Aborting Tasks in BDI Agents

John Thangarajah, James Harland
RMIT University
Melbourne, Australia
{johthan,jah}@cs.rmit.edu.au

David Morley, Neil Yorke-Smith
Artificial Intelligence Center, SRI International
Menlo Park, CA 94025 U.S.A.
{morley,nysmith}@ai.sri.com

ABSTRACT

Intelligent agents that are intended to work in dynamic environments must be able to gracefully handle unsuccessful tasks and plans. In addition, such agents should be able to make rational decisions about an appropriate course of action, which may include aborting a task or plan, either as a result of the agent's own deliberations, or potentially at the request of another agent. In this paper we investigate the incorporation of aborts into a BDI-style architecture. We discuss some conditions under which aborting a task or plan is appropriate, and how to determine the consequences of such a decision. We augment each plan with an optional *abort-method*, analogous to the failure method found in some agent programming languages. We provide an operational semantics for the execution cycle in the presence of aborts in the abstract agent language CAN, which enables us to specify a BDI-based execution model without limiting our attention to a particular agent system (such as JACK, Jadex, Jason, or SPARK). A key technical challenge we address is the presence of parallel execution threads and of sub-tasks, which require the agent to ensure that the abort methods for each plan are carried out in an appropriate sequence.

Categories and Subject Descriptors

I.2.11 [**ARTIFICIAL INTELLIGENCE**]: Distributed Artificial Intelligence—*Intelligent agents*

General Terms

Design, Reliability, Theory

Keywords

Agents:: Architectures: reactive and deliberative; Agents:: formal models of agency

1. INTRODUCTION

Intelligent agents generally work in complex, dynamic environments, such as air traffic control or robot navigation, in which the success of any particular action or plan cannot be guaranteed [13]. Accordingly, dealing with failure is fundamental to agent programming, and is an important element of agent characteristics such as robustness, flexibility, and persistence [21].

In agent architectures inspired by the *Belief-Desire-Intention* (BDI) model [16], these properties are often characterized by the interactions between *beliefs*, *goals*, and *plans* [2].[1] In general, an agent that wishes to achieve a particular set of tasks will pursue a number of plans concurrently. When failures occur, the choice of plans will be reviewed. This may involve seeking alternative plans for a particular task, re-scheduling tasks to better comply with resource constraints, dropping some tasks, or some other decision that will increase the likelihood of success [12, 14]. Failures can occur for a number of reasons, and it is often not possible to predict these in advance, either because of the complexity of the system or because changes in the environment invalidate some earlier decisions. Given this need for deliberation about failed tasks or plans, failure deliberation is commonly built into the agent's execution cycle.

Besides dealing with failure, an important capability of an intelligent agent is to be able to abort a particular task or plan. This decision may be due to an internal deliberation (such as the agent believing the task can no longer be achieved, or that some conflicting task now has a higher priority) or due to an external factor (such as another agent altering a commitment, or a change in the environment).

Aborting a task or plan is distinct from its failure. Failure reflects an inability to perform and does not negate the "need" to perform — for example, a reasonable response to failure may be to try again. In contrast, aborting says nothing about the ability to perform; it merely eliminates the need. Failure propagates from the bottom up, whereas aborting propagates from the top down. The potential for concurrently executing sub-plans introduces different complexities for aborting and failure. For aborting, it means that multiple concurrent sub-plans may need to be aborted as the abort is propagated down. For failure, it means that parallel-sibling plans may need to be aborted as the failure is propagated up.

There has been a considerable amount of work on plan failures (such as detecting and resolving resource conflicts [20, 10]) and most agent systems incorporate some notion of failure handling. However, there has been relatively little work on the development of abort techniques beyond simple dropping of currently intended plans and tasks, which does not deal with the clean-up required. As one consequence, most agent systems are quite limited in their treatment of the situation where one branch of a parallel construct

[1] One can consider both *tasks* to be performed and *goals* to achieve a certain state of the world. A task can be considered a goal of achieving the state of "the task having been performed", and a goal can be considered a task of bringing about that state of the world. We adopt the latter view and use "task" to also refer to goals.

AAMAS'07 May 14–18 2007, Honolulu, Hawai'i, USA.

978-81-904262-7-5 (RPS) ©2007 IFAAMAS

fails (common approaches include either letting the other branch run to completion unhindered or dropping it completely).

In this paper we discuss in detail the incorporation of abort clean-up methods into the agent execution cycle, providing a unified approach to failure and abort. A key feature of our procedure-based approach is that we allow each plan to execute some particular code on a failure and on an abort. This allows a plan to attempt to ensure a stable, known state, and possibly to recover some resources and otherwise clean up before exiting. Accordingly, a central technical challenge is to manage the orderly execution of the appropriate clean-up code. We show how aborts can be smoothly introduced into a BDI-style architecture, and for the first time we give an operational semantics for aborting in the abstract agent language CAN [23, 17]. This allows us to specify an appropriate level of detail for the execution model, without focusing on the specific constructs of any one agent system such as JACK [2], Jadex [14], Jason [6], or SPARK [9]. Our focus is on a single agent, complementary to related work that considers exception handling for single- and multi-agent systems (e.g., [22, 5, 6]).

This paper is organized as follows. In Section 2 we give an example of the consequences of aborting a task, and in Section 3 we discuss some circumstances under which aborts should occur, and the appropriate representation and invocation procedures. In Section 4 we show how we can use CAN to formally specify the behaviour of an aborted plan. Section 5 discusses related work, and in Section 6 we present our conclusions and future work.

