Web Intelligence and Agent Systems: An International Journal 0 (2008) 1–0
IOS Press

Building More Robust Multi-Agent Systems Using a Log-based Approach [1]

Amy Unruh [a] James Bailey [a,*] Kotagiri Ramamohanarao [a]
[a] *Department of Computer Science and Software Engineering, University of Melbourne, Australia*

Abstract. In an agent system, the ability to handle problems and recover from them is important in sustaining stability and providing robustness. We claim that execution *logging* is essential to support agent system robustness, and that agents should have architectural-level support for logging and recovery methods. We describe an infrastructure-level, default methodology for agent problem-handling, based on logging, and supported by declaratively encoding domain-specific knowledge related to changes in goal status and semantic compensations. Via logging, the approach allows repair of already-completed as well as current goals. We define a language, APLR, to support and constrain incremental specification of problem-handling information, with the agents' problem-handling behaviour increasing in sophistication as more knowledge is added to the system. The approach is implemented by mapping the methodology and domain knowledge to 3APL-like plan rules extended to support logging.

Keywords: Robust agents, Agent reliability, Agent safety, Agent recovery

1. Introduction

Situated multi-agent systems are often complex, with decentralised models of control. Changes in the environment as well as agent actions can trigger problems, and unaddressed problems can propagate from one agent's tasks to another's. When problems occur, it is often difficult to characterise the global state of a system of agents working towards a shared set of goals and to determine if its behaviour is correct. Thus the ability to handle problems and recover from them can be important in sustaining a stable and robust agent system. Traditional recovery methods employed in (distributed) database systems are not adequate because of the situated nature of agent actions [22].

We will show that by using two types of declaratively-specified domain-dependent knowledge in conjunction with maintenance of *execution history* for an agent, and by defining a developer-level language in which to organise and specify this information, we can support a default agent problem-handling method termed *RCPH* (Retry-Compensation Problem-Handling). By "problem-handling", we refer both to addressing failures, and situations where the effects of a task are undesirable. A default method is one that has general applicability in the absence of more specific knowledge about how to handle a problem.

RCPH is employed at the agent framework level and underlies a high-level agent development language called *APLR* (Agent Programming Language for Robustness), which provides a *goal atomicity* abstraction to shield an agent developer from the details of problem recovery, allowing increased robustness and consistency of agent behaviour. By means of this methodology a matrix of agent problem-handling behaviours is supported, which increase in sophistication as domain knowledge is added by the developer, and provide a search heuristic over a 'plan repair' space. We further describe an implementation in which the problem-handling algorithm and declarative domain

[1] A preliminary conference version of this paper appeared at IEEE/WIC/ACM IAT'06

[*] Corresponding author: Department of Computer Science and Software Engineering, The University of Melbourne, Victoria 3010, Australia. Email: jbailey@csse.unimelb.edu.au, Tel: +61 3 8344 1319, Fax: +61 3 9348 1184

[1] A preliminary conference version of this paper appeared at IEEE/WIC/ACM IAT'06

[*] Corresponding author: Department of Computer Science and Software Engineering, The University of Melbourne, Victoria 3010, Australia. Email: jbailey@csse.unimelb.edu.au, Tel: +61 3 8344 1319, Fax: +61 3 9348 1184

knowledge are mapped to sets of 3APL-like [11] plan rules.

1.1. Example

In this section, we give a motivating example that illustrates many of the robustness issues we will address in this paper. Fig. 1 describes a scenario of robotic agents in a hospital. It considers some of the problems that could occur as an agent is trying to perform a task, and how the agent might usefully react to them. From this example, several observations about robust behaviour may be made:

- It can be useful and often crucial to 'clean up' problems to keep their effects from propagating to other tasks and agents. Re-attempts alone will often be insufficiently robust; and cleanups can 'reset' state such that retry conditions are met. But, useful cleanups are not necessarily 'undos' of a task: it depends upon application semantics.
- The assumption that a goal has succeeded or failed based on execution history is not robust when situated actions can have variable results or are not fully modelled. E.g., a movement may not take an agent to exactly the location intended.
- Observation of full task results can be delayed, triggering reconsideration of task status. E.g., in making a reservation, an agent may charge a credit card, and later discover that the card was fraudulent, rendering the charge unsuccessful.
- The agent needs to reason about the status of high-level goals (*abstract plans*) as well as leaf goals (*basic actions*). The agent should be able to determine whether a goal has already been achieved. Additionally, goals can fail/succeed unpredictably due to exogenous events.
- An agent should reason about the conditions under which a task should be persisted when there is a problem. E.g., if the agent breaks the equipment, it should not persist in carrying it to the doctor. If the equipment is dropped but not broken, the agent should try to pick it up and continue. In either case, it should persist in working towards its higher-level goal of getting equipment of a given type to the doctor.

