A Framework for Goal-Based Semantic Compensation in Agent Systems

Amy Unruh, James Bailey, and Kotagiri Ramamohanarao

Dept. of Computer Science and Software Engineering
The University of Melbourne, VIC 3010, Australia
{unruh,jbailey,rao}@cs.mu.oz.au

Abstract. We describe an approach to improving the robustness of an agent system by augmenting its failure-handling capabilities in a way that makes the system's behavior more predictable. Our approach is goal-based, both with respect to defining failure-handling knowledge, and in associating this knowledge with agent tasks. By abstracting this knowledge above the level of specific actions or task implementations, it is not tied to specific agent architectures or task plans and is more widely applicable. The failure-handling knowledge is employed via a failure-handling support component associated with each agent through a goal-based interface. The use of this component decouples the specification and use of failure-handling information from the specification of the agent's domain problem-solving knowledge, and reduces the failure-handling information that an agent developer needs to provide.

1 Introduction

The work described in this paper is part of a Department of CSSE, University of Melbourne project to develop methodologies for building more robust multi-agent systems, in which we investigate ways to apply transactional semantics to improve the robustness of agent problem-solving and interaction. Traditional transaction processing systems prevent inconsistency and integrity problems by satisfying the so-called ACID properties of transactions: Atomicity, Consistency, Isolation, and Durability [1]. These properties define an abstract computational model in which each transaction runs as if it were alone and there were no failure. The programmer can focus on developing correct, consistent transactions, while the handling of concurrency and failure is delegated to the underlying engine. In the context of our larger project, we will be developing methodologies to support failure handling, crash recovery and concurrency management in agent systems. In addition, the longer-term project goals include the definition of a language and computational model that will more formally specify aspects of our methodologies. While transactionally motivated, our approach must be applicable in domains outside the traditional database environment; in these domains the effects of many actions may not be delayed (essentially, they always "commit"), and thus correction of problems must be implemented by "forward recovery", or failure *compensation* [1].

In this paper, we focus on one aspect of behavior motivated by transactional semantics: improving the ability of an agent system to recover from task problems by "cleaning up after" or "undoing" its problematic actions– essentially, to compensate its failures. This behavior can make an agent system more stable and predictable; both with respect to the individual agent and with respect to its interactions with other agents: not only are problems compensated, it is more likely that a retry of a failed task will be successful if the system is returned to a desirable state; and it is more likely that the agents' implicit assumptions about the state of their environment will hold. In this way, by making the semantics of the agent system more predictable, the system can become more robust– safer– in its reaction to unexpected or problematic events.

A key issue is that it is usually not computationally feasible to specify an agent's task decompositions prior to its performing a task, nor do we necessarily have enough information about our (potentially infinite) "states of the world" to do so prior to runtime. Not only are we unable to predict how the tasks in a dynamic agent system will unfold (and thus we can't pre-define compensations for those tasks), it is not always desirable to exactly "undo" a task even if it is possible to perform its inverse. Useful compensations are not only application-specific but must be dynamically determined. This means that we cannot directly use those

methods for managing transactional structures which require the transactions and their compensations to be characterized ahead of time. Nor it is a solution to create "compositions" of compensations by associating compensations with each subtransaction and applying them in reverse order [1]; in many domains such compensations will not be correct or useful. Nevertheless, we would like to develop a method that supports a basic motivation behind the use of transaction compensations: that of defining and using information about how to undo problems, in a consistent and well-specified manner, so that we are able to make claims about how an agent system will behave on certain types of failure, and characterize how it will recover in such situations. Given the requirements of a dynamic agent system, several criteria drive our approach.

- We would like to develop general recovery strategies that can leverage the domain knowledge that the agents bring to the system.
- We would like to support an incremental approach to providing robustness, by generating "reasonable" default behavior with little extra-agent domain knowledge, then allowing information to be added in stages to improve performance.
- We would like the failure-handling knowledge for a given domain to be specifiable at a level of abstraction that does not require an agent designer to pay attention to the details of the agent's failure-handling mechanisms.

A primary basis of our approach is that within a problem domain, it is often possible to usefully define *what* to do to address a failure independently of the details of *how* to implement the correction; and to define failure-handling knowledge for a given (sub)task without requiring knowledge of the larger context in which the task may be invoked. Based on these hypotheses, we will present a method for enhancing agent system robustness in which failure-handling knowledge is defined independently of specific task implementations, and which allows the system to respond to certain types of failure in a more robust, consistent, and predictable manner.

1.1 Overview of Approach

Our objective is to make an agent more robust, by improving its failure-handling behavior. We would also like agent designers to be able to specify this more robust behavior in a tractable and consistent manner: by allowing the designers to define failure-handling information in a way that is easy to understand and which does not require them to make tweaks to a given agent's existing domain model; and by providing an underlying support mechanism which takes care of the details of the failure-handling model. Our approach addresses these these objectives.

The first key aspect of our approach is that we employ a *goal-based* specification of failure-handling knowledge. We utilize two types of failure-handling knowledge, which must be provided for each agent in a system. First, for a given domain, and for some subset of the tasks the agent knows how to perform, we associate information about what needs to be achieved to clean up (e.g., compensate) a failure of that task. This information is specified in terms of the *goals* that need to be achieved in order for the compensation to be successful, not in terms of action or plan steps that might implement the goal. Essentially, we are specifying, at an abstract level, *what* to do to handle a failure, not *how* to achieve the implementation. In conjunction with this set of failure-handling definitions, a set of goal-level failure-handling strategy rules is created. These rules effectively specify *when* to employ certain failure-handling knowledge. The rules match on goal-level information about a task, not execution details. Our goal-based formulation of failure-handling knowledge has several advantages: it allows an abstraction of knowledge that can be hard to express in full detail; its use is not tied to a specific agent architecture; and it allows our method to be used in dynamic domains in which it is not possible to pre-specify all relevant failure-handling plans.

The second key aspect of our approach is the definition of a decoupled component and abstract interface that makes use of the goal-level domain knowledge described above in order to support agent failure management[1]. We refer to this component as the agent's FHC (Failure-Handling Component). The FHC performs

[1] With respect to our larger project goals, this framework will also support other aspects of our intended transactional semantics, such as logging, recovery from crashes, and task concurrency management.

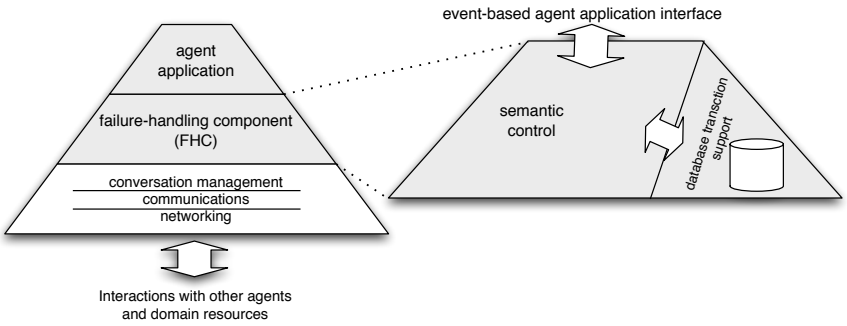

Fig. 1. An agent's FHC. We refer to the domain logic part of the agent, above the failure-handling component, as the "agent application".

high-level monitoring of the agent's problem-solving, and affects its behavior in failure situations without requiring modification of the agent's implementation logic. When multiple distributed agents are working on different subparts of a task, their FHCs communicate failure events, coordinate which agent handles failure of a delegated task, and support autonomous handling of subtask failure.

