Auton Agent Multi-Agent Syst (2017) 31:288–331
DOI 10.1007/s10458-015-9322-4

Aborting, suspending, and resuming goals and plans in BDI agents

James Harland[1] · **David N. Morley**[2,3] · **John Thangarajah**[1] · **Neil Yorke-Smith**[4,5]

Published online: 11 December 2015

Abstract Intelligent agents designed to work in complex, dynamic environments such as e-commerce must respond robustly and flexibly to environmental and circumstantial changes, including the actions of other agents. An agent must have the capability to deliberate about appropriate courses of action, which may include reprioritising tasks—whether goals or associated plans—aborting or suspending tasks, or scheduling tasks in a particular order. In this article we study mechanisms to enable principled suspend, resuming, and aborting of goals and plans within a Belief-Desire-Intention (BDI) agent architecture. We give a formal and combined operational semantics for these actions in an abstract agent language (CAN), thus providing a general mechanism that can be incorporated into several BDI-based agent platforms. The abilities enabled by our semantics provides an agent designer greater flexibility to direct agent operation, offering a generic means to manage the status of goals. We demonstrate the reasoning abilities enabled on a document workflow scenario.

Keywords Intelligent agents · Goal reasoning · Belief-Desire-Intention · BDI agents

✉ Neil Yorke-Smith
nysmith@aub.edu.lb

James Harland
james.harland@rmit.edu.au

David N. Morley
davidmorley@luckmor.com

John Thangarajah
johnt@rmit.edu.au

1 RMIT University, Melbourne, Australia

2 SRI International, Menlo Park, CA, USA

3 Present Address: Google, Inc., Mountain View, CA, USA

4 American University of Beirut, Beirut, Lebanon

5 University of Cambridge, Cambridge, UK

1 Introduction

Intelligent agent technology is widely adopted for developing complex software systems that operate in highly dynamic environments in business, such as in logistics [10,24], process design [2], and electronic trading [51]; in policy and social science, such as in climate change modelling [16]; and in marketing, such as personalized recommendations [14,38]. Their value is recognised by firms particularly for e-commerce and supply chain optimization [1,13,38]. Following the literature, we take an *intelligent agent* to be an rational, autonomous decision-making entity; in this sense it is more akin to a principal in the principal–agent theory of economics.

Such agent-based systems are typically structured in terms of conceptualized mental attitudes over which the agent performs its reasoning. For example, an e-commerce agent might reason over its beliefs about the bidding strategies of other agents. In agent architectures inspired by the *Belief-Desire-Intention (BDI)* model [34], the cognitive state of an agent comprises of *beliefs*, *goals*, and *plans*, amongst other aspects. The BDI architecture is the foundation for numerous deployed agent systems [7,25,32]. In general, an agent wishes to achieve a particular set of goals in the world state and will pursue a number of plans concurrently in order to achieve its goals. At regular intervals in its execution cycle, the agent will update its beliefs and review its current choice of goals and plans. A number of levels of reasoning are typically involved, and a variety of reasoning techniques have been employed in developed agents.

From a reasoning point of view, a key characteristic fundamental to intelligent agents and their success in deployed systems is their ability to deliberate autonomously over their goals. Agents designed to work in complex, dynamic environments must respond robustly and flexibly to environmental and circumstantial changes, and to the communications and actions of other agents. For an agent to respond to changes and to effect the courses of action that it judges to be most appropriate, the agent may choose, for instance, to adopt new tasks, to abort or suspend some of its current tasks, to resume others, to prioritize tasks, or to schedule tasks in a particular order. By 'task' we will mean either a goal or a plan.

This article considers how an agent can respond to the unpredictable dynamics of its environment, by providing mechanisms for the agent to manipulate its tasks. In particular, we focus on the suspension, resumption, and aborting of goals and plans.

Regarding terminology, in this article we use the term *drop* to mean that a task is simply halted and no further execution takes place. We use the term *abort* to mean that the task is halted, but some specific clean-up procedure may be executed, after which no further execution of the task takes place. Hence dropping a goal is the same as aborting a goal with an empty clean-up procedure. Similarly we use the term *suspend* to mean that a task is halted (possibly to be resumed later), and a clean-up procedure may be executed before any further processing, and the term *resume* to similarly imply the execution of a procedure before the task is resumed. In other words, dropping a tasks simply ceases to execute it, whereas the other three provide the option of appropriately executing a clean-up procedure.

Whilst there are a number of successful agents platforms that specifically incorporate goals—either explicitly, such as GOAL [17] and Jadex [6,31], or implicitly, such as events in JACK [52]—few of these systems provide mechanisms for aborting goals (as distinct from simply dropping them), and fewer provide mechanisms for suspending goals. For instance, GOAL provides methods for dropping goals, but not for the more general notion of aborting goals, and does not provide mechanisms for suspending or resuming goals. While both Jason [5] and Jadex provide mechanisms for aborting, suspending and resuming goals, these

mechanisms are less general than those discussed in this article, and in particular, aborting a goal in Jason and Jadex result in the goals simply being dropped. Whilst there may be some circumstances in which simply dropping a goal is appropriate (such as when a goal has succeeded, a goal has become irrelevant, or when an urgent response is required to an emergency), it is generally more appropriate to abort a goal, so that any appropriate 'clean up' behaviour can be specified. Hence the notions of aborting, suspending and resuming goals and plans discussed in this article are more general than current agent programming language implementations, and may be viewed as a specification of the behaviour of such constructs.

1.1 Example scenario

Consider the following example of a hypothetical interaction between a human and an assistive agent, through which we illustrate how abort, suspend and resume behaviours naturally arise.

Alice is a knowledge worker assisted by a learning, personal assistive agent such as *CALO* [28,55].[1] Alice plans to have a paper published at a leading academic conference. She assigns the goal of Support Manuscript Submission (SMS) to her CALO agent to support her in this task. CALO adopts the goal and a plan (which we denote as SMSPLAN) with the following subgoals to achieve it:

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. Obtain Clearance (OC): for publication.
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 subgoals must be performed in order, with the exception of subtasks 3 (OC) and 4 (TWP), which may be performed in parallel.

Suppose now that subgoals 1 and 2 are complete and CALO is performing steps 3 and 4 of the SMSPLAN, and that Obtain Clearance (OC) is performed by a plan consisting of the following subgoals and actions:

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

Now suppose that a change in circumstances causes Alice to attend to another more important task and so to place the paper writing on hold. Given this situation, Alice instructs her CALO agent to *suspend* the SMS goal until further notice. Suspending the SMS goal involves suspending the SMSPLAN plan, the OC subgoal and the TWP subgoal. Notably, suspending the OC subgoal (and the plan that achieves it) requires CALO to notify Alice's manager that Alice no longer requires clearance for publication. CALO can achieve this notification by invoking the Cancel Clearance Request (CCR) goal.

After completing her other, priority, task, Alice then decides to resume writing her paper. She requests CALO to *resume* the SMS goal. This requires CALO to resume the suspended

[1] CALO was a multi-year research project from which Apple, Inc.'s *Siri* evolved.

OC and TWP subgoals. The plan for obtaining clearance requires the clearance request to be re-sent afresh to the manager, according to the rules of Alice's organization.

Suppose that some time later, due to travel issues it emerges that Alice will definitely not be able to attend the conference. Hence, she instructs her CALO agent to *abort* the SMS goal. Aborting the goal implies aborting both the SMSPLAN and the OC subgoal. Aborting the first 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) goal. Aborting the second requires CALO to cancel the clearance request to the manager, again by invoking Cancel Clearance Request.

1.2 Discussion

We note a number of important observations with respect to aborting, suspending and resuming tasks from the scenario. Here we consider both goals—which we take to have a dual declarative–procedural nature [54]—and their subgoals, and also plans—the recipes in process to achieve the adopted goals. We continue to use the term *task* when we mean to refer to both aspects.

First, the decision to suspend or abort a particular task 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 the example. In this article, we do not consider the questions of *why* and *when* tasks should be aborted or suspended but rather concentrate on determining the appropriate actions to be taken once these decisions are made. We thus provide essential capability for an agent system when an agent, according to its expertise, decides or is directed to perform such operations on its tasks.

Second, once the decision is made to abort, suspend or resume the attempt to submit a paper, there are some actions the agent should take, such as cancelling the clearance request on suspend and re-sending the clearance request on resumption. In other words, there are some actions that need to be performed as a consequence of the decision to abort, suspend or resume the task. This is similar to the case for failure, as implemented in agent platforms such as JACK, in that there may also be actions to take when a goal or plan fails.

Note that in the case of failure or aborting a task, 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 may involve compensation via forward recovery actions [3]. We discuss this in more detail in Sect. 2.2.

Third, in the case of aborting, given that tasks may contain subtasks, which may contain further subtasks, it is necessary for a parent task to wait until its children have finished their abort methods. In contrast to this, when suspending, the parent task will be suspended prior to the children since suspending the parent may result in the child tasks being aborted. We give some illustrative examples of this behaviour based on the above scenario in Sect. 3. This point exhibits of one of the technical challenges that we address in this article: determining the precise sequence of actions once a parent goal or plan is aborted or suspended. Note also that if the task is dropped rather than aborted, this process becomes trivial.

Fourth, there is a distinction between aborting (or failing) a goal 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, a plan aborting or failing does not necessarily lead to the parent task aborting or failing.

Finally, we adopt the approach that when a task is suspended, a condition which specifies when it is to be resumed can be denoted. However, when this condition become true, resuming the task is not automatic: rather, the agent must deliberate whether the task is still appropriate to be pursued as circumstances may have changed. For example, when resuming the paper-writing task, if there are no longer any funds available to travel, the task may be aborted rather than resumed. In addition, the suspended goal may need to be remain suspended if a higher-priority goal is resumed at the same time.

1.3 Summary of approach

In this article we study mechanisms to enable principled suspending, resuming, and aborting of achievement goals and their plans within a BDI-style intelligent agent architecture. We provide an operational semantics for the agent execution cycle in the presence of task suspension and aborts in an extended version of the abstract agent language CAN [39], which enables us to specify a BDI-based execution model without limiting our attention to a particular agent platform such as JACK, Jadex, or Jason.

As illustrated in the above scenario, the challenge to be addressed is the relationship between a goal, its subgoals, and the plans that may be in progress towards its achievement. We describe the challenges in terms of goals. The first specific challenge is that a decision to abort or suspend a goal may come at an arbitrary point in the process of achieving it. The goal may have some subgoals which have already been achieved (or aborted), some subgoals which are not yet achieved, and possibly a set of concurrently-executing plans at various stages of completion. Aborting or suspending this potentially complex set of processes will thus involve recursively acting on any subgoals which have not been achieved, as well as determining what to do with plans which are to be aborted or suspended. The second specific challenge is that the resumption of a goal after a period of suspension is not necessarily automatic. The agent will likely want to reconsider its options, since circumstances may have changed. For example, there now may be other goals with higher priority competing for the same resource, or other agents may have changed their requirements. 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.

To address this complexity, we augment each plan with an optional *abort method*, analogous to the failure method found in some agent programming languages. We address the presence of parallel execution threads and of subgoals, which require the agent to ensure that the abort methods for each plan are carried out in an appropriate sequence. We further augment each plan with optional *suspend* and *resume methods*. We annotate each task with one of three tags: *active*, *inactive*, or *suspended*. The latter two tags indicate that work on the task should not proceed; *inactive* indicates that the task should not be executed, while *suspended* also indicates that the task has been properly suspended. We modify the agent's execution cycle to respect these tags, and we take care to treat the case of aborting a suspended task.

The mechanisms to be described take precedence over the agent's normal steps in the BDI execution cycle. That is, any meta-activity of suspension or aborting occurs before regular agent deliberation and action, including intention selection and plan execution.

1.4 Contribution

Our contribution includes (1) principled design of abort, suspend, and resume mechanisms for BDI-style agents; (2) generic methods for adding such features to existing plans; (3) a formal and combined operational semantics in CAN; and (4) a detailed case study to illustrate

the mechanisms developed with a Prolog-based prototype. Further, at a technical level, we show how the addition of abort, suspend and resume mechanisms can be expressed in the original agent language by a systematic transformation process; essentially this 'compiles' the additional mechanisms into CAN, with the minor extension to include a `wait` mechanism. In this way, the semantics of the extended language derive automatically from the semantics of the original language.

This article combines and extends our initial report of semantics for aborting tasks [45] and semantics for suspending and resuming tasks [46]. This includes combining the different mechanisms from the two papers into one coherent semantics (which is not simply a conjunction of the two pieces of work), and developing a comprehensive prototype implementation of the complete semantics, demonstrating how an agent-based system can benefit from these capabilities.

One point to note is that we do not address the issue of providing proofs of the properties of our technique. Whilst this is a desirable outcome, and one that will be addressed in future work, doing so in this article would significantly increase its length. Instead, we concentrate on discussing issues around the integration of abort, suspend and resume facilities, our formalisation of these, and some key examples which have been developed with our prototype implementation.

A key aspect of our approach is that it is based on the transformation of an existing agent program, and specifically one using rules based on those of AgentSpeak [33], which are often referred to as *event-condition-action* rules. Specifically, we provide a transformation of an existing agent program into a more sophisticated one *in the same language* which incorporates abort, suspend and resume mechanisms. However, this transformed program is itself a set of event-condition-action rules, meaning not only that our approach provides a source-to-source transformation that could be used in a compiler or interpreter, but also that *no extra mechanisms are needed to implement it*. Hence our approach provides a formal specification of sophisticated agent behaviour which be used in any language that provides event-condition-action rules in the style of AgentSpeak.

1.5 Outline

We organize the remainder of the article as follows. Section 2 provides background and discusses the significance of our work in light of the literature. Section 3 identifies situations in which a task may be suspended or aborted, and outlines the process for aborting, suspending, and resuming tasks. Section 4 presents the unified operational semantics for these mechanisms and describes its implementation and operation on the earlier scenario. Section 5 concludes with directions for future work.

2 Background

We begin this section by introducing the BDI reasoning architecture, and the concepts of goal and plan.

2.1 Agents, goals, and plans

Among the foundations for intelligent agents architectures, the *Belief-Desire-Intention* (BDI) model [34] ascribes to an agent cognitive factors: *Beliefs* describe predicates held to be true by the agent about the world state and the agent's own state; *Desires* describe states of the

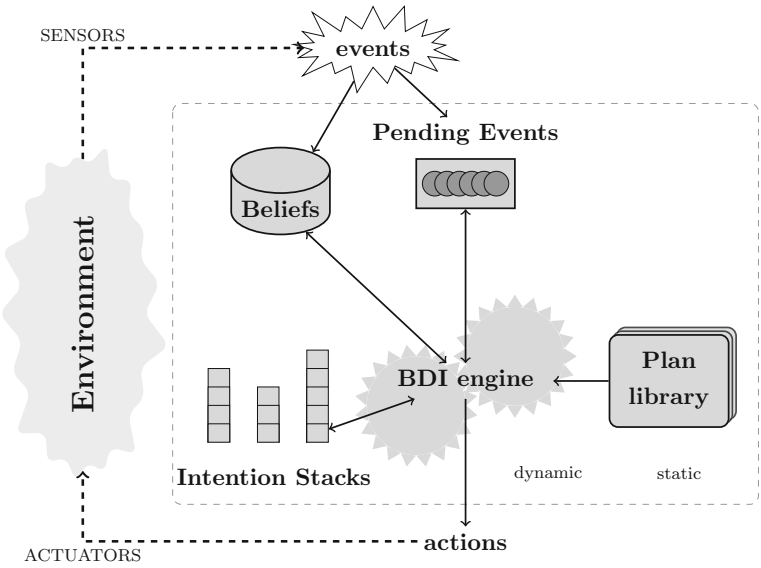

Fig. 1 A typical BDI-style architecture at the conceptual level. Events in the environment are sensed, updating beliefs. A queue of pending events is processed by the BDI reasoning engine, updating beliefs and intentions. Intentions are expanded using the plan library, and actions actuated in the world. Goals are not depicted

world that the agent would like to have brought about; and *Intentions* are means by which an agent commits to achieving its goals. Additional cognitive factors such as obligations and norms can also be provided [12].

In practice, agent architectures in the BDI heritage focus on *goals* more than desires, and *plans* more than intentions [12,23,29,30]. Goals describe a consistent set of desires to which the agent is committed. For example, whilst an agent may desire to empty the ocean, this would not be a realistic goal. Plans describe recipes, consisting of subgoals and actions, that the agent intends to achieve, in order to accomplish its goals.

