Managing Semantic Compensation in a Multi-Agent System

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. Recovery in agent systems is an important and complex problem. This paper describes an approach to improving the robustness of an agent system by augmenting its failure-handling capabilities. The approach is based on the concept of semantic compensation: "cleaning up" failed or canceled tasks can help agents behave more robustly and predictably at both an individual and system level. However, in complex and dynamic domains it is difficult to define useful specific compensations ahead of time. This paper presents an approach to defining semantic compensations abstractly, then implementing them in a situation-specific manner at time of failure. The paper describes a methodology for decoupling failure-handling from normative agent logic so that the semantic compensation knowledge can be applied in a predictable and consistent way– with respect to both individual agent reaction to failure, and handling failure-related interactions between agents– without requiring the agent application designer to implement the details of the failure-handling model. In particular, in a multi-agent system, robust handling of compensations for delegated tasks requires flexible protocols to support management of compensation-related activities. The ability to decouple the failure-handling conversations allows these protocols to be developed independently of the agent application logic.

1 Introduction

The design of reliable agent systems is a complex and important problem. One aspect of that problem is making a system more robust to failure. 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.

Our research is motivated by this principle: we would like to make a system of interacting agents more robust, by improving its failure-handling behavior. We would also like agent designers to be able 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 logic; and by providing an underlying support mechanism which takes care of the details of the failure-handling.

However, in most multi-agent domains, principles of transaction management can not be directly applied. The effects of many actions may not be delayed: such actions "always commit", and thus correction of problems must be implemented by "forward recovery", or failure *compensation* [1]– that is, by performing additional actions to correct the problem, instead of a transaction rollback. In addition, actions may not be "undoable" nor repeatable. Further, it is often not possible to enumerate how the tasks in a dynamic agent system might unfold ahead of time: it is not computationally feasible, nor do we have enough information about the possible "states of the world" to do so.

In this paper, we focus on one aspect of behavior motivated by transactional semantics, that of approximating failure atomicity by *semantic compensation*: improving the ability of an agent system to recover from task problems by "cleaning up after" or "undoing" its problematic actions. The use of semantic compensation in an agent context has several benefits:

- It helps leave an agent in a state from which future actions– such as retries, or alternate methods of task achievement– are more likely to be successful;
- it helps maintain an agent system in a more predictable state: agent interactions are more robust; and unneeded resources are not tied up;
- it can often be applied more generally than methods which attempt to "patch" a failed activity;
- and (in the context of our longer-term project goals) it allows a long "transaction" to be split up into shorter ones, with less chance of deadlocks, and higher concurrency.

We introduce the concept of semantic compensation by presenting an example in a "dinner party" domain. (We use this domain because it has a rich semantics and is easily understandable; its issues can be mapped to analogous problems in more conventional domains). Consider two related scenarios, where a group of agents must 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. Fig. 1A shows one task decomposition for such an activity. The figure uses an informal notation, where subtasks that may be executed concurrently are connected by a double bar; otherwise they are executed sequentially. 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.

Next, consider a scenario which differs in that the party will be held at the host's house. In this case, while the party must be scheduled, a hall does not

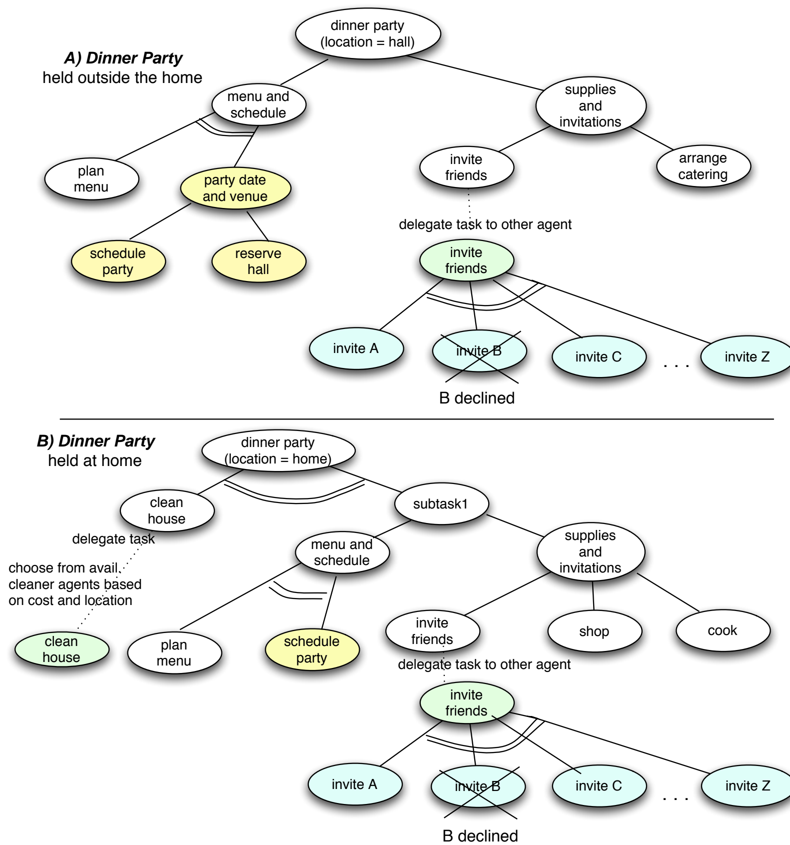

Fig. 1: Planning a dinner party.

need to be reserved. In addition, the hosts will not have the party catered and will shop themselves. Fig. 1B shows a task decomposition for this scenario.

