Golog Speaks the BDI Language

Sebastian Sardina[1] and Yves Lespérance[2]

[1] RMIT University, Melbourne, Australia
[2] York University, Toronto, Canada

Abstract. In this paper, we relate two of the most well developed approaches to agent-oriented programming, namely, BDI (Belief-Desire-Intention) style programming and "Golog-like" high-level programming. In particular, we show how "Golog-like" programming languages can be used to develop BDI-style agent systems. The contribution of this paper is twofold. First, it demonstrates how *practical* agent systems can be developed using high-level languages like Golog or IndiGolog. Second, it provides BDI languages a clear classical-logic-based semantics and a powerful logical foundation for incorporating new reasoning capabilities not present in typical BDI systems.

1 Introduction

BDI (Belief-Desire-Intention) agent programming languages and platforms (e.g., PRS [11], AgentSpeak and Jason [20, 2], Jack [4], and JAM [14]) and the situation calculus-based Golog high-level programming language and its successors (e.g., ConGolog [6], IndiGolog [7, 24], and FLUX [26]) are two of the most well developed approaches within the agent-oriented programming paradigm. In this paper, we analyze the relationship between these two families of languages and show that BDI agent programming languages are closely related to IndiGolog, a situation calculus based programming language where programs are executed on-line in a dynamic environment, supporting sensing actions to acquire information from the environment and exogenous events.

BDI agent programming languages were conceived as a simplified and operationalized version of the BDI (Belief, Desire, Intention) model of agency, which is rooted in philosophical work such as Bratman's [3] theory of practical reasoning and Dennett's theory of intentional systems [8]. Practical work in the area has sought to develop programming languages that incorporate a simplified BDI semantics basis that has a computational interpretation. An important feature of BDI-style programming languages and platforms is their interleaved account of sensing, deliberation, and execution [19]. By executing as they reason, BDI agents reduce the likelihood that decisions will be made on the basis of outdated beliefs and remain responsive to the environment by making adjustments in the steps chosen as they proceed. Because of this, BDI agent programming languages are well suited to implementing systems that need to operate in "soft" real-time scenarios [16, 1]. Unlike in classical planning-based architectures, *execution* happens at each step. The assumption is that the careful crafting of plans' preconditions to ensure the selection of appropriate plans at execution time, together with a built-in mechanism for retrying alternative options, will usually ensure that a successful execution is found, even in the context of a changing environment.

In contrast to this, high-level programming languages in the Golog line aim for a middle ground between classical planning and normal programming. The idea is that the programmer may write a *sketchy* non-deterministic program involving domain specific actions and test conditions and that the interpreter will reason about these and search for a valid execution. The semantics of these languages is defined on top of the *situation calculus*, a popular predicate logic framework for reasoning about action [23]. The interpreter for the language uses an action theory representing the agent's beliefs about the state of the environment and the preconditions and effects of the actions to find a provably correct execution of the program. By controlling the amount of nondeterminism in the program, the high-level program execution task can be made as hard as classical planning or as easy as deterministic program execution. In IndiGolog, this framework is generalized to allow the programmer to control planning/lookahead and support on-line execution and sensing the environment.

In this paper, we show how a BDI agent can be built within the IndiGolog situation calculus-based programming framework. More concretely, we describe how to translate an agent programmed in a typical BDI programming language into a high-level IndiGolog program with an associated situation calculus action theory, such that *(i)* their ultimate behavior coincide; and *(ii)* the original structure of the propositional attitudes (beliefs, intentions, goals, etc.) of the BDI agent and the model of execution are preserved in the IndiGolog translation. We first do this for what we call the *core* engine of BDI systems, namely, the reactive context-sensitive expansion of events/goals. After this, we show how to accommodate more sophisticated BDI reasoning mechanisms such as goal failure recovery. In doing so, we highlight the potential additional advantages of programming BDI agents in the situation calculus, by pointing out different reasoning about action techniques that IndiGolog BDI agents may readily incorporate.

2 Preliminaries

2.1 BDI Programming

BDI agent systems were developed as a way of enabling *abstract plans* written by programmers to be combined and used in real-time, in a way that is both flexible and robust. A BDI system 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 for responding to the event/goal in question. The execution of this plan-strategy 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 in the domain. There are a number of agent programming languages and development platforms in the BDI tradition, such as PRS [11], AgentSpeak and Jason [20, 2], Jack [4], SPARK [17], Jack [4], and JADEX [18]. Our discussion is based on the CAN (**C**onceptual **A**gent **N**otation) family of BDI languages [27, 25], which are AgentSpeak-like languages with a semantics capturing the common essence of typical BDI systems.

A BDI agent Υ is a configuration tuple $\langle \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle$, where $\mathcal{B}$ stands for the agent's current beliefs about the world, generally a set of atoms, Π is the (static) plan-library, $\mathcal{A}$ is the sequence of actions executed so far, and Γ is the multi-set of intentions the agent

is currently pursuing. The *plan library* contains plan rules of the form $e : \psi \leftarrow P$, where e is an event/goal that triggers the plan, ψ is the context for which the plan may be applied (i.e., the precondition of the rule), and P is the body of the plan rule—*P is a reasonable strategy in order to resolve the event/goal e when condition ψ is believed to hold*. The *plan-body P* is a program built from primitive actions A that the agent can execute directly (e.g., $drive(loc1, loc3)$), operations to add $+b$ and delete $-b$ beliefs, tests for conditions $?\phi$, and (internal) subgoaling event posting $!e$ (e.g., $!Travel(mel, yyz)$). Complex plan bodies are built with the usual sequence ; and concurrency $\parallel$ constructs. There are also a number of auxiliary constructs internally used when assigning semantics to programs: the empty (terminating) program *nil*; the construct $P_1 \rhd P_2$, which tries to execute P_1, falling back to P_2 if P_1 is not possible; and $(\!|\psi_1 : P_1, \ldots, \psi_n : P_n|\!)$, which encodes a set of guarded plans. Lastly, the *intention base Γ* contains the current, partially instantiated, plan-body programs that the agent has already committed to for handling some events—since Γ is a multi-set it may contain a program more than once.

As with most BDI agent programming languages, the Plotkin-style operational semantics of CAN closely follows Rao and Georgeff's abstract interpreter for intelligent rational agents [22]: *(i)* incorporate any pending external events; *(ii)* select an intention and execute a step; and *(iii)* update the set of goals and intentions. A *transition relation* $C \longrightarrow C'$, on so-called *configurations* is defined by a set of *derivation rules* and specifies that executing configuration C a *single step* yields configuration C'. A *derivation rule* consists of a, possibly empty, set of premises, typically involving the existence of transitions together with some auxiliary conditions, and a single transition conclusion derivable from these premises. Two transition systems are used to define the semantics of the CAN language. The first transition relation $\longrightarrow$ defines what it means to execute a single intention and is defined in terms of *intention-level* configurations of the form $\langle \Pi, \mathcal{B}, \mathcal{A}, P \rangle$ consisting of the agent's plan-library Π and belief base $\mathcal{B}$, the actions $\mathcal{A}$ executed so far, and the program P being executed. The second transition relation $\Longrightarrow$ is defined in terms of the first and characterizes what it means to execute a whole agent.