Any agent which implements the FHC's interface, and for which necessary failure-handling knowledge is provided to the FHC, can "plug in" to this component. Thus, analogously to exception-handling in a language like Java, the FHC reduces the agent designer's implementation requirements– that is, reduces what needs to be done to "program" the agent's failure-handling behavior– while providing a model that constrains and structures the failure-handling information that needs to be defined. As shown in Figure 1, the FHC sits conceptually below the agent's domain logic component and interfaces with (in a typical architecture) the conversation protocol layer of the agent [2], [3]. As suggested in the figure, the FHC maintains transactionally-supported storage of its information. In the remainder of this paper, for clarity, we refer to the domain logic part of the agent, above the failure-handling component, as the "agent application".

Below, we provide a motivating example for our approach. In Section 2, we describe the two types of goal-based failure-handling knowledge utilized by the approach. In Section 3 we detail the agent's failure-handling component, or FHC, which makes use of this failure-handling knowledge; and describe a prototype which implements much of the approach. In Sections 4 and 5 we finish with a discussion of related work, and summarize.

1.2 Examples

We first introduce the concept of goal-based failure-handling, and its utility in supporting compensation strategies, with some motivating examples. Consider two related "dinner party" scenarios, where a group of agents must plan and carry out the activities necessary to prepare for holding a party. First, consider an example where the party will be held at a rented hall, e.g. for a business-related event. Figure 2 shows one task decomposition for such an activity. The figure uses an informal notation, where subtask ordering is indicated by arrows, and subtasks that may be executed concurrently are connected by a double bar. The subtasks include planning the menu, scheduling the party and arranging to reserve a hall, inviting the guests and arranging for catering. The figure indicates that some of the subtasks (such as inviting the guests) may be delegated to other agents in the system. If the party is planned successfully, but the event must be canceled (e.g., if the host becomes ill), then we can define a number of things that must reasonably be done to properly take care of the cancellation– that is, to "compensate" the party: all party-related reservations should be canceled, extra food used elsewhere if possible, and guests notified and apologies made. The second task decomposition in Figure 2 shows one implementation of these goals. For this particular scenario, no actions are taken with respect to the food, as the caterers will handle it once they get the cancellation.

Next, consider a scenario which differs in that the party will be held at the host's house (perhaps for a smaller number of guests). In this case, while the party must be scheduled, a hall does not need to be

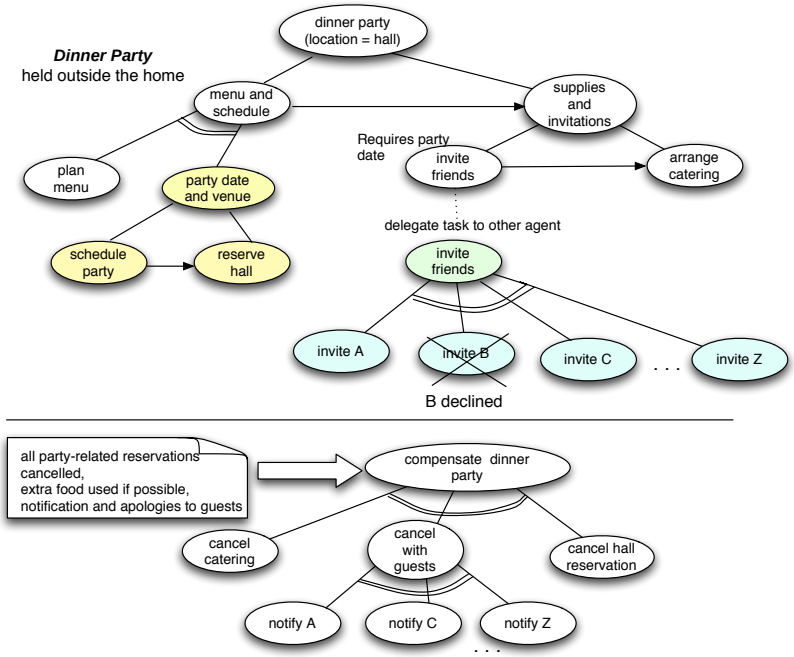

Fig. 2. Planning a dinner party, then canceling it.

reserved. In addition, the hosts will not have the smaller party catered and will shop themselves. Figure 3 shows a task decomposition for this scenario. It differs in several respects from the decomposition in Figure 2. If this second party must be canceled after it is planned, the same set of high-level criteria apply. However, the implementation of the "compensation" is different. We have no reservations to cancel, but we do have extra food, which will be taken to (say) a homeless shelter. Due to the more personal nature of this party, we also send a small gift to each guest.

In both these examples, the party cancellation tasks can be viewed as accomplishing a *semantic compensation*– an approximate "undo" in which some effects may be approximately reversed, but other forward actions may also be necessary to accomplish the "cancellation" semantics. A semantic compensation may include actions that are not the inverse of any of the (sub)tasks of the original task, such as the gift-giving in the second example. In the context of a semantic compensation, some subtasks may "reverse" the effects of previous actions (e.g. canceling the reservation of a meeting hall), but other previous effects may be ignored (no actions are performed to "undo" the effects of the house-cleaning). Still other sub-tasks (dealing with the extra food) may be partially compensatable depending upon context. The definition of such a semantic compensation is task- and domain-specific. In addition, these examples illustrate that in many domains both the implementation of a task, and its semantic compensation, can differ according to context and the "state of the world". So, it can be expensive, or effectively impossible, to define all necessary semantic compensations prior to runtime in terms of specific actions that must be performed in order to effect the compensation. Instead, we claim it is more useful to define semantic compensations *declaratively*, in terms of the *goals* that the agent system needs to achieve in order to accomplish the compensation. The way in which these goals are achieved, for a specific scenario, will depend upon context.