If the party planning fails or the event must be canceled, then a number of things might need to be done to properly take care of the cancellation– that is, to "compensate for" the party planning. However, the specifics of these activities will be different depending upon what has been accomplished prior to cancellation. In the first case (Fig. 1A) we may have to cancel some reservations, but if we have used caterers, then we will not have to deal with any extra food; in the second case (Fig. 1B), we have no reservations to cancel, but we will likely have unused food. In either event, the party cancellation activities can be viewed as accomplishing a *semantic compensation*; clearly, compensation activities must address agent task semantics. An exact 'undo' is not always desirable– even if possible. Further, compensations are both context-dependent and usually infeasible to enumerate ahead of time, and employing a composition of subtask compensations is almost always too simplistic.

In this paper, we describe two primary aspects of our approach to semantic compensation. First, we present a method for operationalizing the concept of

semantic compensation for an agent system. This is accomplished via *goal-based* specification of failure-handling knowledge. 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.

Second, for semantic compensation to be effective, it must be employed consistently and predictably across an agent system, with respect to how individual agents react when a task fails or is canceled, and how the agents in the system *interact* when problems develop with a delegated task. We claim that a fixed method of assigning responsibility for task compensations is not sufficiently robust, and that inter-agent *protocols* are required to determine responsibility.

We implement predictable semantic compensation by factoring an agent's failure-handling from its normative behavior. We define a decoupled, platform-independent agent component that uses goal-based compensation knowledge to support failure management. We refer to this component as the agent's FHC (Failure-Handling Component). The FHC performs 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[1]. As shown in Fig. 2, the FHC sits conceptually below the agent's domain logic component, which we refer to as the "agent application". Analogously to exception-handling in a language like Java, the FHC 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.

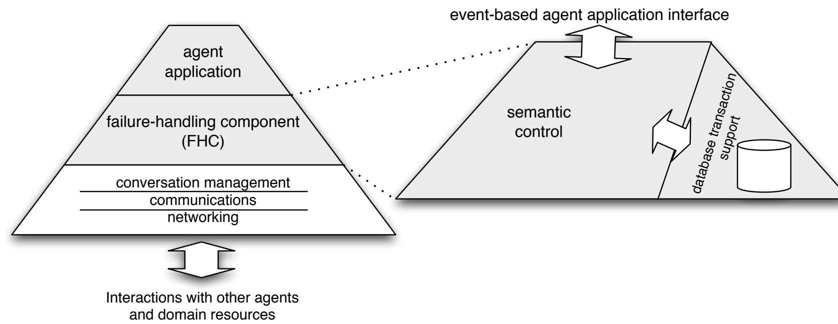

Fig. 2: An agent's FHC. We refer to the domain logic part of the agent, above the failure-handling component, as the "agent application".

In Section 2, we describe our approach to goal-based definition of failure-handling knowledge. Section 3 outlines how the FHC uses this knowledge to provide robust and well-specified reactions to task failures and cancellations,

[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.

and to support predictable and consistent failure-handling-related interactions between the agents in the system, independent of changes in the 'core' agent application logic. In Sections 4 and 5 we finish with a discussion of related work, and conclude.

2 Goal-Based Semantic Compensation

The example of Section 1 suggested how compensation of a high-level task can typically be achieved in different ways depending upon context. It is often difficult to identify prior to working on a task the context-specific details of how a task failure should be addressed or a compensation performed. It can be effectively impossible to define all semantic compensations prior to runtime in terms of specific actions that must be performed.

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, thus making these definitions more widely applicable. We associate *failure-handling goal definitions* with some or all of the tasks (goals) that an 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* a goal is accomplished. The way in which these goals are achieved, for a specific scenario, will depend upon context.

Figures 3A and 3B illustrate this idea. Suppose that in the party-planning examples of Section 1, the host falls ill and the party needs to be canceled. Let the compensation goals for the party-planning task be the following:

– all party-related reservations should be canceled;
– extra food used elsewhere if possible; and
– guests notified and 'apologies' made.