So, the following are some of the intention-level derivation rules for the language:[3]

$$\frac{\Delta = \{\psi : P \mid e : \psi \leftarrow P \in \Pi\}}{\langle \Pi, \mathcal{B}, \mathcal{A}, !e \rangle \longrightarrow \langle \Pi, \mathcal{B}, \mathcal{A}, (\!|\Delta|\!) \rangle} \; Ev \qquad \frac{\langle \Pi, \mathcal{B}, \mathcal{A}, P_1 \rangle \longrightarrow \langle \Pi, \mathcal{B}', \mathcal{A}', P_1' \rangle}{\langle \Pi, \mathcal{B}, \mathcal{A}, P_1 \rhd P_2 \rangle \longrightarrow \langle \Pi, \mathcal{B}', \mathcal{A}', P_1' \rhd P_2 \rangle} \; \rhd$$

$$\frac{\mathcal{B} \models \phi\theta}{\langle \mathcal{B}, \mathcal{A}, ?\phi \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, nil \rangle} \; ? \qquad \frac{\psi : P \in \Delta \quad \mathcal{B} \models \psi\theta}{\langle \Pi, \mathcal{B}, \mathcal{A}, (\!|\Delta|\!) \rangle \longrightarrow \langle \Pi, \mathcal{B}, \mathcal{A}, P\theta \rhd (\!|\Delta \setminus \{\psi : P\}|\!) \rangle} \; Sel$$

Derivation rule Ev captures the first stage in the plan selection process for a (pending) event-goal e, in which the agent collects, from the plan library, the set $(\!|\Delta|\!)$ of the so-called "*relevant*" (guarded) plans that may be used to resolve the pending event. Such set is later used by rules Sel and $\rhd$ to commit to and execute, respectively, an *applicable* strategy/plan P (one whose condition ψ is believed true). Notice in rule Sel how the remaining non-selected plans are kept as backup plans as the second program in the $\rhd$ construct. Finally, rule $?$ accounts for transitions over a basic test program.

---

[3] Configurations must also include a variable substitution θ for keeping track of all bindings done so far during the execution of a plan-body. For legibility, we keep substitutions implicit in places where they need to be carried across multiple rules (e.g., in rule $?$).

On top of these intention-level derivation rules, the set of agent-level derivation rules are defined. Basically, an agent transition involves either assimilating external events from the environment or executing an active intention. Also, in the rules below, the following auxiliary function is used to represent the set of achievement events caused by belief changes: $\Omega(\mathcal{B}, \mathcal{B}') = \{!b^- \mid \mathcal{B} \models b, \ \mathcal{B}' \not\models b\} \cup \{!b^+ \mid \mathcal{B} \not\models b, \ \mathcal{B}' \models b\}$.

$$\frac{\mathcal{E} \text{ is a set of external events} \quad \mathcal{B}' = (\mathcal{B} \setminus \{b \mid -b \in \mathcal{E}\}) \cup \{b \mid +b \in \mathcal{E}\}}{\langle \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \Longrightarrow \langle \Pi, \mathcal{B}', \mathcal{A}, \Gamma \uplus \{!e \mid !e \in \mathcal{E}\} \uplus \Omega(\mathcal{B}, \mathcal{B}') \rangle} \ A_{ext}$$

$$\frac{P \in \Gamma \quad \langle \Pi, \mathcal{B}, \mathcal{A}, P \rangle \longrightarrow \langle \Pi, \mathcal{B}', \mathcal{A}', P' \rangle}{\langle \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \Longrightarrow \langle \Pi, \mathcal{B}', \mathcal{A}', (\Gamma \setminus \{P\}) \uplus \{P'\} \uplus \Omega(\mathcal{B}, \mathcal{B}') \rangle} \ A_{exec}$$

Rule A_{ext} assimilates a set of external events, both achievement ones, of the form $!e$, as well as belief updates events, of the form $+b$ or $-b$—both the belief and intention bases of the agent may be updated. Note that, by means of auxiliary function Ω, a new (achievement) event of the form $!b^+$ or $!b^-$ is posted for each belief b that changes due to an external belief update; such an event may in turn trigger some new behavior.

Rule A_{exec} states that the agent may evolve one step if an active intention P can be advanced one step with remaining intention P' being left to execute. In such a case, the intention base is updated by replacing P with P' and including the belief update events produced by potential changes in the belief base. Observe that the intention base is a *multi-set*, which means that it may contain several occurrences of the same intention.

Relative to the above derivation rules, one can formally define the meaning of an agent as its possible execution traces. (See [27, 25] for the complete semantics.)

Definition 1 (BDI Execution). *A BDI execution E of an agent $\Upsilon_0 = \langle \Pi, \mathcal{B}_0, \mathcal{A}_0, \Gamma_0 \rangle$ is a, possibly infinite, sequence of agent configurations $\Upsilon_0 \cdot \Upsilon_1 \cdot \ldots \cdot \Upsilon_n \cdot \ldots$ such that $\Upsilon_i \Longrightarrow \Upsilon_{i+1}$, for all $i \geq 0$.*

2.2 High-Level Programming in Golog

The *situation calculus* [23] is a logical language specifically designed for representing dynamically changing worlds in which all changes are the result of named *actions*. The constant S_0 is used to denote the initial situation where no actions have yet been performed. Sequences of actions are built using the function symbol *do*: $do(a, s)$ denotes the successor situation resulting from performing action a in situation s. Relations whose truth values vary from situation to situation are called *fluents*, and are denoted by predicate symbols taking a situation term as their last argument (e.g., *Holding*(x, s)). A special predicate *Poss*(a, s) is used to state that action a is executable in s.

Within this language, we can formulate action theories describing how the world changes as the result of the available actions. For example, a *basic action theory* [23] includes domain-independent foundational axioms to describe the structure of situations, one successor state axiom per fluent (capturing the effects and non-effects of actions), one precondition axiom per action, and initial state axioms describing what is true initially (i.e., what is true in the initial situation S_0).