A typical BDI-style agent system, illustrated in Fig. 1, follows three main steps: *perception*, *deliberation*, and *action*. First, *perception* of (changes in) world state via sensors, followed by updating of the agent's beliefs accordingly. Second, *deliberation*, in which the agent reasons about its goals and plans in light of its changed beliefs. For example, the agent may choose, for instance, to adopt new goals, to abort or suspend some of its current goals or plans, to resume others, to plan how to achieve a goal, to (re-)prioritize goals, or to schedule plans in a particular order. Third, after deliberation, the agent may *act* by performing some actions from some of its current plans via actuators. The actions may affect the agent's internal state (e.g., update its beliefs) or the external world (i.e., a situated action). Note that external actions may fail, i.e., the agent may be unable to successfully achieve the actions that it executes.

Goals and plans Among the various types of goals, such as performance, maintenance, testing, and achievement goals [11,15,36,47], the the most common type is the latter, and it is on achievement goals that we focus this article.

We follow Winikoff et al. [54] and define goals to have a dual declarative and procedural aspects. We suppose that a goal has a *success condition* ϕ_S that denotes when the goal may be considered to have succeeded and a *failure condition* ϕ_F that denotes when it may

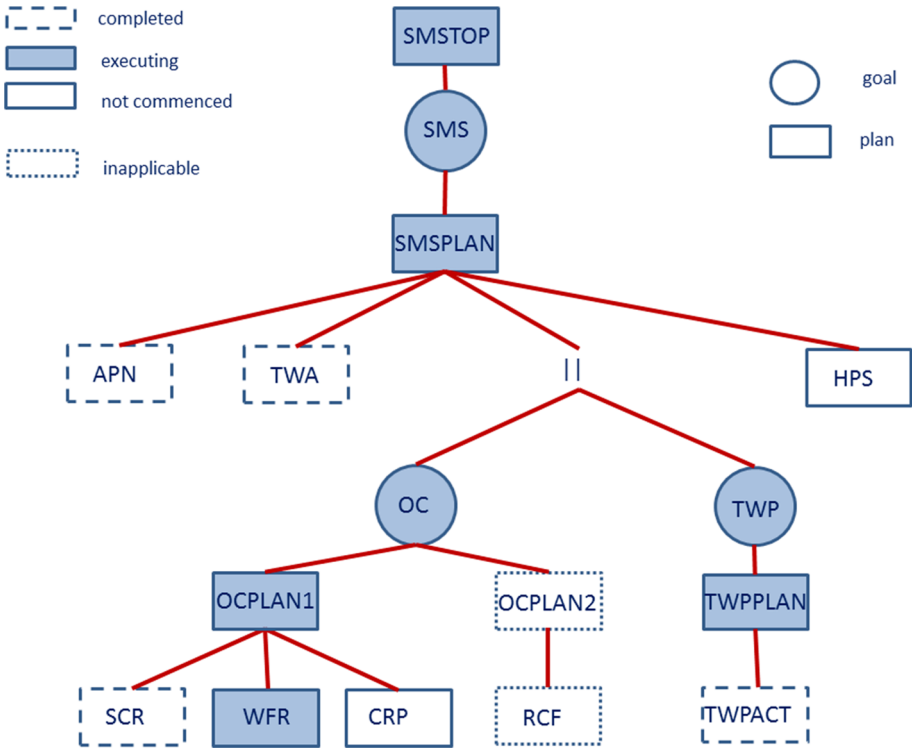

Fig. 2 Example partial goal–plan tree, showing execution part-way through the illustrative scenario described in Sect. 1.1. The *oval* and *rectangular shapes* represent goals and plans, respectively; *shading* and *dashed border* represent executing and completed tasks, respectively

be considered to have failed, capturing the declarative aspects. The goal also has a set of pre-defined *plans* that can be used to achieve the goal, which captures the procedural aspect.

We suppose that a plan has a (triggering) *event*, a *context* (pre-condition) that is a necessary condition before the plan may be executed, an *in-condition* that must be true during the plan execution, and a plan body. A *plan body* consists of actions and sub-goals describing a *program* for achieving the goal that it handles.

Formally, we define both goal and plan structures following the CAN language [40], which we describe in Sect. 4.1.

Goal–plan tree From the above, it can be seen that the hierarchical relationship between the goals and plans of an agent naturally gives rise to a *goal–plan tree* (GPT) structure [27,42,49]. As illustrated in Fig. 2, a goal–plan tree is rooted by a goal and consists of alternating layers of plan and goal nodes. We call the root node the *top-level goal*. The operator on a goal node is disjunctive, indicating that any of the child plans will achieve the goal (if successful). The operator on a plan node is conjunctive, indicating that all of the child subgoals must be achieved, in order for the goal to be successful. This goal–plan tree structure can be used for various deliberation techniques. For example, Thangarajah and Padgham [49] use the goal–plan tree for detecting potential interference and synergies between goals of the agent. In our work we use the goal–plan tree to determine the order in which goals and plans are to be aborted, suspended and resumed.

2.2 Related work

We next survey the literature to provide further background to our contribution and to situate our approach. Table 1 gives an overview comparison of goal handling functionality in agent programming languages.

Failure handling In agent platforms such as GOAL [17], GORITE [37], JACK [52], Jadex [31], Jason [4], and SPARK [26], a library of plans is available for the agent to use. Since plans can contain subgoals, and goals are achieved by executing plans, the failure of goals and plans are mutually interdependent. The main difference between goal failure and plan failure is that when a plan fails, the agent may seek alternative plans to achieve the same goal. The standard behaviour is that when all possible plans for a given goal have been exhausted, then the goal fails.

As an alternative, if the agent is endowed with the ability to generate plans, i.e., new plans that are not in the plan library, then it can attempt to plan upon failure of pre-existing plans—either from first principles [43] or using Hierarchical Task Network planning, as in the abstract agent language CANPLAN2 [40].

As noted in the introduction, platforms such as JACK [52] associate a *failure-method* with each plan and is called upon when a plan fails for reasons described above. The method contains actions (and/or subgoals) as defined by the developer which are intended as clean-up actions. For instance, as a variant of our earlier Manscript Submission scenario, if a plan to obtain clearance for a paper fails due to a legal problem, the failure methods may cancel any travel reservations that may have already been made.

The Jadex platform [6,30,31] similarly associates methods with plan events: success ("passed"), failed, and aborted. The agent designer can override these Java methods in order to perform clean-up actions. Goal failure is handled via a Java exception, which can be caught and thus behaviour (e.g., retry) specified.

The Jason platform [4,5,18] implements an extended variant of the AgentSpeak semantics [33]. Jason relies on internal events and internal actions for some of its mechanisms, such as checking or dropping current goals or intentions. Jason generates a 'goal deletion' event upon plan failure; whereupon a 'clean-up' plan can be executed prior to, e.g., attempting another plan to achieve the goal for which the original plan failed. A failed goal deletion plan causes the goal deletion plan itself, and its triggering parent goal, to be dropped.

The approach to failure-handling of Chessell et al. [8] aims for "compensation" of transactions in a business process that fail or encounter an exception. They attach one or more *compensation activities* to a transaction, and present a semantics that also accommodates selective compensation. They assume that these failure methods do not themselves fail. If they do fail, similarly to Jason, the agent simply stops executing the failed failure method with no further deliberation.

In platforms such as SPARK [26] and to some extent Jadex, the agent designer can specify an alternative behaviour for failed goals and plans, and for failed failure-methods, by means of *meta-level* procedures.

Aborting Aborting a goal or plan differs from crudely dropping it, in that the agent attempts any clean-up that might be necessary. In earlier work, we developed a semantics for aborting goals and plans [45], and later a separate semantics for suspending and resuming goals and plans [46], on the assumption that aborting does not occur in the suspend-and-resume mechanism. We build on these earlier works, for the first time a treatment of all these mechanisms for goal manipulation in one semantics. Further, we now handle attempted resumption of a goal that has pre- or in-conditions attached.

Table 1 Comparison of goal handling functionality in agent programming languages

	Drop	Fail method	Fail	Abort method	Abort	Suspend method	Suspend	Resume	Resume method
GOAL [17]	Yes	Yes	No	Yes	No	No	No	No	No
GORITE [37]	Yes	Yes	No	Yes	No	Yes	No	Yes	No
JACK [52]	Yes	Yes	Yes	Yes	No	No	No	No	No
Jason [4]	Yes	Yes	Yes	Yes	No	Yes	No	Yes	No
Jadex [31]	Yes	Yes	No	Yes	No	Yes	No	Yes	No
SPARK [26]	Yes	Yes	Yes	Yes	Yes	No	No	No	No
This article (CAN)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

The approach to aborting in Jason is typical of agent platforms. Jason does not have abort as such, but does have a `fail_goal` construct which can be used within a plan to force the failure of the parent goal. Thus aborting is, in a sense, reduced to the case of failure. Clean-up must be manually coded and the clean-up code manually invoked.

Suspending Suspending a goal is recognized by Braubach et al. [6] and later by Van Riemsdijk et al. [11,36] as an important operation in a conceptualization of goals, alongside adopting and dropping a goal. An implemented operational approach based on Braubach et al. is found in the Jadex system [31], while Van Riemsdijk et al. do not consider plans, and do not attempt to develop an operationalization.

Java-based Jadex distinguishes three sub-states for adopted goals: *Option*, *Active*, and *Suspended*. State *Option* pertains to goals considered "desirable" by the agent, but the agent only actively pursues goals when they are placed in state *Active*. Suspension occurs (automatically) when the goal's context condition becomes false, and "holds as long as the context stays invalid." Our proposal grants the agent's reasoning to suspend a goal for other reasons, besides necessity due to its context condition.

Jason provides `suspend` and `resume` constructs, which can be used to suspend and resume (respectively) a goal and the plans which are currently executing to achieve it. However, Jason does not provide *abort-*, *suspend-* and *resume-methods* (see Sect. 3 below), which means that our approach is more general than the mechanisms provided by Jason.

Also Java-based, GORITE [20,37] allows goal states `PASSED` (i.e., succeed), `FAILED`, `STOPPED` (i.e., inactive), `BLOCKED` (i.e., suspended), and `CANCELLED` (i.e., aborted). State of a top-level goal is typically (re-)set in the plans for that goal, such as writing `return Goal.States.STOPPED;`. GORITE's mechanisms provide programmatic control to the agent designer, without a formal theoretical basis.

The previously-cited, operational approach of Thangarajah et al. [46] attaches suspend- and resume-methods to goals and plans. As in the case of aborting, default methods are provided in the cases where no dedicated goal- or plan-specific reasoning is required.

The question of what to do when resuming a suspended plan, should circumstances have changed, has been considered in the AI planning literature (e.g., [3]). On the one hand, an agent may appraise world state to see if the plan is still applicable. If not, on the other hand, it may abort the plan and start executing a new plan, or it may attempt to repair the plan to adapt it to the new situation.

Goal deliberation In our present work we develop a principled mechanism for aborting, and suspending and resuming, goals and plans. From a broader point of view, an intelligent agent not only requires mechanisms to manipulate such constructs, but also a (meta-)reasoning ability to decide what such goal and plan management 'actions' to perform, and when. This topic of goal deliberation was included from the early days of the BDI framework [33,35] and study of this reasoning question is ongoing [9,15,23,49,53].

3 Aborting, suspending, and resuming goals and plans

In this section we informally describe the mechanisms for aborting, suspending, and resuming goals and plans and show how they enable important behaviour in agent-based systems. To introduce the representations that are necessary to incorporate these concepts, we begin by examining the current failure handling mechanism in BDI-style systems. We then provide a set of illustrative vignettes, before describing the mechanisms themselves.

The mechanisms for suspending (and resuming) goals and plans are complementary to those that handle failure and aborting of goals and plans. The key aspects that distinguish between failure, aborting, and suspending 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 (if in some cases necessary), the result of some high level deliberation of the agent. The command to abort or suspend or resume can be made at any arbitrary point in the execution hierarchy. Before a goal or plan is aborted, all of its subgoals and subplans need to be aborted; that is, a bottom-up approach is taken. When a goal or plan is suspended (respectively resumed), by contrast, the approach is top-down; the suspend (respectively resume) method of the goal or plan is first executed and then the children are suspended (respectively resumed).

While the mechanisms for suspending and resuming and for failing and aborting goals and plans are thus complementary, they are not independent. That is, a mechanism for aborting a goal cannot operate in isolation to a mechanism for suspending a goal; otherwise, unintended side-effects can occur, leading to inconsistent states.

Abort, suspend and resume reasoning methods Failure of a goal or plan, like aborting, cause the execution of the goal or plan to cease and, consequentially, the agent to reflect over its current goals and plans. As just described, the difference between failing, on the one hand, and aborting, suspending, and resuming, on the other hand, is in the way they arise. In the case of the failure operation, the trigger to cease execution of a goal or plan comes from below in the goal–plan tree, due to the failure of (sub)plans or subgoals. In the case of the other operations, the trigger comes from above, from the parent (or ancestor) GPT nodes that initiated the goal or plan.

For example, in the scenario of Sect. 1, when Alice decides to place her paper writing on hold, she does so by suspending the Support Manuscript Submission (SMS) goal. This in turn triggers the suspension of the plan SMSPLAN, which in turn triggers the suspension of the parallel goals OC and TWP, and so on down to the leaves of the goal–plan tree. In this way suspend operates top-down; we first make the task at the top-level inactive, and only then proceed recursively to the children. Suppose that a problem means that an action in the plan for OC fails, and suppose the failure is significant enough such that the plan fails and the goal fails. From the bottom of the tree upwards, then, OC fails, SMSPLAN fails, and SMS fails.

It is important to note that the original decision to abort, suspend or resume a goal or plan may be due to a number of reasons. As discussed above, the suspension of the parent of a goal will require the suspension of any children of the goal. The suspension of the parent may be due to the suspension of its own parent, or because the agent has decided to suspend this goal in favour of another. Hence whilst suspension of a parent will necessarily imply the suspension of any children, the original signal to suspend may come from a number of sources (see Sect. 4 for more discussion on this point).

A common feature of all the above operations, however, is that clean-up actions may be required when a goal or plan fails, is aborted or suspended, or when a goal or plan is resumed from being suspended. Our approach therefore associates an *abort-method*, *suspend-method* and *resume-method* with each plan much like the *failure-method* in platforms like JACK. These methods enables the agent developer to specify dedicated compensation actions according to how the agent is attempting to perform the plan or goals above it. Note that these methods can be arbitrary programs (see Sect. 4.1 for a precise definition) and so can invoke goals and plans 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 assumption here is that, like the failure-method, the designer of the agent system has the opportunity to specify a sensible clean-up method that takes into consideration the point in the plan at which the abort, suspend, or resume is to be executed. For any plan, the clean-up method is optional: if no clean-up 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 goal when aborting a plan.

An alternative to attaching clean-up methods to each plan is to attach such methods to each atomic action. We choose the former because: (1) action-level clean-up 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-based systems, and (3) the clean-up methods for failure in current platforms are attached to plans.

Note that an explicit representation of the clean-up methods for goals is not required, since goals are achieved by executing some plan or plans. Hence, aborting, suspending or resuming a goal means aborting, suspending or resuming the current plan that is executed to achieve that goal, as we describe in the sequel.

3.1 Motivational vignettes

In order to understand how the agent's abort, suspend, and resume reasoning should function, we consider five situations, based on the Manuscript Submission scenario of the introduction (Sect. 1.1). These five vignettes illustrate situations where it is sensible for an agent to consider aborting, suspending, or resuming some of its goals or plans, and highlight some of the complexities that the agent's reasoning must accommodate.

