Suspending and Resuming Tasks in BDI Agents

John Thangarajah, James Harland
RMIT University
Melbourne, Australia
{johnt,james.harland}@rmit.edu.au

David Morley, Neil Yorke-Smith
Artificial Intelligence Center, SRI International
Menlo Park, CA, U.S.A.
{morley,nysmith}@ai.sri.com

ABSTRACT

Intelligent agents designed to work in complex, dynamic environments must respond robustly and flexibly to environmental and circumstantial changes. An agent must be capable of deliberating about appropriate courses of action, which may include reprioritising goals, aborting particular tasks, or scheduling tasks in a particular order. This paper investigates the incorporation of a mechanism to suspend and reconsider tasks within a BDI-style architecture. Such an ability provides an agent designer greater flexibility to direct agent operation, and it offers a generic means for handling conflicts between tasks. We investigate conditions under which a goal or a plan may be suspended, the process for suspending it, and the appropriate behaviours upon resumption. We give an operational semantics for suspending tasks in terms of the abstract agent language CAN, thus providing a general mechanism that can be incorporated into any BDI-based agent programming language.

Categories and Subject Descriptors

I.2.8 [**ARTIFICIAL INTELLIGENCE**]: Problem Solving, Control Methods, and Search—*Plan execution, formation, and generation*; I.2.11 [**ARTIFICIAL INTELLIGENCE**]: Distributed Artificial Intelligence—*Intelligent agents*

General Terms

Design, Reliability, Theory

Keywords

Agent reasoning::reasoning (single and multi-agent); Agent theories, models and architectures::formal models of agency

1. INTRODUCTION

A fundamental feature of intelligent agent systems is the ability to cope with complex and dynamic environments. In an unpredictable environment, agents must behave robustly and flexibly in order to achieve their goals. In agent architectures inspired by the classic *Belief-Desire-Intention* (BDI) model [14], behavioural properties are often characterized by the interactions between *beliefs*, *goals*, and *plans* [22]. In general, an agent that wishes to achieve a particular set of goals will pursue a number of plans concurrently. At regular intervals in the deliberation cycle, the agent will review its current choice of goals and plans. Its activity as a consequence may involve seeking alternative plans for a particular goal, re-scheduling goals to better comply with resource constraints, or suspending some goals until a more appropriate time.

Cite as: Suspending and Resuming Tasks in BDI Agents, John Thangarajah, James Harland, David Morley and Neil Yorke-Smith, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril,Portugal,pp.405-412.

Consider Alice, a knowledge worker aided by a personal assistive agent such as *CALO* [10]. Alice has in mind to attend the AAMAS conference later in the year, and her CALO agent adopts the goal of BookConferenceTravel (BCT) to assist her. A few days later, Alice also instructs CALO to purchase a laptop for her. Since Alice has limited funds, CALO is unable to perform both tasks for her successfully. In this situation, assuming the laptop purchase to be of lower priority, CALO may decide that the best course of action is to abort the task [17]. However, if more funding is expected in the near future, it is more sensible for CALO to *suspend* purchasing the laptop until additional funding is available.

Observe two points from this example. First, the BCT goal may have sub-goals and sub-plans (obtaining travel authorization, requesting travel quotes, registering for the conference, and so on), that will have to be suspended. However, as the request to suspend may come at an arbitrary point in the process of achieving it, the BCT goal may contain some sub-goals which have already been achieved (or aborted), some sub-goals which are not yet achieved, and possibly a set of concurrently executing plans at various stages of completion. Suspending this potentially complex operation will thus involve recursively suspending any sub-goals which have not been achieved, as well as determining what to do with plans which are to be suspended.

Second, the resumption of a goal after a period of suspension is not necessarily automatic. In the above example, when the additional funding is available, CALO will need to reconsider its options rather than immediately resume the purchasing of the laptop, since circumstances may have changed. For example, there now may be other goals with higher priority competing for the same resource, or Alice may no longer require the laptop. Hence, resuming a suspended goal is a matter of reconsidering the dynamic set of goals, rather than merely waiting for a trigger event to fire.

Whilst there are a number of agents systems that specifically incorporate goals, very few of these provide mechanisms for *suspending* goals. Systems that have some sort of mechanism for suspending goals or plans include Jadex [12], JAM [5] and Jason [6], and the architecture of Beaudoin [1].

Once adopted, goals in Jadex can be in one of the three states *option*, *active*, or *suspended*. Each goal has an associated context. When a goal is first adopted, it is in the option state. Once the context becomes true, it is then in the active state. If the context becomes invalid at some point, the goal moves to the suspended state. Once this occurs, all plans and sub-goals of the suspended goal are terminated. Once the context becomes *valid* again, the goal returns to the option state, which allows the agent to resume the goal whenever it decides it is appropriate. While this approach is similar in spirit to that proposed in this paper, we also provide mechanisms for the suspension and resumption of sub-goals, including default behaviours, which can be overridden if desired.

JAM allows goals to be suspended due to a higher level goal becoming applicable. Thus a goal g can be suspended in order to allow a more important goal to be pursued. Once the higher level goal is completed, the context of g is re-checked, and the goal resumes from the point of suspension. If the context check fails, then the goal is assumed to have failed and is hence dropped. In our approach, we allow a greater variety of options upon resumption, including continuing the suspension of the goal. This facilitates more sophisticated strategies to deliberate over goals, for instance to resolve conflicts or to exploit positive interactions.

Jason allows an agent designer to specify situations under which a goal will be suspended and resumed. This is achieved by the explicit `.suspend()` and `.resume()` constructs to provide the programmer with the requisite primitive tools. We present a fuller infrastructure, in that we provide mechanisms which manage the entire suspension and resumption process for both goals and plans.

The NML1 system of Beaudoin [1] contains management processes which can be used to suspend goals. However, the focus is on methodologies for design of intelligent systems, and hence Beaudoin deals at a design level. We are interested in programming techniques, specifically for BDI-style systems, and in providing appropriate mechanisms for incorporating suspension into a specific deliberation architecture.

In this paper we investigate how to incorporate the ability to suspend goals and plans into the agent execution cycle. Such an ability allows for more sophisticated reasoning over, and execution of, tasks, such as exemplified in the above example. We identify situations in which a task may be suspended (Section 2) and outline the process for suspending and resuming tasks (Section 3). We give an operational semantics for suspension of goals in the abstract agent language CAN [23, 15, 16] (Section 4); we thus provide a clear mechanism which can then be implemented in any of a number of practical systems such as JACK [22], Jason [6], or SPARK [8].