On top of situation calculus action theories, logic-based programming languages can be defined, which, in addition to the primitive actions, allow the definition of complex actions. Golog [15], the first situation calculus agent language, provides all the

usual control structures (e.g., sequence, iteration, conditional, etc.) plus some nonde-
terministic constructs allowing the programmer to write "sketchy" plans. ConGolog [6]
extends Golog to support concurrency. To provide an intuitive overview of the language,
consider the following nondeterministic program for an agent that goes to work in the
morning (shamelessly taken from Ryan Kelly):

> **proc** $goToWork$
> $ringAlarm; (hitSnooze; ringAlarm)^*; turnOffAlarm;$
> $(\pi food)[Edible(food)?; eat(food)];$
> $(haveShower \parallel brushTeeth);$
> $(driveToUni \mid trainToUni);$
> $(Time < 11:00)?$
> **endProc**

While this high-level program provides a general strategy for getting up and going to
work, it is underspecified, and many details, such as what to eat and how to travel to
work, are left open. Program $\delta_1 \mid \delta_2$ nondeterministically chooses between programs
δ_1 and δ_2, $\pi x. \delta(x)$ executes program $\delta(x)$ for *some* legal binding for variable x, and
δ^* performs δ zero or more times. Concurrency is supported by the following three
constructs: $(\delta_1 \| \delta_2)$ expresses the concurrent execution (interpreted as interleaving) of
programs δ_1 and δ_2; $\delta_1 \rangle\rangle \delta_2$ expresses the concurrent execution of δ_1 and δ_2 with δ_1
having higher priority; and $\delta^{\|}$ executes δ zero or more times concurrently. Note that
a concurrent process may become (temporarily) blocked when it reaches a test/wait
action $\phi?$ whose condition ϕ is false (or a primitive action whose precondition is false).
Test/wait actions can also be used to control which nondeterministic branches can be
executed, e.g. $[(\phi?; \delta_1) \mid (\neg\phi?; \delta_2)]$, and to constrain the value of a nondeterministically
bound variable, e.g., $\pi x.[\phi(x)?; \delta(x)]$. Finally, the language also accommodates the
standard if-then-elses, while loops, and recursive procedures.

Finding a legal execution of a high-level program is at the core of the whole ap-
proach. Originally, Golog and ConGolog programs were intended to be executed *of-
fline*, that is, a complete execution was obtained before committing even to the first ac-
tion. However, IndiGolog [7, 24], the latest language within the Golog family, provides
a formal logic-based account of interleaved planning, sensing, and action by executing
programs *online* and using a specialized new construct $\Sigma(\delta)$, the *search operator*, to
perform local offline planning when required.

Roughly speaking, an *online* execution of a program finds a next possible action, ex-
ecutes it in the real world, then obtains sensing information, and repeats the cycle until
the program is completed. Formally, an online execution is a sequence of so-called on-
line configuration of the form (δ, σ), where δ is a high-level program and σ is a history
(see [7] for its formal definition). A history contains the sequence of actions executed
so far as well as the sensing information obtained. Online executions are characterized
in terms of the following two predicates [6]: *Final*(δ, s) holds if program δ may legally
terminate in situation s; and *Trans*(δ, s, δ', s') holds if a single step of program δ in sit-
uation s may lead to situation s' with δ' remaining to be executed. In the next section,
we will generalize the notion of online execution to suit our purposes.

3 BDI Programming in IndiGolog

Programming a BDI agent in the situation calculus amounts to developing a special basic action theory and a special IndiGolog high-level agent program to be executed with it. From now on, let $\Upsilon = \langle \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle$ be the BDI agent to program in IndiGolog.

3.1 The BDI Basic Action Theory

We start by showing how to obtain an action theory $\mathcal{D}^{\Upsilon}$ for our agent Υ. We assume that Υ is stated over a first-order language $\mathcal{L}_{BDI}$ containing finitely many belief and event atomic relations, namely, $b_1(\boldsymbol{x_1}), \ldots, b_n(\boldsymbol{x_n})$ and $e_1(\boldsymbol{x_1}), \ldots, e_m(\boldsymbol{x_n})$.

Let us then define what the fluents and actions available in the situation calculus language $\mathcal{L}_{sitCalc}$ are. First, for every belief atomic predicate $b(\boldsymbol{x})$ in $\mathcal{L}_{BDI}$, the language $\mathcal{L}_{sitCalc}$ includes a relational fluent $b(\boldsymbol{x}, s)$ together with two primitive actions $add_b(\boldsymbol{x})$ and $del_b(\boldsymbol{x})$ which are meant to change the fluent's truth value. Second, for each achievement event type $e(\boldsymbol{x})$ in the domain, there is a corresponding action term $ach_e(\boldsymbol{x})$ in $\mathcal{L}_{sitCalc}$. Finally, for every action atom $A(\boldsymbol{x})$ in $\mathcal{L}_{BDI}$, there is a corresponding action function $A(\boldsymbol{x})$ in $\mathcal{L}_{sitCalc}$.

In addition, the language $\mathcal{L}_{sitCalc}$ shall include one auxiliary distinguished fluent and two actions to model external event handling. Fluent *PendingEv*(s) stands for the multi-set of events that are "pending" and need to be handled, either belief update or achievement events. This fluent is affected by two actions. Whereas action *post*(e) indicates the external posting of event e; action *serve*(e) indicates that (pending) event e has been selected and is being handled. In both actions, argument e is of sort action.

Let us now construct the basic action theory $\mathcal{D}^{\Upsilon}$ corresponding to a BDI agent $\Upsilon = \langle \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle$, as follows:

1. The initial description in $\mathcal{D}^{\Upsilon}$ is defined in the following way:

$$\mathcal{D}_{S_0}^{\Upsilon} = \bigcup_{i=1}^{n} \{\forall \boldsymbol{x}.b_i(\boldsymbol{x}, S_0) \equiv \boldsymbol{x} = \boldsymbol{t_i^1} \vee \ldots \vee \boldsymbol{x} = \boldsymbol{t_i^{k_i}}\} \cup$$
$$\{\forall a.\textit{Exog}(a) \equiv (\exists a')a = post(a')\},$$

where for every $i \in \{1, \ldots, n\}$, $\mathcal{B} \models b_i(\boldsymbol{x}) \equiv [\boldsymbol{x} = \boldsymbol{t_i^1} \vee \ldots \vee \boldsymbol{x} = \boldsymbol{t_i^{k_i}}]$, for some $k_i \geq 0$—$b_i(\boldsymbol{t_i^1}), \ldots, b_i(\boldsymbol{t_i^{k_i}})$ are all the true belief atoms in $\mathcal{B}$ with respect to belief relation b_i (each $\boldsymbol{t_i^j}$ is a vector of ground terms).

2. The following precondition axioms, for every fluent $b(\boldsymbol{x})$ and action type $A(\boldsymbol{x})$:

$$Poss(serve(a), s) \equiv (a \in \textit{PendingEv}(s)) \qquad Poss(A(\boldsymbol{x}), s) \equiv \texttt{True}$$
$$Poss(add_b(\boldsymbol{x}), s) \equiv Poss(del_b(\boldsymbol{x}), s) \equiv \texttt{True} \qquad Poss(post(a), s) \equiv \texttt{True}$$

3. For every domain fluent $b(\boldsymbol{x}, s)$, $\mathcal{D}^{\Upsilon}$ includes the following successor state axiom:

$b(\boldsymbol{x}, do(a, s)) \equiv$
$a = add_b(\boldsymbol{x}) \vee a = post(add_b(\boldsymbol{x})) \vee b(\boldsymbol{x}, s) \wedge (a \neq del_b(\boldsymbol{x}) \wedge a \neq post(del_b(\boldsymbol{x})))$.