1. When an event alters the importance of an existing goal or plan, the agent should deliberate over whether the existing plan(s) should continue. For example, suppose that Alice tasks CALO with a new, high-priority goal to purchase a replacement laptop, but that Alice lacks enough funds to both purchase the laptop and to attend the conference. Reasoning over resource requirements [27,48,50] will cause the agent to realize that it cannot successfully complete both goals. Given that the new goal has greater importance, a rational agent will evaluate the best course of action and suggest, in this case, that Alice abort an existing Support Manuscript Submission goal. To abort the Support Manuscript Submission goal and its associated plan, the agent needs to abort the Obtain Clearance and Track Writing Paper subgoals. Aborting the plan for Obtain Clearance has the special requirement of invoking the Cancel Clearance Request plan. The goal–plan tree applicable here is shown in Fig. 3. Note that execution of abort-methods proceeds in a bottom-up manner; in particular the abort-method for OCPLAN1 will be executed before any abort-method for SMSPLAN.
2. When circumstances change it may be necessary to suspend a goal. For example, as described in the introduction, Alice instructs her CALO agent to suspend the Support Manuscript Submission goal, which in turn suspends the associated plan and is subgoals, Obtain Clearance and Track Writing Paper. Suspending the plan for Obtain Clearance also has the special requirement of invoking the Cancel Clearance Request plan, because the company's policy is that clearance approvals may only be held by active works in progress. Note that execution of suspend-methods proceeds in a top-down manner; in particular, any suspend-method for SMSPLAN will be executed before any suspend-method for OCPLAN1. The goal–plan tree applicable here is shown in Fig. 4.
3. A suspended goal may need to be resumed. For example, when Alice instructs her CALO to resume the Support Manuscript Submission. Here resuming the plan for Obtain

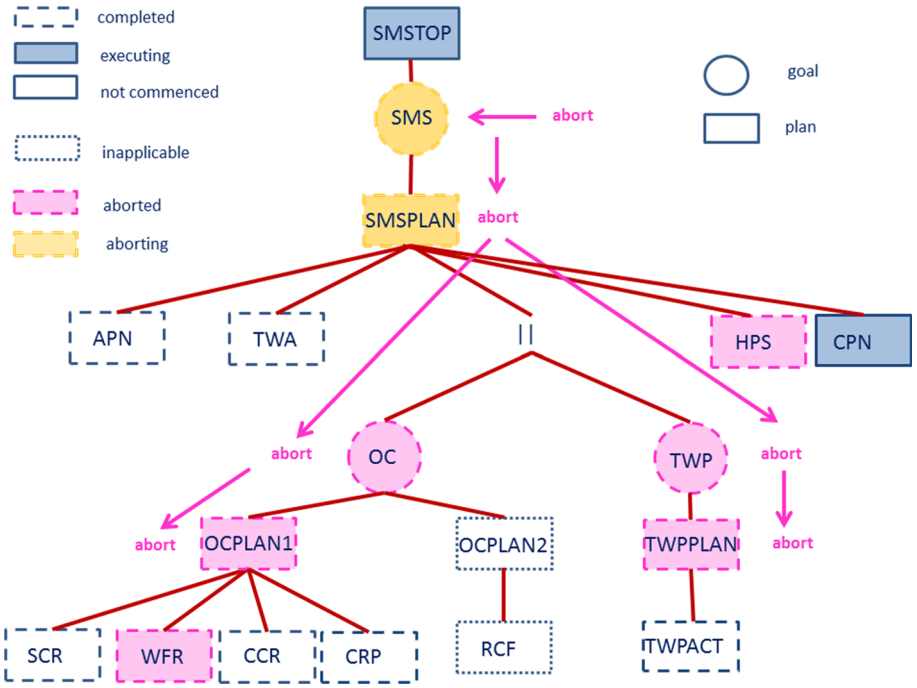

Fig. 3 Goal–plan tree for Vignette 1

Clearance has the special requirement of re-invoking the Send Clearance Request plan. The goal–plan tree applicable here is shown in Fig. 5.

4. A suspended goal may need to be aborted. For example, the request to abort the Support Manuscript Submission goal may occur while the goal is suspended. Here rather than the resume-method for the Obtain Clearance plan being invoked, we need to invoke the abort-method (Cancel Clearance Request, which in this case has already been achieved). The goal–plan tree before the abort is shown in Fig. 6, and after it the goal–plan tree is the same as in Vignette 1, i.e. Fig. 3.
5. When a goal succeeds or fails because of an external factor other than the agent itself, the plan currently being executed to perform the goal should be aborted. For example, suppose Alice is promoted during the process of writing the paper, and that employees of Alice's (new) seniority automatically have clearance for publishing papers. Since Alice no longer needs her manager's clearance for publishing her paper, CALO can abort the *plan* for Obtain Clearance. In doing so it must invoke the abort-method, in this case thus performing Cancel Clearance Request.[2] The Obtain Clearance *goal* is still active, but now it can use a new alternative plan for senior employees that consists solely of the action Register Clearance Form. The goal–plan tree applicable here is shown in Fig. 7.

Observe that the first situation above involves deliberating over the importance of a goal or plan, which depends on various factors such as priority (see, e.g., [21]) and completeness

[2] If there is any difference between how to abort a goal 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.

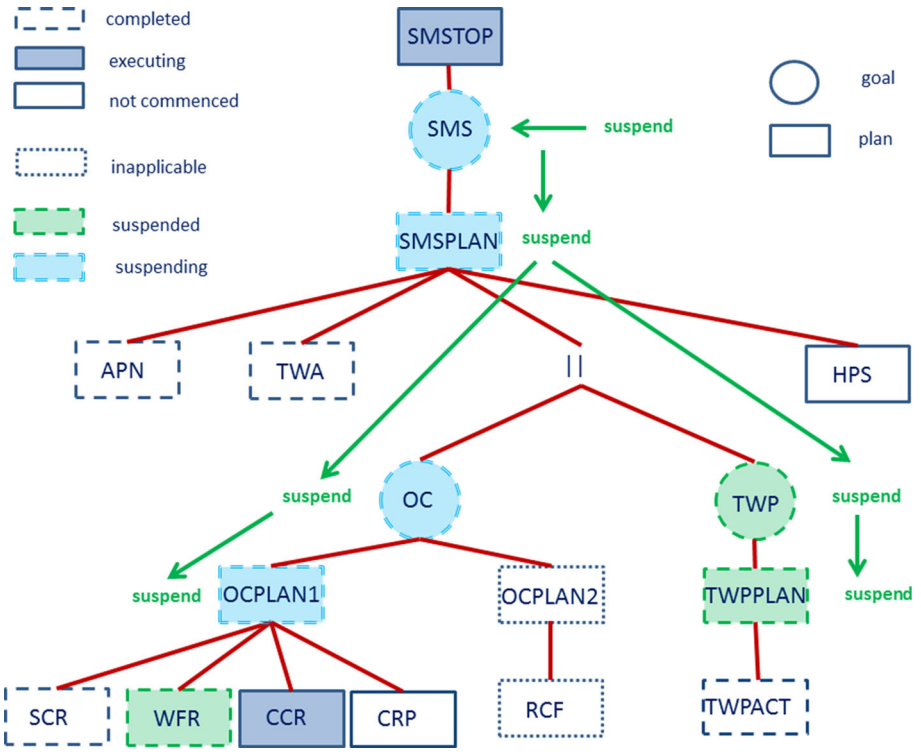

Fig. 4 Goal–plan tree for Vignette 2

([48]). Although this deliberation is beyond the scope of this paper, it is a complementary topic of our future work.

Once a decision has been made to abort, suspend, or resume a goal or plan, the operational semantics we provide in Sect. 4 for aborting, suspending, and resuming describes what happens to the agent's execution of those goals and plans.

We remark that the above situations apply to *achievement* goals, for which the goal is completed when a particular state of the world is brought about (e.g., ensure Alice has clearance). Different forms of reasoning apply to other goal types [11,36] such as maintenance goals. Extending our work to maintenance goals is a future undertaking.

To illustrate the mechanisms below we use our running Manuscript Submission example shown in Fig. 2. In the figure, goals and plans are distinguished by the shape; completed (i.e., successful), currently executing, and future goals and plans are indicated respectively by the border being dashed, bold (and the shape shaded), or light.

3.2 Aborting goals and plans

The intent of aborting a goal or plan is that the task 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, an attempted 'undo' process may cause adverse effects. Therefore, the clean-up methods that we specify are forward recovery procedures that attempt to ensure a stable

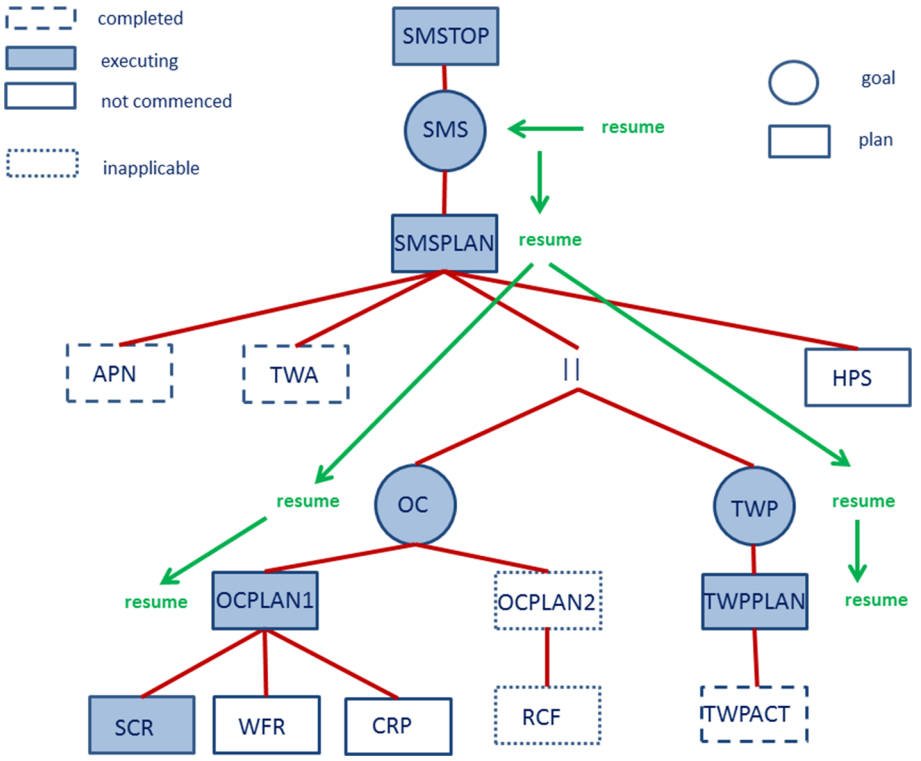

Fig. 5 Goal–plan tree for Vignette 3

state and that also may, if possible, recover resources. In our semantics, we assume that primitive actions are atomic (although they may have some duration) and cannot be aborted (or suspended), i.e., we allow any currently executing action to complete. Note that one reason that a plan may be aborted is due to a failure of its in-condition.

We now informally lay out the agent's action upon aborting goals and plans.

Plans When a plan P is aborted:

1. Abort each subgoal 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 goal of P. We note here that if the parent goal T_P is not to be aborted then the agent *may* choose another applicable plan to satisfy T_P.

Goals When a goal (or subgoal) G is aborted:

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

In order to prevent infinitely cascading clean-up efforts, we assume that abort-, suspend-, and resume-methods will never be aborted, fail, or be suspended; other than this, they

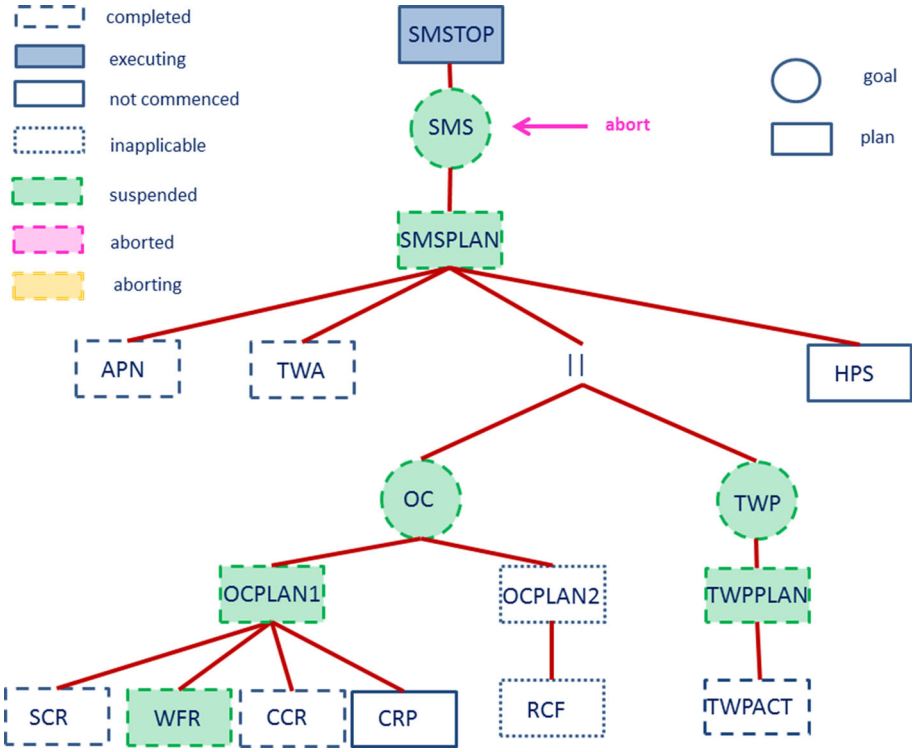

Fig. 6 Goal–plan tree for Vignette 4 (before the abort)

can consist of any legal plan or goal. In reality, however, such methods may fail. In this case—lacking a more sophisticated handling mechanism—the agent simply stops executing the failed method with no further deliberation. While the assumption we make is not a reflection of the full complexity of reality, it is pragmatic in terms of the agent execution cycle. As we saw in Sect. 2.2, such an assumption is not uncommon. As we also saw, platforms such as SPARK allow the agent designer to specify an alternative behaviour for a failed failure-method, and a similar mechanism applies to abort-, suspend-, and resume-methods. We also assume that failure-, abort-, suspend-, and resume-methods always terminate.

Example For example, consider Vignette 1 (see Fig. 3). The abort proceeds top-down from the SMS goal. It has a single active plan, the SMS plan. This plan completed the action Allocate Paper Number and the subgoal Track Writing Abstract, and currently has two active subgoals in parallel, OC and TWP (see Fig. 2), each of which has a single active plan. These two plans have no subgoals. Hence, the agent reaches the bottom of the abort 'trace' and operates bottom-up as follows:

1. Invoke abort-method of plan OCPLAN1. Its abort-method is CCR.
2. Indicate failure of the plan to goal OC.
3. Invoke abort-method of plan TWP. It has a default abort-method.
4. Indicate failure of the plan to goal TWP.
5. Drop goal OC.

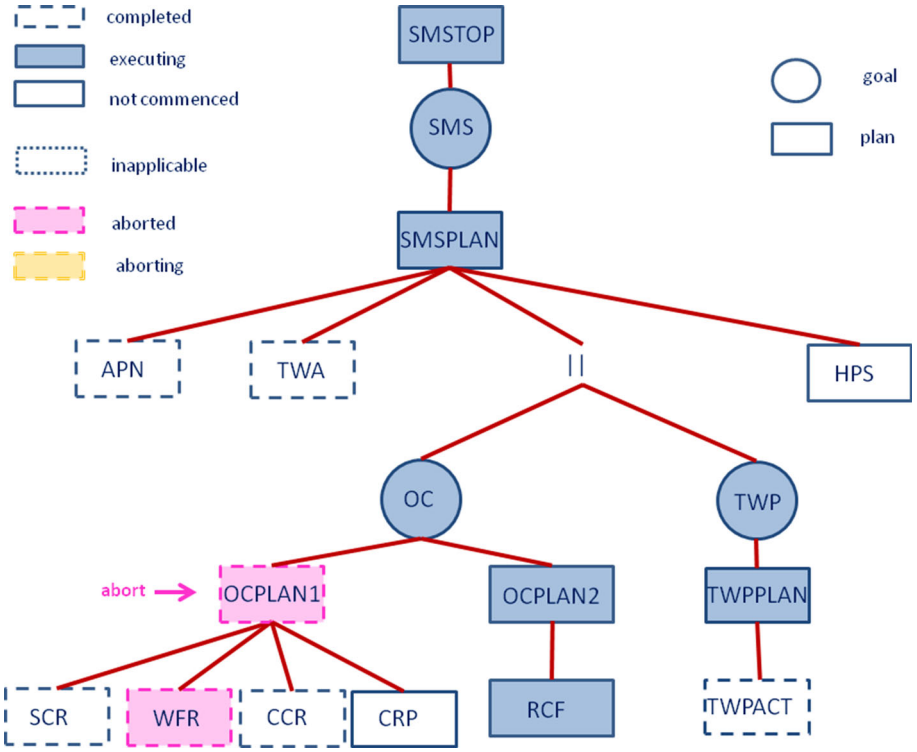

Fig. 7 Goal–plan tree for Vignette 5

6. Drop goal TWP.
7. Invoke abort-method of plan SMSPLAN. Its abort-method is CPN (i.e. Cancel Paper Number).
8. Indicate failure of the plan to goal SMS.
9. Drop goal SMS.

3.3 Suspending and resuming goals and plans

The intent of suspending a goal or plan is that the task is 'paused'; the intent of resuming is that a paused task is returned to execution, although not necessarily at the same state that it was suspended (since the world state may have changed in the interim).

We allow goals and plans to be tagged with either *inactive* or *suspended*, both of which indicate that work on the goal or plan should not proceed; *suspended* also indicates that the goal or plan has been properly suspended as described below. 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.

These mechanisms will be given precise semantics in the next section, as part of our holistic semantics that includes aborting, and accounts for the possible interplay between aborting and suspending or resuming.

3.3.1 Suspending goals and plans

When a task T is to be suspended, we first pause the goal or plan and all its currently active subgoals and subplans 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 goal or plan T, until the agent deliberates and acts on the consequences of the suspension decision, which may include suspending, aborting, terminating, or continuing execution of different subgoals and subplans. Until this deliberation is complete, the lower level goals and plans should not execute; hence the necessity of pausing them.

