Semantic-Compensation-Based Recovery in Multi-Agent Systems

Amy Unruh Henry Harjadi James Bailey
Kotagiri Ramamohanarao
Dept. of Computer Science and Software Engineering
The University of Melbourne, VIC 3010, Australia
{unruh,hharjadi,jbailey,rao}@cs.mu.oz.au

Abstract

In agent systems, an agent's recovery from execution problems is often complicated by constraints that are not present in a more traditional distributed database systems environment. An analysis of agent-related crash recovery issues is presented, and requirements for achieving 'acceptable' agent crash recovery are discussed.

Motivated by this analysis, a novel approach to managing agent recovery is presented. It utilises an event- and task-driven model for employing semantic compensation*, task retries, and checkpointing. The compensation/retry model requires a situated model of action and failure, and provides the agent with an emergent unified treatment of both crash recovery and run-time failure-handling. This approach helps the agent to recover acceptably from crashes and execution problems; improve system predictability; manage inter-task dependencies; and address the way in which exogenous events or crashes can trigger the need for a re-decomposition of a task. An agent architecture is then presented, which uses* pair processing *to leverage these recovery techniques and increase the agent's availability on crash restart.*

1. Introduction

Multi-agent systems are often complex, with decentralised models of control. Actions of the agents are often influenced by the environment in which the system is situated. Unaddressed problems can propagate from one agent to another, in ways that may be difficult to identify. In addition, unexpected changes in the environment can cause problems with agents that were not designed to handle such changes. If problems occur, it is often difficult to characterise the global state of an agent system and to determine if its behaviour is correct. For this reason, the ability to handle failures and recover from them can be important in sustaining a stable agent system. Traditional recovery methods employed in (distributed) database systems are not adequate, although many of the principles are useful.

In this paper, we first discuss issues in agent crash recovery that make application of existing recovery techniques from other fields problematic, and present a definition of agent recovery to an 'acceptable' rather than consistent state.

We then present a novel approach for supporting agent recovery. The approach utilises an event- and task-driven model of when and how to employ techniques for *semantic compensation* and task retries, and to checkpoint agent state. The compensation/retry model requires the agent to implement a situated model of action and failure, resulting in a unified treatment of both crash recovery and run-time failure-handling, and allowing the agent to address a number of facets of the 'acceptable state' objectives described.

Based on this framework, we will describe a high-level agent "recovery procedure" that addresses the objectives of an acceptable recovery state. This procedure includes the use of a technique called *pair processing*, which leverages the agent's recovery capabilities to improve agent availability on startup– shortening the recovery period and reducing the length of time in which transient environmental information may be lost. We then discuss related and future work, and conclude.

2. Issues in Agent Crash Recovery

In the distributed systems and transaction management contexts, the goal of crash recovery is to return a system of (possibly distributed) processes to a *consistent* state after a crash, where a consistent global state is one which may occur during a failure-free, correct running of the computation (that is, such a state would have been reachable during normal operation), and consistency must be achieved both with respect to an individual process and for the system as a whole.

Distributed-system recovery methods typically make a number of assumptions about the context in which their techniques will be employed. They typically assume the existence of a closed system with only controlled processes modifying the data; that some form of rollback is possible (a process can be restored to a previously saved state); and that post-checkpointed requests can be replayed starting from a restored state. [6].

Based on these assumptions, a range of checkpointing and logging techniques have been developed to recover a system of distributed processes to a consistent state after a crash. If a process checkpoints (saves state) before "exporting" any information to other processes, sometimes referred to as *pessimistic independent* checkpointing [16], then restoration of one process won't require cascading rollbacks for others (if the assumptions above are met), but this can be expensive. A focus of many of these techniques is thus how to reduce the checkpointing overload, e.g. by employing *uncoordinated* checkpointing, but to then return the system to a globally consistent state after a crash, e.g. by finding a suitable *recovery line*. If replay is not feasible, then these approaches will not work.

In an agent environment, the underlying assumptions made by these techniques are often violated, so that it is not always possible to achieve consistency of an individual agent on restart. This is the case for several reasons. First, it is not always possible for an agent to revert to a previous checkpointed state. "State", in terms of the agent's behaviour, may include aspects of the environment not under the agent's control. In addition, most situated actions "always commit"– so for agents which interact with their environment, it is usually not possible to perform rollbacks in the traditional database sense. Nor are exact compensations (forward recovery) always possible– an agent can't always undo the effects of an action that modified its environment.

Second, it is possible for transiently observable exogenous events to occur while an agent is down, which there may be no means to 'replay' when the agent comes back up. Thus, some information may be lost during a crash.