That is, the truth-value of fluent b is affected only by actions add_b and del_b, either internally executed or externally sensed from the environment.

More importantly, action theory $\mathcal{D}^{\Upsilon}$ includes a successor state axiom for fluent $PendingEv(do(a,s))$ specifying how the multi-set of pending events changes:

$$PendingEv(do(a,s)) = v \equiv [\gamma(a,v,s) \lor PendingEv(s) = v \land \neg\exists v'.\gamma(a,v',s)];$$

where:

$$\gamma(a,v,s) \stackrel{\text{def}}{=} \big(\bigvee_{i=1}^{n}[\gamma_i^+(a,v,s) \lor \gamma_i^-(a,v,s)] \lor \bigvee_{i=1}^{m}[\gamma_i^e(a,v,s)] \lor$$
$$\exists a'.a = serve(a') \land v = PendingEv(s) \setminus \{a'\}\big);$$

$$\gamma_i^+(a,v,s) \stackrel{\text{def}}{=}$$
$$\exists \boldsymbol{x}. a \in \{add_{b_i}(\boldsymbol{x}), post(add_{b_i}(\boldsymbol{x}))\} \land \neg b_i(\boldsymbol{x}) \land v = PendingEv(s) \uplus \{add_{b_i}(\boldsymbol{x})\};$$

$$\gamma_i^-(a,v,s) \stackrel{\text{def}}{=}$$
$$\exists \boldsymbol{x}. a \in \{del_{b_i}(\boldsymbol{x}), post(del_{b_i}(\boldsymbol{x}))\} \land b_i(\boldsymbol{x}) \land v = PendingEv(s) \uplus \{del_{b_i}(\boldsymbol{x})\};$$

$$\gamma_i^e(a,v,s) \stackrel{\text{def}}{=} \exists \boldsymbol{x}. a = post(ach_{e_i}(\boldsymbol{x})) \land v = PendingEv(s) \uplus \{ach_{e_i}(\boldsymbol{x})\}.$$

That is, an actual change in the belief of an atom, either due to the execution of some intention or an external event, automatically produces a corresponding pending belief update event. Moreover, an external achievement event $ach_e(\boldsymbol{x})$ becomes pending when sensed. On the other hand, an event e ceases to pending when action $serve(e)$ is executed.

4. Theory $\mathcal{D}^{\Upsilon}$ includes unique name axioms for all actions in $\mathcal{L}_{sitCalc}$, as well as the standard domain-independent foundational axioms for the situation calculus ([23]).

This concludes the construction of the BDI basic action theory $\mathcal{D}^{\Upsilon}$.

3.2 The BDI Agent Program

Let us now construct the IndiGolog BDI agent program δ^{Υ} that is meant to execute relative to the BDI action theory $\mathcal{D}^{\Upsilon}$. We start by showing how to inductively transform a BDI plan-body program P into an IndiGolog program δ_P, namely (remember that plan-bodies programs are used to build BDI plans in the plan library):

$$\delta_P = \begin{cases} P & \text{if } P = A \mid nil \\ \phi? & \text{if } P =?\phi \\ add_b(\boldsymbol{t}) & \text{if } P = +b(\boldsymbol{t}) \\ del_b(\boldsymbol{t}) & \text{if } P = -b(\boldsymbol{t}) \\ handle(ach_e(\boldsymbol{t})) & \text{if } P =!e(\boldsymbol{t}) \\ (\delta_{P_1};\delta_{P_2}) & \text{if } P = (P_1;P_2) \\ \delta_{P_1} & \text{if } P = P_1 \triangleright P_2 \\ achieve_e(\boldsymbol{t}) & \text{if } P = (\!|\Delta|\!), \text{for some event } e(\boldsymbol{t}) \end{cases}$$

Notice that achievement events $!e$ occurring in a plan are handled via simple plan invocation, by invoking procedure $handle$. Also, for now, the translation just ignores the second program in $P_1 \triangleright P_2$, as the version of CAN we consider in this section does not try P_2 when P_1 happens fail. We shall revisit this later in Section 4.

Next, we describe how to transform the BDI plans in the agent's plan library. To that end, suppose that $e(\boldsymbol{x})$ is an event in the BDI language $\mathcal{L}_{BDI}$ such with the following $n \geq 0$ plans in Π ($\boldsymbol{v_t}$ denotes all the distinct free variables in the terms $\boldsymbol{t}$):

$$e(\boldsymbol{t_i}) : \psi_i(\boldsymbol{v_{t_i}}, \boldsymbol{y_i}) \leftarrow P_i(\boldsymbol{v_{t_i}}, \boldsymbol{y_i}, \boldsymbol{z_i}), \text{ where } i \in \{1, \ldots, n\}.$$

Then, we build the following high-level Golog procedure with n non-deterministic choices (i.e., as many as plan-rules for the event):

proc $achieve_e(\boldsymbol{x})$
$\quad |_{i \in \{1,\dots,n\}} \; [(\pi \boldsymbol{v_{t_i}}, \boldsymbol{y_i}, \boldsymbol{z_i}).(\boldsymbol{x} = \boldsymbol{t_i} \wedge \psi_i(\boldsymbol{v_{t_i}}, \boldsymbol{y_i}))?; \delta_{P_i}(\boldsymbol{v_{t_i}}, \boldsymbol{y_i}, \boldsymbol{z_i})]$
endProc

Roughly speaking, executing $achieve_e(\boldsymbol{x})$ involves nondeterministically choosing among the n available options in the plan library for event e. See that the first test statement in each option amounts to checking the relevance and applicability of the option. Thus, the execution of $achieve_e(\boldsymbol{x})$ is bound to *block* if no option is relevant or applicable. In particular, the procedure will *always* block if the agent Υ has no plan to handle the event in question—that is, if $n = 0$, the corresponding Golog procedure is simply ?(False).

Let Δ_Π denote the set of Golog procedures as above, one per event in the BDI language, together with the following procedure:

proc $handle(a)$
$\quad |_{i=1}^{n} \; [(\exists \boldsymbol{x_i}.a = add_{b_i}(\boldsymbol{x_i}))?; \; achieve_{b_i^+}(\boldsymbol{x_i})] \; |$
$\quad |_{i=1}^{n} \; [(\exists \boldsymbol{x_i}.a = del_{b_i}(\boldsymbol{x_i}))?; \; achieve_{b_i^-}(\boldsymbol{x_i})] \; |$
$\quad |_{i=1}^{m} \; [(\exists \boldsymbol{x_i}.a = ach_{e_i}(\boldsymbol{x_i}))?; \; achieve_{e_i}(\boldsymbol{x_i})]$
endProc

That is, when a is a legal event (belief update or achievement goal), procedure $handle(a)$ calls the appropriate procedure that is meant to *resolve* the event. Observe that this program contains two nondeterministic programs per belief atom in the domain (one to handle its addition and one to handle its deletion from the belief base), plus one nondeterministic program per achievement event in the domain.