The two examples above assumed that the dinner party was planned successfully, then canceled. One can also consider scenarios in which problems occur during the party planning task. For example, there may be problems in contacting some of the invitees. Depending upon the nature of the problem, it may make sense to try to recover by re-contacting a guest, perhaps using a different contact method (e.g. email instead of phone). Consider also a scenario with the tasks of Figure 3, in which some problem occurs in accomplishing

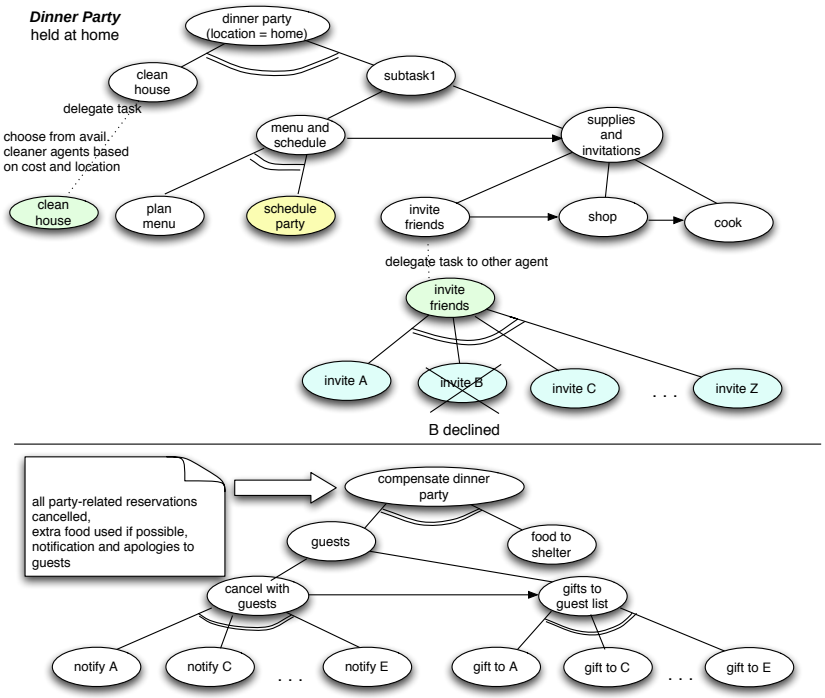

Fig. 3. Planning a dinner party: an alternate scenario.

the "shop" subtask– perhaps the agent doing the shopping has car trouble. Before any retry occurs, it might be useful to first ensure that the agent is oriented (knows its location).

2 Goal-Based Semantic Compensation

In this section, we describe the domain information that needs to be supplied by an agent developer in order to enable the agent's failure-handling component, or FHC. The examples of Section 1.2 illustrated how high-level goals may be achieved in different ways– via different task decompositions– under different circumstances. We saw that this was the case both in achieving a "regular" task, and in accomplishing its compensations. Under such circumstances, it is often difficult to identify prior to working on a task the details of how a task failure should be addressed or a compensation performed. Instead, we specify failure-handling knowledge *in terms of goals*, making it more widely applicable and easier to define. First, we associate *failure-handling goal definitions* with some or all of the tasks (goals) that agent can perform. These definitions specify at a goal[2]– rather than plan– level *what* to do in certain failure situations, and we then rely on the agents in the system to determine *how* it is accomplished. Second, the use of this knowledge is triggered by the goal-level events generated by the agent application's task execution; essentially, triggered by failure, success, or cancellation of a given task. The agent's FHC is provided with information about *when* to employ these definitions, in the form of a set of rules. The rules effectively refine the FHC's default task compensation behavior. These two activities are described in Sections 2.1 and 2.2 below.

We view both of these definitional activities as "assisted knowledge engineering" tasks; we can examine the agent's problem-solving implementation, and in particular its goal-subgoal relationships, and leverage this implementation knowledge to support the definition process. We are researching ways to provide semi-automated support for this process. It is most effective (but not required) to interleave agent application

[2] In this paper, we use 'goal' and 'task' interchangeably; as distinguished from plans, action steps, or task execution. In our usage, goals describe conditions to be achieved, not actions or decompositions.

development and failure-handling specification: as each domain task is defined, the developer can identify the failure-handling semantics that they wish to associate with that task. Because the FHC enforces an explicit and straightforward use of its failure-handling knowledge, the developer need not replicate equivalent behavior in the agent application; thus "domain logic" and failure-handling knowledge may be largely separated, making each easier to modify. In addition, the definitional process can highlight gaps in the agent application's knowledge.

2.1 Goal-Based Failure-Handling Definitions

In this section, we describe the process of creating an agent's goal-based *failure-handling definitions*: defining what goals the agent application is instructed to achieve under failure conditions. Our approach distinguishes between two types of failure-handling task states represented in the agent's FHC: initial task *failure*, and task *cancellation*. The agent application reports task failure to its FHC, including failure *mode* information as available[3]. If a task is moved to a canceled state, this means that any ongoing work on the task will be halted, and its (perhaps partial) effects may be compensated. In contrast to a strict transactional model, which requires that a failure always be compensated, we do not require that this is always the case. We may not want to fully "undo" the effects of a failed task under all circumstances. We may want to first retry task achievement (via the same or a different approach), or it may be unimportant or too expensive to undo a task's effects.

Therefore, we associate two different types of failure-handling goals with each task. The first are *always* invoked on failure of that task, and prior to any retries. We call these the task's *stabilization* goals. They describe what must be achieved to "clean up" a task's effects. Their purpose is to return the system to a more predictable state. The second type of goal may be invoked if a task moves to a canceled state, and they describe what must be achieved to semantically "undo" the task's actions. We call these the task's *compensation* goals. These goals may be invoked on failure, or triggered by the failure of an ancestor task or a domain event. As will be described in Section 2.2, the agent's FHC determines when a task moves to a canceled state and whether a specific subtask compensation is invoked on task cancellation. The stabilization and compensation goal sets for a task must be mutually consistent, and either or both may be the empty set. Under failure circumstances, the union of both goal sets may be invoked.

An agent developer will associate failure-handling definitions with the agent's domain tasks; this information will be used by the agent's FHC. In defining failure-handling goals for a (sub)task, the developer must specify what must be *achieved*– what must be true about the state of the world– for stabilization or compensation of that task to be successful. This information is specified *only in terms of goals*, not actions or plans– the agent application will *determine at runtime how to implement*, or achieve, these goals. The failure-handling goals must be defined as necessary in terms of the parameters of their original associated task. It is important to note that the compensation goals for a high-level task need not correspond to a composition of compensations for the high-level activity's expected subtasks. This was illustrated in the dinner party example.

If a task is to be compensated, then the agent will always work to achieve the compensation goals associated with that task. Thus, the developer must ensure that a given task's compensation describes all the conditions that *must* be met for a successful compensation of that task. In addition, the agent *may*, depending upon context, work to achieve the compensation goals associated with some of its subtasks (as described in Section 2.2). Intuitively, compensation definitions for the lower-level tasks will encode details not included in a higher-level compensation, since these lower-level compensations are defined with respect to a more specific local situation. However, an agent developer must define failure-handling knowledge for an agent task *independently of the context* in which the task may be achieved. That is, we must be able to define a task's compensation semantics independent of the "state of the world" or parent task; we do not know a priori the circumstances under which the task will be invoked or whether it will have fully completed. We believe that encoding our failure-handling knowledge as goals, not plans, will allow us to meet this criteria. The application agent will take current state into account when deciding at runtime how to implement a given failure-handling goal. Table 1 shows some example task failure-handling definitions, from the dinner party

[3] Our prototype currently utilizes two failure modes: *system* faults and *semantic* failures.

scenario of Section 1.2, and for the two classes of failure-handling goals described above. These definitions specify what to do, semantically, to compensate for task problems.