In addition, replay of actions can be problematic: the agent's environment may include limited resources, which can't be accessed or used arbitrary numbers of times. For example, an information source may have a limit on the number of queries it supports per day; or if an object is broken it may not be replaceable.

So, individual agent recovery consistency, as defined in the traditional distributed systems sense, is typically not possible. As a consequence, post-recovery inter-agent system consistency becomes ill-defined, even if a conservative checkpointing scheme is used. Thus, traditional distributed-system recovery methods are usually not directly applicable in an agent context.

In this paper, we describe an approach for agent recovery and run-time failure-handling that addresses these issues. In the remainder of this section, we describe a way of evaluating agent recovery, in terms of *acceptability* criteria, that we claim is more useful in this context than strict consistency.

2.1. Recovering to an 'acceptable' state

"Hospital" domain example tasks:

- *Take inventory of medications*: what is currently available? The process involves moving medications around a stockroom to facilitate counting.
- *Get information to a doctor*: determine the doctor's schedule, find out where they're expected to be; then work out a route to intercept them, as they arrive/depart from a known location (e.g. a surgery room); then go to that location.
- *Give medications to a patient*: devise a medication plan to address a set of symptoms. Not all drugs may be taken with each other.
- *Feed patient*: get food tray from cafeteria, bring to patient's room.

Figure 1. Motivating examples, describing agent tasks in a "hospital" domain. The paper will discuss recovery and repair issues in the context of these tasks.

It is clear from the discussion above that for many agent systems, it will not be possible to achieve *consistent* recovery in a distributed-systems sense of the term. It is thus more useful to consider how an agent might reach a sufficiently *acceptable* state after recovery, and what such an acceptable state might be. Here, we propose an informal definition of 'acceptability' in terms of a set of recovery objectives, illustrated in the example "hospital" agent domain of Fig. 1:

- *The agent recovers to a state that is sufficiently predictable* to avoid propagation of crash-induced errors. For example, in the "hospital" domain of Fig. 1, if an agent crashes and drops a tray while bringing it to a patient, it should clean up the floor (if necessary) upon recovery.
- *After restart, the agent knows 'enough' about the current state of the world and what the other agents expect of it, and its actions reflect this knowledge*:

the agent is not using outdated information about its environment; it doesn't drop important tasks it should be working on; it is able to ascertain if it has been given any new tasks while it was in a crashed state, and it is able to detect whether changes to its environment require it to redo tasks. For example, if an agent is trying to deliver information to a doctor, but the doctor's location changes while the agent is down, it should be able to detect this change upon recovery and re-generate its route.

- *The agent has checkpointed sufficient information so that it doesn't have to redo 'too much' work if it crashes.*

The concept of 'acceptable recovery' drives the approach presented in this paper.

3. Compensation/Retry Failure-Handling

In this section, we present an approach for dealing with certain types of agent problems via an event-driven model for applying task *compensations* and initiating task *re-decompositions* (re-achievement). We first describe the model from a behavioural standpoint, then the architectural mechanisms required to make it work. Then, in the following section, we describe how this model allows us to treat many aspects of recovery and run-time failure-handling in a unified manner, and allows us to address a number of the 'acceptable state' objectives described in Section 2.1. The approach is described in the context of a goal-driven agent that performs context-based (hierarchical) task decomposition, maintains an agenda of currently-active goals and executable actions (i.e., "intentions") and whose subtasks may potentially be delegated to other agents.

A key idea of the model is that it is useful to employ an approach we term *semantic task compensation*, in conjunction with task *retry (re-decomposition)*, to address problems that occur both from crashes, and from task failure. The motivation behind this idea is that the ability of an agent system to recover from problems can be improved by improving the agents' ability to "clean up after" or "undo" effects of their problematic actions. Note, however, that compensation activities must address agent task semantics. An exact 'undo' is not always desirable, even if possible, and the appropriate compensations are context-dependent. 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, and the implicit assumptions made by the agents, in terms of representational validity and state, are more likely to hold;
- it helps maintain an agent system in a more predictable state: agent interactions are more robust; and unneeded resources are not tied up; and
- the approach can often be applied more generally than methods which attempt to "patch" a specific failed activity, and can be usefully viewed as a *default* failure-handling strategy.

However, traditional transaction management methods are usually not appropriate in a situated-agent context. We cannot always characterize 'transactions' and their compensations ahead of time, nor create compositions of compensations by applying subcompensations in a reverse order [8]; in many domains such compensations will not be correct or useful. In addition, in an agent context, failure-handling behaviour should be related to the agent's goals, and take into account which goals are current or have been cancelled.