2. WHEN ARE TASKS SUSPENDED?

In this section we consider the circumstances under which an agent may decide to suspend a goal or plan. We will use the term *task* to refer to either a goal, a plan, or a primitive action. When a task is suspended, the agent attaches a *reconsideration condition*, as described in the next section, and possibly additional *meta information*, such as the anticipated estimated duration of suspension. When the reconsideration condition becomes true, the agent includes the suspended task in the next iteration of its goal deliberation. During that deliberation, it may decide to resume the task, to continue to suspend it (possibly on a different condition), to abort it, or to take other meta-action such as initiating another task.

It should be noted that throughout this paper, the goals we consider are *achievement* goals, i.e., goals which can be dropped once they have been achieved. This class of goals is in contrast with *maintenance* goals [2], which are required to be true over an extended and possibly infinite period of time.

Goals progress through a set of states during their lifecycle. When created, a candidate goal is in the state *initialized*. The agent may *adopt* the goal, and in due course *intend* it by forming an intention from it. An adopted or intended goal may be *suspended* (the focus of this paper), and a suspended goal may be *resumed* to its adopted or intended state. For achievement goals, the lifecycle concludes with the goal being *dropped*, or reaching one of the states *succeeded* or *failed*. The transitions of a goals between such states may be described as a finite state machine.[1] Plans, similarly, progress through different states.

As a running example, we will consider a user-assistive agent such as CALO in a scenario where the user has assigned two tasks, to be performed in collaboration between the human and the agent: to purchase a laptop and to book conference travel to AAMAS.

2.1 Reasons for Suspension

In examining some possible reasons that can motivate an agent to suspend a task, we do not impose that the agent *will necessarily* decide to suspend. The following are situations in which the agent's rational decision could be to suspend a task or tasks; it might decide to take other meta-actions, such as to abort a task.

Conflicts. First, resource conflicts: according to the approach of [20], if two goals are schedulable but have reusable resource requirements that conflict, then the agent suspends (by default) the goal g with the lower priority. This requires identifying the conflicting steps in the two goals, and suspending the relevant steps of g until the conflicting steps in the other goal are complete.

Example: The laptop purchase and conference travel tasks both require funds from a single account. If the agent cannot successfully execute both due to insufficient funds, it may suspend the laptop purchase task, believing the other task to be of higher priority.

Second, effect conflicts: if the effect of one task will violate the pre- or in-conditions of another task, then the agent may suspend one until the conflicting steps of the other are complete [18].

Example: Company policy prohibits submitting a travel authorization while another authorization is pending.

Positive interactions. If two goals have a common step (i.e., plan or sub-goal) then, when one goal reaches that step, the agent may suspend it until the other goal reaches the same step, in order to exploit the synergy of performing the step once for both goals [19]. The suspended task resumes when the other goal reaches this common step.

Example: The final step of both laptop purchase and conference travel tasks is to submit an invoice to division office.

Invalid context. An *applicable* plan for selection is one where the pre-conditions to the plan are true [22]. The *context* of a task governs when a goal or plan continues to be applicable. Should a task's context become false, the agent may suspend the task, and consider resuming it when the context becomes active again.

Example: The context of the conference travel goal include that it be safe to travel; if the government advises that it is not safe, CALO may suspend the goal until it is safe again.

No applicable plan. When there is no applicable plan in the plan library to achieve a goal g, then suspending g is an alternative to failing it, if the agent expects that a plan will become applicable, or if it has the capability of generating a plan [15]. g can be considered for resumption when a plan becomes applicable.

Example: The only plan for CALO to achieve the authorization necessary for travel — the first step of the BCT task — is with the Human Resources office. If the designated HR officer is not available (e.g., on holiday), then rather than failing the task, it can be suspended until the pre-condition of this plan becomes true.

Changing priorities. An agent's commitment to a task and the priority it gives to it may depend on dynamic properties, such as the

[1]Different agent frameworks exhibit variants in goal states and transitions [21, 4, 12]. For our purposes, the precise state evolution is secondary to the inclusion of a suspended state.

availability of resources or the actions of other agents. A change in mental attitude may result in a goal being suspended.

Example: The user, unsure which new laptop to purchase, hears that a new model will be released soon. Hence she suspends her purchase until she has investigated the new model.

Request from another agent. Whether the agent acts on the request to suspend a task depends on factors such as the relationship and authority of the two agents. (In the context of CALO, the user has complete authority.)

Example: User tells CALO to suspend travel arrangements.

2.2 Factors that Affect Commitment

In making rational decisions over task suspension and resumption, an important factor that guides the deliberation process is the commitment the agent has towards a particular goal. As stated, the criteria that may be used to determine this commitment are dynamic, including: goal utility, priority, and deadline; estimated cost of achievement, including estimated amount of resources required [9]; dependencies by other goals: internal dependencies that concern how many other goals depend on the successful completion of the goal, and external dependencies that concern how many other agents depend on it; interactions with other goals, both positive and negative [20, 19, 18]; the level of effort to date; and the estimated likelihood of success [11].

Elaborating the example, suppose the agent has a goal to book a flight as part of a top-level goal of booking conference travel. If cancelled, the penalty, apart from the local financial penalty, is that the conference attendance goal fails, and so forth. Similarly, suppose the goal is a delegated goal by another agent. If dropped, then any obligation to fulfill the goal may be violated.

3. MECHANISMS FOR SUSPENDING AND RESUMING TASKS

Having provided motivations for suspending tasks and illustrative situations when an agent may suspend a task, we now develop mechanisms for suspending and resuming tasks. These mechanisms will be given precise semantics in the next section.

We present *default mechanisms* designed to cover the most common situations, with no additional work required by the agent developer. However, for plans, we allow for the developer to provide a dedicated method to *suspend* a plan and a method to *resume* it. These methods are analogous to the *abort* and *failure* methods of [17]; formally, they are user-defined CAN programs that override the default behaviour. We assume that these methods do not fail and are not themselves suspended.

We allow tasks to be tagged with either *inactive* or *suspended*, both of which indicate that work on the task should not proceed; *suspended* also indicates that the task has been properly suspended. We modify the agent's execution cycle to respect these tags.