The second operation is to suspend the task T 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 goal or plan may, for instance, abort all its lower level goals and plans.

For example, if the SMS goal is to be suspended all the goals and plans indicated as active in Fig. 2 are first paused, and then they are suspended top-down beginning with the SMS plan.

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:

1. Mark the goal as suspended.
2. If the goal has not already begun execution—that is, it is in the current execution stack, but no plan to achieve the goal has been selected yet—then no further steps are required by default.
3. Otherwise—that is, there is a plan in the agent's intention stack associated with the goal—then suspend that plan.

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:

1. Mark the plan instance as suspended.
2. If the plan has no dedicated suspend reasoning method attached, perform the following default procedure:
   - Save the current state of the plan.
   - Do not start executing the next step. This is already taken care of for subgoals as they will have been marked as inactive. Any actions should not be attempted.
   - Suspend all current steps of the plan currently in progress. There may be more than one step since some may be executing in parallel:
     - If the step is a subgoal, suspend it (as above).
     - If the step is an action (including any belief operations, i.e., additions, deletions, or modifications of the agent's belief base), it is allowed to complete as it is considered as atomic.
3. Otherwise—if there is a dedicated suspend reasoning method attached to the plan, then call this method. 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, the agent might abort the plan if the goal 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 above.

- Plans to release resources and perform other clean-up actions. For example, in suspending the Obtain Clearance plan the agent need to perform a Cancel Clearance Request action to stop the clearance request from progressing.

4. If the plan instance is still inactive, mark it as suspended and then suspend all current steps of the plan marker as inactive. There may be more than one step since some may be executing in parallel:
   - If the step is a subgoal, 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.

Example For example, consider Vignette 2 (see Fig. 4). The suspend proceeds top-down from the SMS goal. It has a single active plan, the SMS plan. This plan completed the action Allocate Paper Number and the subgoal Track Writing Abstract, and currently has two active subgoals in parallel, OC and TWP (see Fig. 2 again), each of which has a single active plan. These two plans have no subgoals. Hence, the agent operates top-down as follows:

1. Tag goal SMS and all its children as inactive.
2. Mark the goal as suspended.
3. Mark plan instance SMS as suspended.
4. Execute the default suspend-method on it.
5. Plan SMS has two current active steps, the goals OC and TWP.
6. Mark goal OC as suspended.
7. Mark plan instance OCPLAN1 as suspended.
8. Execute its suspend-method, which invokes CCR.
9. (Plan OCPLAN1 has one current active step, the wait Wait For Response.)
10. Mark goal TWP as suspended.
11. Mark plan instance TWPPLAN as suspended.
12. Execute the default suspend-method on it.

3.3.2 Resuming goals and plans

When a suspended task T is resumed, the default behaviour is to resume the goal or plan. That is, the agent will attempt to resume execution of T 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 designer and the application domain, and if required may be implemented as optional meta-reasoning that overrides the default resumption behaviour.

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

1. Unmark the goal, thereby allowing the goal to progress once any suspended plan instance for it is resumed.
2. If there is a plan instance associated with the suspended goal, then resume that plan. For example, if the SMS goal is resumed then the SMSPLAN plan is resumed.
3. Otherwise—when there is no plan currently selected—re-enable the plan 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:

1. Mark the plan as active.
2. If there is a dedicated resume reasoning method attached to the plan, then call this method.
3. Otherwise—if the plan has no dedicated resume-method attached, perform the following default procedure, beginning by checking the in-conditions of the plan;
   (a) If the in-conditions are true:
      – Allow the next steps of the plan to proceed.
      – Resume any suspended subgoals using the procedure for resuming goals above.
   (b) If the in-conditions are false[3]:
      – Abort the plan.
      – Retry alternative plans if they exist for the top level goal.[4]
4. If the plan is not aborted, unmark the plan, thereby allowing the plan to progress once any suspended subgoals are resumed, and then resume any suspended subgoals 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.[5] As with suspend-methods, a dedicated resume-method is the means to override the default behaviour.

For example, consider the suspended OCPLAN1 plan in Fig. 2. When suspended, the Cancel Clearance Request action was performed, as per company policy. To resume the plan, the agent needs to perform the Send Clearance Request action again.

Note that in-conditions for a plan are different from pre-conditions. Pre-conditions are only required to be true at the point of plan selection, and are not relevant after that point. In-conditions are required to be true at all points in the execution of the plan, and, in particular, on resumption. If the in-condition becomes false at any point (including when the plan is suspended), then the plan is aborted.

Note further that our handling of in-conditions as a trigger for an abort means that, when a plan has been suspended and is resumed, following the above steps, there is no strict need to check in-conditions on resumption. However, it makes sense to explicitly do so on resumption, as there is no point in resuming only to find that the plan should be aborted. Note that the above applies in the case that the plan has no dedicated resume-method, i.e., it is the default behaviour. In the case that the plan does have a resume-method, then the resume-method is executed without first a check of the in-condition. We assume that the resume-method will check the in-condition, and if false it may abort the plan or it may try to repair/adapt the plan. In this way we give the agent designer more flexibility. If the resume-method does not check the in-condition, then the in-condition will be verified in any case as soon as the plan resumes execution.

Example For example, consider Vignette 3 (see Fig. 5). The resume proceeds top-down from the SMS goal. It has a single suspended plan, the SMSPLAN plan. This plan has two suspended subgoals, OC and TWP (see Fig. 2), each of which has a single suspended plan. Hence, the agent operates top-down as follows:

1. Unmark the goal SMS (i.e., remove its suspended mark).
2. Mark the plan SMSPLAN as active.

[3] As we note below, the agent checks in-conditions for falsehood on a regular basis, not just on resumption.

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

[5] If the necessary resources cannot be obtained, the resume-method can fail, upon which by default the agent drops the goal. More elegantly, since we assume that resume-methods should always succeed, the method can indicate to the agent's meta-reasoning to abort the goal.

3. Execute the default resume-method on it.
4. Check the in-condition of plan SMSPLAN. If it is false, then abort SMSPLAN.
5. Unmark the goal OC.
6. Mark the plan OCPLAN1 as active.
7. Execute its resume-method, which calls action SCR.
8. (Once action SCR succeeds) Make the next step in plan OCPLAN1, i.e., the wait, active.
9. (Meanwhile, in parallel) Unmark the goal TWP.
10. Mark the plan TWPPLAN as active.
11. Execute the default resume-method on it.
12. Make the next step in plan TWPPLAN active.

A similar process is followed if the suspended goal is aborted rather than resumed.

Example For example, consider Vignette 4 (see Figs. 3, 6). The abort proceeds top-down from the suspended SMS goal. Apart from various goals and plans in the tree being in the suspended state rather than the active one, the process is the same as for Vignette 1.

Vignette 5 shows a slightly different process, in that a plan is aborted rather than a goal.

Example Consider Vignette 5 (see Fig. 7). The abort proceeds top-down from the OCPLAN1 plan, for which the in-condition fails, and so the plan is aborted. This plan contains the action SCR, which has been completed, the action WFR which is currently being executed, and CPR which has not yet commenced. The agent operates top-down as follows:

1. Abort the action WFR.
2. Invoke the abort-method of plan OCPLAN1. Its abort-method is CCR.
3. Indicate failure of the plan to goal OC.
4. Due to the changed circumstances, the plan OCPLAN1 is now applicable.
5. Execute the plan OCPLAN1.

4 Operational semantics

We now turn to a formal definition of the operations discussed above. A formalization is important because it makes precise the steps that an agent takes to perform the operations. Our approach is to define a transformation which takes an agent program (see Sect. 4.1) as input, and outputs a more specific agent program in which the processes of aborting, suspending and resuming tasks are explicitly defined. In particular, we show how agent programs which can be defined in the formal system CAN [39] can be transformed in such a way that the execution of the transformed program results in the appropriate abort, suspend, and resume operations. In other words, we 'complete' the original program by adding extra parts which define the abort, suspend and resume processes for each task.

4.1 CAN language

CAN is a high-level agent language, in a spirit similar to that of AgentSpeak [33] and Kinny's Ψ [22], 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.

CAN was initially defined by Winikoff et al. [54] and later extended as CANPLAN [41] to include a planning component, and then as CANPLAN2 [39,40] to improve the goal adoption and dropping mechanisms. The extensions also simplified the semantics in the earlier work.

In implemented agent platforms, 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: $\mathsf{Goal}(\phi_s, \phi_f, P)$, 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 abandoned once ϕ_s or ϕ_f become true. This makes $\neg\phi_s \wedge \neg\phi_f$ the equivalent of an in-condition for a goal.

An agent's behaviour is specified by a *plan library* that consists of a collection of *plan clauses* of the form $e : c \leftarrow (i \Rightarrow P)$, 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),[6] i is an in-condition (another logical formula over the agent's beliefs that must be true throughout the plan's execution) and P is the *plan body*.

The plan body is a *program* that is defined recursively as follows:

$$\begin{aligned} P ::= {} & act \mid nil \mid fail \mid +b \mid -b \mid ?\phi \mid !e \mid \phi : P \mid \phi ::P \mid P_1; P_2 \mid P_1 \| P_2 \mid P_1 \triangleright P_2 \mid \\ & \mathsf{Goal}(\phi_s, \phi_f, \{P_1, \ldots P_n\}) \mid (\!|\{\psi_1 : P_1, \ldots, \psi_n : P_n\}|\!) \end{aligned} \tag{1}$$

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 $!e$ is an event[7] that is posted from within the program. The compound constructs are sequencing ($P_1; P_2$), parallel execution ($P_1 \| P_2$), and (sub)goals ($\mathsf{Goal}(\phi_s, \phi_f, \{P_1, \ldots P_n\})$). The execution process will select one of the plans $\{P_1, \ldots P_n\}$ for the subgoal $\mathsf{Goal}(\phi_s, \phi_f, \{P_1, \ldots P_n\})$. When there is only one such plan P, we will often abuse notation and write the goal as $\mathsf{Goal}(\phi_s, \phi_f, P)$.

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. One of the auxiliary compound forms is $(\!|c_1 : P_1, \ldots, c_n : P_n|\!)$, which is a collection of guarded alternatives; if c_i is true, then plan P_i is executed. Typically (but not necessarily) the c_i are mutually exclusive. If they are not, the agent is free to choose any plan P_i whose guard c_i is true. This construct is used to handle events. For any given event e, there will be a corresponding collection of program clauses $e' : \psi_i \leftarrow P_i$. We then use the $(\!|\psi_1 : P_1, \ldots, \psi_n : P_n|\!)$ construct to handle the event by replacing it with the collection of guarded alternatives (see rules *event* and *select* above). 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 fails. $\phi : P$ and $\phi ::P$ are a guard and a blocking guard respectively. When ϕ is true, P is executed in both cases. When ϕ is false, the guard (:) fails, whereas the blocking guard (::) waits until ϕ becomes true. We discuss the two types of guards in more detail below.

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

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

$$\frac{}{\langle \mathcal{B}, nil \parallel P\rangle \longrightarrow \langle \mathcal{B}, P\rangle}\ nil_1 \qquad \frac{}{\langle \mathcal{B}, P;\ nil\rangle \longrightarrow \langle \mathcal{B}, P\rangle}\ nil_2 \qquad \frac{}{\langle \mathcal{B}, nil; P\rangle \longrightarrow \langle \mathcal{B}, P\rangle}\ nil_3$$

$$\frac{}{\langle \mathcal{B}, P; nil\rangle \longrightarrow \langle \mathcal{B}, P\rangle}\ nil_4 \qquad \frac{}{\langle \mathcal{B}, nil \triangleright P\rangle \longrightarrow \langle \mathcal{B}, nil\rangle}\ nil_5 \qquad \frac{}{\langle \mathcal{B}, fail \parallel P\rangle \longrightarrow \langle \mathcal{B}, fail\rangle}\ fail_1$$

$$\frac{}{\langle \mathcal{B}, P \parallel fail\rangle \longrightarrow \langle \mathcal{B}, fail\rangle}\ fail_2 \qquad \frac{}{\langle \mathcal{B}, fail; P\rangle \longrightarrow \langle \mathcal{B}, fail\rangle}\ fail_3 \qquad \frac{}{\langle \mathcal{B}, fail \triangleright P\rangle \longrightarrow \langle \mathcal{B}, P\rangle}\ fail_4$$

$$\frac{\mathcal{B}_1 \models pre(a)}{\langle \mathcal{B}_1, a\rangle \longrightarrow \langle \mathcal{B}_2, nil\rangle}\ act_1 \qquad \frac{\mathcal{B} \not\models pre(a)}{\langle \mathcal{B}, a\rangle \longrightarrow \langle \mathcal{B}, fail\rangle}\ act_2$$

$$\frac{}{\langle \mathcal{B}, nil\rangle \longrightarrow \langle \mathcal{B}, e\rangle}\ nil \qquad \frac{}{\langle \mathcal{B}, +b\rangle \longrightarrow \langle \mathcal{B} \cup \{b\}, e\rangle}\ add \qquad \frac{}{\langle \mathcal{B}, -b\rangle \longrightarrow \langle \mathcal{B} \backslash \{b\}, e\rangle}\ del$$

$$\frac{\mathcal{B} \models \phi}{\langle \mathcal{B}, ?\phi\rangle \longrightarrow \langle B, nil\rangle}\ query_1 \qquad \frac{\mathcal{B} \not\models \phi}{\langle \mathcal{B}, ?\phi\rangle \longrightarrow \langle \mathcal{B}, fail\rangle}\ query_2$$

$$\frac{\mathcal{B}_1 \models \phi \quad \langle \mathcal{B}_1, P_1\rangle \longrightarrow \langle \mathcal{B}_2, P_2\rangle}{\langle \mathcal{B}_1, \phi : P_1\rangle \longrightarrow \langle \mathcal{B}_2, P_2\rangle}\ guard_1 \qquad \frac{\mathcal{B} \not\models \phi}{\langle \mathcal{B}, \phi : P\rangle \longrightarrow \langle \mathcal{B}, fail\rangle}\ guard_2$$

$$\frac{\mathcal{B}_1 \models \phi \quad \langle \mathcal{B}_1, P_1\rangle \longrightarrow \langle \mathcal{B}_2, P_2\rangle}{\langle \mathcal{B}_1, \phi :: P_1\rangle \longrightarrow \langle \mathcal{B}_2, P_2\rangle}\ guard_3 \qquad \frac{\mathcal{B} \not\models \phi}{\langle \mathcal{B}, \phi :: P\rangle \longrightarrow \langle \mathcal{B}, \phi :: P\rangle}\ guard_4$$

$$\frac{\Delta = \{\psi_i\theta : P_i\theta \mid e' : \psi_i \leftarrow P_i \in \Pi \wedge \theta = \mathsf{mgu}(e, e')\}}{\langle \mathcal{B}, !e\rangle \longrightarrow \langle \mathcal{B}, (\!|\Delta|\!)\rangle}\ event \qquad \frac{\psi_i : P_i \in \Delta \quad \mathcal{B} \models \psi_i}{\langle \mathcal{B}, (\!|\Delta|\!)\rangle \longrightarrow \langle \mathcal{B}, P_i \triangleright (\!|\Delta \setminus \{\psi_i : P_i\}|\!)\rangle}\ select$$

$$\frac{\langle \mathcal{B}, P_1\rangle \longrightarrow \langle \mathcal{B}, fail\rangle}{\langle \mathcal{B}, (P_1 \triangleright P_2)\rangle \longrightarrow \langle \mathcal{B}, P_2\rangle}\ \triangleright_{fail} \qquad \frac{\langle \mathcal{B}_1, P_1\rangle \longrightarrow \langle \mathcal{B}_2, P_3\rangle}{\langle \mathcal{B}_1, (P_1; P_2)\rangle \longrightarrow \langle \mathcal{B}_2, (P_3; P_2)\rangle}\ sequence$$

$$\frac{\langle \mathcal{B}_1, P_1\rangle \longrightarrow \langle \mathcal{B}_2, P_3\rangle}{\langle \mathcal{B}_1, (P_1 \parallel P_2)\rangle \longrightarrow \langle \mathcal{B}_2, (P_3 \parallel P_2)\rangle}\ parallel_1 \qquad \frac{\langle \mathcal{B}_1, P_2\rangle \longrightarrow \langle \mathcal{B}_2, P_3\rangle}{\langle \mathcal{B}_1, (P_1 \parallel P_2)\rangle \longrightarrow \langle \mathcal{B}_2, (P_3 \parallel P_1)\rangle}\ parallel_2$$

Fig. 8 Operational rules of CAN

4.1.1 Operational semantics of CAN

A summary of the operational semantics for CAN in line with Winikoff et al. [54] and the later simplifications of Sardiña et al. [39–41] is as follows. The planning-related aspects of CAN are not relevant to our work; we need only the execution-related aspects.

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.