2. MOTIVATING EXAMPLE

Alice is a knowledge worker assisted by a learning, personal assistive agent such as CALO [11]. Alice plans to attend the IJCAI conference later in the year, and her CALO agent adopts the task of Support Meeting Submission (SMS) to assist her. CALO's plan for an SMS task in the case of a conference submission consists of the following sub-tasks:

1. Allocate a Paper Number (APN) to be used for administrative purposes in the company.
2. Track Writing Abstract (TWA): keep track of Alice's progress in preparing an abstract.
3. Apply For Clearance (AFC) for publication from Alice's manager based on the abstract and conference details.
4. Track Writing Paper (TWP): keep track of Alice's progress in writing the paper.
5. Handle Paper Submission (HPS): follow company internal procedures for submitting a paper to a conference.

These steps must be performed in order, with the exception of steps 3 (AFC) and 4 (TWP), which may be performed in parallel.

Similarly, CALO can perform the task Apply For Clearance (AFC) by a plan consisting of:

1. Send Clearance Request (SCR) to Alice's manager.
2. Wait For Response (WFR) from the manager.
3. Confirm that the response was positive, and fail otherwise.

Now suppose that a change in circumstances causes Alice to reconsider her travel plans while she is writing the paper. Alice will no longer be able to attend IJCAI. She therefore instructs her CALO agent to abort the SMS task. Aborting the task implies aborting both the SMS plan and the AFC subplan. Aborting the first plan requires CALO to notify the paper number registry that the allocated paper number is obsolete, which it can achieve by the Cancel Paper Number (CPN) task.[2] Aborting the second plan requires CALO to notify Alice's manager that Alice no longer requires clearance for publication, which CALO can achieve by invoking the Cancel Clearance Request (CCR) task.

We note a number of important observations from the example. First, the decision to abort a particular course of action can come from the internal deliberations of the agent (such as reasoning about priorities in a conflict over resources), or from external sources (such as another agent cancelling a commitment), as in this example. In this paper we only touch on the problem of determining *whether* a task or plan should be aborted, instead concentrating on determining the appropriate actions once this decision is made. Hence, our objective is to determine how to incorporate aborting mechanisms into the standard execution cycle rather than determine what should be aborted and when.

Second, once the decision is made to abort the attempt to submit a paper, there are some actions the agent should take, such as cancelling the clearance request. In other words, aborting a task is not simply a matter of dropping the task and associated active plans: there are some "clean up" actions that may need to be done. This is similar to the case for failure, in that there may also be actions to take when a task or plan fails. In both cases, note that it is not simply a matter of the agent "undo-ing" its actions to date; indeed, this may be neither possible (since the agent acts in a situated world and its actions change world state) nor desirable (depending on the semantics of the task). Rather, cleaning up involves compensation via *forward recovery* actions [3].

Third, there is a distinction between aborting a task and aborting a plan. In the former case, it is clear that all plans being executed to perform the task should be aborted; in the latter case, it may be that there are better alternatives to the current plan and one of these should be attempted. Hence, plan aborting or failure does not necessarily lead to task aborting or failure.

Fourth, given that tasks may contain sub-tasks, which may contain further sub-tasks, it is necessary for a parent task to wait until its children have finished their abort methods. This is the source of one of the technical challenges that we address: determining the precise sequence of actions once a parent task or plan is aborted.

3. ABORTING TASKS AND PLANS

As we have alluded to, failure and aborting are related concepts. They both cause the execution of existing plans to cease and, consequentially, the agent to reflect over its current tasks and intentions. Failure and aborting, however, differ in the way they arise. In the case of failure, the trigger to cease execution of a task or plan comes from below, that is, the failure of sub-tasks or lower-level plans. In the case of aborting, the trigger comes from above, that is, the tasks and the parent plans that initiated a plan.

In BDI-style systems such as JACK and SPARK, an agent's domain knowledge includes a pre-defined plan library of *plan clauses*. Each plan clause has a *plan body*, which is a *program* (i.e., combination of primitive actions, sub-tasks, etc.) that can be executed in response to a task or other event should the plan clause's *context condition* be satisfied. The agent selects and executes instances of plan clauses to perform its tasks. There can be more than one applicable plan clause and, in the event that one fails, another applicable one may be attempted. Plans may have sub-tasks that must succeed

[2]CALO needs only drop the TWA and TWP tasks to abort them: for the sake of simplicity we suppose no explicit clean up of its internal state is necessary.

for the plan to succeed. In such systems, a *plan failure* occurs if one of the actions or sub-tasks within the plan fails.

The agent's action upon plan failure depends on its nature: for example, the agent may declare the task to have failed if one plan has been tried and resulted in failure, or it may retry alternate plans and declare (indeed, must declare) *task failure* only if all possible alternate plans to perform the task have been tried and resulted in failure. Observe that, while task failure can follow from plan failure or a sequence of plan failures, plan failure need not lead to task failure provided the agent can successfully complete an alternate plan. Moreover, task failure can also arise separately from plan failure, if the agent decides to abort the task.

Our approach associates an *abort-method* with each plan. This enables the programmer to specify dedicated compensation actions according to how the agent is attempting to perform the task. Note that our abort-methods can be arbitrary programs and so can invoke tasks that may be performed dynamically in the usual BDI fashion, i.e., the clean-up is not limited to executing a predetermined set of actions. The question remains which abort-method should be invoked, and in what manner. Given the complexity of agent action spaces, it is not possible nor desirable to enumerate a static set of rules. Rather, the agent will invoke its abort-methods dynamically according to the state of execution and its own internal events.