Finally, we define the top-level IndiGolog BDI agent program as follows:

$$\delta^\Upsilon \stackrel{\text{def}}{=} \Delta_\Pi; [\delta_{env} \| (\delta_\Gamma \| \delta_{BDI})]; (\neg \exists e \; PendingEv(e))?, \tag{1}$$

where (assuming that $\Gamma = \{P_1, \dots, P_n\}$):

$$\delta_\Gamma \stackrel{\text{def}}{=} \delta_{P_1} \| \cdots \| \delta_{P_n}; \quad \delta_{env} \stackrel{\text{def}}{=} (\pi a.Exog(a)?; a)^*; \quad \delta_{BDI} \stackrel{\text{def}}{=} [\pi a.serve(a); handle(a)]^{\|}.$$

The set of programs Δ_Π provides the environment encoding the BDI plan library. Program δ_Γ accounts for all current intentions in Υ; if $\Gamma = \emptyset$, then $\delta_\Gamma = nil$. In turn, program δ_{env} models the external environment, which can perform zero, one, or more actions of the form $post(a)$, representing an external achievement event goal ($a = ach_e(\boldsymbol{t})$) or a belief update event ($a = add_b(\boldsymbol{t})$ or $a = del_b(\boldsymbol{t})$).

The most interesting part of δ^Υ is indeed the ConGolog program δ_{BDI}, which implements (part of) the BDI execution cycle. More concretely, δ_{BDI} is responsible for selecting an external event and *spawning* a new "intention" concurrent thread for handling it. To that end, δ_{BDI} picks an event a (e.g., $add_{At}(23, 32)$ or $achieve_{moveTo}(0, 0)$) to be served and executes action $serve(a)$. Observe that an event can be served only if it is currently pending (see action precondition for action $serve(a)$ in Subsection 3.1). After the action $serve(a)$ has been successfully executed, the selected event a is actually handled, by calling procedure $handle(a)$ defined in Δ_Π. More importantly, this is done

in a "new" concurrent thread, so that program δ_{BDI} is still able to serve and handle other pending events. The use of concurrent iteration to spawn a new intention from the "main BDI thread" is inspired from the server example application in [6].

Note that Δ_Π and δ_Γ are domain dependent, i.e., they are built relative to a particular BDI agent Υ, whereas programs δ_{BDI} and δ_{env} are independent of the BDI agent being encoded. Observe also that the whole high-level program δ^Υ may terminate only when no more events are pending.

From now on, let $\mathcal{G}^\Upsilon = \langle \mathcal{D}^\Upsilon, \delta^\Upsilon, \mathcal{A} \rangle$ denote the IndiGolog agent for BDI agent Υ.

3.3 LC-Online Executions

Once we have a BDI IndiGolog program $\mathcal{G}^\Upsilon$ on hand, we should be able to execute it and obtain the same behavior and outputs as with the original BDI agent. Unfortunately, we cannot execute $\mathcal{G}^\Upsilon$ *online*, as defined in [7], as such executions may commit too early to free variables in a program—online executions are sequences of *ground* online configurations. For example, program $\pi x.\texttt{True}?; (x = A)?$ can do a transition where it instantiates x to B to produce the remaining program $B = A?$, which then fails.

What we need, instead, is an account of execution that commits to free variables only when necessary. To that end, we generalize the online execution notion from [7] to what we call *least-committed* online executions. We first define two meta-theoretic versions of relations *Trans* and *Final* as follows:

$$
\begin{aligned}
mTrans(\delta(\boldsymbol{x}, \boldsymbol{y}), \sigma, \delta'(\boldsymbol{x}, \boldsymbol{z}), \sigma') &\stackrel{\text{def}}{=} \\
&Axioms[\mathcal{D}, \sigma] \models \exists \boldsymbol{y} \forall \boldsymbol{x}, \boldsymbol{z}.Trans(\delta(\boldsymbol{x}, \boldsymbol{y}), end[\sigma], \delta'(\boldsymbol{x}, \boldsymbol{z}), end[\sigma']);
\end{aligned}
$$

$$
mFinal(\delta(\boldsymbol{x}, \boldsymbol{y}), \sigma) \stackrel{\text{def}}{=} Axioms[\mathcal{D}, \sigma] \models \exists \boldsymbol{x}.Final(\delta(\boldsymbol{x}), end[\sigma]).
$$

(Here, in $\delta(\boldsymbol{x})$ the vector of variables $\boldsymbol{x}$ contains *all* the free variables mentioned in the program, and different variables vectors are assumed disjoint; $end[\sigma]$ denotes the situation term corresponding to the history σ; and $Axioms[\mathcal{D}, \sigma]$ denotes the complete set of axioms in the IndiGolog theory, which includes the action theory $\mathcal{D}$ for the domain and all the axioms for *Trans* and *Final*.)

We can then define least-committed executions as follows.

Definition 2 (LC-Online Execution). *A least-committed online (lc-online) execution of an IndiGolog program δ starting from a history σ is a, possibly infinite, sequence of configurations $(\delta_0 = \delta, \sigma_0 = \sigma), (\delta_1, \sigma_1), \ldots$ such that for every $i \geq 0$:*

1. *$mTrans(\delta_i, \sigma_i, \delta_{i+1}, \sigma_{i+1})$ holds; and*
2. *for all δ' such that $mTrans(\delta_i, \sigma_i, \delta', \sigma_{i+1})$ and $\delta_{i+1} = \delta'\theta$ for some substitution θ, there exists θ' such that $\delta' = \delta_{i+1}\theta'$.*

A finite lc-online execution $(\delta_0, \sigma_0), \ldots, (\delta_n, \sigma_n)$ is terminating iff $mFinal(\delta_n, \sigma_n)$ or for all δ', σ' $mTrans(\delta_n, \sigma_n, \delta', \sigma')$ does not hold.

We notice that, as expected, it can be shown that an lc-online execution stands for all its ground online instances as defined in [7]. However, by executing programs in a least committed way, we avoid premature binding of variables and eliminate some executions where the program is bound to fail.

3.4 BDI/IndiGolog Bisimulation

We are now ready to provide the main results of the paper. Namely, we show that given any BDI execution of an agent, there exists a matching execution of the corresponding IndiGolog agent, and vice-versa. In addition, the correspondence in the internal structure of the agents is always maintained throughout the executions.