Thus, to operationalize the use of semantic compensation in an agent context, it is necessary both to define the compensations in a way that is effective, and to usefully specify when to initiate both compensations and goal re-achievements, including scenarios where problems occur with tasks delegated between agents.

3.1. Goal-Based Compensation

The agent's task compensations are defined *declaratively*, in terms of goals– statements of what needs to be achieved to effect the compensation– not in terms of plans or action sequences. That is, in defining compensation knowledge for a given domain (sub)task, the agent developer specifies what must be true about the state of the world for compensation of that task to be successful. The declarative definitions thus support context-dependent compensations– the agent application will determine at runtime how to implement, or achieve, these goals. The same semantic compensation may be performed differently under different circumstances.

A goal-based formulation of failure-handling knowledge is useful in several ways:

- 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 prespecify all relevant failure-handling plans.

More detail is provided in [20].

3.2. Failure-Handling Model

Based on these core concepts– the use of goal-based compensation and re-achievement for failure-handling– we describe an approach to handling a class of agent problems. A framework supporting the approach has been implemented [20, 19]. However, here we generalize from that implementation, and focus on the failure-handling model.

In this model, execution may be interrupted by a 'problem event' on a task; the event triggers activity, based on a series of task compensations and re-decompositions, to handle the task problem. There are two types of problem events, *failure* and *revision/cancellation*, each described below. A problem event may be generated by execution-monitoring rules that are part of the agent's domain knowledge, and are triggered by some aspect of the agent's state. Additionally, problem events for a given task may be generated by the agent architecture, e.g. in response to timeouts or messages from other agents.

In general, an agent may either handle a problem event at its source, or recursively delegate the handling of the problem further up the task tree, triggering broader repair activities. In both cases, current execution is first halted[1].

We first describe the way in which the agent handles each type of problem event, then discuss some of the implications of the model.

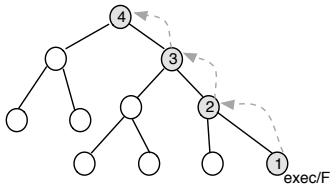

Figure 2. An execution failure at `node 1`.

Failure event: The effect of failure-event handling is to "clean up" the effects of the failure by compensation, then re-attempt the work that was compensated for. The handling of a failure may be delegated up the task tree; if this occurs, then the compensation is broader and more general, and the amount of work to redo the compensated task will be more extensive.

1 In addition, the model supports definition of what we term *stabilization* goals, which– when defined– are employed bottom-up from the point of execution to the failed goal when an execution path is halted and before any explicit failure-handling is initiated. However, this aspect of the model is beyond the scope of this paper.

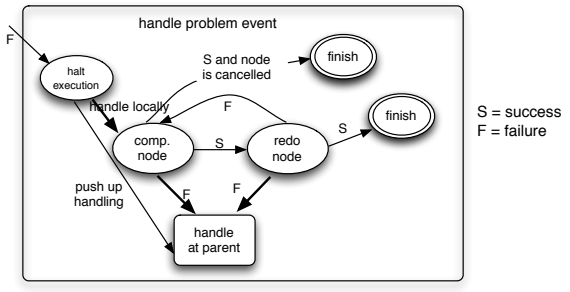

Figure 3. The FSM describing handling of an execution failure event.

Failures can only be triggered for tasks along the current-goal execution path, as shown in Figure 2. Such an event is generated by either an explicit *failure rule*, which detects a problem with a task on the execution path that can not be resolved; by leaf-level execution failure; or by a crash during execution of the task. In the figure, the execution of the leaf task `node 1` can report a failure, but failure events can also be explicitly detected for `nodes 2-4` as well. For example, in the "hospital" domain of Fig. 1, an agent delivering a food tray might crash or trip; these are leaf-level failures. However, if while en route to a doctor, the agent learns that the doctor has left the building, this can trigger failure detection on the higher-level "deliver info" task.

A failure event is handled as follows: the agent will compensate, then re-decompose (retry) the task that received the failure event, or recursively "push up" handling to its parent task. This is illustrated by the nested finite-state machine (FSM) of Figure 3. On failure, the task can be locally compensated, then re-tried. Alternatively, the failure handling can transition recursively to the parent task (if one exists), and a compensation of the parent task attempted. Note that if the task is not a leaf node, then retrying the task in the context of new information may result in a *different decomposition* of the task– an alternate way of performing it given new information [21]. The 'F' and 'S' labels on the arcs indicate failure/success of the (compensation) task. A sucessful compensation will terminate without retry if the task is no longer active (has been cancelled), as is further discussed below. A failure of the retry can cause the compensate/retry cycle to be repeated. At any time, failure of either the compensation or the retry can allow handling to be pushed to the parent task. Domain search control can be used to determine which arc is chosen if more than one exists for a given transition event, as further discussed below.