The figures show the resulting implementations of these compensation goals for the activities in Figs 1A and 1B. Note that the compensation goals for the high-level party-planning task *are the same in both cases*, and indicate what needs to be made true in order for the compensation to be achieved. However, the *implementations* of these goals will differ in the two cases, due to the different contexts in which the agent works to achieve them. When the party was to be held at home (Fig. 3B), extra food must be disposed of, and (due to the more personal nature of the party) gifts are sent to the guests. In the case where the party was to be held in a meeting hall (Fig. 3A), there are reservations that need to be canceled. However, there is no need to deal with extra food (the caterers will handle it) nor to send gifts. Some compensation goals may be already satisfied and need no action, and some tasks may be only partly completed at time of cancellation (e.g. not all guests may be yet invited).

Note that a semantic compensation may include goals that do not address any of the (sub)tasks of the original task, such as the gift-giving in the first

[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.

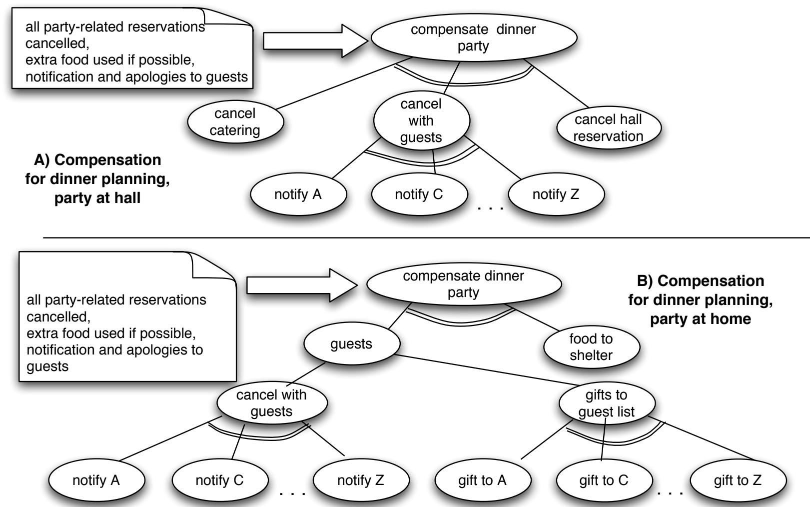

Fig. 3: Compensation of dinner party planning.

example. Some compensation activities may "reverse" the effects of previous actions (e.g. canceling the reservation of a meeting hall), but other previous effects may be ignored (no effort is made to "undo" the effects of the house-cleaning) or partially compensatable (dealing with the extra food) depending upon context. The definition of such a semantic compensation is task- and domain-specific.

A goal-based formulation of failure-handling knowledge has several benefits:

- 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 the compensations to be employed in dynamic domains in which it is not possible to pre-specify all relevant failure-handling plans.

In order to use goal-based compensation in an agent context, the agent developer must provide domain-dependent information prior to runtime. For each (sub)task of an agent's for which failure-handling will be enabled, a set of parameterized failure-handling goals– not plans– must be associated with the task. Our model allows two types of failure-handling knowledge to be associated with each task: goals whose achievement is triggered by the task's failure (*stabilization* goals, which perform immediate local "cleanup"); and *compensation* goals triggered by cancellation. (Cancellation may but need not result from failure).

The goal-based definitions are then instantiated (bound) and used at runtime by the agent's FHC to direct the agent's behavior in failure situations. As will be further described in Section 3, the FHC can present new compensation goals to its agent application, which implements the goals according to context[3]. Failure

[3] If an agent application is given a compensation goal to achieve, this does not necessarily mean that direct work on the goal will be immediately initiated: it may have unmet preconditions.

handling is triggered in the FHC by the agent application's problem-solving and/or task cancellation.

Default failure-management is top-down: when a high-level task is canceled, the agent is given that task's (high-level) compensation goals to achieve. However, the agent's FHC may additionally be provided with information about when to directly employ compensations for subtasks of a canceled task; when to *retry* a failed task (allowing other alternatives to be tried by the agent application); and when not to compensate a failed task; in the form of a set of event-driven, domain-dependent *strategy rules*. The strategy rules refine the FHC's default failure-handling behavior, and allow localized compensations and retries to be spawned.

We view the process of constructing these definitions, and associated strategy rules, as "assisted knowledge engineering"; we can examine and leverage the agent's domain knowledge to support the definition process. (We are researching ways to provide semi-automated support for this process). Because the FHC enforces an explicit and straightforward use of this 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.

Such failure-handling knowledge can be added to an agent system *incrementally*, allowing a progressive refinement of its knowledge about how to react in failure situations, which it takes advantage of when applicable– otherwise, its behavior is as before. The failure-handling knowledge augments, not overrides, the agent's domain logic.

This section provided an overview of a foundation of our approach: the employment of goal-based– rather than plan- or action-based– definitions of semantic compensations. More details are provided in [2]. Our methodology separates the definition of task failure-handling knowledge from the agents' implementation of that knowledge, allowing the compensations to leverage runtime context in a way that makes them both more robust and more widely applicable.