An alternative to attaching an abort-method to each plan is to attach such methods to each atomic action. We choose the former because: (1) action-level abort-methods would incur a greater overhead, (2) plans are meant to be designed as single cohesive units and are the unit of deliberation in BDI systems, and (3) the clean-up methods for failure in current systems are attached to plans.

In order to understand how the agent's abort processing should function, we consider three situations where it is sensible for an agent to consider aborting some of its tasks and plans:

1. When a task succeeds or fails because of an external factor other than the agent itself, the plan currently executed to perform the task should be aborted. For example, suppose company policy changes so that employees of Alice's seniority automatically have clearance for publishing papers. Since Alice now has clearance for publishing her paper, CALO can abort the plan for Apply For Clearance. In doing so it must invoke the abort-method, in this case thus performing Cancel Clearance Request.[3]

2. When two or more sub-programs are executed in parallel, if one fails then the others should be aborted, given that the failure of one branch leads to the failure of the overall task. For example, suppose that part-way through writing the paper, Alice realizes that there is a fatal flaw in her results, and so notifies CALO that she will not be able to complete the paper by the deadline. The failure of the Track Writing Paper task should cause the Apply For Clearance task being executed in parallel to be aborted.

3. When an execution event alters the importance of an existing task or intention, the agent should deliberate over whether the existing plan(s) should continue. For example, suppose that Alice tasks CALO with a new, high-priority task to purchase a replacement laptop, but that Alice lacks enough funds to both purchase the laptop and to attend IJCAI. Reasoning over resource requirements [20, 10] will cause the agent to realize that it cannot successfully complete both tasks. Given that the new task has greater importance, a rational agent will evaluate its best course of action and may decide to abort — or at least suspend — the existing task of submitting a paper and intentions derived from it [12].

[3]If there is any difference between how to abort a task that is externally performed versus how to abort one that is now known to be impossible, the abort-method can detect the circumstances and handle the situation as appropriate.

The operational semantics we provide in Section 4 for aborting tasks and plans captures the first two situations above. The third situation involves deliberating over the importance of a task, which depends on various factors such as task priority. Although this deliberation is beyond the scope of this paper, it is a complementary topic of our future work.

Note that the above situations apply to *achievement goals*, for which the task is completed when a particular state of the world is brought about (e.g., ensure we have clearance). Different forms of reasoning apply to other goal types [4] such as *maintenance* goals [1], where the goal is satisfied by maintaining a state of the world for some period of time (e.g., maintain $100 in cash).

Abort Method Representation

The intent of aborting a task or plan is that the task or plan and all its children cease to execute, and that appropriate clean-up methods are performed as required. In contrast to offline planning systems, BDI agents are situated: they perform online deliberation and their actions change the state of the world. As a result, the effects of many actions cannot be simply undone. Moreover, the "undo" process may cause adverse effects. Therefore, the clean-up methods that we specify are forward recovery procedures that attempt to ensure a stable state and that also may, if possible, recover resources.

The common plan representation in BDI-style systems such as JACK and SPARK includes a *failure-method*, which is the designated clean-up method invoked when the plan fails. To this, we add the *abort-method*, which is invoked if the plan is to be aborted. In our example, the abort-method for the plan for Support Meeting Submission consists of invoking the sub-task Cancel Paper Number. The abort-method need not explicitly abort Apply For Clearance, because the agent will invoke the abort-method for the subtask appropriately, as we outline below.

The assumption here is that, like the *failure-method*, the programmer of the agent system has the opportunity to specify a sensible *abort-method* that takes into consideration the point in the plan at which the abort is to be executed. For any plan, the abort-method is optional: if no abort-method is specified, the agent takes no specific action for this plan. However, the agent's default behavioural rules still apply, for example, whether to retry an alternate plan for the parent task.

Note that an explicit representation of the clean-up methods for tasks is not required, since tasks are performed by executing some plan or plans. Hence, aborting a task means aborting the current plan that is executed to perform that task, as we next describe.

Abort Method Invocation

We now informally lay out the agent's action upon aborting plans and tasks. When a plan P is aborted:

1. Abort each sub-task that is an *active* child of P. An active child is one that was triggered by P and is currently in execution.

2. When there are no more active children, invoke the abort method of plan P.

3. Indicate a plan failure to T_P, the parent task of P. We note here that if the parent task T_P is not to be aborted then the agent *may* choose another applicable plan to satisfy T_P.

When a task (or sub-task) T is aborted:

1. Abort the current active plan to satisfy T (if any).
2. When there are no more active child processes, drop the task. The agent thus no longer pursues T.
3. Note here that when the current active plan for performing T is aborted, no other applicable plans to perform T should be tried as it is the task that is to be aborted.

In order to prevent infinitely cascading clean-up efforts, we assume that abort-methods will never be aborted nor fail. In reality, however, an abort-method may fail. In this case, lacking a more sophisticated handling mechanism, the agent simply stops executing the failed abort-method with no further deliberation. The assumption we make is thus not a reflection of the full complexity of reality, but one that is pragmatic in terms of the agent execution cycle; the approach to failure-handling of [21] makes the same assumption. In systems such as SPARK, the programmer can specify an alternative behaviour for a failed failure- or abort-method by means of *meta-level* procedures. We also assume that failure- and abort-methods terminate in finite time.

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

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

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

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

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

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

Figure 1: Operational rules of CAN.

