakodate, Japan, 2006.
60. Steven Shapiro, Yves Lespérance, and Hector J. Levesque. Goal change. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 582–588, 2005.
61. Yoav Shoham. An overview of agent-oriented programming. In J. M. Bradshaw, editor, *Software Agents*, pages 271–290. The MIT Press, 1997.
62. John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting & exploiting positive goal interaction in intelligent agents. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 401–408, 2003.
63. John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting & avoiding interference between goals in intelligent agents. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 721–726, 2003.
64. John Thangarajah, David Morley, Neil Yorke-Smith, and James Harland. Aborting tasks and plans in BDI agents. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 8–15, 2007.
65. John Thangarajah, James Harland, David Moreley, and Neil Yorke-Smith. Suspending and resuming tasks in BDI agents. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 405–412, 2008.
66. Birna van Riemsdijk, Wiebe van der Hoek, and John-Jules Meyer. Agent programming in Dribble: From beliefs to goals with plans. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 393–400, 2003.
67. Birna van Riemsdijk, Mehdi Dastani, Frank Dignum, and John-Jules Meyer. Dynamics of declarative goals in agent programming. In *Proc. of the International Workshop on Declarative Agent Languages and Technologies (DALT)*, volume 3476 of *LNCS*, pages 1–18. Springer, 2005.
68. Birna van Riemsdijk, Mehdi Dastani, and John-Jules Meyer. Semantics of declarative goals in agent programming. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 133–140, 2005.
69. Andrzej Walczak, Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Augmenting BDI agents with deliberative planning techniques. In *Proc. of the Programming Multiagent Systems Languages, Frameworks, Techniques and Tools workshop (PROMAS)*, pages 113–127, 2006.
70. Daniel S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.
71. David E. Wilkins and Karen L. Myers. A common knowledge representation for plan generation and reactive execution. *Journal of Logic and Computation*, 5(6):731–761, 1995.
72. David E. Wilkins, Karen L. Myers, John D. Lowrance, and Leonard P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI*, 7(1):197–227, 1995.
73. Michael Winikoff, Lin Padgham, James Harland, and John Thangarajah. Declarative & procedural goals in intelligent agent systems. In *Proc. of Principles of Knowledge Representation and Reasoning (KR)*, pages 470–481, 2002.
74. Wayne R. Wobcke. An operational semantics for a PRS-like agent architecture. In M. Stumptner, M., Corbett, D. & Brooks, editor, *AI 2001: Advances in Artificial Intelligence.* Springer-Verlag, Berlin, 2001.

A Proof of Theorem 8

We shall show here the version of the theorem with variables: for any libraries Π and Λ, and belief bases $\mathcal{B}$ and $\mathcal{B}'$ in a bounded agent, and for any action sequences $\mathcal{A}$ and σ, substitutions η and η', and event e:

$$\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \eta, \mathsf{Plan}(!e)\rangle \xrightarrow{\mathsf{bdi}^*} \langle \Pi, \Lambda, \mathcal{B}', \mathcal{A}\cdot\sigma, \eta', \mathit{nil}\rangle \ \text{ iff } \ \sigma \in \mathit{sol}(e\eta, \mathcal{B}, \langle \Pi, \Lambda\rangle).$$

The proof involves an, almost one-to-one, translation between BDI entities (e.g., actions, plan rules, plan bodies) and HTN entities (e.g., operators, methods, tasks networks) based on the relationship among them as discussed in Section 4.1.1, and a proof showing that the BDI and HTN decomposition mechanisms are equivalent.

When it comes to the translation, the only two non-straightforward cases are the ones for *(i)* BDI belief conditions into HTN constraints; and *(ii)* BDI plan-body programs into HTN task networks. For case *(i)*, when l is a literal, ϕ a formula, and n and integer, we take $(l,n)^* = (l,n)$ for the base case; and inductively $(\neg\phi, n)^* = \neg(\phi, n)^*$ and $(\phi_1 \wedge \phi_2, n)^* = (\phi_1, n)^* \wedge (\phi_2, n)^*$. The cases for $(n, \phi)^*$ and $(n_1, \phi, n_2)^*$ are defined in an analogous way.