3 Managing Compensations of Delegated Tasks

The previous section described a method for defining compensations in dynamic and complex agent environments. In this section, we describe a methodology for supporting consistent and predictable system behaviour on failure, while reducing what needs to be done by the agent developer to "program" the agent system to handle failure. To accomplish these goals, we separate each agent's "normative" from failure-handling knowledge by means of a decoupled component that:

- is not tied to a specific agent architecture, but leverages the agent's problem-solving knowledge
- determines *when* to invoke task compensation goals or retries based on the agent's activities and task status
- determines *what* goals to initiate, based on failure-handling knowledge; and
- provides support for multi-agent task compensation scenarios

As introduced in Fig. 2, we label this component the agent's FHC. The FHC maintains an abstraction of the agent's problem-solving history to support its failure management. In this paper we focus on one aspect of that failure handling: the agent's interactions, and how the FHC allows compensation-related interaction protocols to be factored from normative agent conversations.

The need for failure-handling protocols as a core part of the failure-handling methodology can be illustrated by considering compensation scenarios. A task delegation generates an implicit compensation scope for a task; potentially, either the delegator or the 'delegatee'– the agent to which the task was assigned– could be in charge of a compensation if it is later required. Most approaches suggest that a specific 'failure handler' agent/service be used for each activity [3], or that the agent/service that performed the original task will be responsible for its compensation should the need arise [4]. However, no fixed approach for determining which agent should be responsible for compensation, is appropriate all of the time. The agent that performed the original task may be too busy to perform the compensation, unable to perform the compensation, or currently offline/unreachable. If the agent failed at the original task, it should perhaps not take on the compensation of that task. However, we do not want to automatically target a separate failure-handling agent: often the agent that performed the 'forward' task will be the best suited to implement its compensation. Any approach should also accommodate situations where the *delegating* agent is offline.

Consider again the "dinner party" example of Section 2. The `invite guests` task is delegated by the primary "party planning" agent to an "invitation" agent. If the party needs to be canceled, then as part of the compensation process, the `invite guests` task will be compensated by canceling with all confirmed/pending guests. As a default, it makes sense for the invitation agent to contact the guests again– it may have retained internal state useful to the task– but not if it is overloaded or offline. Alternatively, the invitation agent might have failed (and contributed to the failure of the party planning task). In this case, the delegating agent (if online) prefers that the original invitation agent *not* be responsible for the invitation cancellations.

Before compensation for a task can proceed, *responsibility* for the compensation needs to be assigned to one of the agents in the compensation scope. Such an assignment does *not* indicate which agent will actually perform the task; once an agent is responsible for a compensation, it may delegate it. The example above illustrates that a fixed method for assigning compensation responsibility will not always be appropriate. It is more robust to require the relevant agents in the scope of the compensation activity to mutually determine which will be responsible. For this, an interaction *protocol*– supporting a conversation between the agents– is required.

Below, we describe the agent's FHC, and then detail the way in which it is used to support a set of factored compensation-related agent interaction protocols. We describe one of the protocols used by our system, which allows delegated compensations to be robustly managed.

3.1 The Agent's Failure-Handling Component (FHC)

A key aspect of our failure-handling model is the use of an abstract, goal-level representation of the agent's activities. This allows us to support a decoupled and architecture-independent mechanism for managing goal-based compensations. As suggested in Fig. 2, we define a model in which an agent application (the agent's domain logic) sits upon its FHC. The FHC supports a platform-independent API based on a goal-level failure-handling *ontology*, which allows goal instantiation and status information to be exchanged with the agent application. The agent application provides notifications on new goals and goal achievement/failure (with failure modes) to the FHC, and the FHC may introduce new goals to the agent application, to initiate both compensations and retries. In addition, all messages to/from the agent are filtered through its FHC, as will be further described below.

Based on the information from the agent, the FHC maintains a goal-level history of task/subtask information for currently relevant agent tasks: for each such 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. Each node in an FHC task-monitoring tree corresponds to a task subgoal. A node may be in one of the states shown in Fig. 4. The FHC uses this task structure in conjunction with the goal-level failure-handling knowledge described in Section 2, and domain events, to support compensations, task retries/alternatives, and management of task failure and cancellation events. (When a task is canceled, the agent application is instructed to halt all work on it). Compensations cause new task nodes to be created, and compensation tasks may themselves support compensations or retries.

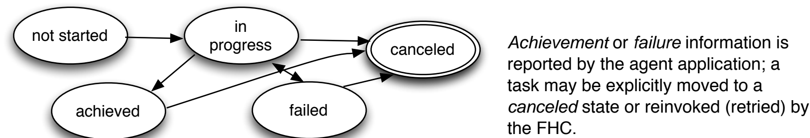

Fig. 4: FHC task node states. 'Canceled' indicates the state of the associated task node only; *not* the status of any corresponding compensation or stabilization activities.