A *transition* $S_0 \longrightarrow S_1$ specifies that executing S_0 for a single step yields configuration S_1. A derivation rule consists of a (possibly empty) set of premises, which are transitions together with some auxiliary conditions (written above the line), and a single transition conclusion derivable from these premises (written below the line).

Figures 8 and 9 gives the operational rules of CAN which are relevant to our work (we omit the planning rules). Rather than present all the rules at once in one larger system, we have found it convenient to introduce the rules for plan execution other than subgoals first, and only then to introduce the subgoal rules. This is due to the conceptual complexity of the goal rules (such as those for dropping a completed goal) compared to those for executing specific plan steps (such as executing two actions in sequence).

The rules in Fig. 8 deal with the execution of plans, including the empty plan *nil* (rules nil_1 to nil_5, *nil*), failures (rules $fail_1$ to $fail_4$), actions (act_1, act_2), belief updates (*add*, *del*), and queries ($query_1$, $query_2$).

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, $\theta = \mathsf{mgu}(e, e')$

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

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

$$\frac{P = P_1 \rhd P_2 \quad \mathcal{B}_1 \not\models \phi_s \vee \phi_f \quad \langle \mathcal{B}_1, P_1\rangle \longrightarrow \langle \mathcal{B}_2, P_3\rangle}{\langle \mathcal{B}_1, \mathsf{Goal}(\phi_s, \phi_f, P)\rangle \longrightarrow \langle \mathcal{B}_2, \mathsf{Goal}(\phi_s, \phi_f, P_3 \rhd P_2)\rangle}\ \mathsf{G}_S \qquad \frac{P = P_1 \rhd P_2 \quad \mathcal{B} \not\models \phi_s \vee \phi_f \quad P_1 \in \{nil, fail\}}{\langle \mathcal{B}, \mathsf{Goal}(\phi_s, \phi_f, P)\rangle \longrightarrow \langle \mathcal{B}, \mathsf{Goal}(\phi_s, \phi_f, P_2 \rhd P_2)\rangle}\ \mathsf{G}_R$$

Fig. 9 Rules for goals in CAN

of e' and the event in 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 $P \rhd \langle\!|\Delta|\!\rangle$ 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. Note that parallel execution is an implicit conjunction; if one of the branches fails, then the parallel execution fails (as specified by rules $fail_1$ and $fail_2$).

We introduce new rules $guard_1$ to $guard_4$, which deal with the guard constructs : and :: in CAN. These are guards and blocking guards respectively, in that when ϕ is false, $\phi : P$ will fail whereas $\phi :: P$ will wait until ϕ becomes true before proceeding. This difference can be seen in the differences between the rules $guard_2$ and $guard_4$. These provide both conditional execution (the : operator) and a mechanism which causes execution to pause until a specific condition is true (the :: operator), and are required in order to be able to suspend and resume goals. We will see how these are used later.

Figure 9 gives simplified rules for dealing with goals, in line with those presented in Sardiña et al. [41]. The first rule states that a goal succeeds when ϕ_s becomes true; the second rule states that a goal fails when ϕ_f becomes 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.

4.1.2 Example of CAN execution

We now show how our example scenario works in CAN (see Figure 2). Recall that the plan for the goal `Obtain Clearance` (or OC) consists of the three sequential tasks `Send Clearance Request` (or SCR), `Wait For Response` (WFR) and `Confirm Response Positive` (CRP), which we will represent as scr, responded::nil and ?positive; +clear respectively, where scr is an action, responded indicates that the response from Alice's manager has been received, positive indicating a positive response and negative a negative one. Note that the waiting behaviour is specified by the use of the :: operator, and the confirmation behaviour by querying whether the response is positive or not. This means that we represent the goal OC in CAN as

Goal(`positive, negative, scr; responded::nil; ?positive; +clear`)

The `Track Writing Paper` (TWP) goal has the single plan (also TWP), and hence can be represented as Goal(`written`, `abandoned`, `twp`). Assuming that the tasks `Allocate Paper Number`, `Track Writing Abstract` and `Handle Paper Submission` (APN, TWA and HPS) are represented as the actions apn, twa, twp and hps respectively, the SMS goal is represented in CAN as

$$\mathsf{Goal}(\texttt{submitted}, \texttt{expired}, \texttt{apn}; \texttt{twa}; (\mathsf{OC} \| \mathsf{TWP}); \texttt{hps})$$

where OC is `Goal(positivescr;responded::nil;?positive;+clear)` `negative` and where TWP is Goal(`written`, `abandoned`, `twp`), as above. We can then use the rules below in order to implement the behaviour of Alice's CALO, as in the scenario, where !*submit* is the event that leads to the initiation of the program corresponding to the plan for the `Support Manuscript Submission` (SMS) goal, !*cleared* is the event that indicates a positive response from her manager, and !*denied* is the event that indicates a negative response.

$$\begin{gathered}\texttt{!submit} : \texttt{true} \leftarrow \mathsf{SMS} \\ \texttt{!cleared} : \texttt{true} \leftarrow \texttt{+responded}; \texttt{+positive} \\ \texttt{!denied} : \texttt{true} \leftarrow \texttt{+responded}; \texttt{+negative}\end{gathered}$$

The first rule is the specification of Alice's behaviour, i.e., once the `!submit` event is received, the goal SMS is adopted. The other two rules specify the changes in the agent's beliefs once the response from Alice's manager is known. A positive response is indicated by the event `!cleared`, which results in the predicates `responded` and `positive` being added to the beliefs. A negative response is indicated by the event `!denied`, which results in the predicates `responded` and `negative` being added to the beliefs.

Given an initial set of beliefs $\mathcal{B}$ for Alice, the event `!submit` is raised, which via the rules *event* and *select* leads to the goal SMS being adopted, so the state of execution is

$$\langle \mathcal{B}, \mathsf{Goal}(\texttt{submitted}, \texttt{expired}, \texttt{apn}; \texttt{twa}; (\mathsf{OC} \| \mathsf{TWP}); \texttt{hps}) \rangle$$

As neither `submitted` nor `expired` is true, this leads via rule G_I to

$$\langle \mathcal{B}, \mathsf{Goal}(\texttt{submitted}, \texttt{expired}, \texttt{apn}; \texttt{twa}; (\mathsf{OC} \| \mathsf{TWP}); \texttt{hps} \triangleright \texttt{P}) \rangle$$

where P is `apn; twa; (`OC‖TWP`); hps` and then via G_S and the the success of apn and twa (rule act_1) to

$$\langle \mathcal{B}, (\mathsf{OC} \| \mathsf{TWP}); \texttt{hps} \rangle$$

This in turn leads to the execution of the goal OC in parallel with the goal TWP (rules G_S and $parallel_1$), and hence to

$$\langle \mathcal{B}, ((\texttt{responded::nil}; \texttt{?positive}; \texttt{+clear}) \| \texttt{twp}); \texttt{hps} \rangle$$

after the success of the action scr (rule act_1). This corresponds to the point in Vignette 2 when Alice instructs her agent to suspend the SMS goal, i.e., Alice is waiting for the response from her manager.

If the response is positive (i.e., the event !*cleared* occurs), then Alice's beliefs are updated to $\mathcal{B}'$ to include this via the *event* and *select* rules (so that $\mathcal{B}' \models \mathsf{positive}$), and so we have (rule $wait_3$, nil_3)

$$\langle \mathcal{B}', (\texttt{?positive}; \texttt{+clear} \| \texttt{twp}); \texttt{hps} \rangle$$

which in turn reduces to (rules $query_1, nil_1$)

$$\langle \mathcal{B}', \texttt{twp}; \texttt{hps} \rangle$$

Assuming that Alice then writes the paper, so that TWP succeeds, and Alice then follows the company internal procedures (so that HPS succeeds), which updates her beliefs to $\mathcal{B}''$, we then have

$$\langle \mathcal{B}'', \texttt{nil} \rangle$$

and as hps has succeeded, we now have $\mathcal{B}'' \models \mathsf{submitted}$, and so Alice's goal of submitting her paper to the conference has succeeded.

4.2 Transformation

We now show how to incorporate failure, abort, suspension and resumption methods into a CAN program.

Overview Given an agent program consisting of a set of rules R of the form $e : c \leftarrow P$ with the in-condition for P being i, our objective is to transform a given set of rules of this form into a corresponding set of such rules that incorporates failure, abort, suspend and resume methods as appropriate. In other words, we want to 'extend' the program given in P above to include the methods P_F, P_A, P_S and P_R, which are respectively the failure-method, abort-method, suspend-method and resume-method.

The new rule, which replaces the one above, behaves in the following way:

- If P terminates successfully, then there is no change in behaviour.
- If P terminates with failure, then P_F is executed.
- If P is aborted, then P_A is executed.
- If P is suspended, then P_S is executed.
- If P is suspended and then resumed, P_R is executed before P resumes execution.
- If i becomes false, then P is aborted.

Roughly speaking, for a given plan body P, we add infrastructure around P to monitor whether it fails or is aborted (in which case the failure or abort plan for P is executed, as appropriate) as well as a parallel goal which monitors suspension and resumption conditions for P.

In order to incorporate just the failure method, we could simply replace P in the above rule with $P \rhd P_F$. However, the behaviour of abort, suspend and resume requires the ability to interrupt the execution at any point, and so we need to be able to 'guard' each step in the execution, so that it can be aborted or suspended at any step.

The basic idea (which we will refine and extend later) is to replace P in the above rule with

$$\begin{aligned} +active; \big(&Goal(\neg active, abort \vee fail, (P; -active) \rhd Fail) \| \\ &Goal(\neg active, false, suspend \vee resume::(Exec; fail))\big) \end{aligned} \tag{2}$$

This construct executes two goals in parallel instead of the single plan P, but with the same termination behaviour. This means that if P succeeds, then both goals should succeed, but if P fails or is aborted, then we should have that $G_1 \parallel G_2$ should fail, for which it is only necessary that one of G_i fail. In the above construct, this is achieved by the failure condition

for the first goal (i.e., $abort \vee fail$), which will become true if P is aborted or fails, and thus ensure that the parallel construct above will fail, as desired.

We now describe rule (2) in detail. It contains two goals. The plan for the first goal executes P. If P succeeds, then the predicate $active$ will be true throughout the execution, and will become false once P terminates, which means that the goal succeeds (due to the success condition being $\neg active$). If P fails, the code $Fail$ is executed, but the goal will still succeed. However, if we ensure that the code in $Fail$ sets the failure condition of the goal to be true, then the goal will fail, as desired. We can also use the same mechanism to 'catch' an abort signal for the goal, so the failure condition of the goal becomes the disjunction of the failure of P and the receipt of an abort for P. There is still some work to do, though, as we need to be able to run either the failure method P_F or the abort method P_A, as appropriate. Hence we use a construct of the form

$$\Big(Goal(\neg active, abort \vee fail, (P; -active) \rhd Fail) \Big) \rhd P'$$

for this, where P' chooses between running the failure method P_F (indicated by $fail$ being true) or the abort method P_A (indicated by $abort$ being true). In either case, as P has not successfully terminated, our replacement for it must fail, and so the final form for this first parallel goal is

$$(Goal(\neg active, abort \vee fail, (P; -active) \rhd Fail)) \rhd (P'; -active; fail).$$

Note also that P' is only executed if P fails or is aborted; if P succeeds, then

$$Goal(\neg active, abort \vee fail, (P; -active) \rhd Fail)$$

succeeds, and so P' is not executed.

The second goal in (2) will do nothing until P is suspended or resumed (because of the guard ::). If P is suspended, $Exec$ will run P_S. If P is suspended and then resumed, $Exec$ will run P_R before resuming P. Note that $active$ remains true when P is suspended, which means that this second goal will only complete when P terminates (successfully or otherwise), i.e., when $\neg active$ is true. Note also that as the failure condition for this goal is $false$ and $Exec$ is followed by $fail$, the plan for the second goal will fail, which means that the plan will be re-run, and hence the waiting behaviour will be resumed due to rule G_I.

The final form of the transformation, given below, is a little more intricate, not just because of the need for an explicit definition of the procedures $Fail$ and $Exec$, but also to take into account the relationships between goals and their parents, as discussed in Sect. 3.

Signals The above discussion gives the general idea of the transformation we detail below. However, we need some infrastructure around the transformation process, particularly due to the relationships between goals and their parents. We define a transformation below that includes extra information about the status of the agent's current goals and plans. As P may contain subgoals, which in turn may lead to the execution of further plans, we need to be able to keep track of the ancestry of a given plan or goal, so that if an ancestor of the current plan being executed is suspended or aborted, then we need to suspend the current plan.

In order to keep track of task ancestry, we add a *context* parameter to the goals and plans in P.

As the only parts of P that can be aborted, suspended or resumed are goals or top-level plans, we generate a new context for each goal in P, and for each plan associated with a goal. For example in Fig. 2, we only need contexts for the goals SMS, OC and TWP and

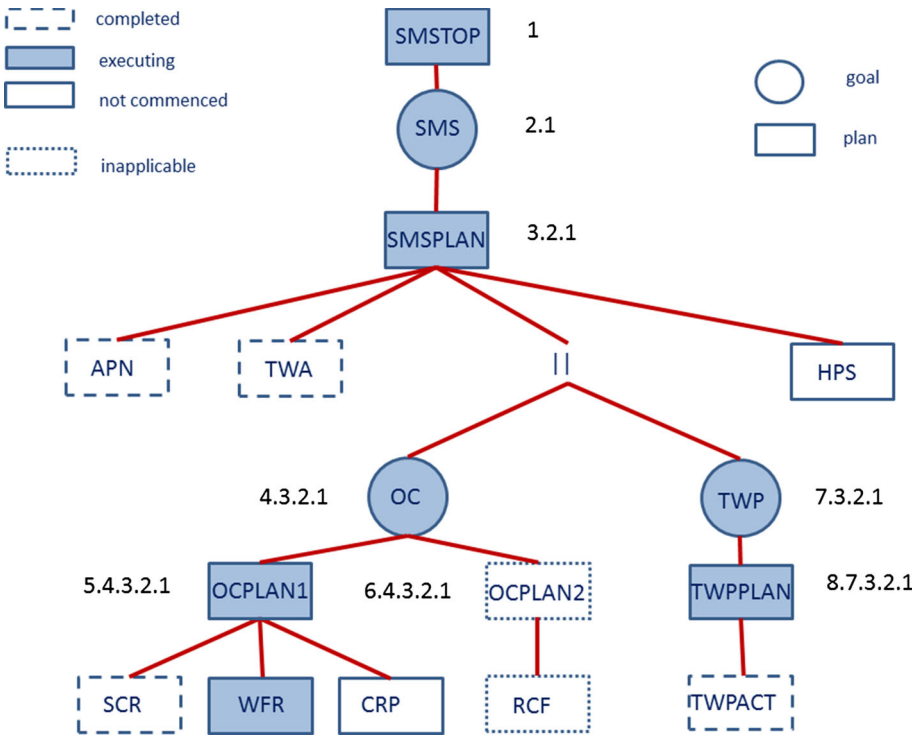

Fig. 10 Goal–plan tree for the scenario (compare Fig. 2) with context labels

plans SMSTOP, SMSPLAN, OCPLAN1, OCPLAN2 and TWPPLAN. This goal–plan tree, updated with context identifiers, can be seen in Fig. 10.

It suffices to use a simple scheme for creating contexts. Each context is a list of identifiers; creating a new context value for execution of a subgoal 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 $n.v$. We will use this mechanism, together with the predicates below, to provide 'signals' to the agent, which will then trigger the appropriate behaviour.

As discussed in Sect. 3, our scope is limited to dealing with the consequences of a decision to abort, suspend or resume a goal, and not to determine when such an action should be taken. Accordingly, we assume the existence of three auxiliary 'signal' predicates $suspend(v)$, $abort(v)$ and $resume(v)$, which indicate a decision by the agent to suspend, abort and resume (respectively) a particular task. This means that when an agent decides to abort a particular task with identifier v, $abort(v)$ becomes true in the agent's beliefs (and similarly for suspend and resume). However, this information needs to be 'filtered' in order to determine a precise sequence of actions to be perform in response to any one of these signals. For this reason we also introduce some auxiliary predicates in addition to the three above.