The mechanisms outlined below take precedence over the agent's normal steps in the execution cycle. That is, any meta-activity of suspension must occur before regular agent deliberation and action, including intention selection and plan execution.

To illustrate the mechanisms we use the example shown in Figure 1, where the top-level goal[2] is to Book Conference Travel (BCT). In the figure, goals, plans, actions, and waits are distinguished by the shape; completed (successful) tasks are indicated, and tasks currently executing are shown with bold outline.

[2]The top-level goal is the goal that a plan is achieving; it may be a sub-goal of another goal. Both top-level goals and sub-goals can be suspended.

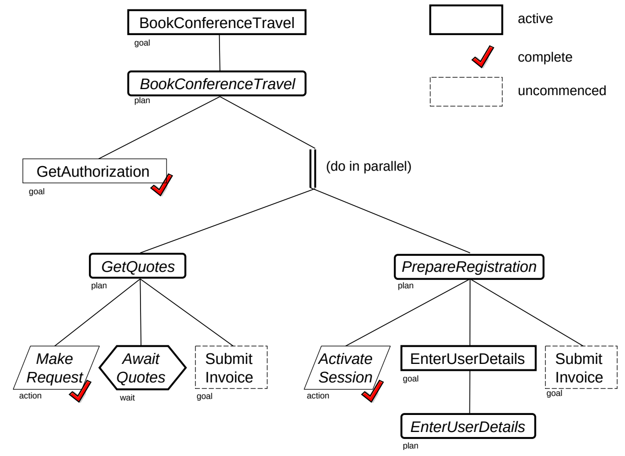

Figure 1: Book Conference Travel example.

3.1 Suspending Tasks

When a task is to be suspended, we first pause the task and all its currently active sub-tasks by tagging them *inactive* in a recursive manner. This first operation is important in order to stop the current steps in execution of the to-be-suspended task, until the agent deliberates and acts on the consequences of the suspension decision, which may include suspending, aborting, terminating, or continuing execution of different sub-tasks. Until this deliberation is complete, the lower level tasks should not execute; hence the necessity of pausing them.

The second operation is to suspend the task and its active children, again in a top-down recursive manner, by means of the mechanism explained below. The reason for performing the suspension top-down is because the suspend method of a parent task may, for instance, abort all its lower level tasks.

For example, if the BCT goal is to be suspended, all the tasks indicated as active in Figure 1 are first paused, and then the tasks are suspended top-down beginning with the BCT plan.

Recall that a task is either a goal, a plan, or an action. We assume that actions are atomic (although they may have some duration) and are suspended locally by a plan if necessary. Hence, suspending actions are handled when plans are suspended. Every plan for achieving an intention has a *parent* goal from which it arose.

Goals. First let us consider the case of suspending a goal. When the agent has determined that a particular goal is to be suspended, it takes the following steps:

- Mark the goal as suspended.
- Attach a *reconsideration condition* to the goal, specifying the conditions under which the agent may again consider pursuing the goal. (The default action on reconsideration is to resume the goal.) For example, if the BCT goal is suspended due to lack of funding, the goal may be reconsidered when funding increases.
- If the goal has not already begun execution — that is, it is in the current intention stack, but no plan instance to achieve the goal has been selected yet — then no further steps are required by default.
- Otherwise — that is, there is a plan instance in the agent's intention stack associated with the goal — then suspend that plan instance.

Plans. When the agent has determined that a plan is to be suspended (notably, as a consequence of suspending a goal, as just described), it takes the following steps:

- If the plan instance is the highest level task to be suspended, then attach a reconsideration condition as with goals above.
- If there is a dedicated suspend reasoning method attached to the plan, then call it. The suspend method may include:
  - The decision to abort the plan, continue execution of the plan, or suspend the plan as per the default method.

    For example, in suspending the PrepareRegistration plan, the agent might abort the plan if the task is simple enough to redo; run the plan to completion if the plan does not consume any resources and is not time consuming; or suspend the plan by following the steps described.
  - Procedures to release resources and perform other clean-up actions. For example, in suspending the GetQuotes plan the agent may have to inform the company's travel office that the conference travel is on hold.
- If the plan instance is still marked *inactive*, mark it as suspended and then suspend all current steps of the plan marked as inactive. There may be more than one step since some may be executing in parallel:
  - If the step is a sub-goal, suspend as a goal above.
  - If the step is an action (including any belief operations), it is allowed to complete as it is considered as atomic.
  - If the step is a wait, then make the wait inactive. For example, in Figure 1, WaitForQuotes is suspended by simply removing the wait, so that the agent is no longer waiting to receive quotes.

There is a potential optimization to this mechanism. When suspending a plan p, if p has not executed any of its steps as yet, then it suffices to simply *drop* the plan. To drop a plan means to remove it from the agent's intention stack; we do not fail the plan (as in plan failure), as this would potentially trigger the agent to retry alternate plans for the parent goal of p. We do not include this optimization in the coming operational semantics as it requires a significant change to the CAN language that we use to formalize the above mechanism. Note that, while increasing efficiency, this optimization may potentially change the behaviour of the agent in terms of plan selection; nonetheless, the top level behaviour of achieving the goal does not change.

3.2 Resuming Tasks

When a reconsideration condition attached to a suspended task becomes true, the default behaviour is to resume the task. That is, the agent will attempt to resume execution of the task from the point that is was suspended. However, as noted previously, upon reconsideration the agent may choose to abort the goal, drop the goal, select alternate plans to achieve the goal, or some other choice of behaviour. This decision will depend on the developer and the application domain, and if required may be implemented as an optional meta-procedure that overrides the default resumption behaviour.

Goals. When an agent wishes to resume a suspended goal, it takes the following steps:

- Unmark the goal, thereby allowing the goal to progress once any suspended plan instance for it is resumed.
- If there is a plan instance associated with the suspended goal, then resume that plan. For example, if the BCT goal is resumed then the BCT plan is resumed.
- Otherwise — when there is no plan currently selected — re-enable the procedure invocation. That is, allow the normal plan selection mechanisms for the goal to execute.