Any agent application which correctly supports this API, and for which failure-handling knowledge is provided, may be "plugged into" the FHC; it is not architecture-specific. The agent application performs planning, task decomposition and execution. The FHC tracks task decomposition and reacts to task failures– reported via the API– by instructing agents to achieve repair goals. That is, an agent's FHC makes decisions about *what* failure-handling goals should be achieved, and *when* they will be requested of the agent. The agent's application logic is then invoked to implement the tasks and determine the de-

tails 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 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.

[2] provides additional detail, describes the API, and discusses the requirements on an agent application to correctly support the interface with the FHC. In particular, the agent must utilize a goal/subgoal representation of its task problem solving, and communicate changes in this information to its FHC. It must also be able to determine whether or not a given goal is already achieved, and support instructions to start/halt work on a goal.

Below, we focus on one specific aspect of the FHC. Its representation and maintenance of goal-level agent information allows failure-handling interaction protocols to be specified and implemented orthogonally from the agent application logic, to the benefit of system robustness and behavioral consistency. In the following, we assume that, as shown in Fig. 2, the agent architecture includes a "conversation" layer, which ensures that the agent's incoming/outgoing messages adhere to the system's prescribed interaction protocols [5, 6]. We assume that this layer supports 'are you alive' pings to the agent(s) participating in active conversations, and generates error events if these other agents are not reachable.

3.2 Compensation Interaction Protocols

The agent application's 'regular' interaction protocols will determine how subtasks are allocated and delegated. These protocols may involve negotiation or bidding [5] or market mechanisms [7], as well as direct sub-task assignments or subscriptions, and typically include the reporting of error/task results.

We do not require the FHC to understand the semantics of these conversations. We separate failure-handling conversations the agent's 'regular' protocols, and implement them in the FHC. Based on the way in which we decouple the FHC from the agent application, and the way in which the FHC represents and maintains high-level task information as communicated from the agent, we are able to support failure-handling protocols in a manner transparent to the agent application. This factoring obviates the need for the agent application to implement handling the compensation-related protocols itself. The agent application developer does not have to build the agent applications to support these protocols, and as long as the agent's implementation of its FHC API is correct, the protocol's correct implementation is *independent of changes* to the agent logic. Similarly, the compensation-related interaction protocols may be changed without impacting the agent application.

To implement robust compensation-related interactions via the agent's FHCs, we have two requirements. First, the FHC must be able to explicitly detect con-

versation 'timeout' events, when an agent is offline (down or unreachable). As described above, we assume this information is provided by the agent's underlying conversation layer. Second, we must be able to associate, or "connect" the pair of task nodes– one in the delegating agent's FHC task tree structure and one in the delegatee's– that correspond to the same delegated task, without requiring the FHCs to parse the conversations that led to the delegation. With this, compensation information can propagate from one agent's FHC to another using the associated nodes.

"Connecting" delegated tasks across agents. To connect FHC task nodes across delegations, we make three requirements of the agent application as part of its implementation of the FHC API. First, we require that agents represent a task to be allocated explicitly *as* a task, so that they can communicate the creation of such tasks to their FHC. Second, we require that the agents 'know' when they are sending out messages related to assigning or allocating a task, and can associate those messages with their local representation of the task. Third, we require that receiving agents know when they are creating a task *based on* an incoming message, and are able to associate this new task with the relevant message ID (MID). This level of awareness on the part of the agent application is necessary to allow the FHC to operate independently of any specific agent delegation protocols. We then require the agent to implement the following aspects of the FHC API to support task node association:

1. When an agent begins a task delegation activity for which communication will be required, it notifies the FHC of the new task (goal). The FHC will build a new task node associated with the task.
2. When the agent sends out messages related to delegation of that task, it associates these messages– which are passed via the FHC– with the UID of the new goal. The FHC annotates the outgoing message information with the UID of the corresponding FHC task node (TNID).
3. The FHCs of the receiving agent(s) process the message envelope to record MID/TNID associations.
4. If an agent is assigned a delegated task (either directly or after an exchange), the delegating agent informs its FHC of the assignment, and the receiving agent annotates its `new task` notification to its FHC with the MID of the message by which the task was assigned. Based on its MID/TNID bookkeeping, the FHC of the receiving agent will create a new task node with an ID that links it to the parent task node in the delegating agent.

Fig. 5 illustrates this process with an example ContractNet-like delegation protocol. The FHC is not required to parse the agent's messages; it is only necessary for the agent application to implement the interface correctly with respect to indicating associated task/delegation message correspondences. It is this bookkeeping that allows compensation-related protocols to be factored from the agent's regular conversations. The agent's normal conversations take care of any task delegation for a given scenario, but the FHCs of the agent involved ensure that the delegator/delegatee relationship for a task is made explicit. If

a task later needs to be compensated, both delegator and delegatee can can be involved in determining responsibility.