4. OPERATIONAL SEMANTICS

We provide the semantics for the task and plan failure and aborting processes outlined above. We use the CAN language initially defined in [23] and later extended as CANPLAN in [17] to include a planning component and then as CANPLAN2 in [18] to improve the goal adoption and dropping mechanisms. The extensions also simplified the semantics in the earlier work. We use some of these simplifications for providing a brief summary of the CAN language in Section 4.1. Following a presentation of the operational semantics of our approach in Section 4.2, in Section 4.3 we provide a worked example to clarify the semantics that we present.

4.1 CAN Language

CAN is a high-level agent language, in a spirit similar to that of AgentSpeak [15] 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 [2].

In practical systems, tasks are typically translated into events that trigger the execution of some plans. This is also true in the CAN language, but, in order to maintain the persistence of goals, a goal construct is introduced. This is denoted by $\mathsf{Goal}(\phi_s, P, \phi_f)$, where ϕ_s is the success condition that determines when the goal is considered achieved, ϕ_f is a fail condition under which it is considered the goal is no longer achievable or relevant, and P is a program for achieving the goal, which will be aborted once ϕ_s or ϕ_f become true.

An agent's behavior 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)[4] and P is the plan body. The plan body is a *program* that is defined recursively as follows:

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

[4]An omitted c is equivalent to $true$.

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, and $!e$[5] is an event[6] that is posted from within the program. The compound constructs are sequencing ($P_1; P_2$), parallel execution ($P_1 \| P_2$), and goals ($\mathsf{Goal}(\phi_s, P, \phi_f)$).

The above defines the *user language*. In addition, a set of auxiliary compound forms are used internally when assigning semantics to constructs. *nil* is the basic (terminating) program. When an event matches a set of plan clauses these are collected into a set of guarded alternatives ($(\!|c_1 : P_1, \ldots, c_n : P_n|\!)$). The other auxiliary compound form, $\triangleright$, is a choice operator dual to sequencing: $P_1 \triangleright P_2$ executes P_1 and then executes P_2 only if P_1 failed.

A summary of the operational semantics for CAN in line with [23] and following some of the simplifications of [17] is as follows. A *basic configuration* $S = \langle \mathcal{B}, \mathcal{G}, \Gamma\rangle$ consists of the current belief base $\mathcal{B}$ of the agent, the current set of goals $\mathcal{G}$ being pursued (i.e., set of formulae), 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).

Figure 1 gives some of the operational rules. The $Event$ rule handles task events by collecting all *relevant plan clauses* for the event in question: for each plan clause $e' : \psi_i \leftarrow P_i$, if there is a most general unifier, $\theta = \mathsf{mgu}(e, e')$ of e' and the event in

[5]Where it is obvious that e is an event we will sometimes exclude the exclamation mark for readability.

[6]Typically an achievement goal.

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

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

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

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

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

Figure 2: Rules for goals in CAN.

question, then the rule constructs a guarded alternative $\psi_i\theta : P_i\theta$. The $Select$ rule then selects one *applicable plan body* from a set of (remaining) relevant alternatives: program $P \rhd (\!|\Delta|\!)$ states that program P should be tried first, falling back to the remaining alternatives, $\Delta \setminus P$, if necessary. This rule and the $\rhd_{fail}$ rule together are used for failure handling: if the current program P_i from a plan clause for a task fails, rule $\rhd_{fail}$ is applied first, and then if possible, rule $Select$ will choose another applicable alternative for the task if one exists. Rule $Sequence$ handles sequencing of programs in the usual way. Rules $Parallel_1$ and $Parallel_2$ define the possible interleaving when executing two programs in parallel.

Figure 2 gives simplified rules for dealing with goals, in line with those presented in [17]. The first rule states that a goal succeeds when ϕ_s become true; the second rule states that a goal fails when ϕ_f become true. The third rule G_I initializes the execution of a goal-program by updating the goal base and setting the program in the goal to $P \rhd P$; the first P is to be executed and the second P is used to keep track of the original program for the goal. The fourth rule G_S executes a single step of the goal-program. The final rule G_R restarts the original program (encoded as P_2 of pair $P_1 \rhd P_2$) whenever the current program is finished but the desired and still possible goal has not yet been achieved.

4.2 Aborting Intentions and Handling Failure

We next introduce the ability to specify handler programs, in the form of failure- and abort-methods, that deal with the clean-up required when a given program respectively fails or is aborted. We do not associate failure- and abort- methods with plan clauses or with tasks (events), but rather we introduce a new program construct that specifies failure- and abort- methods for an arbitrary program. The *FAb*(P, P_F, P_A) construct executes the program P. Should P fail, it executes the failure handling program P_F; should P need to be aborted, it executes the abort handling program P_A. Thus to add failure- and abort- methods P_F and P_A to a plan clause $e : c \leftarrow P$, we write $e : c \leftarrow$ *FAb*(P, P_F, P_A).

With the introduction of the ability to abort programs, we modify the parallel construct to allow the failure of one branch to abort the other. We must take into consideration the possible existence of abort-methods in the aborted branch. Similarly, with the Goal construct we can no longer completely abandon the program the goal contains as soon as the success or failure condition holds; we must now take into consideration the existence of any abort-methods applicable to the program.

We provide the semantics of an augmented agent language containing the *FAb* construct by defining a source transformation, similar to macro-expansion, that maps a plan library containing the *FAb*(P, P_F, P_A) construct into (almost) standard CAN. The one non-standard extension to CAN is a *wait-until-condition* construct. We explain this simple modification of the parallel construct below when we come to translation of the Goal construct. First we describe the general nature of the source transformation, which proves to be quite simple for most of the language constructs, and then we concentrate on the three more complex cases: the *FAb*, parallel, and Goal constructs.