Plans. When an agent wishes to resume a suspended plan (notably, as a consequence of resuming a goal, as just described), it takes the following steps:

- Mark the plan as active.
- If there is a dedicated resume reasoning method attached to the plan, then call this method.
- If the plan has no dedicated resume method attached, perform the following default procedure, beginning by checking the in-conditions of the plan:
  - If the in-conditions are true:
    * Allow the next steps of the plan to proceed.
    * Resume any suspended sub-goals using the procedure for resuming goals above.
  - If the in-conditions are false:
    * Abort the plan.
    * Retry alternative plans if they exist for the top-level goal.[3]
- If the plan is not aborted, unmark the plan, thereby allowing the plan to progress once any suspended sub-goals are resumed, and then resume any suspended sub-goals using the procedure for resuming goals above.

The optional resume method attached to a plan can be used for special operations such as re-acquiring resources. As with suspend methods, a dedicated resume method is the means to override the default behaviour. For example, consider the suspended EUD plan in Figure 1. Assume that an in-condition for this plan is that there is an active web-session and that on resumption this condition is no longer true. If the default resume mechanism is followed, the agent will abort this plan and retry alternative plans for the goal EnterUserDetails. This will cause the goal to fail as no alternatives exist. Hence, it is desirable to have a resume method that re-starts the same plan in this situation.

Note that when a plan is resumed, we check for in-conditions but not for pre-conditions. Unlike in-conditions that are required to be true during the execution of the plan, pre-conditions are required to be true only at the point of plan selection. Once the plan is selected they are no longer relevant. Given that the plan choice has already occurred before the plan is suspended, by default we do not check for pre-conditions on resumption. Nevertheless, there may be situations where the pre-conditions need to be checked and re-established if necessary. If required, this reasoning must be encoded into the resume reasoning method of the plan.

4. OPERATIONAL SEMANTICS

In this section we provide a formal semantics for task suspension and resumption. We do this using the CAN [23, 15, 16] language; we provide an overview of CAN in Section 4.1.

Major changes to the CAN transition rules to accommodate new features can lead to unexpected consequences. Instead, we introduce a manageable, localised modification to CAN and implement suspension and resumption via a source transformation, in the style

[3]According to the agent's meta-decisions upon plan failure.

of [17]. This enables suspension and resumption to take place at arbitrary points in the relevant tasks, managed by assertion of certain predicates, which are then used to control the transformed plans in an appropriate manner, as we will describe. We provide a detailed description of the transformation process in Sect. 4.2, although for simplicity we do not include a mechanism for aborting goals nor for handling reconsideration conditions. In Sect. 4.3 we illustrate this approach using the Book Conference Travel example of Figure 1.

4.1 The CAN Language

We give our formal semantics in terms of CAN [23, 15, 16]. CAN is a high-level agent language, in a spirit similar to that of AgentSpeak [13] and Kinny's Ψ [7], both of which attempt to extract the essence of a class of implemented BDI agent systems. CAN provides an explicit goal construct that captures both the declarative and procedural aspects of a goal. Goals are persistent in CAN in that, when a plan fails, another applicable plan is attempted. This equates to the default failure handling mechanism typically found in implemented BDI systems such as JACK [22].

An agent's behaviour is specified by a *plan library*, denoted by Π, that consists of a collection of *plan clauses* of the form $e : c \leftarrow P$, where e is an event, c is a context condition (a logical formula over the agent's beliefs that must be true in order for the plan to be applicable) that can be omitted if $true$ and P is the plan body. The plan body is a *program* that is defined recursively as follows:

$$\begin{aligned} P ::= &\ act \mid +b \mid -b \mid ?\phi \mid !e \mid P_1; P_2 \mid P_1 \| P_2 \mid \mathsf{Goal}(\phi_s, P_1, \phi_f) \\ &\mid P_1 \rhd P_2 \mid (\!|\{\psi_1 : P_1, \ldots, \psi_n : P_n\}|\!) \mid nil \end{aligned}$$

where $P_1, \ldots, P_n$ are themselves programs, act is a primitive action that is not further specified, and $+b$ and $-b$ are operations to add and delete beliefs. The belief base contains ground belief atoms in the form of first-order relations but could be orthogonally extended to other logics. It is assumed that well-defined operations are provided to check whether a condition follows from a belief set ($B \models c$), to add a belief to a belief set ($B \cup \{b\}$), and to delete a belief from a belief set ($B \setminus \{b\}$). $?\phi$ is a test for condition ϕ. $!e$ is an event, typically an achievement goal, that is posted from within the program. The compound constructs are sequencing ($P_1; P_2$), parallel execution ($P_1 \| P_2$), a (finite) set of guarded plans $(\!|\{\psi_1 : P_1, \ldots, \psi_n : P_n\}|\!)$, $P_1 \rhd P_2$ which, executes P_1 and then P_2 only if P_1 has failed; and goals ($\mathsf{Goal}(\phi_s, P, \phi_f)$).

A summary of the operational semantics for CAN in line with [23] and following some of the simplifications of [15] is as follows. A *basic configuration* $S = \langle \mathcal{B}, P \rangle$ consists of the current belief base $\mathcal{B}$ of the agent and the current program P being executed (i.e., the current intention).

A transition $S_0 \longrightarrow S_1$ specifies that executing S_0 for a single step yields configuration S_1. $S_0 \longrightarrow^* S_n$ is the usual reflexive transitive closure of $\longrightarrow$: S_n is the result of one or more single-step transitions. A derivation rule $\dfrac{S' \longrightarrow S_r}{S \longrightarrow S'_r}$ consists of a (possibly empty) set of premises, which are transitions together with some auxiliary conditions (numerator), and a single transition conclusion derivable from these premises (denominator).

Figures 2 and 3 contain a summary of the important rules of CAN for our purposes. For further details about CAN and its features we refer to [23, 15, 16].

4.2 Suspending and Resuming Goals

Part of the power of our approach to suspending and resuming tasks is to allow for the existence of suspend and resume methods on plans. We represent these methods within CAN using programs that we associate with the plans during a syntactic transformation of the plan library. The syntactic transformation also attaches "wait-until" guards on components of the plans to allow pausing the plans when necessary.

A *wait-until* guard construct $\phi : P$, that does not execute P until ϕ becomes true, was added to CAN in [17]. In this paper, we utilize a slightly different version of this construct to ensure that the wait is passed though on transitions for task expansion and plan selection. We use these guards to prevent further execution of sub-tasks in a plan when the plan is suspended; when a guarded task is expanded to a collection of plans, we need to ensure that the guard is kept on the plan selection as well. Similarly, when a plan is selected, we want to make sure that on failure of that plan, the selection of a replacement plan is also guarded.