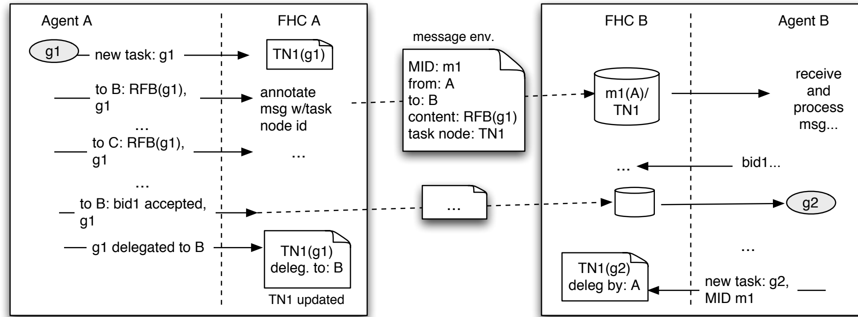

Fig. 5: "Connecting" a related task nodes in two agent's FHCs, with an example bid delegation protocol. Agent A's application logic associates RFB (request for bid) messages with their associated subtask ID (`g1`), and communicates this to its FHC. Agent A's FHC annotates these outgoing messages with `g1`'s task node ID (`TN1`). When B's bid is accepted, A's FHC notes the delegation, and B's FHC uses the request MID to associate the new task with `TN1`.

Failure-Handling Protocols. The protocols related to compensation scope utilize inter-agent FHC task node associations, and must encompass several related types of communications necessary to support predictable failure-handling in delegated task contexts. These include:

1. When a canceled task is to be compensated, determination of which agent will be responsible for the compensation.
2. Propagation of a task cancellation notification to a sub-task delegatee and collection of the cancellation results by the parent task. (Cancellation requires task 'halt'; sub-task agents report if cancellation was successful. Recall that "canceled" refers to the FHC task node for the original task, not the status of any subsequent compensation tasks).
3. Notification of sub-task failure and failure mode from delegatee to delegator. (The agents' "regular" protocols may support exchange of sub-task failure information as well; but the respective agents' FHCs ensure that this information is always exchanged regardless of other prescribed interactions).

We require that these protocols encompass situations where a participating agent has crashed or gone offline. In addition, we would like them to support 'reasonable' autonomy of agents while enforcing predictability of behavior. In defining the protocols, we assume that reliable message delivery and acknowledgment is supported by the agents' underlying communication layers.

We define our protocols using Petri nets [8]. A Petri net consists of *places* (depicted as ovals) and *transitions* (depicted as rectangles), which are linked by arrows. A transition in a Petri net is enabled if each incoming place has at least one token. An enabled transition can be fired by removing a token from each incoming place and placing a token on each outgoing place. We follow the

notation used in [9], which may be mapped to FIPA AUML [10] diagrams: we have two types of places, one corresponding to protocol states and the other to messages and events[4]. An exchange of messages adheres to a given protocol if each successive message, when activated by a token, enables a transition to a new (protocol state) place. In our semantics, for all protocol states with subsequent transition(s), one and only one of the transitions must occur.

Fig. 6 shows Protocol 1 above: *determination of compensation responsibility.* This protocol is a pairwise conversation between a delegator and delegatee. It can only be initiated from a system state in which the "forward" task has been successfully canceled, and cancellation information propagated, as supported by Protocol 2. From Protocol 1's start states (`s0` and `s1`), if the delegator (parent) is offline, the delegatee (child) must take responsibility. Otherwise, the delegatee is given autonomy to decide independently of the delegator whether it will take responsibility for the compensation, *unless* it has itself failed. If the delegatee rejects the compensation responsibility, is offline, or fails in the compensation, the delegator must take responsibility[5]. If a delegated task has failed, then is subsequently canceled and compensation required, the delegating agent (if online) must decide whether or not the delegatee will be allowed to make decisions about the compensation. As long as at most one of the agents are offline, the protocol ensures that *one and only one agent in the compensation scope* will take responsibility for the compensation.

For example, suppose in Fig. 6 that the delegatee (C) is offline when the protocol is initiated at state `s1`– a completed task is canceled. C's "offline" event causes a transition to `s3`. Because one and only one transition from a protocol state must occur, the delegator (P) must take responsibility for the compensation, and indicates this to C (message (1)). If/when C comes back online, it will retrieve this message and will not address the compensation. Alternatively, suppose that C is online and declines the compensation responsibility (2), but then receives notification that P is offline. From this state (`s4`), C must now take on the compensation (3), if possible.

A compensation is treated by the agent application as a new task like any other, though the FHC of the responsible agent logs the association with its original task. The protocol described above does not determine task delegation. The FHC-based interactions determine only which agent is initially *responsible* for the new compensation task; then, as is possible with any task, the responsible agent may decide to delegate it.

[4] The protocol below includes agent communication timeouts; other compensation-related events are `cancel` and `failure`.