Task	Compensation Goals	Stabilization Goals
plan dinner party	all party reservations canceled & extra food used if possible & notification and apologies to guests	n/a
invite guests	guests notified of cancellation	n/a
reserve hall	hall reservation canceled	n/a
shop for supplies	supplies returned	at known location

Table 1. Simplified task failure-handling definitions. The compensation goal for the 'plan dinner party' task describes what *must* be achieved if that task is canceled. Stabilization goals, always invoked immediately on failure, are defined for one of the tasks.

2.2 Failure-Handling Strategies

Given failure-handling goal definitions for domain tasks, the agent's FHC must be instructed when to use them, via a set of domain-dependent *failure-handling strategy rules*. Essentially, these rules serve to specify the degree to which sub-tasks should be autonomously compensated on failure (in effect, whether the failure should be handled locally or "thrown" to the parent); or on cancellation of a parent task. As above, specification of these rules requires knowledge of the domain semantics.

The strategy rules conceptually map to event-condition-action rules [4], and match against the abstract goal-level information about task history and status maintained by the FHC (as will be detailed in Section 3), as well as the failure-handling definitions associated with domain tasks. They are triggered by 'goal events' communicated from the agent application (including failure *mode* information as available), or changes to status of the FHC tree nodes. The strategy rules:

- Determine when to cancel a task based on its failure;
- Define when to perform task retries;
- Enumerate those domain events that should trigger task cancellation;
- Determine whether to invoke a given (sub)task compensation if it is part of a canceled task; and
- Provide failure-task assignment support [not discussed here].

Note that these failure-handling strategy rules *match against the abstract failure events* generated by the FHC, not against the agent application's internal state. We believe that this level of abstraction is sufficient to generate a useful range of failure-handling behaviors, while allowing a simpler specification on the part of the agent designer. Specific compensation and failure-handling strategies for an application are generated by defining and employing *strategy rule sets*.

Figure 4 shows a fragment of a strategy rule set for the "plan dinner party" domain of Section 1.2. Tasks of type 'invite a guest' are considered failed if no result is generated in a given time interval. These tasks may be retried after achieving any associated stabilization goals. It is important to note that *the agent application determines how to perform the retry*; these rules just decide whether to initiate it. That is, it is the agent application that contains the knowledge of *contingency plans*– alternative ways to try to accomplish a goal. If a given event is detected ('host falls ill'), then the "plan dinner party" task is halted and canceled. Achievement of the root compensation goals, as specified in Table 1, is initiated. The agent application begins work to achieve those goals. (Note that these compensation goals don't address all of the original task's effects). At the same time, the strategy rules initiate autonomous achievement of "invite guests" and "reserve-hall" subtask compensations *if those subtasks were part of the original task*.

This simple example illustrates an important point: a strategy to effect a high-level task compensation will always include invocation of the compensation goals for that task goal, and may include invocations of the compensations for *some*–but not necessarily all– of its subtasks. This can be useful in two respects. First, it allows localized fixes to be spawned in an autonomous manner at the same time that a more deliberate

IF a task node reaches failure [as reported by its agent application],
  THEN initiate the stabilization goals associated with the node. (implicit)
IF a node reaches failure, and there is no retry of that task in progress, and no strategy rules block its cancellation,
  THEN mark the node 'canceled'. (implicit)
IF a failed node transitions to 'canceled', THEN initiate the compensation goals associated with that node. (implicit)
IF a domain event causes a node to transition to 'canceled',
  THEN initiate the compensation goals associated with that node. (implicit)
IF a root node fails, THEN always mark it 'canceled'.
IF Task <dinner party planning> is 'in progress', and <the host falls ill>,
  THEN mark its node 'canceled'.
IF the agent application does not return a goal status notification for an <invite a guest> task N units of time after the task was initiated,
  THEN mark its node 'failed'.
IF an <invite a guest> node reports (system) failure, and less than N retries of the node have occurred,
  THEN log that a retry is in progress for that node.
IF a node reports success of its stabilization goal and a retry is in progress,
  THEN re-initiate the task goal associated with the node after M units of time and increment the retry count for that goal.
IF the <invite guests> task is marked 'canceled', and it is a descendent of the <dinner party planning> task,
 THEN initiate the compensation for the <invite guests> subtask.
IF the <reserve-hall> task is marked 'canceled', and it is a descendent of the <dinner party planning> task,
 THEN initiate the compensation for the <reserve-hall> subtask.

Fig. 4. A fragment of a simple FHC failure-handling strategy rule set. If a node is canceled, the cancellation notification propagates automatically to its children, moving them to a canceled state as well (and halting all work on the task). The rules marked "implicit" are domain-independent and hardwired by the FHC; we list them here for clarity.

top-down compensation planning process is occurring, in order to react more quickly to certain problems[4]. This means that in a distributed system, where different agents are performing sub-parts of a task, agents may be *reactively signaled* to fix failures. Second, the compensation definitions for the lower-level tasks may encompass details not included in the higher-level compensation, since they are specific to *this* task decomposition. Thus a more complete compensation may be effected.

Domain characteristics will influence the choice of strategy; and, though not shown here, the strategies can encompass failure-handling strategies *for* failure-handling tasks themselves. Experiments are in progress to determine what strategies are most effective in various domains. We anticipate that we will be able to provide a set of default strategy rules, to be overridden and augmented in specific cases.

3 The Agent's FHC

As was introduced in Figure 1, each agent application in our agent system sits upon a failure-handling component, which we term its *FHC*, with communication via a goal-based interface. The FHC employs the two forms of goal-based knowledge described in the previous section. The interface allows the FHC to track the agent application's problem-solving at a high level, and to affect its behavior in failure situations. In a multi-agent task scenario, the FHCs implement communication of failure-handling information between the agents involved in the task, coordinate which agent handles failure of a delegated task, and allow sub-task agents to autonomously perform compensating activities on failure if appropriate.

An agent's problem-solving logic resides in its agent application, which performs planning, task decomposition and execution. The FHC tracks the agent application's task decomposition and status of its task (sub)goals, and– via its strategy rules and failure-handling definitions– reacts to task failures by instructing agents to perform repair tasks. That is, an agent's FHC makes autonomous decisions about *what* failure-handling tasks will be performed, and *when* they will be requested. The agents' application logic is then invoked to implement the tasks and determine the details of how to correct for the failures. The FHC's failure-handling *augments*, not overrides, the agent application's.