Our transition rules for the guard operator ':' are given in Figure 4. Intuitively, when ϕ is believed to be false, the guarded plan $\phi : P$ should continue to wait (rule *false*). Not only do we wish expansion of a guarded achievement goal event ϕ:!e into a collection of plan bodies to be delayed while ϕ is false, we also want the resultant plan selection to be delayed should ϕ become false before plan selection. Thus rule *Event* copies the guard on the event to a guard on the plan body collection. Similarly, we want plan re-selection

$$\frac{\Delta = \{\psi_i\theta : P_i\theta \mid e' : \psi_i \leftarrow P_i \in \Pi \wedge \theta = \mathsf{mgu}(e, e')\}}{\langle \mathcal{B}, !e \rangle \longrightarrow \langle \mathcal{B}, (\!|\Delta|\!) \rangle}\ Event$$

$$\frac{\psi_i : P_i \in \Delta \quad \mathcal{B} \models \psi_i}{\langle \mathcal{B}, (\!|\Delta|\!) \rangle \longrightarrow \langle \mathcal{B}, P_i \rhd (\!|\Delta \setminus \{\psi_i : P_i\}|\!) \rangle}\ Select$$

$$\frac{\langle \mathcal{B}, P_1 \rangle \not\longrightarrow}{\langle \mathcal{B}, (P_1 \rhd P_2) \rangle \longrightarrow \langle \mathcal{B}, P_2 \rangle}\ \rhd_{fail}$$

$$\frac{\langle \mathcal{B}, P_1 \rangle \longrightarrow \langle \mathcal{B}', P_1' \rangle}{\langle \mathcal{B}, (P_1; P_2) \rangle \longrightarrow \langle \mathcal{B}', (P'; P_2) \rangle}\ Sequence$$

$$\frac{\langle \mathcal{B}, P_1 \rangle \longrightarrow \langle \mathcal{B}', P' \rangle}{\langle \mathcal{B}, (P_1 \| P_2) \rangle \longrightarrow \langle \mathcal{B}', (P' \| P_2) \rangle}\ Parallel_1$$

$$\frac{\langle \mathcal{B}, P_2 \rangle \longrightarrow \langle \mathcal{B}', P' \rangle}{\langle \mathcal{B}, (P_1 \| P_2) \rangle \longrightarrow \langle \mathcal{B}', (P' \| P_1) \rangle}\ Parallel_2$$

Figure 2: Operational rules of CAN.

$$\frac{\mathcal{B} \models \phi_s}{\langle \mathcal{B}, \mathsf{Goal}(\phi_s, P, \phi_f) \rangle \longrightarrow \langle \mathcal{B}, true \rangle}\ \mathsf{G}_s$$

$$\frac{\mathcal{B} \models \phi_f}{\langle \mathcal{B}, \mathsf{Goal}(\phi_s, P, \phi_f) \rangle \longrightarrow \langle \mathcal{B}, fail \rangle}\ \mathsf{G}_f$$

$$\frac{P = \mathsf{Goal}(\phi_s, P', \phi_f) \quad P' \neq P_1 \rhd P_2 \quad \mathcal{B} \not\models \phi_s \vee \phi_f}{\langle \mathcal{B}, P \rangle \longrightarrow \langle \mathcal{B}, \mathsf{Goal}(\phi_s, P' \rhd P', \phi_f) \rangle}\ \mathsf{G}_I$$

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

$$\frac{P = P_1 \rhd P_2 \quad \mathcal{B} \not\models \phi_s \vee \phi_f \quad P_1 \in \{true, fail\}}{\langle \mathcal{B}, \mathsf{Goal}(\phi_s, P, \phi_f) \rangle \longrightarrow \langle \mathcal{B}, \mathsf{Goal}(\phi_s, P_2 \rhd P_2, \phi_f) \rangle}\ \mathsf{G}_R$$

Figure 3: Rules for goals in CAN.

$$\frac{\mathcal{B} \not\models \phi}{\langle \mathcal{B}, (\phi{:}P) \rangle \longrightarrow \langle \mathcal{B}, (\phi{:}P) \rangle} \; :false \qquad \frac{\Delta = \{\psi_i\theta : P_i\theta \mid e' : \psi_i \leftarrow P_i \in \Pi \wedge \theta = \mathsf{mgu}(e, e')\} \quad \mathcal{B} \models \phi}{\langle \mathcal{B}, \phi{:}!e \rangle \longrightarrow \langle \mathcal{B}, \phi{:}(\!|\Delta|\!) \rangle} \; :Event$$

$$\frac{\psi_i : P_i \in \Delta \quad \mathcal{B} \models \psi_i \quad \mathcal{B} \models \phi}{\langle \mathcal{B}, \phi{:}(\!|\Delta|\!) \rangle \longrightarrow \langle \mathcal{B}, P_i \rhd \phi{:}(\!|\Delta \setminus \{\psi_i : P_i\}|\!) \rangle} \; :Select \qquad \frac{P \neq !e \quad P \neq (\!|\Delta|\!) \quad \mathcal{B} \models \phi}{\langle \mathcal{B}, (\phi{:}P \rangle \longrightarrow \langle \mathcal{B}, P \rangle} \; :true$$

Figure 4: Rules for the modified guard construct.

on failure to be delayed should ϕ become false, thus requiring rule *Select* to preserve the guard on the plan body collection. In all other cases, we drop the guard when it is satisfied (rule *true*).

The identification of which components to pause is achieved by introducing *context* parameters to tasks. We translate each task $!e$ into a task $!e(v)$ with an extra parameter v that specifies the context (i.e., the identity) of the particular instance of that task. That is, two distinct occurrences of the task $!e$ that are being executed concurrently would have two distinct values for the context parameter. Each time a plan calls for the execution of a sub-task, we generate a new context to pass in as the context parameter for that sub-task.

We use a very simple scheme for creating contexts. Each context is a list of identifiers; creating a new context value for execution of a sub-goal amounts to prepending a new identifier n onto the start of the existing context v to obtain $n.v$. We refer to the *parent context* v^* of a context v. The parent $(n.v)^*$ of $n.v$ is v, i.e., the tail of $n.v$.

For the purposes of examples in this paper, we will use associate each occurrence of an event/sub-goal in the plan library with a single digit and use strings of digits as contexts. For example, executing sub-goal number 3 in context 27 gives subcontext 327. The parent context of 327 is 27 (written as $327^* = 27$). We will denote the root context (equivalent to the empty string of digits) by $\perp$. Thus we have $27^* = 7$ and $7^* = \perp$.