[5] The protocol in the figure is simplified for readability: the compensation information is not parameterized, and it does not include the case where the delegatee, based on its local knowledge, does not believe that the cancellation needs to be compensated. Note also that in this protocol, the delegator is not required to report on compensation results to the delegatee. An alternate protocol could require this as well.

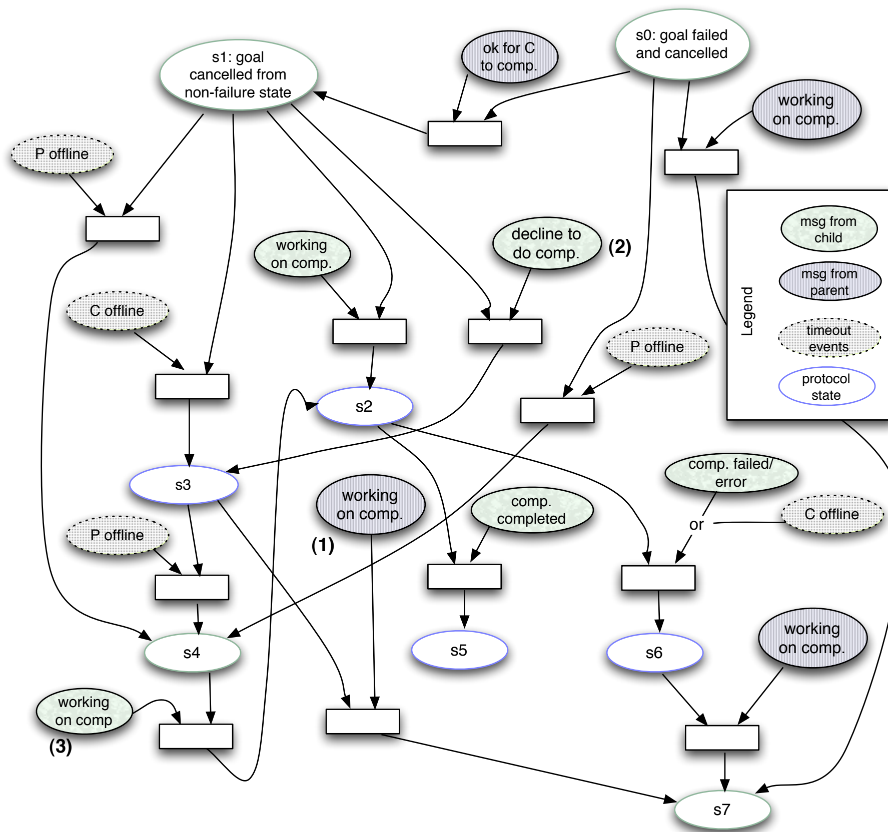

Fig. 6: The protocol to determine compensation responsibility. 'P' is the delegating ("parent") agent, and 'C' is the delegatee ("child") agent, with respect to the given canceled goal. The 'protocol state' labels are used only to distinguish the states. The numbered messages are referenced in the example.

3.3 Prototype Implementation and Initial Experiments

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[11]. 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 interface uses Jess "fact" syntax to communicate goal-based events.

The prototype implements the goal-based semantic compensation model described here. The protocols it uses, while not yet specified declaratively, are implemented as described here with respect to the messages that must be exchanged by the delegating and delegatee agents. However, the current prototype is more limited than the model in that it does not yet support pings/timeout

events, and its means of "connecting" delegated task nodes across agents is hardwired to a specific delegation protocol.

We have performed initial experiments in several relatively simple problem domains. Our prototype has helped us to conclude that this approach is feasible, and suggests 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 strategy rules for more complex problem domains in which agent interaction will feature prominently.

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 [12], SAGAs, [13], and ConTracts [14]. Earlier related project work has explored models for the implementation of transactional plans in BDI agents [15–18], and a proof-of-concept system using a BDI agent architecture with a closed nested transaction model was constructed [19].

In [20], Greenfield et al. discuss a number of issues that can arise when employing "traditional" compensation models, similar to those raised here. In Nagi et al. [21, 22] 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 et al. [23, 24] describe an approach to MAS exception-handling utilizing sentinels associated with each agent. For a given domain, "sentinels" are developed that intercept the communications to/from each agent and handle certain coordination exceptions for the agent. The exception-detecting and -handling knowledge for that shared model resides in their sentinels. Entwisle et al.[25] take a related approach in which decoupled exception-handling agents utilize a knowledge base to monitor, diagnose, and handle problems in a system. In our approach, while we decouple high-level handling knowledge, the agents retain the logic for failure detection and task implementation; some agents may be designed to handle certain compensations.

Chen and Dayal [26] 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).

[27] 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 [4] 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. 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 [28], 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 [29, 30].