Of the additional auxiliary predicates, some (i.e., $active(v)$, $failed(v)$, $suspended(v)$) are asserted and retracted as appropriate in the course of the execution of the agent; others (see below) are defined in terms of these and the above three signals. The current state of a given tasks with identifier v is given by $active(v)$, $failed(v)$, $suspended(v)$. When $active(v)$ is true, it means the corresponding task has commenced execution (but may be currently

Table 2 Rules for signals and other predicates

$SC(v)$	$\equiv$ (suspend(v)$\vee$ suspended(parent(v)) $\wedge$ pause(v)
$RC(v)$	$\equiv$ resume(v) $\vee$ ($\neg$ suspended(parent(v)) $\wedge$ suspended(v) $\wedge$ $\neg$ suspend(v))
$AC(v)$	$\equiv$ to_abort(v) $\wedge$ $\neg\exists v'$(v =parent(v') $\wedge$ active(v'))
$FC(v)$	$\equiv$ failed(v)
ready(v)	$\equiv$ $\neg$ pause(v) $\wedge$ $\neg$ suspended(v)
pause(v)	$\equiv$ to_suspend(v) $\vee$ to_abort(v)
to_abort(v)	$\equiv$ abort_decided(v) $\vee$ to_abort(ancestor(v))
abort_decided(v)	$\equiv$ abort(v) $\vee$ $\neg$ in_condition(v)
to_suspend(v)	$\equiv$ suspend_decided(v) $\vee$ to_suspend(ancestor(v))
suspend_decided(v)	$\equiv$ suspend(v) $\wedge$ $\neg$ resume(v)

suspended). If $active(v)$ subsequently becomes false, it means that the task has terminated (with either success or failure). $failed(v)$ is true when the corresponding task has terminated unsuccessfully. $suspended(v)$ is true when the corresponding task is currently suspended. On the resumption of the task, this becomes false. For a task whose execution has commenced but which has not been suspended or aborted, $active(v)$ is true.

Note that it is possible to provide similar behaviour by using events rather than signals and beliefs. We have chosen not take the event-based approach, as using events would require specific rules to handle them, which would have the same effect. As this is an internal agent decision, rather than something triggered by an external event (such as a decision from Alice's manager), we believe it is conceptually simpler to model these as changes in the agent's beliefs.

The remaining predicates are defined in Table 2. Because we do not prescribe a particular form for the agent's beliefs, we give these in terms of logical equivalences. However, these can easily be transformed into rules in the manner of Prolog [19,44], or other similar systems.

In the table, $FC(v)$, $AC(v)$, $SC(v)$ and $RC(v)$ are the failure, abort, suspend and resume conditions respectively for the task v. When this condition is true, this indicates that P_F, P_A, P_S or P_R respectively should be executed. These are the conditions that will be used in the final form of the two parallel goals, discussed in the overview above. Further, in the table:

- $pause(v)$ indicates a task that is to be aborted or suspended, but hasn't been aborted or suspended yet.
- $ready(v)$ indicates a tasks that can be executed now (i.e., it is neither suspended nor marked for suspension).
- $abort\_decided(v)$ indicates either an abort signal is true ($abort(v)$) or that the relevant in-condition has become false.
- $to\_abort(v)$ indicates a task that is to be aborted, possibly due to its parent being aborted. This is not the same as $AC(v)$ as it may be necessary to wait until the task has no active parent.
- $in\_condition(v)$ indicates that the in-condition of the task is true. Hence if this becomes false, the task should be aborted.
- $suspend\_decided(v)$ indicates that a suspend signal is true ($suspend(v)$) and that the task is not to be resumed yet.

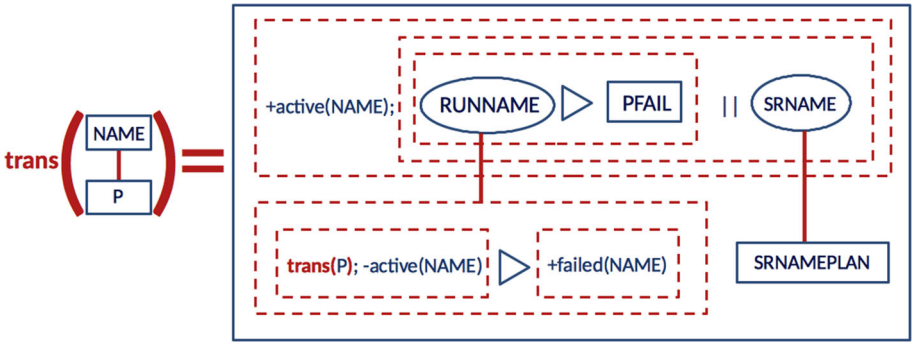

Fig. 11 Translation for a plan

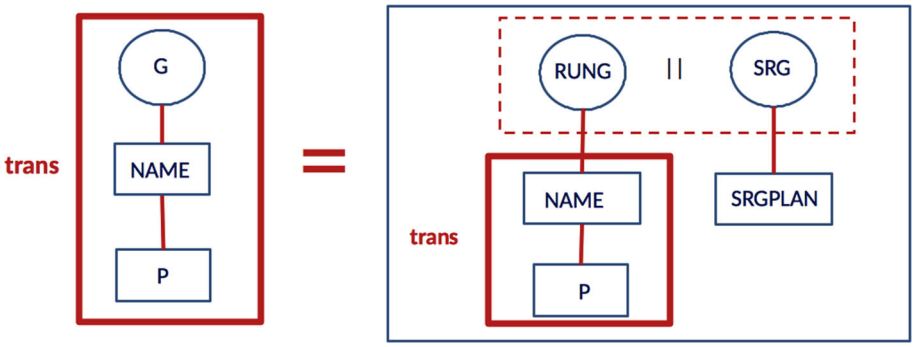

Fig. 12 Translation for a goal

- $to\_suspend(v)$ indicates a task that is to be suspended, possibly due to its parent being suspended.
- $active(v)$, $failed(v)$ and $suspended(v)$ are facts asserted as appropriate.
- $suspend(v)$, $abort(v)$ and $resume(v)$ are used to signal changes.
- $ready(v)$ together with $FC(v)$, $AC(v)$, $SC(v)$ and $RC(v)$ will be used in the definition of the transformation.
- $pause(v)$, $abort\_decided(v)$, $to\_abort(v)$ and $abort\_decided(v)$ are internal, in that they are only used to define $ready(v)$, $FC(v)$, $AC(v)$, $SC(v)$ and $RC(v)$.

Transformation We are now in a position to define the transformation from the set of rules R together with the abort, failure, suspension and resumption procedures as follows. Illustrations of this transformation are given in Figs. 11 and 12. As discussed above, the way this is done is to translate the rules of the program into ones which add infrastructure (in the form of tasks executed in parallel) to provide the correct behaviour on termination, and to monitor signals to abort, suspend and resume.

Each rule $e : c \leftarrow P$ in R is transformed to a new rule

$$e : c \leftarrow transplan(v, P, P_F, P_A, P_S, P_R) \tag{3}$$

where $transplan$ is as defined as follows:

$$transplan(c, P, P_F, P_A, P_S, P_R) = +active(v); \Big(\mathbf{G}(v, P, \langle\!|\{AC(v) : P_A, FC(v) : P_F\}|\!\rangle) \\ \| \ \mathsf{Goal}(\neg active(v), false, (\Pi(v, P_S, P_R); fail)) \Big)$$

where

$$\mathbf{G}(v, P, P') = \Big(\mathsf{Goal}(\neg active(v), AC(v) \vee FC(v), (trans(v, P); -active(v) \triangleright + failed(v))\Big) \triangleright (P'; -active(v); fail)$$

$$\Pi(v, P_S, P_R) = SC(v) \vee RC(v) :: (\!|\{SC(v) : +suspended(v); P_S, RC(v) : -suspended(v); P_R\}|\!)$$

and

$$\begin{aligned}
&trans(v, act) && = ready(v) :: act\\
&trans(v, +b) && = ready(v) :: +b\\
&trans(v, -b) && = ready(v) :: -b\\
&trans(v, nil) && = nil\\
&trans(v, fail) && = fail\\
&trans(v, !e) && = ready(v) ::!e\\
&trans(v, P_1; P_2) && = trans(v, P_1); trans(v, P_2)\\
&trans(v, P_1 \triangleright P_2) && = trans(v, P_1) \triangleright trans(v, P_2)\\
&trans(v, P_1 \parallel P_2) && = trans(v, P_1) \parallel trans(v, P_2)\\
&trans(v, (\!|\{\psi_1 : P_1, \ldots, \psi_n : P_n\}|\!)) && = (\!|\{(ready(v) :: \psi_1) : trans(v, P_1), \ldots, (ready(v) :: \psi_n) : trans(v, P_n)\}|\!)\\
&trans(v, \phi : P) && = ready(v) :: (\phi : trans(v, P))\\
&trans(v, \phi :: P) && = (ready(v) \wedge \phi) :: trans(v, P)\\
&trans(v, \mathsf{Goal}\Big(\phi_s, \phi_f, P\Big)) && = \mathsf{Goal}(ready(v) \wedge \phi_s, (ready(v) \wedge \phi_f) \vee AC(v),\\
& && \qquad transplan(n.v, P, P_F, P_A, P_S, P_R)\Big)\\
& && \quad \parallel \mathsf{Goal}\Big(\neg active(n.v), false, \Pi(n.v, nil, nil); fail\Big)
\end{aligned}$$

As seen above, the plan P in the rule is replaced by the parallel execution of two goals, one of which executes P with additional context information, whereas the other monitors the status of the suspension and resumption of P.

For many cases, the main effect of this transformation is to prefix each execution step with $ready(v)$, so that the execution only proceeds if $ready(v)$ is true, and waits otherwise. For subgoals, we also update the context identifier. The other aspect of note is the transformation for subgoals, which uses the same process as the second parallel goal of the transformation.

Note that there is no need for a translation of the in-condition of a plan, as this is handled by the rules for the signals (see abort_decided(v) above), so that if the in-condition becomes false, the signal abort(v) becomes true.

4.3 Worked example

We now describe how the above transformation operates on the Manuscript Submission scenario. In order to apply the transformation, we first need the context identification scheme. The resultant GPT with context labels is shown in Fig. 10. Second, we need to specify the failure-, abort-, suspend- and resume-methods for the `SMS` plan. These are given in Tables 3 and 4. We have found it convenient to assume that each plan has a unique name (`SMSPLAN,`

Table 3 Goals in the scenario with their success and failure conditions, and related plans

Goal name	Success	Failure	Plan name	Plan
SMS	*submitted*	*expired*	SMSPLAN	`apn; twa; (OC‖TWP); hps`
OC	*positive*	*negative*	OCPLAN1	`scr; responded::nil; ? positive; +clear`
			OCPLAN2	`rcf`
TWP	*written*	*abandoned*	TWPPLAN	twp

Table 4 Plans in the scenario with their failure-, abort-, suspend-, and resume-methods

Plan name	Failure	Abort	Suspend	Resume
SMSPLAN	Nil	CPN	Nil	Nil
OCPLAN1	Nil	CCR	CCR	SCR
OCPLAN2	Nil	Nil	Nil	Nil
TWPPLAN	Nil	Nil	Nil	Nil

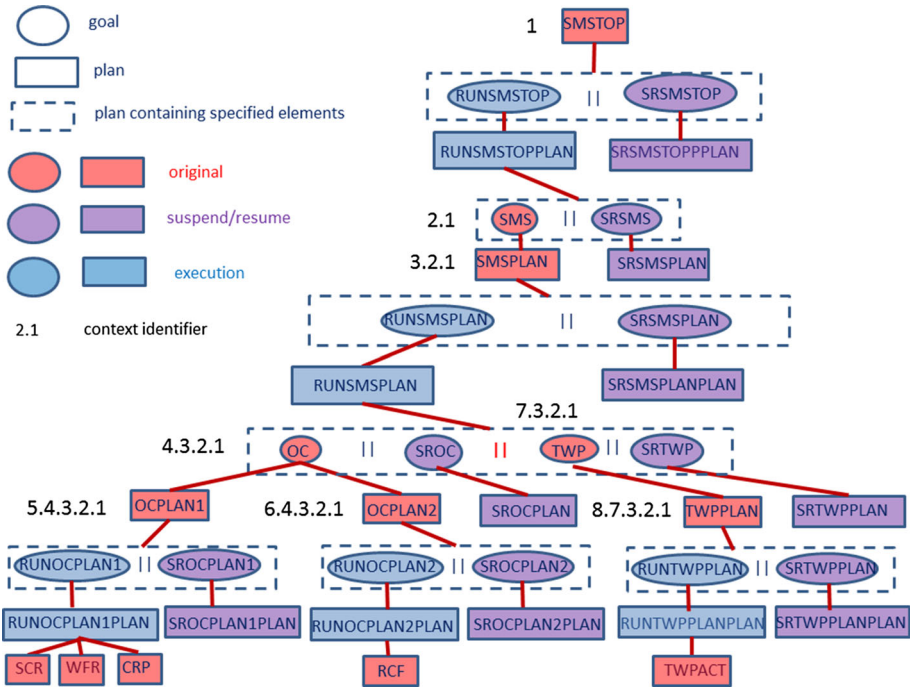

Fig. 13 Goal–plan tree for the translated rules

`OCPLAN1`, `OCPLAN2`, `TWPPLAN`, etc). This makes it simple to specify attributes such as the in-condition, fail-, abort-, suspend- and resume-methods of the plan, and to associate plans with goals.

It should be noted that the transformation of a goal G (named `Goal`) and an associated plan P (named `Plan`) typically results in three extra goals, and three extra plans. One goal is from the second parallel goal in the above transformation, whose role is to monitor the suspend and resume signals for `Goal`, and so we name this goal `SRGoal` (and its associated plan as `SRGoalPlan`). The other two goals come from the transformation of `Plan`, which involves a similar goal to monitor suspend and resume signals for the plan (`SRPlan`) and an associated plan (`SRPlanPlan`), and a further goal `RunPlan`, which is used to monitor abort signals, and whose associated plan (`RunPlanPlan`) contains the body of the transformed plan. As shown in Fig. 13, `RunPlanPlan` may contain further goals and hence continue the goal–plan tree.

Now in our example, there is only the one rule that needs to be transformed, i.e., the one below. The other two rules are only used to process events into belief changes in the agent,

and hence are not processes that will be aborted, suspended or resumed, and hence we do not transform them. The rule

$$!\texttt{submit} : \texttt{true} \leftarrow \mathsf{Goal}(\texttt{submitted}, \texttt{expired}, \texttt{apn}; \texttt{twa}; (\mathsf{OC} \| \mathsf{TWP}); \texttt{hps})$$

is transformed as below

$$!\texttt{submit} : \texttt{true} \leftarrow +\texttt{active}(1); (\mathtt{G}_1 \| \mathtt{G}_2)$$

where $C_2 = 2.1$, $C_3 = 3.2.1$, $C_4 = 4.3.2.1$, $C_5 = 5.4.3.2.1$, $C_6 = 6.4.3.2.1$, $C_7 = 7.3.2.1$, $C_8 = 8.7.3.2.1$ (we will use these names as an abbreviation below), and

$$\begin{aligned}
\mathtt{G}_1 &= \mathbf{G}(2.1, SMS, (\!|\{FC(1) : nil, \neg FC(1) : nil\}|\!)) \\
&= \mathbf{G}(2.1, SMS, nil) \\
&= \mathsf{Goal}\Big(\neg active(C_2), AC(C_2) \vee FC(C_2), trans(C_3, SMSPLAN); \\
&\qquad -active(C_2) \rhd +failed(C_2)\Big) \rhd -active(C_2); fail
\end{aligned}$$

$$\begin{aligned}
&trans(C_3, SMSPLAN) \\
&= trans(C_3, \mathsf{Goal}(\texttt{submitted}, \texttt{expired}, \texttt{apn}; \texttt{twa}; (\mathsf{OC} \| \mathsf{TWP}); \texttt{hps})) \\
&= \mathsf{Goal}\Big(ready(C_3) \wedge submitted, ready(C_3) \wedge expired, trans(C_3, apn; twa; (\mathsf{OC} \| \mathsf{TWP}); hps)\Big) \| \\
&\quad \mathsf{Goal}\Big(\neg active(C_3), false, SC(C_3) \vee RC(C_3) :: \\
&\quad (\!|\{SC(C_3) : +suspended(C_3); nil, RC(C_3) : -suspended(C_3); nil\}|\!); fail\Big) \\
&= \mathsf{Goal}\Big(ready(C_3) \wedge submitted, ready(C_3) \wedge expired, \\
&\quad ready(C_3) :: apn; ready(C_3) :: twa; trans(C_4, \mathsf{OC}) \parallel trans(C_7, \mathsf{TWP}); ready(C_3) :: hps\Big) \| \\
&\quad \mathsf{Goal}\Big(\neg active(C_3), false, SC(C_3) \vee RC(C_3) :: \\
&\qquad (\!|\{SC(C_3) : +suspended(C_3), RC(C_3) : -suspended(C_3)\}|\!); fail\Big)
\end{aligned}$$

$$\begin{aligned}
&trans(C_4, \mathsf{OC}) \\
&= trans(C_4, \mathsf{Goal}(positive, negative, \{trans(C5, OCPLAN1), trans(C6, OCPLAN2)\}) \\
&= \mathsf{Goal}\Big(ready(C_4) \wedge positive, ready(C_4) \wedge negative, \\
&\quad \{trans(C5, OCPLAN1), trans(C6, OCPLAN2)\}\Big) \| \\
&\quad \mathsf{Goal}\Big(\neg active(C_4), false, SC(C_4) \vee RC(C_4) :: \\
&\quad (\!|\{SC(C_4) : +suspended(C_4); nil, RC(C_4) : -suspended(C_4); nil\}|\!); fail\Big)
\end{aligned}$$

$$\begin{aligned}
&trans(C_5, \mathsf{OCPLAN1}) \\
&= trans(C_5, positive, negative, scr; responded :: nil; ?positive; +clear) \\
&= ready(C_5) :: scr; (ready(C_5) \wedge responded) :: nil; ready(C_5) ::?positive; ready(C_5) :: +clear\Big)
\end{aligned}$$

$$trans(C_6, \mathsf{OCPLAN2}) = trans(C_6, rcf) = ready(C_6) :: rcf$$

$$\begin{aligned}
&trans(C_7, \mathsf{TWP}) \\
&= trans(C_7, \mathsf{Goal}(written, abandoned, twp)) \\
&= \mathsf{Goal}(ready(C_7) \wedge written, ready(C_7) \wedge abandoned, ready(C_7) :: twp) \| \\
&\quad \mathsf{Goal}\Big(\neg active(C_7), false, SC(C_7) \vee RC(C_7) ::
\end{aligned}$$

$$(\!|\{SC(C_7) : +suspended(C_7); nil,\ RC(C_7) : -suspended(C_7); nil\}|\!);\ fail\Big)$$

$$\mathtt{G}_2 = \mathsf{Goal}(\neg active(1),\ false,\ SC(1) \vee RC(1) :: (\!|SC(1) : +suspended(1); nil,\ RC(1) : -suspended(1); nil|\!);\ fail)$$

The structure of the resulting GPT is shown in Fig. 13. To simplify the presentation, only the connections between goals and plans are shown; the dashed boxes typically contain actions which do not affect the structure of the tree.