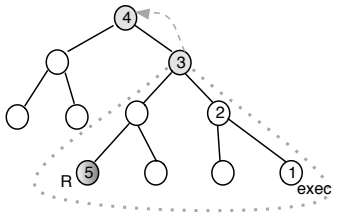

Figure 4. A revision event (R) triggers compensation and task re-decomposition of the ancestor task common to the revision and execution points.

Revision/cancellation event: A *revision* event is applicable to any node (whether completed or not) in a currently-active task tree, where the root task node has not yet been achieved. Such an event is generated by an explicit *revision rule*, part of the agent's domain knowledge, which carries the semantics that some aspect of the agent's state indicates that the task should not have been performed, and that the parent task in which the problematic task was used, needs to be reworked. Information from other agents, or revoking or cancelling a previously-assigned task, can trigger such a rule.

The important difference with the failure-handling process above, is that here, *revision events can be detected for successfully-completed subtasks*, not just currently-executing tasks. For example, consider the "give medications" task of Section 2. If the patient is allergic to one of the medications, further use of that drug must be cancelled, a compensation must be performed if necessary (e.g. by giving epinephrine), and then the "give medications" task must be re-addressed: a replacement must be found, and a new plan must be generated, one which doesn't include any drugs that are contraindicated with the replacement.

As the example suggests, by undoing the effects of a completed task, any dependent tasks will be affected; this is addressed by the event-handling process, which is illustrated in Figure 4 and is as follows. To handle a revision event, the agent must first compensate the task that received the revision event– `node 5` in the figure. Then, it determines the task that is the *common ancestor* of both the revision event task, and the currently-executing task[2]. This is the level at which dependencies caused by compensating must be addressed. In the figure, `node 3` is the common ancestor task of both node 5 (which received the revision event), and `node 1` (the currently-executing task). At that common node, the agent follows the procedure of Figure 3– it compensates, then re-decomposes the task of `node 3`, or may recursively push handling to *its* parent task. The dotted line of Fig. 4 suggests the task decomposition that will be revisited. If a root node of a task hierarchy receives a revision/cancellation event, the agent compensates it only; there are no parent dependencies to force a re-decomposition.

3.2.1. Discussion. The two different types of problem handling above illustrate a distinction between the approach described here and most compensation-based failure-handling in the workflow and transaction management literature. In particular, here compensation activity may be triggered by events on tasks that have completed successfully, as well as by execution problems.

In general, a problem will be most effectively addressed at the lowest level possible; this should be the default. However, in some cases a local compensation may be problematic (or attempts at a local fix may fail repeatedly), and the agent should effect a broader, higher-level compensation instead. This is analogous to 'throwing' exceptions upward.

In the procedures above, we don't specify how many retries should occur or when handling of a problem should be pushed up– in general, this requires domain-specific knowledge, which can be encoded as search control on the selection of an FSM transition when multiple transitions are possible, allowing incremental refinement of an agent's default failure-handling behaviour. That is, we distinguish the specification of the domain-independent failure-handling process (the FSM) from any domain-specific choices on the allowable transitions at a given stage in the failure-handling.

The examples above did not include scenarios where failure occurs during a compensation. However, while it is beyond the scope of this paper to discuss in detail, such scenarios are supported consistently in the model above as well. Compensation tasks may also have associated compensation definitions. The nested FSM execution model allows errors during compensation to be addressed locally (as discussed above) or captured by the parent context of the original error, again depending upon the domain knowledge conditioning the FSM transitions.

The failure-handling model above can be viewed as a type of exception handling. However, in contrast with other approaches to agent exception handling, e.g. [18, 2], in our approach there is no explicitly separate "handler" method– the agent's domain knowledge is leveraged to implement the compensation and retry goals that are the building blocks of the failure-handling process. In addition, our failure-

2 This tree shows the case for ordered subtasks; for concurrent subtasks, similar reasoning is applied.

handling model operates at a different level of granularity, in that failures *during* a compensation or retry are considered in the context of their enclosing failure-handling effort.

3.3. Agent Execution and Failure Model

For situated agents, action execution can have unexpected or non-deterministic results, and exogenous events can change things independent of any action of the agent. This means that:

- Finishing all subgoals in a task decomposition doesn't necessarily imply success of the parent goal. This can be the case even if execution of each subgoal occurred without any explicit error results.
- Success/failure of a goal can be triggered by exogenous events as well as subgoal results.
- Similarly, exogenous events can impact or undo the effects of previously-achieved (sub)goals.