The most complex part of the translation involves obtaining HTN network tasks from BDI plan-bodies programs. With that at hand, it is trivial to build the set of methods from a plan library. To that end, when P is a plan-body program in the CANPlan user-language, we define $\mathcal{T}(P,n)$ to be P's corresponding task network with task labels starting from n, and is defined inductively as follows:

Base Case: if $P = \mathit{act}$, then $\mathcal{T}(P,n) = [\{n : \mathit{act}\}, \mathit{true}]$; if $P = \mathit{nil}$, then $\mathcal{T}(P,n) = [\emptyset, \mathit{true}]$; if $P = !e$, then $\mathcal{T}(P,n) = [\{n : e\}, \mathit{true}]$; and if $P = ?\phi$, then $\mathcal{T}(P,n) = [\{n : \mathit{noOp}\}, (\phi, n)^*]$. Primitive task noOp is the standard dummy action with no effects.

Inductive Case: Suppose that $\mathcal{T}(P_1,n) = [d_1, \phi_1]$ and $\mathcal{T}(P_2, n+|d_1|+1) = [d_2, \phi_2]$. Then, if $P = P_1; P_2$, we define $\mathcal{T}(P,n) = [d_1 \cup d_2, \phi_1 \wedge \phi_2 \wedge (n+|d_1| \prec n+|d_1|+1)]$; and if $P = P_1 \| P_2$, we define $\mathcal{T}(P,n) = [d_1 \cup d_2, \phi_1 \wedge \phi_2]$. (Note the difference between sequence and concurrency; the former imposes an extra ordering constrain in the network.)

Finally, the corresponding set of HTN methods for a BDI plan library Π is defined as $\mathcal{T}(\Pi) = \bigcup_{e:\psi \leftarrow P \in \Pi} \{\mathcal{T}_{ev}(e : \psi \leftarrow P)\}$, where $\mathcal{T}_{ev}$ is as follows ($\mathcal{T}(P,1) = [s_P, \phi_P]$):

$$\mathcal{T}_{ev}(e : \psi \leftarrow P) = (e,\ [\{0 : \mathit{noOp}\} \cup s_P,\ \phi_P \wedge (\psi, 0)^* \wedge \bigwedge_{(n:t)\in s_P} 0 \prec n]).$$

With the formal relationship between BDI and HTN entities established, one can then demonstrate—by induction on the structure of plan bodies—that a *successful execution* resulting from the operational rules of CANPlan corresponds directly to a *complete task decomposition* in HTN systems. To that end, one shows that $\langle \mathcal{B}, \mathcal{A}, \eta, P\rangle \xrightarrow{\mathsf{plan}_k} \langle \mathcal{B}', \mathcal{A}\cdot\mathcal{A}', \eta', \mathit{nil}\rangle$ if and only if there exists a sequence of task networks $d_0 \cdot \ldots \cdot d_n$, with $d_0 = \mathcal{T}(P, \ell)$, for some $\ell \geq 0$, such that $d_i \in \mathit{red}(d_{i-1}, \mathcal{B}, \langle \mathcal{T}(\Pi), \Lambda\rangle)$, for each $i \in \{1, \ldots, n\}$, and $\mathcal{A}' \in \mathit{comp}(d_n, \mathcal{B}, \langle \mathcal{T}(\Pi), \Lambda\rangle)$. (Here, $\mathit{comp}(d, \mathcal{B}, \mathcal{D})$ is the set of all plan *completions* of a network d containing only primitive tasks, and ignoring all dummy noOp operators (i.e., plans for which the constraint formula ϕ in d is satisfied), and $\mathit{red}(d, \mathcal{B}, \mathcal{D})$ is the set of all *reductions* of d in $\mathcal{B}$ by methods in $\mathcal{D}$. See [26].)

The proof is done first on induction on k. So, if $k = 0$, then $\mathcal{A}' = \epsilon$ and $P = \mathit{nil}$. We then take $d_n = d_0 = [\emptyset, \mathit{true}]$ and $\epsilon \in \mathit{comp}(d_n, \mathcal{B}, \langle \mathcal{T}(\Pi), \Lambda\rangle)$ holds trivially. Next, suppose the claim holds for all numbers less than some $k \geq 1$ and that $\langle \mathcal{B}, \mathcal{A}, \eta, P\rangle \xrightarrow{\mathsf{plan}_{k+1}}$