We keep track of information about the state of execution of a task with context v using the following belief predicates:

- h_v: asserted into the belief set to suspend the execution of the task with context v. This has the effect of immediately pausing execution on that task and all its sub-tasks and initiating the execution of the suspension methods. To resume the execution, this predicate is removed from the belief set.

- c_v: asserted into the belief set by a suspension method to activate the sub-task with context v, allowing it to continue executing to completion. This should only be set by the parent task[4].

- s_v: asserted after the completion of the suspension method for the task with context v to indicate that the suspension method has been executed and the task is in a suspended (as opposed to merely paused) state. This predicate is removed from the belief set only once the resume method has been executed.

With the ability to track task execution state, we can define the following predicates:

- $H_v = h_v \vee (H_{v^*} \wedge \neg c_v)$: execution of the task with context v should be paused. Either this task has been explicitly marked as paused, or the parent should be paused and this task has not been marked as able to proceed regardless.

[4]In this presentation of a suspension/resumption mechanism, the simplest non-default operation on sub-tasks is task resumption. Other behaviors, such as aborting require introducing the complexity of the semantics of abort into the presentation.

- $a_v = \neg H_v \wedge \neg s_v$: the task v is active, that is it should not be paused and it is not currently suspended. When we translate the plans for a task with context v, we place wait guards that wait on a_v on each of the basic components, to prevent them from executing when we want to suspend v.

In parallel with executing a plan for a task with context v, we must also execute a program that will invoke the suspend and resume methods as necessary, as described informally in Section 3.

We need to run the suspend method represented by program P^s when we find that v is paused or when an ancestor is paused and the suspend methods down to the the parent of v have been executed. (We need to wait until the parent suspend method is executed in case that method decides to allow v to execute to completion by asserting c_v and thereby prevent the suspension of the task.) This condition is expressed as $h_v \vee (H_v \wedge s_{v^*})$. After the suspend method is executed, we add s_v to the belief set, allowing suspend methods for children to execute as needed.

To initiate the resumption of a suspended task with context v, the h_v fact is removed from the belief set. This immediately causes $H_{v'}$ to become false for v' being v and its descendants. These tasks do not start executing immediately, since $s_{v'}$ is still true.

We run the resume method, P^r, of a suspended task, v, when we find that H_v is no longer true and its parent's (v^*) resume method has been executed. The completion of the parent's resume method is indicated by the removal of the fact s_{v^*}. Thus the condition for executing P^r is expressed as $\neg H_v \wedge \neg s_{v^*}$. After the resume method is executed, we remove s_v from the belief set, allowing the resume methods for children to execute as needed. This also causes a_v to become true, allowing execution of the task to continue to the extent that it can until any sub-tasks it may be waiting on are resumed.

Thus the program to suspend and resume v can be defined as

$$\pi(v, P^s, P^r) = \\ (h_v \vee (H_v \wedge s_{v^*})){:}P^s; +s_v; (\neg H_v \wedge \neg s_{v^*}){:}P^r; -s_v$$

Let P be a plan body transformed to include contexts and appropriate wait guards (see below). Let us introduce the notation $\mathit{while}(P, q_v, \pi(v, P^s, P^r))$ as an abbreviation for

$$+q_v; ((P; -q_v \rhd -q_v; ?\mathit{false}) \parallel \mathsf{Goal}(\neg q_v, \pi(v, P^s, P^r), \mathit{false}))$$

This will assert q_v (a new proposition denoting that P is executing) into the belief set and start executing P. Whether P succeeds or fails, q_v will be removed from the belief set after it terminates. After q_v is added to the belief set, the Goal construct will repeatedly execute $\pi(v, P^s, P^r)$ until q_v is removed from the belief set, i.e., until P has completed.

We are now in a position to describe the syntactic transformation of the plans in the plan library. Given a plan clause of the form $e : c \leftarrow P$ with a suspend method P^s (default nil) and a resume method P^r (default nil) for e, we introduce a new context variable v and create a replacement plan clause:

$$e(v) : c \leftarrow \mathit{while}(\mu_v(P), q_v, \pi(v, P^s, P^r))$$

where the plan context condition c is unchanged and

$\mu_v(nil) = nil$
$\mu_v(P_1; P_2) = \mu_v(P_1); \mu_v(P_2)$
$\mu_v(P_1 \triangleright P_2) = \mu_v(P_1) \triangleright \mu_v(P_2)$
$\mu_v(\llparenthesis \psi_1 : P_1, \ldots, \psi_n : P_n \rrparenthesis) = \llparenthesis \psi_1 : \mu_v(P_1), \ldots, \psi_n : \mu_v(P_n) \rrparenthesis$
$\mu_v(P_1 \parallel P_2) = \mu_v(P_1) \parallel \mu_v(P_2)$
$\mu_v(\phi{:}P) = (a_v \wedge \phi){:}\mu_v(P)$
$\mu_v(\mathsf{Goal}(\phi_s, P, \phi_f)) = \mathsf{Goal}(a_v \wedge \phi_s, \mu_v(P), a_v \wedge \phi_f)$
$\mu_v(!e) = a_{n.v}{:}!e(n.v)$ for the n-th sub-goal in the library
$\mu_v(act) = a_v{:}act$
$\mu_v(+b) = a_v{:}{+b}$
$\mu_v(-b) = a_v{:}{-b}$

The first rule states that the mapping has no effect on the null program nil. The next four rules simply apply the mapping to the arguments of the operators. For the guard construct, note that it is not simply a matter of waiting for the guard to become true — we must have that the task is active. Hence we add the condition a_v to the guard (sixth rule). Similar remarks apply to the Goal construct (seventh rule), in the goal can only terminate if it is active. For sub-tasks, we create a new context $n.v$ for the sub-task. For this and the last three rules we ensure that progress only occurs if the plan (or the sub-task) is active.

4.3 Example

4.3.1 Transformation of Plan Library

Consider plan clauses corresponding to the example of Figure 1.