So, for a situated agent, execution monitoring must be supported if the agent is to do robust failure handling. In the context of the compensation-based failure-handling model described above, this translates to several requirements. The agent must be able to sense and react to exogenous events; and sense, rather than 'model' the results of its actions (since its actions may have unpredictable results). In addition, a goal-based agent should be able to monitor and *explicitly detect goal status changes*, both success (achievement), and failure, based on state information, not execution history. Thus:

- Success of a parent task is determined by explicit detection of achievement, not inferred by the completion of its subtasks. Further, an agent must not infer failure based solely on a current inability to achieve a goal, though achievement *timeouts* may cause domain-specific derivation of explicit failure.
- Unachieved current tasks remain active. Thus, if all the subtasks of a task are achieved, but the parent task itself is not, it stays on the agent's agenda. For example, consider an agent's task of giving some information to a doctor. The agent may successfully go to the place the doctor is expected to be, but if the doctor does not turn up there after all, the "locate doctor" task remains active (unless timeout occurs).
- Already-achieved tasks on the agenda are detected as such and are removed (not re-executed), thus allowing only necessary work to be performed in expanding a compensation task, or in re-decomposing a task. For example, if a compensation requires an agent to locate a doctor, and the doctor is currently visible to the agent (i.e., already located), this triggers the detection of task achievement, causing the removal of the sub-task from the agenda.

If these requirements are not met, then problem events can't be detected, and compensations and re-decompositions– which must be expanded in the context of work that needs to be done– can't be properly employed.

In addition to the above, the agent has a further architectural requirement to support these methodologies. It must maintain an *abstract execution log/history*, via persistent transactional storage, maintained across crashes. This log records not only 'leaf task' execution, but task status information, and parent-child relationships, with each root task current to the agent represented as a tree. That is, the hierarchical task relationships, as well as the goal failure and success events in the execution history, are preserved in the log. The logged information enables the problem-handling reasoning described above, as well as refinements of the default behaviour via search control rules conditioned on the history information.

4. A Unified Approach to Recovery and Run-Time Failure Handling

Section 2 discussed why traditional recovery methods, such as those applied in distributed systems, aren't directly applicable in most agent contexts. Rollback is typically not possible, and on recovery from a crash, agents must consider and accommodate changes in the environment during their recovery process, and must compensate for effects of the crash.

The compensation/retry model of Section 3, in conjunction with its underlying logging framework, provides a foundation for addressing these issues, and for treating aspects of both recovery and run-time failure-handling in a unified manner. To do this, two additional extensions to the model above are required. First, *agent crash points are treated as failure events.* An agent crash causes a failure event to be posted for the task that was currently executing at the time of crash, on recovery. This information is derivable from the agent's persistent execution log. Second, we extend the persistent transactional logging framework described in the previous section to support checkpointing of agent state information, as is further discussed below.

By supporting these capabilities, and by employing the compensation/retry model in the context of situated goal and execution monitoring, the agent exhibits a recovery behavior that addresses a number of the 'ac-

ceptable state' objectives described in Section 2.1. The following behaviour is supported at crash recovery:

First, on recovery from a crash as well as a failure, compensation helps 'reset' the agent if it was left in an inconsistent or ill-defined state on crash, and works toward releasing unneeded resources, thus allowing the agent to behave more predictably, and to allow the other agents in the system to operate more successfully.

A useful analogy is that of a *transaction*, which can be viewed as a series of operations which takes a system from one consistent state to another [8]. Our approach to semantic compensation can be viewed in the same way– post-crash compensation approximates a rollback, leaving the agent and the system as a whole more consistent and well-defined.

For example, if an agent crashes while it is performing the "count medications" task in the stockroom, its count may no longer be valid after it restarts (e.g., while it has been down, other agents may have removed items from the room). The agent will need to start over, but for a correct (re)count, it should first "clean up" what it had been doing, by restoring all the items it was counting to their normal places in the room.

Second, the agent's execution monitoring model allows the effects of a crash to be sensed in the same way as unexpected execution results, these effects may similarly trigger failure/revision events. That is, changes in the world trigger revision events in a unified manner, without needing to distinguish whether the agent was down in the interval during which the changes occurred. For example, if an agent is trying to intercept a doctor, and the doctor's location changes, this will trigger a revision of the "plan route" task regardless of whether the agent was offline during the time of the change.

Third, if non-achievement of a subgoal causes its active parent to be no longer directly achievable, but the parent has not failed, then work will continue on the parent goal– it will remain on the agenda, with further task (re-)decompositions applied to it. This helps ensure that post-recovery, the agent continues to work on relevant goals.