$\langle \mathcal{B}', \mathcal{A}\cdot\mathcal{A}', \eta', \mathit{nil}\rangle$. We then perform induction on the the structure of program P. For the base case, suppose that $P = act$ and thus $d_0 = [\{1 : act\}, \mathit{true}]$. Then $\mathcal{A}' = act$, $P' = \mathit{nil}$, and $\mathcal{B} \models prec(act)$. Then, we take $d_n = d_0$ and it follows that $act \in \mathit{comp}(d_n, \mathcal{B}, \langle T(\Pi), \Lambda\rangle)$. The case for test is similar. Consider now the case of posting of events, that is, $P = !e$ and thus $d_0 = [\{0 : e\}, \mathit{true}]$. Clearly, it has to be the case that *(i)* $\langle \mathcal{B}, \mathcal{A}, \eta, !e\rangle \xrightarrow{\mathsf{plan}_2} \langle \mathcal{B}, \mathcal{A}, \eta', P \rhd (\!|\Delta|\!)\rangle$; and *(ii)* $\langle \mathcal{B}, \mathcal{A}, \eta', P \rhd (\!|\Delta|\!)\rangle \xrightarrow{\mathsf{plan}_{k-1}} \langle \mathcal{B}', \mathcal{A}\cdot\mathcal{A}', \eta'', \mathit{nil}\rangle$. Since the backup program $(\!|\Delta|\!)$ is irrelevant in any plan-type derivation, $\langle \mathcal{B}, \mathcal{A}, \eta', P\rangle \xrightarrow{\mathsf{plan}_{k-2}} \langle \mathcal{B}', \mathcal{A}\cdot\mathcal{A}', \eta'', \mathit{nil}\rangle$. By the hypothesis induction on k and point *(ii)*, there exists a sequence of task networks $d_1 \cdot \ldots \cdot d_n$, with $d_1 = \mathcal{T}(P, 1) = [s, \phi]$, such that $d_i \in \mathit{red}(d_{i-1}, \mathcal{B}, \langle T(\Pi), \Lambda\rangle)$, for each $i \in \{2, \ldots, n\}$, and $\mathcal{A}' \in \mathit{comp}(d_n, \mathcal{B}, \langle T(\Pi), \Lambda\rangle)$. Now, by point *(i)*, there must exist a plan rule $e : \psi \leftarrow P \in \Pi$ whose context conditions ψ hold in $\mathcal{B}$, that is, $\mathcal{B} \models \psi$. This implies that there is a method in $\mathcal{T}(\Pi)$ of the form $M_e = (e,\ [\{0 : noOp\} \cup s,\ \phi \wedge (\psi, 0)^* \wedge \bigwedge_{(n:t)\in s} 0 \prec n])$.

Let us now take the modified sequence $d'_1 \cdot \ldots \cdot d'_n$, where $d'_i = [s_i \cup \{0 : noOp\}, \phi_i \wedge (\psi, 0)^* \wedge \bigwedge_{(n:t)\in s} 0 \prec n]$, for all $i \in \{1, \ldots, n\}$. Since $\mathcal{B} \models \psi$ and $noOp$ is the empty operator, it is not hard to see that $d'_i \in \mathit{red}(d'_{i-1}, \mathcal{B}, \langle T(\Pi), \Lambda\rangle)$, for each $i \in \{2, \ldots, n\}$, and $\mathcal{A}' \in \mathit{comp}(d'_n, \mathcal{B}, \langle T(\Pi), \Lambda\rangle)$. (Recall that, without loss of generality, $\mathit{comp}(d, \mathcal{B}, \mathcal{D})$ ignores $noOp$ primitive tasks.) Furthermore, due to method M_2, $d_0 \in \mathit{red}(d'_1, \mathcal{B}, \langle T(\Pi), \Lambda\rangle)$ and sequence $d'_0 = d_0 \cdot d'_1 \cdot \ldots \cdot d'_n$ is such that $d'_i \in \mathit{red}(d'_{i-1}, \mathcal{B}, \langle T(\Pi), \Lambda\rangle)$, for each $i \in \{1, \ldots, n\}$, and $\mathcal{A}' \in \mathit{comp}(d'_n, \mathcal{B}, \langle T(\Pi), \Lambda\rangle)$.