The first plan clause $BCT :\leftarrow !GA; (!GQ \parallel !PR)$ with neither suspend nor resume methods becomes:

$BCT(v) :\leftarrow while(P_v^{BCT}, q_v, \pi(v, nil, nil))$ where
$P_v^{BCT} = a_{0.v}{:}!GA(0.v); (a_{1.v}{:}!GQ(1.v) \parallel a_{2.v}!PR(2.v))$

Note that we have $\pi(v, nil, nil)$ as there is neither a suspend method nor a resume method for BCT. Note also that P_v^{BCT} is derived from the rule for BCT by adding a guard to each part.

The second plan clause $GQ :\leftarrow MR; AQ{:}nil; !SI$ with a suspend method consisting of the goal InformRequestSuspended, $!IRS$, and a resume method consisting of the goal InformRequestResumed, $!IRR$, becomes:

$GQ(v) :\leftarrow while(P_v^{GQ}, q_v, \pi(v, !IRS(v), !IRR(v)))$ where
$P_v^{GQ} = a_v{:}MR; (a_v \wedge AQ){:}nil; a_{3.v}{:}!SI(3.v)$

Note the difference between the context used in the guard conditions for the event $!SI$ and the actions MR and nil.

Finally, the plan clause $PR :\leftarrow AS; !EUD; !SI$ with a resume method consisting of the goal ReactivateSession, $!RS$ becomes:

$PR(v) :\leftarrow while(P_v^{PR}, q_v, \pi(v, nil, !RS(v))$ where
$P_v^{PR} = a_v{:}AS; a_{4.v}{:}!EUD(4.v); a_{5.v}{:}!SI(5.v)$ Note that until some $h_{v'}$ or $s_{v'}$ is in the belief set, a_v will be true for every v. Thus the $a_v{:}P$ guards do not prevent the execution of the guarded programs P until one of these beliefs is added.

4.3.2 Goal Expansion Transitions

Now suppose we have expanded and partially executed the goal $!BCT(0)$ working towards the situation shown in Figure 1. At some point we will have reached an intermediate state where we are ready to execute

$$P_0^{BCT} = a_{00}{:}!GA(00); (a_{10}{:}!GQ(10) \parallel a_{20}!PR(20)).$$

After successfully executing $a_{00}{:}!GA(00)$, we can transition $a_{10}{:}!GQ(10)$ to $a_{10}{:}\llparenthesis\{P\}\rrparenthesis$ where

$$P = while(P_{10}^{GQ}, q_{10}, \pi(10, !IRS(10), !IRR(10))).$$

This in turn transitions into $P \triangleright a_{10}{:}\llparenthesis\{\}\rrparenthesis$. Within P we have

$$P_{10}^{GQ} = a_{10}{:}MR; (a_{10} \wedge AQ){:}nil; a_{310}{:}!SI(310).$$

After successfully executing $a_{10}{:}MR$, we will be left with

$$(a_{10} \wedge AQ){:}nil; a_{310}{:}!SI(310).$$

In the same way, we will have within the other parallel branch

$$a_{420}{:}!EUD(420); a_{520}{:}!SI(520)$$

4.3.3 Suspension

Suppose h_0 is added to the beliefs, indicating that we want to suspend the top-level goal $!BCT(0)$. This causes H_0 to become true, and since $10^* = 20^* = 0$, H_{10} and H_{20} also become true. Similarly, H_{420} becomes true.

Looking at the AwaitQuotes wait, since H_{10} is true, a_{10} is false. Thus the condition $(a_{10} \wedge AQ)$ will not become true even if the arrival of the quotes causes AQ to become true. Thus we have made the wait inactive.

Similarly, even if execution of $!EUD(420)$ has started, by a_{420} becoming false, its execution will be paused.

We now have all the sub-goals of $!BCT(0)$ paused, but the suspend methods have not been executed.

Consider the suspend and resume methods that are now running in parallel branches thanks to the *while* construct. For the top-level goal $!BCT(0)$ we have

$$\pi(0, nil, nil) =$$
$$(h_0 \vee (H_0 \wedge s_\perp)){:}nil; +s_0; (\neg H_0 \wedge \neg s_\perp){:}nil; -s_0$$

Since h_0 is now true, the empty suspend method nil is executed and the belief s_0 is asserted. This program goes no further thanks to the guard condition $\neg H_0 \wedge \neg s_\perp$. The assertion of s_0 causes the initial guard condition within

$$\pi(20, nil, !RS(20) =$$
$$(h_{20} \vee (H_{20} \wedge s_0)){:}nil; +s_{20}; (\neg H_{20} \wedge \neg s_0){:}!RS(20); -s_{20}$$

to become true. This leads to the execution of the empty suspend method nil for the sub-goal $!PR(20)$ and the assertion of s_{20}. A similar process happens for the sub-goal $!GQ(10)$. By this mechanism, the suspend methods are executed first for the top-level goal and then progressively down the sub-goals, with appropriate s_v beliefs being added.

4.3.4 Resumption

Now let us look at how resumption works. To resume $!BCT(0)$ we delete the belief h_0. This immediately causes H_v to become false for $v = 0$ and any descendant. However, after the suspend methods for $v = 0$ and descendants were executed, s_v was added to the belief set. Thus a_v is still false for each of these v and ordinary execution is still suspended.

For $BCT(0)$, the guard condition $(\neg H_0 \wedge \neg s_\perp)$ now holds and the (empty) resume method is executed. The condition s_0 is then deleted from the belief set. This leads to the guard condition $(\neg H_{20} \wedge \neg s_0)$ becoming true and the resume method $!IRR(20)$ for $!PR(20)$ being executed and s_{20} being removed from the beliefs. In this way the resume methods are executed top-down and the conditions a_0, a_{10}, a_{20}, etc., are made true again, enabling execution of the original goal/sub-goals to proceed.

4.3.5 Selective Sub-Goal Reactivation

Suppose when suspending the PrepareRegistration sub-goal there is some reason why we want not to suspend the EnterUserDetails sub-goal but instead to continue it to completion. We can achieve this by making the suspend method for $!PR(v)$ be $+c_{4.v}$. When we execute this method for $!PR(20)$ we will add the belief c_{420} before adding s_{20}. Since $H_{420} = h_{420} \vee (H_{20} \wedge \neg c_{420})$, this has the effect of causing H_{420} to become false and a_{420} true, thereby reactivating $!EUD(420)$. To avoid leaving the c_{420} belief around when $!PR(20)$ resumes, the resume method should delete that belief.