It should be noted that context identifiers are only needed in the goal–plan tree for goals and for top-level plans (i.e., plans with names). This is because these are the only parts of the goal–plan tree which can be aborted or suspended. Actions such as SCR and HPS in Fig. 10 cannot be aborted or suspended, and hence do not need context identifiers.

In the execution of the goal–plan tree in Fig. 13, note that if there are no tasks are suspended or resumed, then none of the plans in purple boxes in the figure (i.e., those whose name commences with SR) are executed. This is because the first step in each plan is a guard, which will only proceed if the suspension or resumption condition becomes true. If there are no aborts, and no plan fails, then the plans in the blue boxes merely ensure that the plans in the red boxes (the original ones) are executed.

Consider Vignette 2 (see Fig. 4 in Sect. 3.1). As shown in Fig. 4, the signal to suspend the goal SMS occurs when the (parallel) goals OC and TWP are executing. When the signal is received, as there is no suspend-method for SMS, the goal is marked as suspended (by inserting the predicate *suspended(2.1)* into the agent's beliefs). This immediately flags (via the rules for *pause(v)* and *ready(v)* in Table 2) that all descendants of SMS are to be suspended, and in particular that the guard *ready(v)* is false for SMSPLAN, OC, TWP, OCPLAN1, and TWPPLAN. This immediately halts the execution all of these, and makes the suspension conditions (*SC(v)* in Table 2) true for OC and TWP. This means that the plans SROCPLAN and SRTWPPLAN become active, which results in the suspend-methods for the goals OCPLAN1 and TWPPLAN being run (SCR and nil respectively) and then the goals OC and TWP are marked as suspended (by inserting *suspended(4.3.2.1)* and *suspended(7.3.2.1)* into the agent's beliefs). This triggers the suspension of OCPLAN1 and TWPPLAN, which is indicated by the insertion of *suspended(5.4.3.2.1)* and *suspended(8.7.3.2.1)*. Due to the instances of *suspended* in the agent's beliefs, none of the active plans can proceed, which thus correctly implements the suspension of the goal SMS.

When SMS is resumed, as in Vignette 3, the plan SRSMSPLAN will re-commence execution, as the resume condition for SMS is now true. This deletes *suspended(2.1)* from the agent's beliefs, thus marking SMS as now active. The definition of *RC(v)* in Table 2 then ensures that *resume(3.2.1)* is true, and so the plan SRSMSPLANPLAN removes *suspended(3.2.1)* from the agent's beliefs. As the in-condition for SMSPLAN is still true, the next step is to resume the goals OC and TWP. As *suspended(3.2.1)* is no longer true, *RC(4.3.2.1)*and *RC(7.3.2.1)* are now both true, and so the plans SROCPLAN and SRTWPPLAN remove *suspended(4.3.2.1)* and *suspended(7.3.2.1)* respectively from the agent's beliefs. This in turn triggers the resumption of OCPLAN1 and TWPPLAN via the execution of SROCPLAN1PLAN and SRTWPPLANPLAN, and hence SCR is executed, as it is the resume-method for OCPLAN1. This in turn means that *suspended(5.4.3.2.1)* and *suspended(8.7.3.2.1)* are deleted from the agent's beliefs, which means *ready* is now true for all contexts, and so the execution of the goal SMS has been successfully resumed.

If the suspended goal SMS is aborted (as in Vignette 4) rather than resumed (as in Vignette 3), then the addition of the signal *abort(2.1)* to the agent's beliefs makes—again

via the rules in Table 2—*to_abort(2.1), to_abort(3.2.1), to_abort(4.3.2.1), to_abort(7.3.2.1), to_abort(5.4.3.2.1)* and *to_abort(8.7.3.2.1)* all true. This means that *ready* is false for all of these contexts, meaning that all execution is paused. Since OCPLAN1 and TWPPLAN have no active children, this means that the abort condition for OCPLAN1 and TWPPLAN are both true (i.e., *AC(5.4.3.2.1)* and *AC(8.7.3.2.1)* are true), and hence the failure conditions for the goals RUNOCPLAN1 and RUNTWPPLAN are both true. This means that *failed(5.4.3.2.1)* and *failed(8.7.3.2.1)* are both added to the agent's beliefs, and due to the rightmost ▷ construct in the definition of **G**, this will execute the abort-methods CCR and nil for the plans OCPLAN1 and TWPPLAN respectively. This also makes OCPLAN1 and TWPPLAN inactive, meaning that the abort conditions for OC and TWP are now both true. As these abort conditions are in the failure conditions of the translated goals, both goals now fail, and are hence made inactive. This makes the abort condition for SMSPLAN true (i.e., *AC(3.2.1)*), and so the goal RUNSMSPLAN fails. Again by the definition of **G**, this means that the abort-method SPN for SMSPLAN is executed, and then SMSPLAN is then made inactive. This in turn triggers the abort condition for SMS, which then fails. This completes the abort process.

4.4 Implementation

We have developed a prototype implementation of our system. This implementation, which we refer to as *Orpheus*, consists of around 1100 lines of Prolog, and includes a generic interpreter for CAN, an implementation of the above translation and some code specific to the scenario involving Alice and her submission. We have implemented the basic scenario (i.e., in which no tasks are aborted or suspended) as well as Vignettes 1–5. The output of our implementation on the basic Alice example is given in Table 5, and on the Vignettes in Tables 6, 7, 8 and 9.

Our interpreter follows the typical *observe-think-act* cycle of an agent system (Fig. 1). The first step is to process any events, which may involve updating the agent's beliefs. Then the status of the agent's goals are evaluated, to see if any goals should be aborted, suspended or resumed. The final stage is then to execute one step of the goal–plan tree, before restarting the cycle again. Due to the presence of the ‖ operator in CAN, there may be several parallel threads executing at the same time in the goal–plan tree. For example, in Fig. 10 there are two parallel threads, one starting with OC and the other with TWP. Hence the execution stage performs one step (if possible) for each of the parallel threads in the tree.

The implementation has been a useful tool for testing our ideas, and is available from the authors.[8] It should be stressed that Orpheus is not intended as (yet another) agent programming language. Rather, it is a tool to be used to explore ways in which aborting, suspending and resuming tasks can be managed, and as a means of providing experimental evaluation. In particular, the main focus of the implementation is to show the behaviour of the translated rules, and that these correctly implement the desired behaviour.

Vignettes Given the transformation of the goal–plan tree in Fig. 13 for our scenario, we now show how each of our motivational vignettes work with this transformed goal–plan tree. All of the executions shown below are a summary of execution traces generated by our implementation, by providing an appropriate input sequence of events to the transformed goal–plan tree for our scenario in Fig. 13. The full execution traces, together with all relevant code, is available from the authors.

[8] http://www.cs.rmit.edu.au/~jah/orpheus.

Table 5 Normal execution for Alice's SMS goal

Step	Beliefs	Events	Plans
0	∅	$!submit$	
1	∅		$(\!\vert true : SMS(SMSPLAN)\vert\!)$
2	∅		$SMS(SMSPLAN)$
3	∅		$SMS(apn; twa; (OC \Vert TWP); hps)$
4	*allocated*		$SMS(twa; (OC \Vert TWP); hps)$
5	*tracked,* allocated		$SMS((OC \Vert TWP); hps)$
6	tracked, allocated		$SMS((OC \Vert TWP); hps))$
7	*requested*, tracked, allocated		$SMS((OC(responded::nil; ?positive; +clear) \Vert TWP(nil)); hps))$
8	*written,* requested, tracked, allocated		$SMS(OC(responded::nil; ?positive; +clear); hps))$
9	written, requested, tracked, allocated		$SMS(OC(responded::nil; ?positive; +clear); hps))$
10	written, requested, tracked, allocated		$SMS(OC(responded::nil; ?positive; +clear); hps))$
11	*positive, responded*, written, requested, tracked, allocated	$!cleared$	$SMS(OC(responded::nil; ?positive; +clear); hps)$
12	*clear*, positive, responded, written, requested, tracked, allocated		$SMS(hps)$
13	*submitted,* clear, positive, responded, written, requested, tracked, allocated		nil

In order to save space, we will sometimes denote a goal such as SMS as SMS(apn; twa; $(OC \Vert TWP)$; hps) rather than Goal($submitted, expired, apn; twa; (OC \Vert TWP)$; hps). In general, this means we will write a goal G with currently executing plan P as G(P) rather than the longer form using Goal($\cdots, \cdots, P$).

The normal execution sequence for the SMS goal is given in Table 5. The event $submit$ initiates the SMSTOP plan, which then initiates the SMS goal (step 2). After the success of the APN and TWA actions, the agent executes the goals OC and TWA concurrently (step 5). The TWA finishes first, leaving the agent waiting for a response from Alice's manager (steps 7-10). Eventually the event $cleared$ is received (step 11), indicating a positive response from Alice's manager, and so the agent executes the HPS action and the plan SMSTOP terminates with success.

The execution sequence for Vignette 1 is given in Table 6. This proceeds as above, except that the $abort(SMS)$ event is received at step 7. The TWPPLAN plan has no abort-method, and hence is immediately dropped. The OCPLAN1 plan has the abort-method CCR, and so this is executed (step 8) before the abort-method CPN for the SMS goal is executed (step 9). Once this is done, the goal SMS is dropped, leaving the SMSTOP plan terminating with failure.

Table 6 Execution for Vignette 1

Step	Beliefs	Events	Plans
0	$\emptyset$	$!submit$	
1	$\emptyset$		$(\!\|true : SMS(SMSPLAN)\|\!)$
2	$\emptyset$		$SMS(SMSPLAN)$
3	$\emptyset$		$SMS(apn; twa; (OC \| TWP); hps)$
4	*allocated*		$SMS(twa; (OC \| TWP); hps)$
5	*tracked*, allocated		$SMS((OC \| TWP); hps)$
6	tracked, allocated		$SMS((OC \| TWP); hps))$
7	*requested*, tracked, allocated	$!abort(SMS)$	$SMS((OC(responded{::}nil; ?positive; +clear) \|$ $TWP(nil)); hps))$
8	*abort(OC), abort(TWP), abort(SMS)*, requested, tracked, allocated		$SMS(OC(ccr) \| nil))$
9	*aborted(OC), aborted(TWP)*, abort(SMS), requested, tracked, allocated		$SMS(cpn)$
10	aborted(OC), aborted(TWP), abort(SMS), requested, tracked, allocated		$SMS(nil)$
11	aborted(OC), aborted(TWP), *aborted(SMS)* requested, tracked, allocated		*fail*

Table 7 Execution for Vignettes 2 and 3

Step	Beliefs	Events	Plans
0	$\emptyset$	$!submit$	
1	$\emptyset$		$(\!\|true : SMS(SMSPLAN)\|\!)$
2	$\emptyset$		$SMS(SMSPLAN)$
3	$\emptyset$		$SMS(apn; twa; (OC \| TWP); hps)$
4	*allocated*		$SMS(twa; (OC \| TWP); hps)$
5	*tracked,* allocated		$SMS((OC \| TWP); hps)$
6	tracked, allocated		$SMS((OC \| TWP); hps))$
7	*requested,* tracked, allocated tracked, allocated	$!suspend(SMS)$	$SMS((OC(responded{::}nil; ?positive; +clear) \|$ $TWP(nil)); hps))$
8	*suspend(OC), suspend(TWP), suspend(SMS)*, requested, tracked, allocated		$SMS(OC(ccr) \| nil))$
9	*suspended(OC), suspended(TWP),* suspend(SMS), requested, tracked, allocated		$SMS(cpn)$

Table 7 continued

Step	Beliefs	Events	Plans
10	suspended(OC), suspended(TWP), suspend(SMS), requested, tracked, allocated		$SMS(nil)$
11	suspended(OC), suspended(TWP), *suspended(SMS),* requested, tracked, allocated		
12	suspended(OC), suspended(TWP), suspended(SMS), requested, tracked, allocated	$!resume(SMS)$	
13	*resume(OC), resume(TWP),* *resume(SMS),* requested, tracked, allocated		$SMS((OC(responded{::}nil; ?positive; +clear) \parallel TWP(nil)); hps))$
14	requested, tracked, allocated	$!cleared$	$SMS((OC(responded{::}nil; ?positive; +clear) \parallel TWP(nil)); hps))$
15	*clear, positive, responded, written,* requested, tracked, allocated		$SMS(; hps))$
16	*submitted*, clear, positive, responded, written, requested, tracked, allocated		nil

The execution sequence for Vignettes 2 and 3 is given in Table 7. The difference between this and Vignette 1 is that at step 7 the event $suspend(SMS)$ is received. As TWPPLAN has no abort-method, it is immediately suspended (step 8), whilst the abort-method CCR is executed for OCPLAN1. Once this is done, OC is suspended (step 9), allowing subsequent execution of the abort-method CPN for SMSPLAN, and then the suspension of the goal SMS (step 11). This is the endpoint of Vignette 2. When the $resume(SMS)$ event is received in Vignette 3 (step 12), as there are no resume-methods for any of the suspended plans, the goals SMS, OC and TWP are all immediately resumed, and the rest of the execution is the same as in Table 5.

The execution sequence for Vignette 4, in Table 8, is the same as the previous table up to step 12, where instead of $resume(SMS)$, the event received is $abort(SMS)$. The resulting execution is then the same as steps 7-11 of Vignette 1.

The execution for Vignette 5, in Table 9, again proceeds similarly for steps 1-6. At step 7, though, the event received is $promote(alice)$, which makes the in-condition for OCPLAN1 false, and hence means this plan should be aborted (step 8). Note, though, that the TWP goal proceeds uninterrupted, and hence succeeds at this point, while the abort-method CCR for OCPLAN1 is executed. After this, OCPLAN2 is now applicable, due to $senior(alice)$ now being true, and hence becomes the plan executed to achieve OC (step 9). Once this succeeds (step 11), the paper is submitted, and the plan SMSTOP terminates with success.