B Complete Operational Semantics for CANPlan

Defined between agent configurations of the form $C = \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \Gamma\rangle$.

$$\frac{C \xRightarrow{\mathsf{int}} C_1 \quad C_1 \xRightarrow{\mathsf{event}} C_2 \quad \langle C, C_2\rangle \xRightarrow{\mathsf{goal}^*} \langle C, C'\rangle \quad \langle C, C'\rangle \not\xRightarrow{\mathsf{goal}}}{C \xRightarrow{\mathsf{CAN}^{\mathcal{A}}} C'}\ A_{\mathsf{CANPlan}}$$

$$\frac{\langle id, P, \eta\rangle \in \Gamma \quad \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \eta, P\rangle \xrightarrow{\mathsf{bdi}} \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}', \mathcal{A}', \eta', P'\rangle}{\langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \Gamma\rangle \xRightarrow{\mathsf{int}} \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}', \mathcal{A}', (\Gamma \setminus \{\langle id, P, \eta\rangle\}) \cup \{\langle id, P', \eta'\rangle\}\rangle}\ A_{\mathsf{int}}$$

$$\frac{\mathcal{B}' = (\mathcal{B} \setminus \{b \mid -b \in \mathcal{E}(C)\}) \cup \{b \mid +b \in \mathcal{E}(C)\} \quad \gamma^! = \{!e \mid !e \in \mathcal{E}(C)\}}{C = \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \Gamma\rangle \xRightarrow{\mathsf{event}} \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}', \mathcal{A}, \Gamma \Cup \gamma^!\rangle}\ A_{\mathsf{ev}}$$

$$\frac{\langle id, P, \eta\rangle \in \Gamma \quad \mathcal{DG}(P) = \emptyset \quad \langle \mathcal{B}, \mathcal{A}, \eta, P\rangle \not\xrightarrow{\mathsf{bdi}}}{\langle C, \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \Gamma\rangle\rangle \xRightarrow{\mathsf{goal}} \langle C, \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \Gamma \setminus \{\langle id, P, \eta\rangle\}\rangle\rangle}\ A^1_{\mathsf{goal}}$$

$$\frac{C_{init}[\mathcal{B}] \not\models b \quad \mathcal{B} \models b \quad e = +b \quad e \notin \mathcal{EG}(\Gamma) \quad \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, !e\rangle \longrightarrow}{\langle C_{init}, \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \Gamma\rangle\rangle \xRightarrow{\mathsf{goal}} \langle C_{init}, \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \Gamma \Cup \{!e\}\rangle\rangle}\ A^2_{\mathsf{goal}}$$

$$\frac{C_{init}[\mathcal{B}] \models b \quad \mathcal{B} \not\models b \quad e = -b \quad e \notin \mathcal{EG}(\Gamma) \quad \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, !e\rangle \longrightarrow}{\langle C_{init}, \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \Gamma\rangle\rangle \xRightarrow{\mathsf{goal}} \langle C_{init}, \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \Gamma \Cup \{!e\}\rangle\rangle}\ A^3_{\mathsf{goal}}$$

$$\frac{\psi \rightsquigarrow P \in \mathcal{M} \quad C[\mathcal{B}] \not\models \psi\theta \quad \mathcal{B} \models \psi\theta \quad \langle \mathcal{B}, \mathcal{A}, \theta, P\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \theta', P'\rangle \quad \nexists\langle id, P'\rangle \in \Gamma}{\langle C, \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \Gamma\rangle\rangle \xRightarrow{\mathsf{goal}} \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \Gamma \Cup \{P'\theta\}\rangle}\ A^4_{\mathsf{goal}}$$

$$\frac{\langle id, P, \eta\rangle \in \Gamma \quad \langle \mathcal{B}, \mathcal{A}, \eta, P\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \eta', P'\rangle \quad |\mathcal{DG}^{end}(\mathcal{B}, P\eta)| < |\mathcal{DG}^{end}(\mathcal{B}, P'\eta')|}{\langle C, \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \Gamma\rangle\rangle \xRightarrow{\mathsf{goal}} \langle C, \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, (\Gamma \setminus \{\langle id, P, \eta\rangle\}) \cup \{\langle id, P', \eta'\rangle\}\rangle\rangle}\ A^5_{\mathsf{goal}}$$

B.1 Intention-Level Semantics

Defined between intention-level configurations of the form $C = \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, P \rangle$. We use two labelled intention-level transitions $\xrightarrow{\mathsf{bdi}}$ and $\xrightarrow{\mathsf{plan}}$; both are assumed when none is specified.