We start by characterizing when a BDI agent and an IndiGolog agent configuration "match." To that end, we shall use relation $\Upsilon \approx \mathcal{G}$, which, intuitively, holds if a BDI agent Υ and an IndiGolog agent $\mathcal{G}$ represent the same (BDI) agent system. Formally, relation $\langle \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \approx \langle \mathcal{D}, \delta, \sigma \rangle$ holds iff

1. $\delta = \Delta_\Pi; [\delta_{env} \| (\delta_{\Gamma'} \| \delta_{BDI})]; ?(\neg \exists e \; PendingEv(e))$, for some $\Gamma' \subseteq \Gamma$ such that $\Gamma = \Gamma' \uplus \{a \mid Axioms[\mathcal{D}, \sigma] \models a \in PendingEv(end[\sigma])\}$;
2. $\mathcal{A}$ and σ contain the same sequence of *domain* actions;
3. for every ground belief atom b: $\mathcal{B} \models b$ iff $Axioms[\mathcal{D}, \sigma] \models b[end[\sigma]]$;
4. $\mathcal{D} = \mathcal{D}^{\Upsilon'}$, for some $\Upsilon' = \langle \Pi, \mathcal{B}', \mathcal{A}, \Gamma \rangle$.

The first condition states that the IndiGolog program is of the form shown in equation (1) above (see Section 3.2), but where some active intentions may still be "pending." In other words, some intentions in Γ that have not yet started execution may not show up yet as concurrent processes in δ, but they are implicitly represented as "pending" in fluent $PendingEv(s)$. The second requirement states that both agents have performed the same sequence of domain primitive actions, that is, actions other than the internal ones $serve(a)$, $post(a)$, $add_b(\boldsymbol{x})$, and $del_b(\boldsymbol{x})$. The third condition requires both agents to coincide on what they *currently* believe. Observe that the *initial* beliefs of the IndiGolog agent do not necessarily need to coincide with those of the BDI agent, as long as the *current* beliefs do (i.e., the beliefs that hold after history σ); in fact the BDI agent configuration does not specify what it believed initially, while the IndiGolog agent's action theory does. Lastly, the IndiGolog agent executes relative to a basic action theory whose dynamics are as described in Section 3.1.

First of all, it is possible to show that the encoding of initial BDI agents, that is agents that have not yet performed any action, into IndiGolog agents described above is in the $\approx$ relation with the original BDI agent.

Theorem 1. *Let Υ be an initial BDI agent (that is, $\mathcal{A} = \epsilon$). Then, $\Upsilon \approx \langle \mathcal{D}^\Upsilon, \delta^\Upsilon, \mathcal{A} \rangle$.*

The importance of a BDI agent and an IndiGolog agent being in the $\approx$ relation is that their respective transitions can then always be simulated by the other type of agent To demonstrate that, we first show that any BDI transition can be replicated by the corresponding IndiGolog agent. Observe that IndiGolog may need several transitions to replicate the BDI transition when it comes to assimilating external events; whereas BDI agents incorporate sets of external events in a single transition, the IndiGolog agent incorporates one event per transition. Also, IndiGolog agents ought to execute the special action $serve(a)$ to start handling external achievement events.

Theorem 2. *Let Υ be a BDI agent and $\langle \mathcal{D}, \delta, \sigma \rangle$ an IndiGolog agent such that $\Upsilon \approx \langle \mathcal{D}, \delta, \sigma \rangle$. If $\Upsilon \implies \Upsilon'$, then there exists a program δ' and a history σ' such that $mTrans^*(\delta, \sigma, \delta', \sigma')$ holds relative to action theory $\mathcal{D}$, and $\Upsilon' \approx \langle \mathcal{D}, \delta', \sigma' \rangle$.*

Furthermore, in the other direction, any step in a BDI IndiGolog execution can always be "mimicked" by the corresponding BDI agent.

Theorem 3. *Let Υ and $\langle \mathcal{D}, \delta, \sigma \rangle$ be a BDI and an IndiGolog agents, respectively, such that $\Upsilon \approx \langle \mathcal{D}, \delta, \sigma \rangle$. Suppose that $mTrans(\delta, \sigma, \delta', \sigma')$ holds relative to action theory $\mathcal{D}$, for some IndiGolog program δ' and history σ'. Then, either $\Upsilon \approx \langle \mathcal{D}, \delta', \sigma' \rangle$ or there exists a BDI agent Υ' such that $\Upsilon \Longrightarrow \Upsilon'$ and $\Upsilon' \approx \langle \mathcal{D}, \delta', \sigma' \rangle$.*

So, when the IndiGolog agent performs a transition it remains "equivalent" to the BDI agent or to some evolution of the BDI agent. The former case applies only when the transition in question involved the execution of a *serve(a)* action to translate a pending event into a concurrent process.

Putting both theorems together, our encoding allows IndiGolog to bisimulate BDI agents.

4 BDI Failure Handling

Since BDI systems are meant to operate in dynamic settings, plans that were supposed to work may fail due to changes in the environment. Indeed, a plan may fail because a test condition $?\phi$ is not believed true, an action cannot be executed, or a sub-goal event does not have any applicable plans. The BDI language we have discussed so far has no strategy towards failed plans or intentions, once an intention cannot evolve, it simply remains in the intention base *blocked*. In this section, we discuss how BDI programming languages typically address plan/intention failure and show how the above IndiGolog encoding can be extended accordingly. In particular, we show how agents can *abandon* failed intentions and *recover* from problematic plans by trying alternative options.

Before getting into technical details, we shall first introduce a new construct into the IndiGolog language. In "Golog-like" languages, a program that is *blocked* may not be dropped for the sake of another program. To overcome this, we introduce the construct $\delta_1 \rhd \delta_2$ with the intending meaning that δ_1 should be executed, falling back to δ_2 if δ_1 becomes blocked:[4]

$$Trans(\delta_1 \rhd \delta_2, s, \delta', s') \equiv (\exists \gamma.Trans(\delta_1, s, \gamma, s') \wedge \delta' = \gamma \rhd \delta_2) \ \vee$$
$$\neg\exists\gamma, s''.Trans(\delta_1, s, \gamma, s'') \wedge Trans(\delta_2, s, \delta', s');$$

$$Final(\delta_1 \rhd \delta_2, s, \delta', s') \equiv Final(\delta_1, s) \vee \neg\exists\gamma, s''.Trans(\delta_1, s, \gamma, s'') \wedge Final(\delta_2, s).$$

4.1 Dropping Impossible Intentions

It is generally accepted that intentions that cannot execute further may simply be *dropped* by the agent — rational agents should not pursue intentions/goals that are deemed impossible [21, 5]. This is indeed the behavior of AgentSpeak agents.[5]

The BDI language of Section 2.1 can be easily extended to provide such an intention-dropping facility, by just adding the following agent-level operational rule:

$$\frac{P \in \Gamma \quad \langle \Pi, \mathcal{B}, \mathcal{A}, P \rangle \not\longmapsto}{\langle \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \Longrightarrow \langle \Pi, \mathcal{B}, \mathcal{A}, \Gamma \setminus \{P\} \rangle} \ A_{clean}$$

---

[4] One could easily extend these definitions to only allow dropping a blocked δ_1 under given conditions; this could be used to implement "time outs" or allow blocking for synchronization.