A key issue is that the *FAb* constructs may be nested, either directly or indirectly. Let us call each instantiation of the construct at execution time a *possible abort point* (pap). Where these constructs are nested, it is important that before the failure- or abort-method of a parent *pap* is executed, the failure- or abort-methods programs of the children *pap*s are executed first, as described earlier in Section 3. The need to coordinate the execution of the abort-methods of nested *pap*s requires that there be some way to identify the parents and children of a particular *pap*. We achieve this as part of the source transformation by explicitly keeping track of the context of execution as an extra parameter on the events and an extra variable within each plan body.[7]

The source transformation replaces each plan clause of the form $e : c \leftarrow P$ with a plan clause $e(v) : c \leftarrow \mu_v(P)$ where v is a free variable, not previously present in the plan clause. This variable is used to keep track of the context of execution.

The value of the context variable is a list of identifiers, where each new *pap* is represented by prepending a new identifier to the context. For example, if the identifiers are integers, the context of one *pap* may be represented by a list $[42, 1]$ and the context introduced by a new *pap* may be represented by $[52, 42, 1]$. We will refer to *pap*s by the context rather than by the new identifier added, e.g., by $[51, 42, 1]$ not 51. This enables us to equate the ancestor relationship between *pap*s with the list suffix relationship on the relevant contexts, i.e., v is an ancestor of v' if and only if v is a suffix of v'.

For most CAN constructs, the context variable is unused or passed unchanged:

$\mu_v(act) = act$
$\mu_v(+b) = +b$
$\mu_v(-b) = -b$
$\mu_v(nil) = nil$
$\mu_v(!e) = !e(v)$
$\mu_v(P_1; P_2) = \mu_v(P_1); \mu_v(P_2)$
$\mu_v(P_1 \rhd P_2) = \mu_v(P_1) \rhd \mu_v(P_2)$
$\mu_v((\!|\psi_1 : P_1, \ldots, \psi_n : P_n|\!)) = (\!|\psi_1 : \mu_v(P_1), \ldots, \psi_n : \mu_v(P_n)|\!)$

It remains to specify the transformation $\mu_v(\cdot)$ in three cases: the *FAb*, parallel, and Goal constructs. These are more complex in that the transformed source needs to create a new *pap* identifier dynamically, for use as a new context within the construct, and to keep track of when the *pap* is active (i.e., currently in execution) by adding and removing beliefs about the context.

Let us introduce the primitive action $prependID(v, v')$ that creates a new *pap* identifier and prepends it to list v giving list v'. We also introduce the following predicates:

- $a(v)$ — the *pap* v is currently *active*.
- $abort(v)$ — the *pap* v should be *aborted* (after aborting all of its descendants).

[7] An alternative would be to use meta-level predicates that reflect the current state of the intention structure.

- $f(v)$ — the program of *pap* v has failed.
- $ancestorof(v, v') \equiv v = v' \vee ancestorof(v, tail(v'))$ — the *pap* v is an ancestor of *pap* v'.
- $nac(v) \equiv \neg\exists v'.(a(v') \wedge ancestorof(v, v') \wedge v \neq v')$ — v has *no active children*.
- $sa(v) \equiv \exists v'.abort(v') \wedge ancestorof(v', v)$ — we *should abort* v, i.e., $abort$ is true of v or some ancestor; however, we need to wait until no children of v are active.
- $san(v) \equiv sa(v) \wedge nac(v)$ — we *should abort* v *now* if we should abort v and v has no active children.

First let us consider the case of the *FAb* construct. The idea is that, whenever a new *pap* occurs, the $prependID(v, v')$ action is used to create a new *pap* identifier list v' from the existing list v. We then add the belief that v' is the active context, i.e., $+a(v')$, and start processing the program within the *pap* using v' instead of v as the context. We need to make sure that we retract the belief that v' is active at the end, i.e., $-a(v')$.

We use the Goal construct to allow us to drop the execution of a program within a *pap* v' when it is necessary to abort. While executing the program P, we know that we need to drop P and invoke its abort-method if some ancestor of P has been told to abort. This is represented by the predicate $sa(v')$ being true. However, we need to make sure that we do this only after every child *pap* has had the chance to invoke its abort-method and all these abort-methods have completed: if we drop the program too soon, then execution of the abort-methods of the children will also be dropped. Therefore, the condition we actually use in the Goal construct to test when to drop the program is $san(v')$. This condition relies on the fact that as the children *pap*s complete, they remove the relevant a facts.

Our use of the Goal construct is for its ability to *drop* the execution of a program when conditions are met. To leave aside the "repeat execution until a condition is met" aspect, we must ensure that the success or failure condition of the construct is satisfied once the execution of the program succeeds or fails. We make sure of this by retracting the fact $a(v')$ on success and asserting the fact $f(v')$ on failure, and by having the appropriate success and failure conditions on the Goal. Hence, if the Goal construct fails, then the program either was aborted or it failed. We invoke the relevant failure- or abort- method, retract the $a(v')$ fact, and then fail.

Putting all this together, we formally define $\mu_v(\mathit{FAb}(P, P_A, P_F))$ to be the following, where v' is a new variable distinct from any other in the agent's plan library:

$$prependID(v, v'); +a(v');$$
$$\textsf{Goal}\ \big(\ \neg a(v'), (\mu_{v'}(P); -a(v') \triangleright +f(v')), san(v') \vee f(v')\ \big)$$
$$\triangleright\ (((?sa(v'); \mu_v(P_A)) \triangleright\ \mu_v(P_F)); -a(v'); ?false)$$

Second, we must transform the parallel operator to ensure that the failure of one branch safely aborts the other. Here we construct two new contexts, v' and v'', from the existing context v. If one branch fails, it must abort the other branch. At the end, if either branch was aborted, then we must fail.