$$\frac{\theta_r \in \mathsf{ren}(\mathsf{vars}(\Pi), \mathsf{vars}(\eta)) \quad \Delta = \{\psi \wedge \widehat{\theta} : P \mid \ e' : \psi \leftarrow P \in \Pi\theta_r,\ \theta \in \mathsf{mgu}(e\eta, e')\} \neq \emptyset}{\langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \eta, !e \rangle \longrightarrow \langle \Pi, \Lambda, \mathcal{M}, \mathcal{B}, \mathcal{A}, \eta, e : (\!|\Delta|\!) \rangle} \mathit{Event}$$

$$\frac{\begin{array}{ll} \psi : P \in \Delta & \theta_r \in \mathsf{ren}(\mathsf{vars}(\psi\eta) \cup \mathsf{vars}(P\eta), \mathsf{vars}(\eta)) \quad \mathcal{B} \models (\psi\eta\theta_r)\theta \\ P_s = P\eta\theta_r; ?(\widehat{\theta_r}) & \theta_{free} \in \mathsf{ren}(\mathsf{vars}(P_s), \mathsf{vars}(\eta\theta)) \qquad \Delta' = \{\langle \psi \wedge (\neg\widehat{\theta_r})\theta : P \rangle\} \end{array}}{\langle \mathcal{B}, \mathcal{A}, \eta, e : (\!|\Delta|\!) \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \eta\theta(\theta_{free}^{-1}), P_s \rhd e : (\!|(\Delta \setminus \{\psi : P\}) \cup \Delta'|\!) \rangle} \mathit{Sel}$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, \eta, P_1 \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \eta', P' \rangle}{\langle \mathcal{B}, \mathcal{A}, \eta, P_1 \rhd P_2 \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \eta', P' \rhd P_2 \rangle} \rhd_{step} \qquad \frac{}{\langle \mathcal{B}, \mathcal{A}, \eta, \mathit{nil} \rhd P' \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \eta, \mathit{nil} \rangle} \rhd_{end}$$

$$\frac{P_1 \neq \mathit{nil} \quad \langle \mathcal{B}, \mathcal{A}, \eta, P_1 \rangle \not\xrightarrow{\mathsf{bdi}} \quad \langle \mathcal{B}, \mathcal{A}, \eta, P_2 \rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', \eta', P_2' \rangle}{\langle \mathcal{B}, \mathcal{A}, \eta, P_1 \rhd P_2 \rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', \eta', P_2' \rangle} \rhd_f^{\mathsf{bdi}}$$

$$\frac{\mathsf{vars}(b\eta) = \emptyset}{\langle \mathcal{B}, \mathcal{A}, \eta, +b \rangle \longrightarrow \langle \mathcal{B} \cup \{b\eta\}, \mathcal{A}, \eta, \mathit{nil} \rangle} +b \qquad \frac{\mathsf{vars}(b\eta) = \emptyset}{\langle \mathcal{B}, \mathcal{A}, \eta, -b \rangle \longrightarrow \langle \mathcal{B} \setminus \{b\eta\}, \mathcal{A}, \eta, \mathit{nil} \rangle} -b$$

$$\frac{\mathsf{vars}(a\eta) = \emptyset \quad a' : \psi \leftarrow \Phi^-; \Phi^+ \in \Lambda \quad a'\theta = a\eta \quad \mathcal{B} \models \psi\theta}{\langle \Lambda, \mathcal{B}, \mathcal{A}, \eta, a \rangle \longrightarrow \langle \Lambda, (\mathcal{B} \setminus \Phi^-\theta) \cup \Phi^+\theta, \mathcal{A} \cdot a\eta, \eta, \mathit{nil} \rangle} \mathit{do} \qquad \frac{\mathcal{B} \models (\phi\eta)\theta}{\langle \mathcal{B}, \mathcal{A}, \eta, ?\phi \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \eta\theta, \mathit{nil} \rangle} ?$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, \eta, P_1 \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \eta', P_1' \rangle}{\langle \mathcal{B}, \mathcal{A}, \eta, P_1; P_2 \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \eta', P_1'; P_2 \rangle} \mathit{Seq}_1 \qquad \frac{\langle \mathcal{B}, \mathcal{A}, \eta, P \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \eta', P' \rangle}{\langle \mathcal{B}, \mathcal{A}, \eta, \mathit{nil}; P \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \eta', P' \rangle} \mathit{Seq}_2$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, \eta, P_1 \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \eta', P' \rangle}{\langle \mathcal{B}, \mathcal{A}, \eta, P_1 \parallel P_2 \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \eta', P' \parallel P_2 \rangle} \parallel_1 \qquad \frac{\langle \mathcal{B}, \mathcal{A}, \eta, P_2 \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \eta', P' \rangle}{\langle \mathcal{B}, \mathcal{A}, \eta, P_1 \parallel P_2 \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \eta', P_1 \parallel P' \rangle} \parallel_2$$