[5] There has been work on more sophisticated treatments of plan failure in AgentSpeak [2].

That is, an agent may choose to just drop an intention from its intention base if it cannot execute further in the current mental state. To mimic this behavior in our BDI IndiGolog formalization, we slightly modify the domain-independent program δ_{BDI} as follows:

$$\delta_{BDI} \stackrel{\text{def}}{=} [\pi a.serve(a); (handle(a) \rhd (\texttt{True})?)]^{\|}.$$

Here, a pending event is handled within the scope of a $\rhd$, which basically allows the intention thread to simply terminate if it becomes blocked. Notice that, as with BDI languages, for procedure $handle(a)$ to be blocked, every sub-goal event triggered by the handling of a (represented in the IndiGolog program as simple procedure calls) ought to be blocked. Observe also that in this approach, only the main program corresponding to a top-level event may be dropped, not lower-level instrumental subgoals.

4.2 BDI Goal Failure Recovery

Merely dropping a whole intention when it becomes *blocked* provides a rather weak level of commitment to goals. The failure of a plan should not be equated to the failure of its parent goal, as there could be alternative ways to achieve the latter. For example, suppose an agent has the goal to quench her thirst, and in the service of this goal, she adopts the subgoal of buying a can of soda [25]. However, upon arrival at the store, she realizes that all the cans of soda are sold out. Fortunately though, the shop has bottles of water. In this situation, it is irrational for the agent to drop the whole goal of quenching her thirst just because soda is not available. An AgentSpeak agent may do so. Similarly, we do not expect the agent to fanatically insist on her subgoal and just wait indefinitely for soda to be delivered. What we expect is the agent to merely drop her commitment to buy soda and adopt the alternative (sub)goal of buying a bottle of water, thereby achieving the *main* goal.

As a matter of fact, one of the typical features of implemented BDI languages is that of plan-goal failure recovery: if a plan happens to fail for a goal, usually due to unexpected changes in the environment, another plan is tried to achieve the goal. If no alternative strategy is available, then the goal is deemed failed and failure is propagated to higher-level motivating goals, and so on. This mechanism thus provides a stronger level of *commitment to goals*, by decoupling plan failure from goal failure. To accommodate failure handling, we further extend the BDI language of Section 2.1, by providing the following additional derivation rule for construct $\rhd$:

$$\frac{\langle \Pi, \mathcal{B}, \mathcal{A}, P_1 \rangle \not\longrightarrow \quad \langle \Pi, \mathcal{B}', \mathcal{A}', P_2' \rangle \longrightarrow \langle \Pi, \mathcal{B}', \mathcal{A}', P_2' \rangle}{\langle \Pi, \mathcal{B}, \mathcal{A}, P_1 \rhd P_2 \rangle \longrightarrow \langle \Pi, \mathcal{B}', \mathcal{A}', P_2' \rangle} \rhd_f$$

That is, if the current strategy P_1 is *blocked* but the alternative backup program P_2 is able to evolve, then it is legal to drop P_1 and switch to P_2. Observe that due to derivation rules Ev and Sel, $P_2 = (\!|\Delta|\!)$ will encode the set of *relevant* plans that have not yet been tried for the event being addressed. From now on, let the CAN language refer to our extended BDI language, with both new derivation rules A_{clean} and $\rhd_f$ for failure included.

Hence, due to the interaction between derivation rules Ev, Sel and $\rhd_f$, a CAN BDI agent executes a program $P_1 \rhd (\!|\Delta|\!)$ in order to resolve an goal event $!e$. When the

current strategy P_1 being pursued is not able to make a step, the agent may check the set of alternatives $(|\Delta|)$ in the hope of finding another option P_2 for addressing e. If one is found, the agent may opt to abandon its strategy P_1 and continue with P_2. (Details can be found in [27, 25].)

Let us now describe how to replicate this failure recovery behavior within our IndiGolog framework of Section 3. For simplicity, we shall assume that, as with actions, only *ground* posting of subgoal events are allowed in the BDI language. This means that all variables $\boldsymbol{x}$ in an event posting $!e(\boldsymbol{x})$ are considered *inputs* to the event. If an event is meant to return data, it must do so by using of the belief base. To support failure recovery, we slightly modify how plans in the plan library Π are converted into ConGolog procedures. Specifically, for each event $e(\boldsymbol{x})$, we define the following procedure (and make procedure $achieve_e(\boldsymbol{x})$ simply call $achieve'_e(\boldsymbol{x}, [1, \dots, 1])$):

proc $achieve'_e(\boldsymbol{x}, \boldsymbol{w})$ // $\boldsymbol{w}$ is an n-long vector
$|_{i \in \{1, \dots, n\}} \, [(\pi \boldsymbol{v_{t_i}}, \boldsymbol{y_i}, \boldsymbol{z_i}).(\boldsymbol{x} = \boldsymbol{t} \wedge \psi_i(\boldsymbol{v_t}, \boldsymbol{y}) \wedge w = 1)?; \delta_{P_i}(\boldsymbol{v_{t_i}}, \boldsymbol{y_i}, \boldsymbol{z_i}) \triangleright \Phi_i(\boldsymbol{x}, \boldsymbol{w})]$
endProc

where $\Phi_i(\boldsymbol{x}, \boldsymbol{w}) \overset{\text{def}}{=} achieve'_e(\boldsymbol{x}, [w_1, \dots, w_{i-1}, 0, w_{i+1}, \dots, w_n])$.

Vector $\boldsymbol{w}$ has one component per plan rule in the library for the event in question; its i-th component w_i states whether the i-th plan in Π is *available* for selection. Condition $(\boldsymbol{x} = \boldsymbol{t} \wedge \psi_i(\boldsymbol{v_t}, \boldsymbol{y}) \wedge w = 1)$ checks whether event $e(\boldsymbol{x})$ is relevant, applicable, and available. Program Φ_i determines the *recovery strategy*, in this case, recursively calling the procedure to achieve the event, but removing the current plan from consideration (by setting its component in $\boldsymbol{w}$ to 0). Due to the semantics of $\triangleright$, recovery would only be triggered if procedure $achieve'_e(\boldsymbol{x}, \boldsymbol{w})$ may execute one step, which implies that there is indeed an available plan that is relevant and applicable for the event.

It turns out that these are the only modifications to the encoding of Section 3 required to mimic the behavior of CAN agents with failure handling in the IndiGolog high-level language.

Theorem 4. *Theorems 2 and 3 hold for CAN agents under the extended translation to IndiGolog agents.*

More interestingly, the proposed translation can be adapted to accommodate several alternative accounts of execution and failure-recovery. For example, goal failure recovery can be disallowed for an event by just taking $\Phi_i(\boldsymbol{x}, \boldsymbol{w}) \overset{\text{def}}{=} ?(\texttt{False})$ above. Similarly, a framework under which *any* plan may be (re)tried for achieving a goal event, regardless of previous (failed) executions, is obtained by taking $\Phi_i(\boldsymbol{x}, \boldsymbol{w}) \overset{\text{def}}{=} achieve_e(\boldsymbol{x})$. In this case, the event is "fully" re-posted within the intention.