Table 8 Execution for Vignette 4

Step	Beliefs	Events	Plans
0	$\emptyset$	$!submit$	
1	$\emptyset$		$(\!\vert true : SMS(SMSPLAN) \vert\!)$
2	$\emptyset$		$SMS(SMSPLAN)$
3	$\emptyset$		$SMS(apn; twa; (OC \Vert TWP); hps)$
4	*allocated*		$SMS(twa; (OC \Vert TWP); hps)$
5	*tracked*, allocated		$SMS((OC \Vert TWP); hps)$
6	tracked, allocated		$SMS((OC \Vert TWP); hps))$
7	*requested*, tracked, allocated	$!suspend(SMS)$	$SMS((OC(responded::nil; ?positive; +clear) \Vert TWP(nil)); hps))$
8	*suspend(OC), suspend(TWP), suspend(SMS)* requested, tracked, allocated		$SMS(OC(ccr) \Vert TWP(nil)))$
9	*suspended(OC), suspended(TWP),* suspend(SMS), requested, tracked, allocated		$SMS(cpn)$
10	suspended(OC), suspended(TWP), suspend(SMS), requested, tracked, allocated		$SMS(nil)$
11	suspended(OC), suspended(TWP), *suspended(SMS)*, requested, tracked, allocated		
12	suspended(OC), suspended(TWP), suspended(SMS), requested, tracked, allocated	$!abort(SMS)$	
13	*abort(OC), abort(TWP), abort(SMS)*, requested, tracked, allocated		$SMS((OC(ccr) \Vert TWP(nil)))$
14	*aborted(OC), aborted(TWP)* abort(SMS), requested, tracked, allocated		$SMS(cpn)$
15	aborted(OC), aborted(TWP) abort(SMS), requested, tracked, allocated		$SMS(nil)$
16	aborted(OC), aborted(TWP) *aborted(SMS)*, requested, tracked, allocated		$fail$

Table 9 Execution for Vignette 5

Step	Beliefs	Events	Plans
0	junior(alice)	$!submit$	
1	junior(alice)		$(\!\vert true : SMS(SMSPLAN) \vert\!)$
2	junior(alice)		$SMS(SMSPLAN)$
3	junior(alice)		$SMS(apn; twa; (OC \Vert TWP); hps)$
4	*allocated*, junior(alice)		$SMS(twa; (OC \Vert TWP); hps)$
5	*tracked,* allocated		$SMS((OC \Vert TWP); hps)$
6	tracked, allocated, junior(alice)		$SMS((OC \Vert TWP); hps)$
7	*requested*, tracked, allocated junior(alice)	$!promote(alice)$	$SMS((OC(responded::nil; ?positive; +clear) \Vert TWP(nil)); hps)$
8	*abort(OCPLAN1)*, *senior(alice)*, *written*, requested, tracked, allocated		$SMS(OC(ccr); hps)$
9	aborted(OCPLAN1), senior(alice), written, requested, tracked, allocated		$SMS(OC(OCPLAN2); hps)$
10	aborted(OCPLAN1), senior(alice) written, requested, tracked, allocated		$SMS(OC(rcf); hps)$
11	*clear*, aborted(OCPLAN1), senior(alice) written, requested, tracked, allocated		$SMS(hps)$
12	*submitted,* clear, aborted(OCPLAN1), senior(alice), written, requested, tracked, allocated		nil

5 Summary and further work

Recognizing that an important ability of an agent-based system is to respond dynamically to changes, this article has investigated how to incorporate the ability to abort, suspend, and resume goals and plans into the execution cycle of a BDI-style intelligent agent. Such capabilities support an agent's more sophisticated reasoning over goals and plans, and its execution of plans, and thus the flexibility and capability of agent-based systems. We gave a combined operational semantics in the abstract agent language CAN, thus providing a clear mechanism which can be implemented in a number of practical agent platforms. A prototype implementation shows the operational correctness of our mechanisms on a detailed case study.

We extended CANwith additional constructs, namely the constructs ; and ::, in order to give required additional expressiveness. This provides us with an execution model for agent programs. We also showed how to compile an agent program, extended to include incorporating facilities for aborting, suspending and resuming tasks, into CAN, thus providing a precise operational definition of these operations. This methodological approach holds promise for similar approaches to computational extensions of the agent programming facilities provided in CAN. In particular, this shows the way forward for writing meta-interpreters for agent languages, in the spirit of similar methods for Prolog (such as those described by Sterling and Shapiro [44]). Future work includes exploring and proving formal properties.

Our semantics is developed for achievement goals; in maintenance goals, one must consider the consequences of, particularly, suspending a maintenance goal that is actively responding to a violation of its maintenance condition [15]. We leave exploration of these directions to future work.

One issue that sometimes arises is that once a goal succeeds, any plan currently being executed to achieve the goal is abandoned. While this is generally harmless, an item of further work is to examine the circumstances under which it may be better to abort such plans, rather than simply stop executing them.

A similar (and more significant) issue is that there are some circumstances in which it may be appropriate to 'proactively' abort a goal. 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. This means that the goal Track Writing Paper fails (or is aborted). As this is in parallel with the Obtain Clearance goal, it seems rational to abort the Obtain Clearance goal at the same time, as the conjunction of these two goals cannot now succeed. This may be thought of as a 'consequential' extension to the `.fail_goal` construct of Jason, in that the failure of one goal causes the abort of another. This will require an extension to our translation, particularly for the $\|$ construct, but involves some delicacy, as it is not simply a matter of any abort in one branch leading to the abort of the other. For example, aborting G_1 in $(G_1 \triangleright G_2) \parallel G_3$ means that G_2 is adopted, but not that $G_1 \triangleright G_2$ will necessarily fail. Hence there needs to be some further analysis of the goal–plan tree in order to correctly apply this idea. Note that the main issue here is how to determine which goals should be aborted when a given goal fails or is aborted. Once this decision is known, it is straightforward to implement behaviour of this nature using the methods of this article.

A further extension is to allow the agent to explicitly attach reconsideration conditions to suspended goals. In the current framework, a suspended goal is resumed when a resume signal is raised. The idea here is to explicitly state conditions under which a suspended goal may be resumed. This would be particularly useful in cases in which it is known at the time of suspension when the goal should be resumed, such as when a goal is suspended as a means of resolving conflicts between goals. Note that this is not simply a matter of extending the definition of the resume signal, similar to how the failure of an in-condition is handled. As there may be a significant time lapse between the goal's suspension and its resumption, it is important to allow the agent to reconsider whether to resume the goal or not. This will require some care in the way that the translation is extended.

Acknowledgements This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-07-D-0185/0004. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, or the Air Force Research Laboratory. The authors thank the JAAMAS reviewers, and the reviewers of the AAMAS 2007 and 2008 conferences at which preliminary parts of this work were presented. NYS thanks the Operations group at the Cambridge Judge Business School and the fellowship at St Edmund's College, Cambridge, where this work was performed.

References

1. Amini, M., Wakolbinger, T., Racer, M., & Nejad, M. G. (2012). Alternative supply chain production-sales policies for new product diffusion: An agent-based modeling and simulation approach. *European Journal of Operational Research*, *216*(2), 301–311.
2. Baldoni, M., Baroglio, C., Marengo, E., Patti, V., & Capuzzimati, F. (2014). Engineering commitment-based business protocols with the 2CL methodology. *Journal of Autonomous Agents and Multi-Agent Systems*, *28*(4), 519–557.

3. Boella, G., & Damiano, R. (2008). A replanning algorithm for decision theoretic hierarchical planning: Principles and empirical evaluation. *Applied Artificial Intelligence*, *22*(10), 937–963.
4. Bordini, R. H., & Hübner, J. F. (2010). Semantics for the Jason variant of AgentSpeak (plan failure and some internal actions). In *Proceedings of ECAI'10*, Lisbon, Portugal (pp. 635–640).
5. Bordini, R. H., Hübner, J. F., & Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason*. New York: Wiley.
6. Braubach, L., Pokahr, A., Moldt, D., & Lamersdorf, W. (2004). Goal representation for BDI Agent systems. In *Proceedings of 2nd international workshop on programming multi-agent systems (ProMAS'04)*, New York, NY (pp. 9–20).
7. Burmeister, B., Arnold, M., Copaciu, F., & Rimassa, G. (2008). BDI-agents for agile goal-oriented business processes. In *Proceedings of AAMAS'08 (Industry Track)*, Estoril, Portugal (pp. 37–44).
8. Chessell, M. G., Vines, C., Butler, D., Ferreira, M., & Henderson, P. (2002). Extending the concept of transaction compensation. *IBM Systems Journal*, *41*(4), 743–758.
9. da Costa Pereira, C., & Tettamanzi, A. (2010). Belief-goal relationships in possibilistic goal generation. In *Proceedings of ECAI'10*, Lisbon, Portugal (pp. 641–646).
10. Dai, B., & Chen, H. (2011). A multi-agent and auction-based framework and approach for carrier collaboration. *Logistics Research*, *3*(2–3), 101–120.
11. Dastani, M., van Riemsdijk, M. B., & Winikoff, M. (2011). Rich goal types in agent programming. In *Proceedings of AAMAS'11*, Taipei, Taiwan (pp. 405–412).
12. Dignum, F., Kinny, D., & Sonenberg, E. (2002). From desires, obligations and norms to goals. *Cognitive Science Quarterly*, *2*(3–4), 407–430.
13. Groves, W., Collins, J., Gini, M. L., & Ketter, W. (2014). Agent-assisted supply chain management: Analysis and lessons learned. *Decision Support Systems*, *57*, 274–284.
14. Hang, C. W., & Singh, M. P. (2012). Generalized framework for personalized recommendations in agent networks. *Journal of Autonomous Agents and Multi-Agent Systems*, *25*(3), 475–498.
15. Harland, J., Morley, D. N., Thangarajah, J., & Yorke-Smith, N. (2014). An operational semantics for the goal life-cycle in BDI agents. *Journal of Autonomous Agents and Multi-Agent Systems*, *28*(4), 682–719.
16. Heath, B., Hill, R., & Ciarallo, F. (2009). A survey of agent-based modeling practices (January 1998 to July 2008). *Journal of Artificial Societies and Social Simulation*, *12*(4), 9.
17. Hindriks, K. V., de Boer, F. S., van der Hoek, W., & Meyer, J. J. C. (2000). Agent programming with declarative goals. In *Proceedings of ATAL'00*, LNCS 1986, Boston, MA (pp. 228–243).
18. Hübner, J. F., & Bordini, R. H. (2015). Jason: A Java-based interpreter for an extended version of Agentspeak. Retrieved July 02, 2015 from http://jason.sourceforge.net.
19. Huntbach, M. M., & Ringwood, G. A. (1999). *Agent-oriented programming: From prolog to guarded definite clauses*. LNCS 1630. Berlin: Springer.
20. Jarvis, D., Jarvis, J., Rönnquist, R., & Jain, L. C. (2013). *Development using the GORITE BDI framework, multiagent systems and applications* (Vol. 46). Heidelberg: Springer.
21. Khan, S. M., & Lespérance, Y. (2010). A logical framework for prioritized goal change. In *Proceedings of AAMAS'10*, Toronto, Canada (pp. 283–290).
22. Kinny, D. (2001). The Psi calculus: An algebraic agent language. In *Proceedings of ATAL'01*, Seattle, WA (pp. 32–50).
23. Lorini, E., van Ditmarsch, H. P., & Lima, T. D. (2010). A logical model of intention and plan dynamics. In *Proceedings of ECAI'10*, Lisbon, Portugal (pp. 1075–1076).
24. Máhr, T., & de Weerdt, M. (2005). Distributed agent platform for advanced logistics. In *Proceedings of AAMAS'05*, Utrecht, The Netherlands (pp. 155–156).
25. Mikic-Fonte, F. A., Burguillo-Rial, J. C., & Nistal, M. L. (2012). An intelligent tutoring module controlled by BDI agents for an e-learning platform. *Expert Systems with Applications*, *39*(8), 7546–7554.
26. Morley, D., & Myers, K. (2004). The SPARK agent framework. In *Proceedings of AAMAS'04*, New York, NY (pp. 714–721).
27. Morley, D., Myers, K. L., & Yorke-Smith, N. (2006). Continuous refinement of agent resource estimates. In *Proceedings of AAMAS'06*, Hakodate, Japan (pp. 858–865).
28. Myers, K., Berry, P., Blythe, J., Conley, K., Gervasio, M., McGuinness, D., et al. (2007). An intelligent personal assistant for task and time management. *AI Magazine*, *28*(2), 47–61.
29. Myers, K. L., & Yorke-Smith, N. (2005). A cognitive framework for delegation to an assistive user agent. In *Proceedings of AAAI 2005 fall symposium on mixed-initiative problem-solving assistants*, Arlington, VA (pp. 94–99).
30. Pokahr, A., Braubach, L., & Lamersdorf, W. (2005). A goal deliberation strategy for BDI agent systems. In *Proceedings of the third German conference on Multi-Agent System TEchnologieS (MATES'05)*, Koblenz, Germany (pp. 82–94).

31. Pokahr, A., Braubach, L., & Lamersdorf, W. (2005). Jadex: A BDI reasoning engine. In R. H. Bordini, M. Dastani, J. Dix, & A. E. Fallah-Seghrouchni (Eds.), *Multi-agent programming* (pp. 149–174). Berlin: Springer.
32. Pěchouček, M., & Mařík, V. (2008). Industrial deployment of multi-agent technologies: Review and selected case studies. *Journal of Autonomous Agents and Multi-Agent Systems*, *17*, 397–431.
33. Rao, A.S. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of seventh European workshop on modelling autonomous agents in a multi-agent world (MAAMAW'96)*, Eindhoven, The Netherlands (pp. 42–55).
34. Rao, A. S., & Georgeff, M. P. (1991). Modeling rational agents within a BDI-architecture. In *Proceedings of KR'91*, Cambridge, MA (pp. 473–484).
35. Rao, A. S., & Georgeff, M. P. (1992). An abstract architecture for rational agents. In: *Proceedings of KR'92*, Cambridge, MA (pp. 439–449).
36. van Riemsdijk, M. B., Dastani, M., & Winikoff, M. (2008). Goals in agent systems: A unifying framework. In *Proceedings of AAMAS'08*, Estoril, Portugal (pp. 713–720).
37. Rönnquist, R. (2007). The goal oriented teams (GORITE) framework. In *Proceedings of ProMAS'07*, LNCS 4908, Honolulu, HI (pp. 27–41).
38. Rosaci, D., & Sarnè, G. M. L. (2012). A multi-agent recommender system for supporting device adaptivity in e-commerce. *Journal of Intelligent Information Systems*, *38*(2), 393–418.
39. Sardiña, S., & Padgham, L. (2007). Goals in the context of BDI plan failure and planning. In *Proceedings of AAMAS'07*, Honolulu, HI (pp. 16–23).
40. Sardiña, S., & Padgham, L. (2011). A BDI agent programming language with failure handling, declarative goals, and planning. *Journal of Autonomous Agents and Multi-Agent Systems*, *23*(1), 18–70.
41. Sardiña, S., de Silva, L., & Padgham, L. (2006). Hierarchical planning in BDI agent programming languages: A formal approach. In *Proceedings of AAMAS'06*, Hakodate, Japan (pp. 1001–1008).
42. Shaw, P. H., Farwer, B., & Bordini, R. H. (2008). Theoretical and experimental results on the goal-plan tree problem. In *Proceedings of AAMAS'08*, Estoril, Portugal (pp. 1379–1382).
43. de Silva, L., Sardiña, S., & Padgham, L. (2009). First principles planning in BDI systems. In *Proceedings of AAMAS'09*, Budapest, Hungary (pp. 1105–1112).
44. Sterling, L., & Shapiro, E. (1994). *The Art of Prolog* (2nd ed.). Cambridge: MIT Press.
45. Thangarajah, J., Harland, J., Morley, D., & Yorke-Smith, N. (2007). Aborting tasks in BDI agents. In *Proceedings of AAMAS'07*, Honolulu, HI (pp. 8–15).
46. Thangarajah, J., Harland, J., Morley, D., & Yorke-Smith, N. (2008). Suspending and resuming tasks in BDI agents. In *Proceedings of AAMAS'08*, Estoril, Portugal (pp. 405–412).
47. Thangarajah, J., Harland, J., Morley, D., & Yorke-Smith, N. (2010). On the life-cycle of BDI agent goals. In *Proceedings of ECAI'10*, Lisbon, Portugal (pp. 1031–1032).
48. Thangarajah, J., Harland, J., Morley, D. N., & Yorke-Smith, N. (2014). Quantifying the completeness of goals in BDI agent. In *Proceedings of ECAI'14*, Prague, Czech Republic (pp. 879–884).
49. Thangarajah, J., & Padgham, L. (2011). Computationally effective reasoning about goal interactions. *Journal of Automated Reasoning*, *47*(1), 17–56.
50. Thangarajah, J., Winikoff, M., Padgham, L., & Fischer, K. (2002). Avoiding resource conflicts in intelligent agents. In *Proceedings of ECAI-02*, Lyon, France (pp. 18–22).
51. Wellman, M. P., Greenwald, A., & Stone, P. (2007). *Autonomous bidding agents: Strategies and lessons from the trading agent competition*. Cambridge: MIT Press.
52. Winikoff, M. (2005). JACK intelligent agents: An industrial strength platform. In *Multi-Agent programming* (pp. 175–193). New York: Springer.
53. Winikoff, M. (2011). A formal framework for reasoning about goal interactions. In *Proceedings of AAMAS'11*, Taipei, Taiwan (pp. 1107–1108).
54. Winikoff, M., Padgham, L., Harland, J., & Thangarajah, J. (2002). Declarative and procedural goals in intelligent agent systems. In *Proceedings of KR'02*, Toulouse, France (pp. 470–481).
55. Yorke-Smith, N., Saadati, S., Myers, K., & Morley, D. (2012). The design of a proactive personal agent for task management. *International Journal on Artificial Intelligence Tools*, *21*(2), 90–119.