5. DISCUSSION

Intelligent agents designed to work in complex, dynamic environments must respond robustly and flexibly to environmental and circumstantial changes. One response in reconsidering the current course of action is suspension of one of its tasks until a more appropriate time. This paper has investigated conditions under which a goal or a plan may be suspended and the appropriate mechanisms for suspending and resuming. We have provided an operational semantics for suspending tasks in terms of the abstract agent language CAN. Goals that are suspended are reconsidered when appropriate and are possibly resumed, according to the general mechanism for resuming tasks also described.

We have considered achievement goals, with the assumption that the agent will not attempt to suspend a task that is being aborted, nor already suspended (and likewise not attempt to resume a task being aborted, nor one not suspended). Our operational semantics does not specify the behaviour on resuming a goal that has pre- or in-conditions attached. Upon resuming a task, these conditions should be checked. If they are no longer true, the agent has two courses of action: abort the goal[5], or continue to suspend the goal and adopt an intention to attempt to make the pre- and in-conditions true once again. Hayashi et al. [3] describe a means of tracking the conditions of a hierarchy of suspended goals, and the necessary actions to re-establish their feasibility upon resumption.

The mechanisms for suspending tasks complements those that handle failure and aborting of tasks [17]. The key aspects that distinguish failure, aborting, and suspending of tasks are as follows. Failure occurs locally at the current lowest level of execution, and the failure is propagated upwards in a bottom-up manner. Failure is typically unintentional, whereas aborting and suspending are deliberate, the result of some high level deliberation of the agent. The command to abort or suspend can be made at any arbitrary point in the execution hierarchy. Before a task is aborted all its sub-tasks need to be aborted; that is, a bottom-up approach is taken. When a task is suspended, by contrast, the approach is top-down where the suspend method of the task is first executed and then if appropriate[6] the children are suspended.

The current semantics that we have provided does not allow for aborting tasks. This is to avoid the complexity of both suspending and aborting. The semantics also does not allow for attaching a reconsideration condition when a task is suspended, and we have made the assumption that suspend and resume methods do not fail. Further investigation is required to determine an appropriate representation of these conditions and extend the semantics accordingly.

A main thrust of our future work is to implement the suspension mechanism in the SPARK [8] agent system. On the conceptual side, we are interested in examining the interaction between suspending and aborting tasks, and extending our operational semantics to the case of maintenance goals.

Acknowledgements. We thank Lin Padgham and the anonymous reviewers for their comments. The first author acknowledges the support of the Australian Research Council and Agent Oriented Software under grant LP0453486. The work of the two authors at SRI International was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of DARPA or the Department of Interior-National Business Center.

[5]Note it is more appropriate to abort the goal, not merely drop it as in Jadex [12], because there may be clean up actions that need to be performed.

[6]For example, if the top level goal is dropped then the sub-tasks are also dropped and not suspended.

6. REFERENCES

[1] L. Beaudoin. *Goal Processing in Autonomous Agents*. PhD thesis, School of Computer Science, University of Birmingham, 1994.

[2] S. Duff, J. Harland, and J. Thangarajah. On proactivity and maintenance goals. In *Proc. of AAMAS'06*, pages 1033–1040, 2006.

[3] H. Hayashi, S. Tokura, F. Ozaki, and T. Hasegawa. On-line interruption planning using Dynagent: Integrating deliberation and emergency deliberation. In *Proc. of ICAPS'07 Workshop on Moving Planning and Scheduling Systems into the Real World*, 2007.

[4] K. Hindriks, F. de Boer, W. van der Hoek, and J.-J. Meyer. Formal semantics for an abstract agent programming language. In *Proc. of ATAL'98*, pages 215–229, 1998.

[5] M. Huber. JAM: A BDI-theoretic mobile agent architecture. In *Proc. of AGENTS'99*, pages 236–243, 1999.

[6] J. F. Hübner, R. H. Bordini, and M. Wooldridge. Programming declarative goals using plan patterns. In *Proc. of DALT'06*, 2006.

[7] D. Kinny. The Psi calculus: an algebraic agent language. In *Proc. of ATAL'01*, pages 32–50, 2001.

[8] D. Morley and K. Myers. The SPARK agent framework. In *Proc. of AAMAS'04*, pages 714–721, 2004.

[9] D. Morley, K. L. Myers, and N. Yorke-Smith. Continuous refinement of agent resource estimates. In *Proc. of AAMAS'06*, 2006.

[10] K. Myers, P. Berry, J. Blythe, K. Conley, M. Gervasio, D. McGuinness, D. Morley, A. Pfeffer, M. Pollack, and M. Tambe. An intelligent personal assistant for task and time management. *AI Magazine*, 28(2):47–61, 2007.

[11] A. Pfeffer. Functional specification of probabilistic process models. In *Proc. of AAAI-05*, pages 663–669, 2005.

[12] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In *Multi-Agent Programming*. Springer, 2005.

[13] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proc. of Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, 1996.

[14] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In *Proc. of KR'92*, pages 439–449, 1992.

[15] S. Sardiña, L. de Silva, and L. Padgham. Hierarchical planning in BDI agent programming languages: a formal approach. In *Proc. of AAMAS'06*, pages 1001–1008, 2006.

[16] S. Sardiña and L. Padgham. Goals in the context of BDI plan failure and planning. In *Proc. of AAMAS'07*, pages 16–23, 2007.

[17] J. Thangarajah, J. Harland, D. Morley, and N. Yorke-Smith. Aborting tasks in BDI agents. In *Proc. of AAMAS'07*, pages 8–15, 2007.

[18] J. Thangarajah, L. Padgham, and M. Winikoff. Detecting and avoiding interference between goals in intelligent agents. In *Proc. of IJCAI'03*, pages 721–726, 2003.

[19] J. Thangarajah, L. Padgham, and M. Winikoff. Detecting and exploiting positive goal interaction in intelligent agents. In *Proc. of AAMAS'03*, pages 401–408, 2003.

[20] J. Thangarajah, M. Winikoff, L. Padgham, and K. Fischer. Avoiding resource conflicts in intelligent agents. In *Proc. of ECAI-02*, 2002.

[21] B. van Riemsdijk, M. Dastani, and M. Winikoff. Goals in agent systems: A unifying framework. In *Proc. of AAMAS'08*, 2008.

[22] M. Winikoff. JACK intelligent agents: An industrial strength platform. In *Multi-Agent Programming*. Springer, 2005.

[23] M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative and procedural goals in intelligent agent systems. In *Proc. of KR'02*, pages 470–481, 2002.