Finally, if the agent crashes without having an up-to-date state checkpoint, explicit task status detection helps avoid unnecessary redo of work. For example, an agent's checkpointed state may record that it still needs to administer medication to a patient, but state information from the external world (e.g. the patient's chart) will allow it to determine after restart if the task has already been done.

The task structure recorded in the agent's persistent execution log supports both compensation/retry reasoning, as described above, and recovery bookkeeping. Section 3 described its role in managing task compensations and retries. However, this persistent history serves two additional purposes.

1. It allows a crashed agent to rebuild its runtime agenda on restart.

2. The task-based logging mechanism can be leveraged to allow the agent to checkpoint at task achievement points. Task-based checkpointing imposes a more coherent semantics on the saved information than would an arbitrary checkpoint interval, and is more easily synchronized with respect to communication with other agents, as such communication is typically task-oriented. Consequently, task-based compensation/retry is more coherently supported.

Further, the hierarchical task structure *allows checkpointing at different granularities* (frequencies) based on the level of the task tree for which task completion triggers a checkpoint. Note that if the state of the external world is likely to change quickly while the agent is down, triggering task re-decomposition after a crash, then frequent checkpointing of task results may not be cost-effective; some of the agent's low-level task results are likely to be discarded after the crash. Instead, the agent may recover more effectively by checkpointing only at the completion of high-level tasks.

4.1. Supporting Recovery: Process Pairs

The view of recovery handling presented above assumes that on restart from a crash, initialisation from some saved state is performed, from which the agent's accommodation to the changes in its tasks and environment, as described above, can proceed. An important factor in this recovery process is how fast the recovery can take place. For example, if an agent crashes while trying to intercept a doctor, the doctor may move out of sensing range if the agent is down too long. The faster the agent can restart, the less likely it is to miss important transient external events, the less the world is likely to change, and the less work the agent is likely to require upon restart.

Pair processing[3] is a well-known technique for improving process reliability [8]. A process pair is a collection of two processes which provide a service. At any one time, one of the processes is *primary*, and delivers the service. If the primary fails, its *shadow* takes over. The two processes 'ping' each other to determine that each is still alive. Because the shadow runs in-memory, this can provide quicker recovery than a full restart.

We have applied pair processing to a design and implementation of an agent architecture for recovery.

3 Sometimes referred to as a 'primary/backup' model.

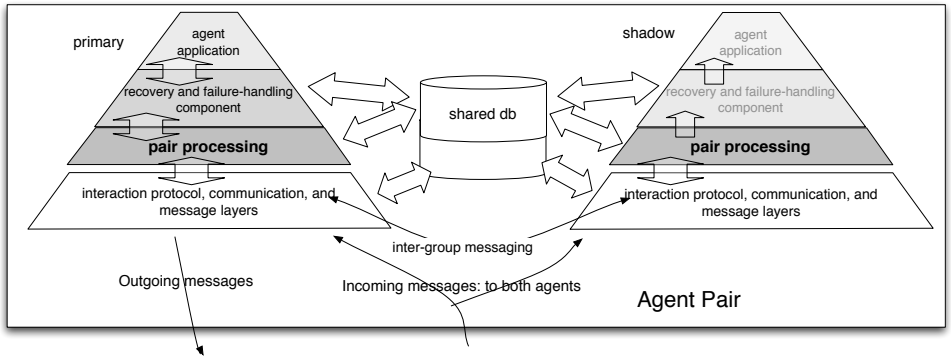

Figure 5. An agent architecture for supporting pair processing and recovery.

Key aspects of this architecture are that that it leverages the failure-handling and recovery functionality described above while allowing this core functionality to remain decoupled, and can support efficient incremental updating of the shadow's state with the primary's.

As shown in Figure 5, our design uses a shared persistent transactional data store between the pair members. A layered architecture factors pair processing from recovery reasoning, from the agent's domain reasoning. The primary persists its recovery bookkeeping and state information to the data store, from which the shadow recovers it as necessary. Each pair member has the same agent architecture, but an agent operates differently in primary vs. shadow mode.

Design for recovery requires correct implementation of persistence requirements; thus, an important aspect of our design is that each layer works in concert to implement the persistence necessary to support the model. On receiving information from an adjacent layer, a layer must persist any necessary information before acknowledging to the sending layer that it was received. On passing information to an adjacent layer, it must not remove any information from persistent memory until acknowledged by the receiving layer.

The pair processing layer, in conjunction with its underlying messaging layer, provides transparent and persistent addressing for the pair. The other agents in the system address the pair by its logical name, not its component agents[4]. Because both pair members receive the messages, message persistence across crashes is supported[5], and via their pings, each agent in the pair acts as a *sentinel* to the other. By factoring pair processing from recovery functionality, and by providing transparent pair addressing, agents in the system can run with or without using pair processing.