The key point here is that, due to the fact that the BDI execution and recovery model is represented in our BDI IndiGolog at the *object* level, one can even go further and design more sophisticated accounts of execution and failure recovery for BDI agents. It is straightforward, for instance, to model the kind of goal failure recovery originally proposed for AgentSpeak, in which the system would automatically post a distinguished *failure goal* (denoted $!{-}g$); the programmer may then choose to provide plans to handle such failure events. A failure handling plan could, for example, carry out some cleanup tasks and even re-post the failed event [20, 2]. This type of behavior can be easily

achieved by taking $\Phi_i(\boldsymbol{x}, \boldsymbol{w}) \stackrel{\text{def}}{=} ach_{fail\_e}(\boldsymbol{x}); ?(\texttt{False})$, and allowing the programmer to provide plan rules in the library for handling the special event $fail\_e(\boldsymbol{x})$. Notice that the event is *posted* so it would eventually create a new intention all-together; the current plan would then immediately be blocked/failed.

5   Discussion

In this paper, we have shown how one can effectively program BDI-style agent systems in the situation calculus-based IndiGolog high-level programming language. The benefits of this are many. First, we gain a better understanding of the common features of BDI agent programming languages and "Golog-like" high-level programming languages, as well as of what is specific to each type of language, and what is required to reproduce BDI languages in the latter. We also get a new classical-logic situation calculus-based semantics for BDI agent programming languages. This opens many avenues for enhancing the BDI programming paradigm with reasoning capabilities, for instance, model-based belief update capabilities, lookahead planning capabilities, plan/goal achievement monitoring capabilities, etc. Our account also shows how one can essentially compile the BDI execution engine of a BDI agent into an object-level IndiGolog program, about which we can reason in the situation calculus. From the perspective of situation calculus-based high-level programming languages, we have enhanced IndiGolog with a more general semantic account of program execution, i.e. least-committed online executions, and we have also introduced a novel language construct that is useful for failure handling. Moreover, our work opens up new perspectives for developing logic-based agent programming languages with BDI features.

There has only been limited work on relating "Golog-like" and BDI programming languages. Hindriks *et al.* [13] show that ConGolog can be bisimulated by the agent language 3APL under some conditions, which include the agent having complete knowledge. In [12], it is also shown that AgentSpeak can be encoded into 3APL. Our results, thus, are complementary, in showing the inverse relationship.

Also related is the work of Gabaldon [10] on encoding Hierarchical Task Network (HTN) libraries in ConGolog. There are similarities between our work and his in the way procedural knowledge is encoded in ConGolog. This is is not surprising, as HTN planning systems and BDI agents have many similarities [9]. But note that in HTNs, and hence in Gabaldon's translation, the objective is *planning* and not reactive execution. We on the other hand, focus on capturing the typical execution regime of BDI agent systems, rather than on performing lookahead planning to synthesize a solution. As a result, we address issues such as external events and plan failure that do not arise in HTN planning.

References

1. S. S. Benfield, J. Hendrickson, and D. Galanti. Making a strong business case for multiagent technology. In *Proc. of AAMAS*, pages 10–15, New York, NY, USA, 2006. ACM Press.

2. R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley, 2007. Series in Agent Technology.
3. M. E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, 1987.
4. P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. JACK intelligent agents: Components for intelligent agents in Java. *AgentLink Newsletter*, 2,1999. Agent Oriented Soft. Pty. Ltd.
5. P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence Journal*, 42:213–261, 1990.
6. G. De Giacomo, Y. Lespérance, and H. J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Art. Intelligence Journal*, 121(1–2):109–169, 2000.
7. G. De Giacomo and H. J. Levesque. An incremental interpreter for high-level programs with sensing. In H. J. Levesque and F. Pirri, editors, *Logical Foundations for Cognitive Agents: Contributions in Honor of Ray Reiter*, pages 86–102. Springer, Berlin, 1999.
8. D. Dennett. *The Intentional Stance*. The MIT Press, 1987.
9. J. Dix, H. Muñoz-Avila, D. S. Nau, and L. Zhang. IMPACTing SHOP: Putting an AI planner into a multi-agent environment. *Annals of Mathematics and Artificial Intelligence*, 37(4):381–407, 2003.
10. A. Gabaldon. Programming hierarchical task networks in the situation calculus. In Proc. of *AIPS'02 Workshop on On-line Planning and Scheduling*, Toulouse, France, April 2002.
11. M. P. Georgeff and F. F. Ingrand. Decision making in an embedded reasoning system. In *Proc. of IJCAI'89*, pages 972–978, Detroit, USA, 1989.
12. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Meyer. A formal semantics for an abstract agent programming language. In *Proc. of ATAL'98*, pages 215–229, 1998.
13. K. V. Hindriks, Y. Lespérance, and H. J. Levesque. An embedding of ConGolog in 3APL. Technical Report UU-CS-2000-13, Dept. of Computer Science, University Utrecht, 2000.
14. M. J. Huber. JAM: A BDI-theoretic mobile agent architecture. In *Proceedings of the Annual Conference on Autonomous Agents (AGENTS)*, pages 236–243, New York, USA, 1999.
15. H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.
16. M. Ljungberg and A. Lucas. The OASIS air-traffic management system. In *Proceedings of the Pacific Rim International Conference on Artificial Intelligence (PRICAI)*, 1992.
17. D. Morley and K. L. Myers. The SPARK agent framework. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 712–719, New York, USA, 2004.
18. A. Pokahr, L. Braubach, and W. Lamersdorf. JADEX: Implementing a BDI-infrastructure for JADE agents. *EXP-In search of innovation (Special Issue on JADE)*, 3(3):76–85, 9 2003.
19. M. E. Pollack. The uses of plans. *Artificial Intelligence Journal*, 57(1):43–68, 1992.
20. A. S. Rao. Agentspeak(L): BDI agents speak out in a logical computable language. In LNCS, vol. 1038, p. 42–55. Springer, 1996.
21. A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In *Proc. of KR'91*, pages 473–484, 1991.
22. A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In *Proc. of KR'92*, pages 438–449, 1992.
23. R. Reiter. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001.
24. S. Sardina, G. De Giacomo, Y. Lespérance, and H. J. Levesque. On the semantics of deliberation in IndiGolog – From theory to implementation. *Annals of Mathematics and Artificial Intelligence*, 41(2–4):259–299, Aug. 2004.
25. S. Sardina and L. Padgham. Goals in the context of BDI plan failure and planning. In *Proc. of AAMAS*, pages 16–23, Hawaii, USA, May 2007. ACM Press.
26. M. Thielscher. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 5(4–5):533–565, 2005.
27. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative & procedural goals in intelligent agent systems. In *Pro. of KR*, pages 470–481, Toulouse, France, April 2002.