Let v' and v'' be new variables distinct from any other in the agent's plan library. We define $\mu_v(P_1 \| P_2)$ to be:

$$prependID(v, v'); prependID(v, v''); +a(v'); +a(v'');$$
$$\big(\ \textsf{Goal}\ \big(\neg a(v'), (\mu_{v'}(P_1); -a(v') \triangleright +f(v')), san(v') \vee f(v')\ \big)$$
$$\quad \triangleright\ (+abort(v''); -a(v''))$$
$$\|$$
$$\quad \textsf{Goal}\ \big(\neg a(v''), (\mu_{v''}(P_2); -a(v'') \triangleright +f(v'')), san(v'') \vee f(v'')\ \big)$$
$$\quad \triangleright\ (+abort(v'); -a(v''))$$
$$\big); ?\neg abort(v') \wedge \neg abort(v'')$$

Finally, we need to modify occurrences of the Goal construct in two ways: first, to make sure that the abort handling methods are not bypassed when the success or failure conditions are satisfied, and second, to trigger the aborting of the contained program when either the success or failure conditions are satisfied.

To transform the Goal construct we need to extend standard CAN with a *wait-until-condition* construct. The construct $\phi : P$ does not execute P until ϕ becomes true. We augment the CAN language with the following rules for the guard operator ':':

$$\frac{\mathcal{B} \models \phi}{\langle \mathcal{B}, \mathcal{G}, (\phi : P\rangle \longrightarrow \langle \mathcal{B}, \mathcal{G}, P\rangle}\ :_{true}$$

$$\frac{\mathcal{B} \not\models \phi}{\langle \mathcal{B}, \mathcal{G}, (\phi : P)\rangle \longrightarrow \langle \mathcal{B}, \mathcal{G}, (\phi : P)\rangle}\ :_{false}$$

In order to specify $\mu_v(\textsf{Goal}(\phi_s, P, \phi_f))$, we generate a new *pap* and execute the program within the Goal construct in this new context. We must ensure that belief $a(v')$ is removed whether the Goal succeeds or fails. We shift the success and failure condition of the Goal construct into a parallel branch using the wait-until-condition construct, and modify the Goal to use the *should abort now* condition $san(v')$ as the success condition. The waiting branch will trigger the abort of the program should either the success or failure condition be met. To avoid any problems with terminating the wait condition, we also end the wait if the *pap* is no longer active.

Let v' be a new variable distinct from any other in the agent's plan library. We define $\mu_v(\textsf{Goal}(\phi_s, P, \phi_f))$ to be:

$$prependID(v, v'); +a(v');$$
$$\big(\ \textsf{Goal}\ (\ san(v'), \mu_{v'}(P), false)\ ; -a(v'); ?\phi_s\ )\quad \|$$
$$\quad \phi_s \vee \phi_f \vee \neg a(v') : +abort(v')\ \big)$$

The program P will be repeatedly executed until $san(v')$ becomes true. There are two ways this can occur. First, if either the success condition ϕ_s or the failure condition ϕ_f becomes true, then the second branch of the parallel construct executes. This causes $abort(v')$ to become true, and, after the descendant *pap*s' abort-methods are executed, $san(v')$ becomes true. In this case, P is now dropped, the $a(v')$ is removed, and the entire construct succeeds or fails based on ϕ_s. The second way for $san(v')$ to become true is if v' or one of its ancestors is aborted. In this case, once the descendant *pap*s' abort-methods are executed, $san(v')$ becomes true, P is dropped, the $a(v')$ belief is removed (allowing the second parallel branch to execute, vacuously instructing v' to abort), and the first parallel branch fails (assuming ϕ_s is false).

4.3 Worked Example

Let us look at translation of the IJCAI submission example of Section 2. We will express tasks by events, for example, the task Allocate a Paper Number we express as the event APN. Let the output of the Apply For Clearance task be `Y` or `N`, indicating the approval or not of Alice's manager, respectively. Then we have (at least) the following two plan clauses in CAN, for the Support Meeting Submission and Apply For Clearance tasks, respectively:

$$SMS(m) : isconf(m) \leftarrow$$
$$\quad \mathit{FAb}(!APN; !TWA; \big(!AFC \,\|\, !TWP\big); !HPS, !CPN, !CPN)$$
$$AFC : true \leftarrow \quad \mathit{FAb}(!SCR; !WFR(r); ?r = \texttt{Y}, nil, !CCR)$$

Note that Support Meeting Submission has a parameter m, the meeting of interest (IJCAI, in our example), while Apply For Clearance has no parameters.

Let us look first at the translation of the second plan clause, for AFC, since it is the simpler of the two. Let v'' and v'''' denote new variables. Then we have as the translated plan clause:

$$
\begin{array}{l}
AFC(v'') : true \leftarrow \\
\quad prependID(v'', v''''); +a(v''''); \\
\quad \mathsf{Goal}\,\big(\, \neg a(v''''), \\
\qquad (!SCR(v''''); !WFR(r, v''''); ?r = \mathrm{Y}; -a(v'''') \triangleright +f(v'''')), \\
\qquad san(v'''') \vee f(v'''')\,\big) \\
\quad \triangleright (((?sa(v''''); !CCR(v'')) \triangleright nil); -a(v''''); ?false)
\end{array}
$$

We can see that an extra context parameter has been added to each task and that the old plan body now appears inside a Goal construct. Should the old plan body succeed, belief $a(v'''')$ is retracted, causing the Goal to succeed. If the old plan body fails, or if the task is to be aborted, the Goal construct fails. This is followed by the execution of CCR (in the case of an abort), the retraction of $a(v'''')$, and failure.