The shadow agent receives pair-addressed messages from the other agents, but does not perform domain tasks while it is in shadow mode. If the primary crashes or becomes unresponsive, the shadow detects this via its monitoring, kills the primary, and switches itself to primary (launching a new shadow). To do this, it must synchronise its message queue with the old primary's persisted message information, and its recovery layer must then instantiate itself and its agent logic layer from the primary's checkpointed information.

The use of a shared database allows this model to support an important benefit of pair processing– the shadow, while it is running in-memory, can leverage the database to efficiently *incrementally* update itself with the primary's checkpointed changes. Because the shadow is not yet 'active', the updating does not compete with other tasks. Thus, if the primary agent crashes, the shadow has already completed much of the instantiation process and may be running quickly.

5. Related Work

In addition to the distributed systems methodologies described in Section 2, several agent-oriented research directions are relevant to our approach as well.

The SPARK [11] agent framework is designed to support situated agents, whose action results must be sensed, and for which failure must be explicitly detected. ConGolog's treatment of exogenous events and execution monitoring has similar characteristics [4]. While these languages do not directly address crash recovery, their action model and task expressions are complementary to the recovery approach described here.

In the Cougaar agent system[17], it is not required that the agents use pessimistic checkpointing; thus, a

4 Currently, we are using JGroups to support agent communication, but this approach would map to a FIPA-based architecture as well.

5 Here, a single-fault model is assumed– but the two agents need not be on the same machine.

procedure is defined to allow them to perform an approximate synchronization after a crash, by exchanging task information. Barga et al. [1] also address interprocess recovery issues, by proposing "interaction contracts" which allow the processes to rely on each other for implementing certain persistence needs, thus allowing recovery guarantees. However, in both cases, this work differs from ours in that their models do not incorporate consideration of constraints from or changes to a situated agents' environment that would invalidate replay of work or require task changes in the context of recovery.

Section 3.2.1 compared our approach to that of building explicit within-gent exception-handling logic. Other approaches encode handler logic within separate monitoring/sentinel agents, e.g. [14, 9]. 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 exception-handling knowledge for that shared model resides in the sentinels. In our approach, while we decouple the failure-handling model from the agent's domain knowledge, the agent's domain logic is leveraged for failure detection and task implementation. Sentinel-based fault detection approaches, e.g.[14], also have relevance to pair-processing: an important aspect of agent recovery is detecting when an agent is behaving so incorrectly that it should be restarted.

Eiter et al. [5] describe a method for recovering from execution problems by backtracking to a diagnosed point of failure, based on execution monitoring, from which the agent continues towards its original plan. The backtracking is enabled by building a library of reverse plans corresponding to action sequences. Thus, their compensations are defined at a plan segment level rather than a goal level, and do not address scenarios where higher-level semantic compensation is required. However, the failure-handling model we describe in this paper can be viewed as falling into the same class of 'plan repair' approaches as does the system above. Effectively, for this class of repair and recovery approaches, the use of compensation/reversal is employed as a search control heuristic over the plan repair space.

In Nagi et al. [13], [12] an agent's problem-solving drives 'transaction structure' in a manner similar to that of our approach. 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.

Workflow systems encounter many of the same recovery issues as agent systems. Recent process modeling research attempts to formalize some of these approaches in a distributed environment. For example, BPEL&WS-Coordination/Transaction [2] 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 nested failure-handling model. However, as discussed in Section 3.2.1, our approach doesn't require explicit definition of separate handler methods, and operates at a different level of granularity.

Pears et al. [15] describe a framework that incorporates server-level exception-handling and the use of process pairs in a mobile agent context. However, in their domain they do not address issues in updating the shadow with the primary's state after the initial replication. Fedoruk et al. [7] propose an approach to agent replication, which has some similarities with the primary/shadow model of our approach. However, in their discussion of state replication and "switchover", they do not take into account the situated recovery issues addressed here.

6. Summary and Future Work

In this paper, we have first analysed issues in agent crash recovery, and suggested that criteria for recovery to an *acceptable* rather than consistent state has more utility in an agent context. We then described an approach to managing agent recovery that addresses some of these criteria, which allows a **unified treatment** of both crash recovery and run-time failure handling, centered around an event- and task-driven model for employing semantic compensation and re-decomposition of the agent's tasks. A notable feature of this model is the way in which compensations can be systematically applied to completed as well as currently-executing tasks. By treating crashes as execution failure points, the agent is able to support an integrated reaction to environmental and task changes that require repair.