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, \eta, \mathit{nil} \parallel P_2 \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \eta', P_2 \rangle} \parallel_{t_1} \qquad \frac{}{\langle \mathcal{B}, \mathcal{A}, \eta, P_1 \parallel \mathit{nil} \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \eta', P_1 \rangle} \parallel_{t_2}$$

$$\frac{\mathsf{vars}(\phi_s) \cup \mathsf{vars}(\phi_f) = \emptyset \quad \mathcal{B} \not\models (\phi_s \vee \phi_f)\eta \quad \langle \mathcal{B}, \mathcal{A}, \eta, !e \rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', \eta', P \rangle}{\langle \mathcal{B}, \mathcal{A}, \eta, \mathsf{Goal}(\phi_s, !e, \phi_f) \rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', \eta', \mathsf{Goal}(\phi_s, P \rhd P, \phi_f) \rangle} \mathsf{G}_{adopt}^{\mathsf{bdi}}$$

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, \eta, \mathsf{Goal}(\phi_s, !e, \phi_f) \rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}, \mathcal{A}, \eta, (!e; ?\phi_s) \rangle} \mathsf{G}_{adopt}^{\mathsf{plan}}$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, \eta, P_1 \rangle \not\longrightarrow \quad \langle \mathcal{B}, \mathcal{A}, \eta, P_2 \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \eta', P_2' \rangle}{\langle \mathcal{B}, \mathcal{A}, \eta, \mathsf{Goal}(\phi_s, P_1 \rhd P_2, \phi_f) \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \eta', \mathsf{Goal}(\phi_s, P_2' \rhd P_2, \phi_f) \rangle} \mathsf{G}_{restart}$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, \eta, P_1 \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \eta', P' \rangle \quad \mathcal{B} \not\models \phi_s\eta \quad \mathcal{B} \not\models \phi_f\eta}{\langle \mathcal{B}, \mathcal{A}, \eta, \mathsf{Goal}(\phi_s, P_1 \rhd P_2, \phi_f) \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \eta', \mathsf{Goal}(\phi_s, P' \rhd P_2, \phi_f) \rangle} \mathsf{G}_{step}$$

$$\frac{\mathcal{B} \models \phi_s\eta}{\langle \mathcal{B}, \mathcal{A}, \eta, \mathsf{Goal}(\phi_s, P, \phi_f) \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \eta, \mathit{nil} \rangle} \mathsf{G}_{succ} \qquad \frac{\mathcal{B} \models \phi_f\eta}{\langle \mathcal{B}, \mathcal{A}, \eta, \mathsf{Goal}(\phi_s, P, \phi_f) \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \eta, ?\mathit{false} \rangle} \mathsf{G}_{fail}$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, \eta, P \rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}', \mathcal{A}', \eta', P' \rangle \quad \langle \mathcal{B}', \mathcal{A}', \eta', P' \rangle \xrightarrow{\mathsf{plan}*} \langle \mathcal{B}'', \mathcal{A}'', \eta'', \mathit{nil} \rangle}{\langle \mathcal{B}, \mathcal{A}, \eta, \mathsf{Plan}(P) \rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', \eta', \mathsf{Plan}(P') \rangle} \mathsf{P}$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, \eta, P \rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}', \mathcal{A}', \eta', P' \rangle}{\langle \mathcal{B}, \mathcal{A}, \eta, \mathsf{Plan}(P) \rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}', \mathcal{A}', \eta', \mathsf{Plan}(P') \rangle} \mathsf{P}_{\mathsf{P}} \qquad \frac{}{\langle \mathcal{B}, \mathcal{A}, \eta, \mathsf{Plan}(\mathit{nil}) \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \eta, \mathit{nil} \rangle} \mathsf{P}_{end}$$