The translation of the first plan clause, for SMS, is more complex, because of the parallel construct that introduces nested *pap*s:

$$
\begin{array}{l}
SMS(m, v) : isconf(m) \leftarrow \\
\quad prependID(v, v'); +a(v'); \\
\quad \mathsf{Goal}\,\big(\, \neg a(v'), \\
\qquad ((!APN(v'); \\
\qquad\quad !TWA(v'); \\
\qquad\quad prependID(v', v''); prependID(v', v'''); +a(v''); +a(v'''); \\
\qquad\quad \big(\, \mathsf{Goal}\,\big(\, \neg a(v''), \\
\qquad\qquad (!AFC(v''); -a(v'') \triangleright +f(v'')), \\
\qquad\qquad san(v'') \vee f(v'')\,\big) \\
\qquad\quad \triangleright (+abort(v'''); -a(v'')) \\
\qquad\quad \| \\
\qquad\qquad \mathsf{Goal}\,\big(\, \neg a(v'''), \\
\qquad\qquad (!TWP(v'''); -a(v''') \triangleright +f(v''')), \\
\qquad\qquad san(v''') \vee f(v''')\,\big) \\
\qquad\quad \triangleright (+abort(v''); -a(v''')) \\
\qquad\quad \big)\,; \\
\qquad\quad ?\neg abort(v'') \wedge \neg abort(v'''); \\
\qquad\quad !HPS(v'); \\
\qquad\quad -a(v')) \\
\qquad \triangleright +f(v')), \\
\qquad san(v') \vee f(v')\,\big) \\
\quad \triangleright (((?sa(v'); !CPN(v)) \triangleright !CPN(v)); -a(v'); ?false)
\end{array}
$$

Here we can see that if the task $!TWP(v''')$ fails then $f(v''')$ will be asserted, failing the Goal construct that contains it, and leading to $abort(v'')$ being asserted. If the $!WFR(r, v'''')$ task in the expansion of $!AFC(v'')$ is still executing and has no active child *pap*s, then $sa(v'')$ and $sa(v'''')$ will be true; however, only $san(v'''')$ and not $san(v'')$ will be true. This set of conditions will cause the Goal construct in the first plan clause to fail, dropping execution of $!WFR(r, v'''')$. The task $!CCR(v'')$ will be executed. Once this task completes, belief $a(v'''')$ is retracted, causing $san(v'')$ to become true, leading to the first Goal construct of the second plan clause to fail.

While the translated plan clauses appear complicated, observe that the translation from the initial plan clauses is entirely automated, according to the rules set out in Section 4.2. The translated plan clauses, with the semantics of CAN augmented by our wait-until-condition construct, thus specify the operation of the agent to handle both failure and aborting for the example.

5. RELATED WORK

Plan failure is handled in the extended version of AgentSpeak found in the Jason system [6]. Failure "clean-up" plans are triggered from goal deletion events $-!g$. Such plans, similar to our failure methods, are designed for the agent to effect state changes (act to "undo" its earlier actions) prior to possibly attempting another plan to achieve the failed goal g.

Given Jason's constructs for dropping a goal with an indication of whether or not to try an alternate plan for it, Hübner et al. [6] provide an informal description of how a Jason agent modifies its intention structure when a goal failure event occurs. In a *goal deletion plan*, the programmer can specify any "undo" actions and whether to attempt the goal again. If no goal deletion plan is provided, Jason's default behaviour is to not reattempt the goal. Failure handling is applied only to plans triggered by addition of an achievement or test goal; in particular, goal deletion events are not posted for failure of a goal deletion plan. Further, the informal semantics of [6] do not consider parallel sub-goals (i.e., the CAN $\|$ construct), since such execution is not part of Jason's language.

The implementation of Hübner et al. [6] requires Jason's internal actions. A requirement for implementing our approach is a reflective capability in the BDI agent implementation. Suitable implementations of the BDI formalism are JACK [2], Jadex [14], and SPARK [9]. All three allow *meta level* methods that are cued by meta events such as goal adoption or plan failure, and offer introspective capabilities over goal and intention states.

Such meta level facilities are also required by the approach of Unruh et al. [21], who define goal-based *semantic compensation* for an agent. Failure-handling goals are invoked according to failure-handling strategy rules, by a dedicated agent *Failure Handling Component* (FHC) that tracks task execution. These goals are specified by the agent programmer and attached to tasks, much like our *FAb*(P, P_F, P_A) construct associates failure and abort methods with a plan P. Note, however, that in contrast to both [6] and our semantics, [21] attach the failure-handling knowledge at the goal, not plan, level. Their failure-handling goals may consist of *stabilization goals* that perform localized, immediate "clean-up" to restore the agent's state to a known, stable state, and *compensation goals* that perform "undo" actions. Compensation goals are triggered on aborting a goal, and so not necessarily on goal failure (i.e., if the FHC directs the agent to retry the failed goal and the retry is successful).