In the following sections, we will describe an approach to making BDI [17] agent systems more robust by supporting architectural-level problem-handling.

Primary aspects of our approach include the following:

- **Architectural-level** problem-handling should be employed to support robust and consistent behaviour in multi-agent systems. Domain-independent aspects of a problem-handling methodology should not be coded in an ad-hoc manner by the developer of an agent system.
- **Maintenance of execution history** is key to supporting robustness and recovery mechanisms in agents. For sensible recovery, it is necessary to know what has been done previously. Our approach supports this via systematic abstract logging at the architectural level, factored from the application semantics, rather than requiring ad-hoc coding by the developer. Our use of declarative domain-specific agent knowledge is exploited in conjunction with logging to produce more robust agent behaviour. A powerful aspect of the approach is that the log allows the agent not only to address problems with tasks it is currently executing, but problems with *completed tasks whose effects later need to be changed*.
- **Declarative specification of problem-handling domain knowledge** is key in allowing separation of the domain-independent aspects of a problem-handling approach from developer-specified domain-dependent knowledge. In particular we express goal status information and *semantic compensation* knowledge declaratively.
- Use of a **high-level agent programming language** enforces robustness abstractions, insulates the user from the agent's infrastructure-level problem-handling, and supports modular system design.

Our approach supports a *spectrum of default methods that increase in sophistication* as the user provides more domain knowledge, yet provides sensible 'baseline' behaviour. It defines repair in terms of semantic compensation and goal persistence, and exploits logging in conjunction with the incremental provision of declarative goal information. A constraint that shapes our approach is that our model must be consistent across multi- and single-agent scenarios in an open agent system. That is, we should not require tight control/synchronization between the agents; at run-time, an agent should not rely on knowing the details of the tasks the other agents are performing. Thus, the approach allows modular, goal-centred problem-handling knowledge to be composed by loosely-coupled agents. We implement the methodology by mapping it to 3APL-like [11] plan rules, providing a semantics for the approach.

An agent is given a high-level goal of getting a piece of equipment to a doctor. It must locate the doctor and obtain equipment of the given type. The agent can either retrieve the equipment from a hospital storeroom, or if there is no equipment of that type available, arrange for it to be delivered from another hospital and take delivery. The agent then carries the equipment to the doctor's location . Other agents will be moving about at the same time.

The agent might encounter these problems during its task:

- The agent **drops but does not break the equipment it is carrying**. It should re-attempt its '`carry`' goal (perhaps using an alternative decomposition of the goal). However, it will need to pick up what it has dropped before the re-attempt can be successful.

- The agent **drops and breaks the equipment it is carrying**.

• The breakage should be cleaned up. Other agents will have problems navigating the corridor; and the materials may be hazardous.

• If the item is broken, then the agent should not persist in carrying that piece of equipment, but will still try to get some instance of that equipment type to the doctor.

• After arranging for cleanup, if the agent is in a narrow corridor, it should move to a central hall– it will be in the way if it remains, and it should move to a standard dispatch area.

- **The doctor can not be located in a given period of time**. This timeout should cause failure.

- While carrying the equipment, the agent learns that **the doctor is no longer in the original room**. The agent should not continue to carry the equipment to the original location, but should attempt to learn the new location of the doctor and take the equipment there.

Fig. 1. Some problems that could occur as an agent is performing tasks in a 'hospital' environment.

The subsequent sections of the paper are organized as follows. In Section 2 we describe the concept of *atomic goals*, which are the foundation of the approach described. In Section 3, we define a problem-handling model, called RCPH, which realises the goal atomicity semantics, based on a *compensation-retry* model for addressing goal achievement issues, and supported by reasoning about goal status and persistence. The approach is embodied in a high-level agent programming language, called APLR, which supports the goal atomicity model.