The use of the FHC reduces the agent developer's implementation requirements, by providing a model that structures and supports the failure-handling information that needs to be defined. (The FHC implementation must be specific to a given agent framework, including its communication mechanisms, and 'connects'

[4] Note that as in the example, the higher-level compensation definition may encompass some of the subtask compensation goals and there may be redundant efforts to achieve the same goal– e.g. to cancel a reservation. The agents should be able to detect whether a given goal is already achieved. This is discussed further in Section 3.4.

the agent application to the lower agent layers). The motivation behind the use of the FHC is analogous to that of the exception-handling mechanism in a language like Java; the developer is assisted in generating desired agent failure-handling behavior, and the result is easier to understand and predict than if the knowledge were added in an ad-hoc fashion. In this section, we first describe the FHC's interface to the application agent, and the information that the FHC monitors and maintains to support its failure-handling policies. Then, Section 3.4 discusses the requirements imposed by this approach on the agent applications in a system.

3.1 Goal-Oriented FHC/Agent Application Interface

The interface between the FHC and its agent application is based on *goal-oriented events*. An agent application reports new (sub)goals to its FHC as it generates its task decompositions, and reports changes in goal status and failure modes as it works on the goals. An agent's FHC determines whether/when an agent can start work on a new goal, and can tell it to halt work on a goal. More specifically, the interface supports the events listed in Figure A-1. This goal-based interface allows us to decouple the failure-handling component's failure-handling strategies from the architectural details of the agent; and thus the method is not tied to a specific agent architecture, but can be used with any agent that can implement the interface (see Section 3.4). We believe that this interface has sufficient expressive power to support a range of useful failure-handling behaviors, and will verify this hypothesis in a range of application domains.

3.2 FHC Task Structures

To provide failure-handling support, the FHC monitors and maintains information about the agent application's problem-solving, as communicated via its interface. It does this at a goal level of abstraction: for each agent task, a tree structure is built in the FHC to syntactically track the goals and subgoals generated as the agent application's problem-solving progresses. We call such a tree a *task-monitoring tree*. The monitored information does not include task details, only goal information. Thus the FHC's structures are typically 'lightweight'.

Fig. 5. FHC task node states.

Each node in a task-monitoring tree corresponds to a task subgoal, with its associated failure-handling definitions. A node may be in one of the states shown in Figure 5. 'Canceled' indicates the given node state only; *not* the status of any corresponding compensation or stabilization activities (for which new FHC nodes will be generated). When a task is canceled, all ongoing work on it is halted.

The task-monitoring trees support the FHC's failure-handling. Figure 6(a) shows a simple example of the way in which a task-monitoring tree is incrementally built to track the progress of an agent application task. The agent application reports new goals, and waits for the signal from its FHC before starting them[5]. The agent application, using its domain knowledge, performs the actual problem solving. The agent application need not be using an explicit "tree" representation itself; it only needs to be able to report its subgoals as they are generated. FHC trees are retained for all tasks that have a currently-executing root or that have a repair relationship to a currently-executing task. Thus the FHC retains abstract information about *subtask history*– though the agent application may have discarded its finished goal structures– and can access this information in applying compensations.

[5] While not discussed in this paper, the FHC is also used to support task coordination; and thus it may potentially postpone the start of a task.

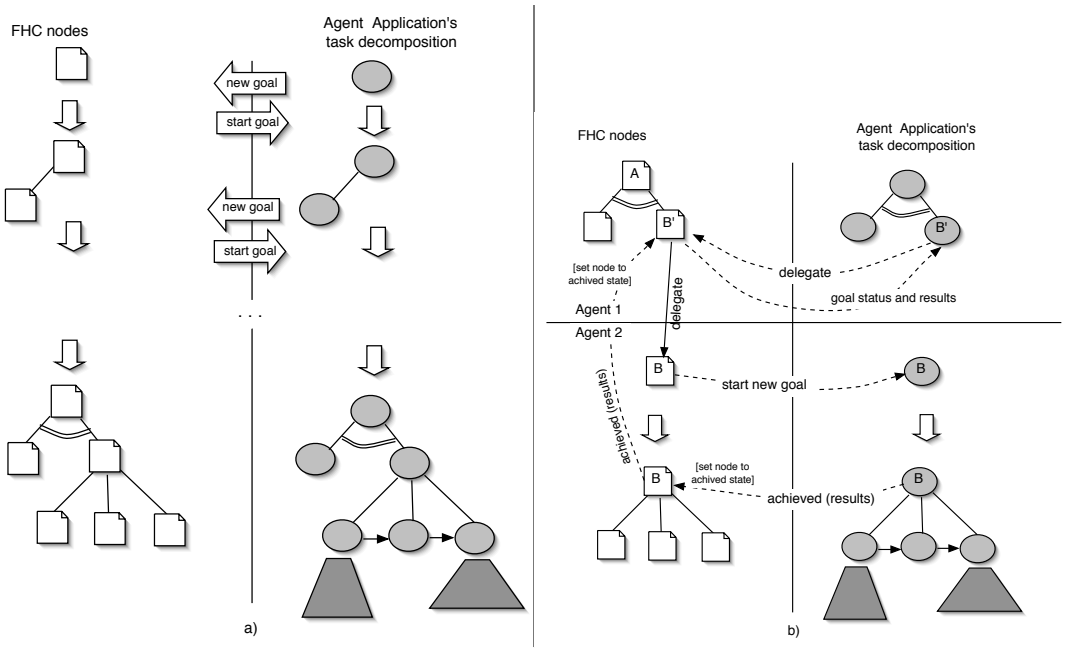

Fig. 6. Constructing an FHC "task tree", by monitoring problem-solving. Each FHC node contains information about task status and associated failure-handling definitions. Task details are not tracked.

Inter-Agent Failure Handling. As suggested in Figure 1, all inter-agent communications are filtered through the agent's FHC before they reach the protocol/communication layers of the agent architecture; and similarly the FHC intercepts the messages to the agent application. (Recall that the FHC's implementation must be specific to a given agent framework). This allows the FHC to annotate outgoing messages, and similarly to pre-process incoming messages. By this means, it adds information to task delegation request and response/status messages, so that the FHC of each agent participating in a task knows how to "connect" its task nodes to those of the other agents. Via the annotated messages, a *distributed FHC task structure* is created jointly by the FHCs of the agents that are performing various parts of a task, by creating proxy nodes in parent agents for delegated tasks. Notification of node status change– including task results and cancellations– are propagated between agents via their respective FHCs, and failure-handling behavior is invoked in their agent applications as appropriate.

Thus, the FHCs in a multi-agent system effectively support a distributed failure-handling protocol[6]. As failure information propagates to agents working on subtasks, these agents may *autonomously compensate* for a subtask's activities as appropriate. Figure 6(b) suggests a task delegation interaction. The FHC performs a number of domain-independent operations on this (distributed) structure.