The approach can be viewed as a default recovery and failure-handling behaviour, applicable when more specific patching/replanning information is not available, and to which refinements can be made incrementally. In helping the agent to recover acceptably from crashes and execution problems, the approach can prevent fault propagation between agents, improve system predictability; help manage inter-task dependencies; and address the way in which exogenous events or crashes can trigger the need for a re-decomposition of a task. The use of a *process pairs*-based agent architec-

ture can leverage such recovery techniques and increase the agent's responsiveness and availability on restart.

Our existing implementations have provided proof-of-concept demonstrations for key aspects of the approach described above– both the compensation/retry model and the pair processing framework– and we are in the process of further integrating, formalizing, and testing the model. A language such as 3APL [10, 3] offers a useful starting point for such a an effort: it provides a more formal semantics than our current implementation, and it supports goal-based reasoning and context-based task decomposition. However, its use would require extension of the existing language. 3APL does not model exogenous events, and does not explicitly model execution failure or success, allow for non-deterministic outcomes from an action execution, nor allow definition of explicit rules for detection of goal failure/success. As part of our current research, we are specifying and implementing a variant of 3APL that supports these changes.

References

[1] R. Barga, D. Lomet, and G. Weikum. Recovery guarantees for general multi-tier applications. In *18th International Conference on Data Engineering*, 2002.

[2] Francisco Curbera, Rania Khalaf, Nirmal Mukhi, Stefan Tai, and Sanjiva Weerawarana. The next step in web services. *COMMUNICATIONS OF THE ACM*, Vol. 46, No. 10, 2003.

[3] M. Dastani, B. van Riemsdijk, F. Dignum, and J.J. Meyer. A programming language for cognitive agents: Goal-directed 3APL. In *First Workshop on Programming Multiagent Systems*, AAMAS '03, 2003.

[4] G. de Giacomo, Y. Lesperance, and H. J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artif. Intell.*, 121(1-2):109–169, 2000.

[5] T. Eiter, E. Erdem, and W. Faber. Plan reversals for recovery in execution monitoring. In *Non-Monotonic Reasoning*, 2004.

[6] E. N. (Mootaz) Elnozahy, L. Alvisi, Y. Wang, and D. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3), 2002.

[7] A. Fedoruk and R. Deters. Improving fault-tolerance by replicating agents. In *AAMAS '02*, pages 737–744. ACM Press, 2002.

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[9] Mark Klein, Juan-Antonio Rodriguez-Aguilar, and Chrysanthos Dellarocas. Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. *Autonomous Agents and Multi-Agent Systems*, 7:179–189, 2003.

[10] F. Koch. 3APL-M: A platform for lightweight deliberative agents, 2004.

[11] D. Morley and K. Myers. The SPARK agent framework. In *AAMAS '04*, NY, NY, 2004.

[12] K. Nagi and P. Lockemann. Implementation model for agents with layered architecture in a transactional database environment. In *AOIS '99*, 1999.

[13] K. Nagi, J. Nimis, and P. Lockemann. 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.

[14] S. Parsons and M. Klein. Towards robust multi-agent systems: Handling communication exceptions in double auctions. In *AAMAS '04*, 2004.

[15] S. Pears, J. Xu, and C. Boldyreff. Mobile agent fault tolerance for information retrieval applications: An exception handling approach. In *The Sixth International Symposium on Autonomous Decentralized Systems*, 2003.

[16] D. Scales and M. Lam. Transparent fault tolerance for parallel applications on networks of workstations. In *USENIX Annual Technical Conference*, 1996.

[17] R. Snyder, D. MacKenzie, and R. Tomlinson. Robustness infrastructure for multi-agent systems. In *Open Cougaar 2004*, 2004.

[18] F. Souchon, C. Dony, C. Urtado, and S. Vauttier. Improving exception handling in multi-agent systems. In *Advances in Software Engineering for Multi-Agent Systems*. Springer-Verlag Lecture Notes in Computer Science, 2003.

[19] A. Unruh, J. Bailey, and K. Ramamohanarao. Managing semantic compensation in a multi-agent system. In *The 12th International Conference on Cooperative Information Systems*, Cyprus, 2004. Springer Verlag LNCS.

[20] A. Unruh, J. Bailey, and K. Ramamohanarao. A framework for goal-based semantic compensation in agent systems. In *1st International Workshop on Safety and Security in Multi-Agent Systems, AAMAS '04. To appear in Springer Verlag LNCS*, 2005.

[21] Aidong Zhang, Marian Nodine, Bharat Bhargava, and Omran Bukhres. Ensuring relaxed atomicity for flexible transactions in multidatabase systems. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 67–78, Minneapolis, Minnesota, United States, 1994. ACM Press.