We are not aware of existing work which explicitly uses protocols for flexibly managing compensation responsibility. Recently, there have been efforts to specify languages for web service composition and coordination. For example, BPEL4WS and WS-Coordination/Transaction [3, 31] provide a way to specify business process composition, scoped failure-handling logic, and coordination contexts and protocols. Each BPEL activity may have a compensation handler, which may be invoked by the activity's failure handler. Compensation and fault handlers may be arbitrary processes. Our approach to semantic compensation has some similarities to the BPEL model, with strategy rules (Section 2) and protocols serving to coordinate "inner scope" compensation. However, our model pushes the implementation of failure handling to the agent application logic.

5 Conclusion

We have described an approach to increasing 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 becomes 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 determining when to employ it. By abstracting

the agent's failure-handling knowledge to a goal level, it can be decoupled from agent domain implementations and employed via the use of a failure-handling component with which the agent application interfaces, supporting predictable behavior and decreasing the requirements on the agent developer. Our methodology separates the definition of failure-handling knowledge from the agents' implementation of that knowledge, allowing the compensations to leverage runtime context in a way that makes them both more robust and more widely applicable.

For semantic compensation to be effective and predictable, it is necessary to control not only how individual agents react when a task fails or is canceled, but how the agents in the system *interact* when problems develop with a task assigned by one agent to another. Flexible interaction protocols are necessary to achieve a useful degree of control. We have described a method for factoring compensation-related protocols from the agent application to its failure-handling component, allowing them to be developed independently, and have described one key such protocol used by our system.

Project 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 [15]. Our experiments will also help us to evaluate the ways in which a failure-handling strategy is tied to the problem representation.

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. Unruh, A., Bailey, J., Ramamohanarao, K.: A framework for goal-based semantic compensation in agent systems. In: 1st International Workshop on Safety and Security in Multi-Agent Systems, AAMAS '04. (2004)
3. Business Process Execution Language for Web Services (BPEL4WS).: `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel` (2003)
4. Mikalsen, T., Tai, S., Rouvellou, I.: Transactional attitudes: Reliable composition of autonomous web services. In: Workshop on Dependable Middleware-based Systems. (2002)
5. FIPA: http://fipa.org, http://www.fipa.org/specs/fipa00029/
6. 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)
7. Walsh, W., Wellman, M.: Decentralized supply chain formation: A market protocol and competitive equilibrium analysis. JAIR **Vol. 19** (2003) 513–567
8. Reisig, W.: Petri nets: An introduction. EATCS Monographs on Theoretical Computer Science (1985)
9. Poutakidis, D., Padgham, L., Winikoff, M.: Debugging multi-agent systems using design artifacts: The case of interaction protocols. In: First International Joint Conference on Autonomous Agents and Multi-Agent Systems. (2002)

10. Odell, J., Parunak, H., Bauer, B.: Extending uml for agents. In: Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence. (2000)
11. Friedman-Hill, E.: Jess in Action. Manning Publications Company (2003)
12. 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
13. Garcia-Molina, H., Salem, K.: SAGAs. In: ACM SIGMOD Conference on Management of Data. (1987)
14. Reuter, A., Schwenkreis, F.: Contracts - a low-level mechanism for building general-purpose workflow management-systems. Data Engineering Bulletin **18** (1995) 4–10
15. 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)
16. 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)
17. 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
18. Smith, V.: Transaction oriented computational models for multi-agent systems. Internal Report, University of Melbourne (2003)
19. Busetta, P., Ramamohanarao, K.: An architecture for mobile BDI agents. In: 1998 ACM Symposium on Applied Computing. (1998)
20. Greenfield, P., Fekete, A., Kuo, D.: Compensation is not enough. In: 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC'03), Brisbane, Australia (2003)
21. 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)
22. Nagi, K., Lockemann, P.: Implementation model for agents with layered architecture in a transactional database environment. In: AOIS '99. (1999)
23. 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)
24. Klein, M., Rodriguez-Aguilar, J.A., Dellarocas, C.: Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. Autonomous Agents and Multi-Agent Systems **7** (2003) 179–189
25. Entwisle, S., Loke, S., Krishnaswamy, S.: Aoex: An agent-based exception handling framework for building reliable, distributed, open software systems. In: Submitted to IAT2004. (2004)
26. Chen, Q., Dayal, U.: Multi-agent cooperative transactions for e-commerce. In: Conference on Cooperative Information Systems. (2000) 311–322
27. T. Strandens, R.K.: Transaction compensation in web services. In: Norsk Informatikkonferanse. (2002)
28. Alonso, G., Agrawal, D., El Abbadi, A., Kamath, M., Gunthor, R., Mohan, C.: Advanced transaction models in workflow contexts. In: ICDE. (1996)
29. Casati, F.: A discussion on approaches to handling exceptions in workflows. SIGGROUP Bulletin **Vol 20. No. 3** (1999)
30. Rusinkiewicz, M., Sheth, A.P.: Specification and execution of transactional workflows. Modern Database Systems: The Object Model, Interoperability, and Beyond (1995) 592–620
31. 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)