- It records changes in task status and new subtask nodes (as communicated by the agent application).
- It manages initiation of new failure-handling goals, including creation of new FHC nodes, their relationships to existing nodes, and task invocation.
- It notifies a parent task node when its child has failed or succeeded; and *failure* of a node triggers initiation of the task's associated stabilization goals with the agent application.
- When a task is *canceled*, *halt* instructions are propagated to its child nodes and their associated application agents.
- When a task is *canceled*, notification of task cancellation is propagated to its child nodes. (The agent's strategy rules determine whether a compensation for a given canceled subtask will be initiated).

[6] Currently, the 'failure-handling ontology' with which we annotate the agent messages, as well as our failure-handling interaction protocols, are ad-hoc. Work is in progress to formalize both.

– It marks failed tasks as canceled, and initiates associated compensations of canceled tasks, as instructed by its strategy rules.
– It initiates task retries as instructed by its strategy rules.

These mechanisms, built into the FHC, provide a support framework for utilizing the agent's failure-handling knowledge and recording necessary history. An agent developer does not need to replicate similar mechanisms in every agent application nor ensure that they are applied consistently.

3.3 Retries and Failure-Handling Goals

Section 2 described the concept of strategy rule sets, which specified when the agent's FHC should initiate various repairs and retries. Here, we provide examples of how retries and compensations are supported by the FHC.

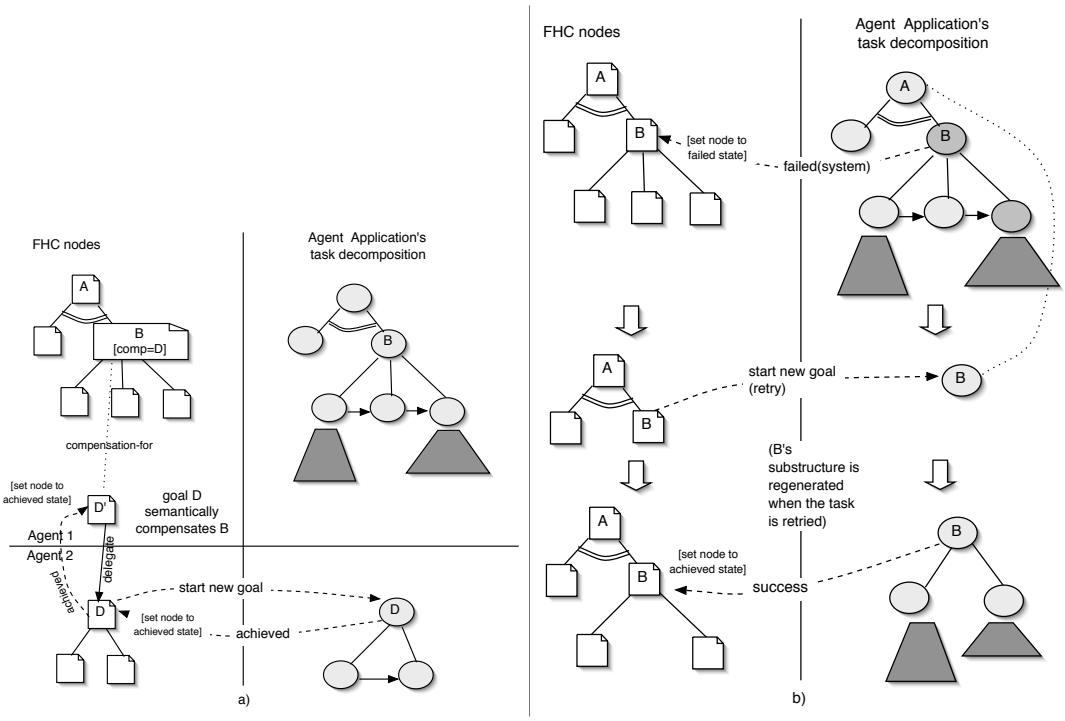

Fig. 7. Compensation and retry scenarios.

Figure 7(a) shows a compensation scenario. In the FHC of Agent 1, the decision is made to cancel subtask B– this may be due to a failure in executing B, or due to a failure in another component of the larger task. B's associated failure-handling definitions indicate that goal D should be used to compensate it. E.g., in the party-planning domain, B might be the "invite friends" subgoal, with a compensation of "cancel with guests". For simplicity of the example we show one compensation goal; there can be multiple goals. D is instantiated with respect to B's bindings, and a new task tree node is created for D in A1. The FHC logs that this new task is the *compensation-for* B. A1's FHC interacts with a system broker to locate some agent that can implement D. Agent 2 is identified, and D is assigned to A2. For the compensation task to be successfully implemented, A2 must be able to determine what needs to be done to compensate for (a perhaps partially executed) B. Note that the execution of D, in A2, may cause additional compensatable subtasks to be created and monitored in A2's FHC.

Figure 7(b) shows a retry scenario. The agent application reports a (system) failure of subtask B to the FHC. For simplicity of the example, no *stabilization* goals have been defined for B– if they had, then a scenario similar to the compensation example would have have occurred first, in order to perform a stabilization task immediately after failure. The FHC decides to retry task B. It instructs the agent application to perform this goal again. It also informs the agent that this is the (Nth) retry of the goal– the agent can take advantage of the fact that the goal is a retry, but is not explicitly required to. Note that it is the *domain knowledge of the agent application*, not of the FHC, which is utilized to figure out *how* to perform the retry. That is, it is the agent application that retains the knowledge of contingency plans– alternative ways to try to accomplish a goal [5]. Note also that the FHC's failure-handling knowledge is used *in conjunction with the agent application's knowledge* of how to deal with problems. The agent application will not report goal failure to its FHC until it runs out of ways to handle the problem itself. However, once it reports failure, a consistent failure-handling strategy, based on the FHC definitions, is then employed system-wide.

In both the examples above, the FHC– not the agent application– is responsible for selecting the agents that will be tasked to perform retry, compensations, and stabilization goals, and for initiating the problem-solving by these agents. Interaction with the agent application that originally performed the task is not (necessarily) required. Under some circumstances, it may make sense to delegate a task, e.g. a compensation task, to the same agent that performed the original corresponding task; this agent may be able to leverage its internal state to perform the cancellations more effectively, or may be already "near" the objects that need to be affected. In other circumstances, e.g. when retrying certain tasks, it may make sense to delegate the retry to an alternate agent that might be able to implement the task in a different way.

3.4 Requirements on the Agent Applications and Agent System

Our goal-based agent/FHC interface imposes certain requirements on the agent application. For an agent to implement the interface "correctly", the following must be true:

- The agent must be able to perform goal-based problem-solving.
- An agent application must attempt to implement the goal-based instructions it receives from its FHC[7]. These instructions may include the *halt* of an existing task as well as new goal initiations.
- To robustly support retries, compensations, and stabilization goals, an agent must be able to determine whether or not a given goal needs to be achieved; whether it has failed, and in what failure mode. (In general, detection of failure can be undecidable; however, the FHC will infer failure if the agent does not respond with task status within a given domain-dependent time interval.)
  Ideally, this determination of goal status will be based on information globally available to the agent system, *not* on a given agent's local state; this allows agents to "take over" compensating another agent's activities if necessary. As a simple example, an agent tasked to cancel a reservation made by a second agent should be able to detect whether or not the reservation exists; the information it requires to make this determination needs to be globally available to all agents, since it is not known a priori which agent will do the cancellation– it may not be the same one that made the reservation. The greater the extent to which task results are globally persisted, the more effective the system's failure-handling will be. A detailed discussion of this issue is beyond the scope of this paper. However, it has close ties to the mechanisms necessary to support crash recovery, and thus increasing the ability of the agents in the system to globally persist task result information increases both their robustness and their recovery capabilities. This is a current research focus of our project.