The FHC approach is defined at the goal level in order to facilitate abstract specification of failure-handling knowledge; the FHC decides when to address a failure and what to do (i.e., what failure-handling goals to invoke), separating this knowledge from the how of implementing corrective actions (i.e., what plan to execute to meet the adopted failure-handling goal). This contrasts with simplistic plan-level failure handling in which the what and how are intermingled in domain task knowledge. While our approach is defined at the plan level, our extended BDI semantics provides for the separation of execution and failure handling. Further, the FHC explicitly maintains data structures to track agent execution. We leverage the existing execution structures and self-reflective ability of a BDI agent to accomplish both aborting and failure handling without additional overhead. FHC's failure-handling strategy rules (e.g., whether to retry a failed goal) are replaced by instructions in our P_F and P_A plans, together with meta-level default failure handlers according to the agent's nature (e.g., blindly committed).

The FHC approach is independent of the architecture of the agent itself, in contrast to our work that is dedicated to the BDI formalism (although not tied to any one agent system). Thus no formal semantics are developed in [21]; the FHC's operation is given as

a state-based protocol. This approach, together with state checkpointing, is used for multi-agent systems in [22]. The resulting architecture embeds their failure handling approach within a pair processing architecture for agent crash recovery.

Other work on multi-agent exception handling includes *AOEX*'s distributed exception handling agents [5], and the similar *sentinels* of [8]. In both cases, failure-handling logic and knowledge are decoupled from the agents; by contrast, while separating exception handling from domain-specific knowledge, Unruh et al.'s FHC and our approach both retain failure-handling logic within an agent.

6. CONCLUSION AND FUTURE WORK

The tasks and plans of an agent may not successfully reach completion, either by the choice of the agent to abort them (perhaps at the request of another agent to do so), or by unbidden factors that lead to failure. In this paper we have presented a procedure-based approach that incorporates aborting tasks and plans into the deliberation cycle of a BDI-style agent, thus providing a unified approach to failure and abort. Our primary contribution is an analysis of the requirements on the operation of the agent for aborting tasks and plans, and a corresponding operational semantics for aborting in the abstract agent language CAN.

We are planning to implement an instance of our approach in the SPARK agent system [9]; in particular, the work of this paper will be the basis for SPARK's abort handling mechanism. We are also developing an analysis tool for our extended version of CAN as a basis for experimentation.

An intelligent agent will not only gracefully handle unsuccessful tasks and plans, but also will deliberate over its cognitive attitudes to decide its next course of action. We have assumed the default behaviour of a BDI-style agent, according to its nature: for instance, to retry alternatives to a failed plan until one succeeds or until no alternative plans remain (in which case to fail the task). Future work is to place our approach in service of more dynamic agent reasoning, such as the introspection that an agent capable of reasoning over task interaction effects and resource requirements can accomplish [19, 12].

Related to this is determining the cost of aborting a task or plan, and using this as an input to the deliberation process. This would in particular influence the commitment the agent has towards a particular task: the higher the cost, the greater the commitment.

Our assumption that abort-methods do not fail, as discussed above, is a pragmatic one. However, this is an issue worthy of further exploration, either to develop weaker assumptions that are also practical, or to analyze conditions under which our assumption is realistic. A further item of interest is extending our approach to failure and abort to maintenance goals [1]. For such goals a different operational semantics for abort is necessary than for achievement goals, to match the difference in semantics of the goals themselves.

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.

7. REFERENCES

[1] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal representation for BDI Agent systems. In *Proc. of Second Intl. Workshop on Programming Multi-Agent Systems (ProMAS'04)*, 2004.

[2] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. JACK intelligent agents — components for intelligent agents in Java. AgentLink News, Issue 2, 1999.

[3] M. G. Chessell, C. Vines, D. Butler, C. M. Ferreira, and P. Henderson. Extending the concept of transaction compensation. *IBM Systems Journal*, 41(4), 2002.

[4] M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer. Goal types in agent programming. In *Proc. of AAMAS'06*, 2006.

[5] S. Entwisle, S. Loke, S. Krishnaswamy, and E. Kendall. Aoex: An agent-based exception handling framework for building reliable, distributed, open software systems. In *Proc. of Seventh Joint Conf. on Knowledge-Based Software Engineering*, 2006.

[6] J. F. Hübner, R. H. Bordini, and M. Wooldridge. Programming declarative goals using plan patterns. In *Proc. of 4th Intl. Workshop on Declarative Agent Languages and Technologies*, 2006.

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

[8] M. Klein, J. A. Rodríguez-Aguilar, and C. Dellarocas. Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. *Autonomous Agents and Multi-Agent Systems*, 7(1–2):179–189, 2003.

[9] D. Morley and K. Myers. The SPARK agent framework. In *Proc. of AAMAS'04*, 2004.

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

[11] 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, 2007. To appear.

[12] K. L. Myers and N. Yorke-Smith. A cognitive framework for delegation to an assistive user agent. In *Proc. of AAAI 2005 Fall Symposium on Mixed-Initiative Problem-Solving Assistants*, 2005.

[13] L. Padgham and M. Winikoff. *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley and Sons, 2004.

[14] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In R. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni, editors, *Multi-Agent Programming*. Springer, 2005.

[15] 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.

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

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

[18] S. Sardina and L. Padgham. Goals in the context of bdi plan failure and planning. In *Proc. of AAMAS'07*, 2007.

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

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

[21] A. Unruh, J. Bailey, and K. Ramamohanarao. A framework for goal-based semantic compensation in agent systems. In *Proc. of First Intl. Workshop on Safety and Security in Multi-Agent Systems*, 2004.

[22] A. Unruh, H. Harjadi, J. Bailey, and K. Ramamohanarao. Semantic-compensation-based recovery management in multi-agent systems. In *Proc. of Second IEEE Symposium on Multi-Agent Security and Survivability (IEEE MAS&S'05)*, 2005.

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