In addition to the requirements that the agent applications must meet, we assume that the multi-agent system supports some form of brokering or service discovery [6]. The agents' FHCs may autonomously instruct other agents to perform failure-handling tasks; these agents must be locatable based on the description of the task, or goal, that they will be asked to perform. In a system of fixed agents, this information can be hardwired, but in an open system it must be discovered dynamically.

[7] In this paper, we do not explore issues of levels of commitment or goal prioritization.

3.5 Prototype Implementation

We have implemented a prototype multi-agent system in which for each agent, an FHC interfaces with an agent application logic layer. The agents are implemented in Java, and the FHC of each agent is implemented using a Jess core [7], with the strategy rule sets of the FHC implemented as Jess rule sets. The agent-application components of our prototype are simple goal-based problem-solvers to which an implementation of the FHC interface was added. Currently, the transaction layer/agent application interface uses Jess "fact" syntax to communicate goal-based events. Inter-agent communication in our prototype system is primitive; in the future we plan to situate the implementation of the FHC on an established agent system framework, which can provide communication, brokering, and interaction protocol support. Using our prototype, we have performed initial experiments in several relatively simple problem domains. Preliminary results are promising, and suggest that our approach to defining and employing goal-based failure-handling information generates useful behavior in a range of situations. Work is ongoing to define failure-handling knowledge and strategies for more complex problem domains in which agent interaction will feature prominently. In the process, the FHC framework will be used as a testbed for evaluating and comparing different failure-handling strategies. Section 5 discusses some of the issues we expect to encounter and investigate in these more complex domains, and discusses future work in the larger project context.

4 Related Work

Our approach is motivated by a number of transaction management techniques in which sub-transactions may commit, and for which forward recovery mechanisms must therefore be specified. Examples include open nested transactions [1], flexible transaction [5], SAGAs, [8], and ConTracts [9]. Earlier related project work has explored models for the implementation of transactional plans in BDI agents [10–13], and a proof-of-concept system using a BDI agent architecture with a closed nested transaction model was constructed [14].

In Nagi et al. [15], [16] an agent's problem-solving drives 'transaction structure' in a manner similar to that of our approach (though the maintenance of the transaction structure is incorporated into their agents, not decoupled). However, they define specific compensation plans for (leaf) actions, which are then invoked automatically on failure. Thus, their method will not be appropriate in domains where compensation details must be more dynamically determined.

Parsons and Klein [17] describe an approach to MAS exception-handling utilizing sentinels associated with each agent. For a given specific domain, such as a type of auction, "sentinels" are developed that intercept the communications to/from each agent and handle certain coordination exceptions for the agent. All of the exception-detecting and -handling knowledge for that shared model resides in their sentinels. In our approach, while we decouple high-level handling knowledge, the agent retains the logic for failure detection and task implementation; agents may be added to a system to handle certain failures if need be.

Chen and Dayal [18] describe a model for multi-agent cooperative transactions. Their model does not directly map to ours, as they assume domains where commit control is possible. However, the way in which they map nested transactions to a distributed agent model has many similarities to our approach. They describe a peer-to-peer protocol for failure recovery in which failure notifications can be propagated between separate 'root' transaction hierarchies (as with cooperating transactions representing different enterprises).

[19] describe a model for implementing compensations via system ECA rules in a web service environment– the rules fire on various 'transaction events' to define and store an appropriate compensating action for an activity, and the stored compensations are later activated if required. Their event- and context-based handling of compensations have some similarities to our use of strategy rules. However, in their model, the ECA rules must specify the compensations directly at an action/operation level prior to activation (and be defined by a central authority).

WSTx [20] addresses transactional web services support by providing an ontology in which to specify *transactional attitudes* for both a service's capabilities and a client's requirements. A WSTx-enabled 'middleware' may then intercept and manage transactional interactions based on this service information. In

separating transactional properties from implementation details, and in the development of a transaction-related capability ontology, their approach has similar motivations. However, their current implementation does not support parameterized or multiple compensations.

Workflow systems encounter many of the same issues as agent systems in trying to utilize transactional semantics: advanced transactional models can be supported [21], but do not provide enough flexibility for most 'real-world' workflow applications. Existing approaches typically allow user- or application-defined support for semantic failure atomicity, where potential exceptions and problems may be detected via domain rules or workflow 'event nodes', and application-specific fixes enacted [22, 23]. Recent process modeling research attempts to formalize some of these approaches in a distributed environment. For example, BPEL&WS-Coordination/Transaction [24] provides a way to specify business process 'contexts' and scoped failure-handling logic, and defines a 'long-lived transaction' protocol in which exceptions may be compensated for. Their scoped contexts and coordination protocols have some similarities to our failure-handling strategy rules and FHC interaction protocols.

5 Discussion and Future Work

We have described an approach to improving robustness in a multi-agent system. The approach is motivated by transactional semantics, in that its objective is to support *semantic compensations* for tasks in a given domain; we assume environments in which we cannot wait to "commit" actions. We augment an agent's failure-handling capabilities by improving its ability to "clean up after" and undo its failures, and to support retries. This behavior makes the semantics of an agent system more predictable, both with respect to the individual agent and with respect to its interactions with other agents; thus the system can become more robust in its reaction to unexpected events.

Our approach is goal-based, both with respect to defining failure-handling knowledge for agent tasks, and in deciding when to employ it. Goal-based events, in conjunction with failure-handling goal definitions, are used to trigger domain-dependent *strategy rules* which determine when to achieve a compensation or stabilization goal. Agents implementing subtasks may autonomously repair local problems as appropriate. By abstracting the agent's failure-handling knowledge above the level of task plans, it can be decoupled from the individual agent implementations, and is employed via the use of a FHC with which the agent application interfaces. The FHC makes autonomous decisions about what failure-handling knowledge to employ, and when to do so, but utilizes the agent's domain knowledge to determine how to achieve or implement, the failure-handling goals and retries. The agent applications may attempt to resolve problems themselves first– failures are only "pushed" to the FHC if that does not happen.

Our methodology separates the definition of goal-based failure-handling knowledge for tasks, from the agents' implementations of those tasks. Our hypothesis, borne out by our initial experiments, is that in many situations this knowledge can in fact be productively separated. One goal of our ongoing tests will be to characterize for what types of domains and scenarios this approach works, and when it is not as appropriate. Our experiments will also help us to evaluate the ways in which a failure-handling strategy is tied to the problem representation. For example, task/subtask granularity influences retry strategies.

Each agent's FHC works to handle failures generated during the agent's problem-solving, thus making the agents in the system locally more robust. However, the use of compensation and stabilization goals for the tasks in a domain can have global benefits as well. They effectively "clean up" the tasks' effects on failure. Thus, system robustness is increased, both with respect to resource management– resources that are no longer needed are reliably released– and with respect to *predictability* of the agents' environment. This latter is a subtle but important point: the agents, modeling the "world" of a given domain, will make implicit assumptions about that domain. The greater the extent to which the agents' world can be returned to a state that reflects those implicit assumptions, the fewer problems the agents will encounter and the more effective they can be.

This project is in its early stages. Work will continue in evaluating our failure-handling methodologies, and to further develop our prototype. Evaluation will include development of scenarios in additional domains– with emphasis on scenarios that require multi-agent interaction; analysis and characterization of failure-handling strategies (including strategies for dealing with cascading failures and analysis of the overhead

incurred by the use of the FHC infrastructure); and will also include tests in which we "plug in" different application agent architectures on top of the FHC, e.g. a BDI agent [10]. In the process, we anticipate the development of a failure-handling ontology with which to exchange information and advertise repair capabilities; and the formalization of inter-FHC interaction protocols to exchange failure-handling information between both parent-child and peer tasks [20], [25], [24].

In the larger scope of our project, a number of research directions are planned. These include extension of the FHC to support crash recovery mechanisms and aspects of task concurrency management. Recovery in particular is closely tied to the failure-handling approach described here; both with respect to requirements on detection of goal status, and in that structures produced by the FHC's task-monitoring mechanisms may serve as a key input to the recovery process as well. In addition, we plan the development of a language and computational model in which aspects of our approach will be formalized.

Acknowledgments. This research is funded by an Australian Research Council Discovery Grant.

References

1. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)
2. FIPA: http://www.fipa.org
3. Nodine, M., Unruh, A.: Facilitating open communication in agent systems. In Singh, M., Rao, A., Wooldridge, M., eds.: Intelligent Agents IV: Agent Theories, Architectures, and Languages. Springer-Verlag (1998)
4. Dayal, U., Hanson, E., Widom, J. In: Active database systems. (1995)
5. Zhang, A., Nodine, M., Bhargava, B., Bukhres, O.: Ensuring relaxed atomicity for flexible transactions in multidatabase systems. In: Proceedings of the 1994 ACM SIGMOD international conference on Management of data, Minneapolis, Minnesota, United States, ACM Press (1994) 67–78
6. Cassandra, A., Chandrasekara, D., Nodine, M.: Capability-based agent matchmaking. In Sierra, C., Gini, M., Rosenschein, J.S., eds.: Proceedings of the Fourth International Conference on Autonomous Agents, Barcelona, Catalonia, Spain, ACM Press (2000) 201–202
7. Friedman-Hill, E.: Jess in Action. Manning Publications Company (2003)
8. Garcia-Molina, H., Salem, K.: SAGAs. In: ACM SIGMOD Conference on Management of Data. (1987)
9. Reuter, A., Schwenkreis, F.: Contracts - a low-level mechanism for building general-purpose workflow management-systems. Data Engineering Bulletin **18** (1995) 4–10
10. Rao, A.S., Georgeff, M.P.: An abstract architecture for rational agents. In: *Third International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann (1992)
11. Busetta, P., Bailey, J., Ramamohanarao, K.: A reliable computational model for BDI agents. In: 1st International Workshop on Safe Agents. Held in conjunction with AAMAS2003. (2003)
12. Ramamohanarao, K., Bailey, J., Busetta, P.: Transaction oriented computational models for multi-agent systems. In: 13th IEEE International Conference on Tools with Artificial Intelligence, Dallas, IEEE Press (2001) 11–17
13. Smith, V.: Transaction oriented computational models for multi-agent systems. Internal Report, University of Melbourne (2003)
14. Busetta, P., Ramamohanarao, K.: An architecture for mobile BDI agents. In: 1998 ACM Symposium on Applied Computing. (1998)
15. Nagi, K., Nimis, J., Lockemann, P.: Transactional support for cooperation in multiagent-based information systems. In: Proceedings of the Joint Conference on Distributed Information Systems on the basis of Objects, Components and Agents, Bamberg (2001)
16. Nagi, K., Lockemann, P.: Implementation model for agents with layered architecture in a transactional database environment. In: AOIS '99. (1999)
17. Parsons, S., Klein, M.: Towards robust multi-agent systems: Handling communication exceptions in double auctions. In: Submitted to The 2004 Conference on Autonomous Agents and Multi-Agent Systems. (2004)
18. Chen, Q., Dayal, U.: Multi-agent cooperative transactions for e-commerce. In: Conference on Cooperative Information Systems. (2000) 311–322
19. T. Strandens, R.K.: Transaction compensation in web services. In: Norsk Informatikkonferanse. (2002)
20. Mikalsen, T., Tai, S., Rouvellou, I.: Transactional attitudes: Reliable composition of autonomous web services. In: Workshop on Dependable Middleware-based Systems. (2002)
21. Alonso, G., Agrawal, D., El Abbadi, A., Kamath, M., Gunthor, R., Mohan, C.: Advanced transaction models in workflow contexts. In: ICDE. (1996)

22. Casati, F.: A discussion on approaches to handling exceptions in workflows. SIGGROUP Bulletin **Vol 20. No. 3** (1999)
23. Rusinkiewicz, M., Sheth, A.P.: Specification and execution of transactional workflows. Modern Database Systems: The Object Model, Interoperability, and Beyond (1995) 592–620
24. Curbera, F., Khalaf, R., Mukhi, N., Tai, S., Weerawarana, S.: The next step in web services. COMMUNICATIONS OF THE ACM **Vol. 46, No. 10** (2003)
25. Nodine, M., Unruh, A.: Constructing robust conversation policies in dynamic agent communities. In: Issues in Agent Communication. Springer-Verlag (2000)

Appendix

- Agent application to FHC notifications:
  - Notification of new problem-solving goal (including parent goal and delegation information)
  - Messages to other agents (will be routed through the FHC)
  - notification of change in goal status as problem-solving progresses: *achieved*, *failed* (including failure *mode* as available), or *halted*.
  - (implicit) timeout (no response from agent in given time interval)
- FHC to agent application notifications:
  - Instructions to start work on a goal, and information about whether the goal is a *retry* (as discussed in Section 3.3)
  - Instructions to stop work on a goal
  - Notification of change in delegated goal status: *achieved*, *failed*, or *halted*.
  - Create and start a new goal(generated from a task's associated failure-handling goal definitions).
  - Messages from other agents

Fig. A-1. Goal-based events supported by the interface between the agent application and its failure-handling component. The reported events must be expressed in a shared system ontology in order to support task-related communication with other agents.