Then, in Section 4, we describe a BDI-specific realisation of the model, supported by specification of declarative knowledge about goal status and goal compensation definitions; and logging. Section 5 illustrates the way in which the approach maps to a multi-dimensional spectrum of problem-handling behaviours, and outlines our implementation. Section 6 discusses related work, and Section 7 concludes.

2. Foundation of Approach: Goal Atomicity Semantics

A foundation of our problem-handling methodology is the definition and realisation of what we term *atomic goals*. These are conceptual entities for which state transitions and predictable terminating states are defined, supporting a goal atomicity model that is motivated by transaction semantics. Here, we define and motivate the concept of atomic goals; the following sections then describe how they are supported and realised by our approach.

Atomic goals are problem-solving abstractions for which processing to reach terminating states is not exposed. An atomic goal may be in one of the following states:

- `not_yet_active`,
- one of two *active* states: `in_progress`, `cancelling`
- one of the following three termination states: `succeeded`, `cancelled`, or `aborted`

The terminating state reflects only whether the work succeeded, was cancelled, or was aborted. (Note that 'Failure' is not a terminating state). Thus, arbitrary processing may be performed to reach a terminating state– e.g., it may include work done to address problems (such as reversing action effects and performing retries).

More specifically, the following conditions on atomic goal transitions are defined:

1. For an atomic goal to reach either the `succeeded` or `cancelled` states, explicit state transition conditions for success or cancellation, associated with the atomic goal, must be met. This means that a change in goal state will not be triggered solely by execution history, but only by the environmental state resulting from those actions.
2. Once a goal is initiated, the goal can only terminate as `succeeded` unless cancellation is initiated or the goal is aborted.
3. Once cancellation is initiated, the goal can only terminate as `cancelled` unless aborted– that is, it can only terminate as `cancelled` or `aborted`.
4. When a goal is terminated, its active descendants are terminated as well.

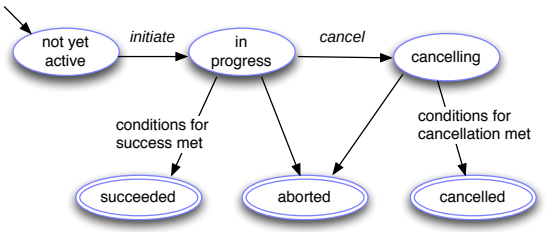

Fig. 2. Atomic goal states and their transitions.

This semantics is captured in Figure 2. It supports robust agent behaviour in several ways:

- The move to a termination state, based on the termination conditions, is decoupled from the agent's activities towards the goal.
- Cancellation will not be initiated, and the `cancelled` goal state may not be transitioned to, without an explicit decision to cancel. Thus, an agent will keep working to reach the `succeeded` state for all its goals unless an explicit decision is made not to do so.
- After the cancellation process is initiated, the `succeeded` state may not be transitioned to. This enforces atomicity of recovery activities.

Thus, provision of this semantics enables a foundation for supporting robust agent behaviour. More specifically, goal atomicity is an approach to supporting and reasoning about agent *goal persistence*, and consequently can be viewed as supporting an agent *commitment* strategy [24,18].

The definition above does not specify *what* conditions define transition to the `succeeded` or `cancelled` states– this is domain-dependent knowledge. If a cancellation were to be an exact undo of the work done towards the goal, then the definition above maps to transaction-like atomicity. However, the examples of Section 1.1 showed that exact transactional atomicity is usually not possible in a situated agent context. 'Cancellation' semantics are only restricted by the atomic goal's state transition constraints and may be realised in any way that is sensible for the application domain.

In the approach described below, we operationalise these semantics for BDI [17] agents by specifying how the atomic goal states are defined and identified, and a methodology for reaching the termination states, in a way that implements a useful definition for *cancellation* and addresses the issues raised in the examples to support robust agent problem-solving. In conjunction, we define an agent-developer-level programming language called **APLR** (Agent Programming Language for Robustness), which supports both the atomic goal abstraction, and specification of the domain information that to supports it.

The model above supports concurrent goal execution, but does not address concurrency management, in the sense that it does not specify goal semantics with respect to isolation and/or collaboration. We return to this topic in Section 7.

3. Supporting goal atomicity: the RCPH Problem-Handling Model

Our approach is centred on a problem-handling model we term RCPH (Retry-Compensation Problem-Handling). The RCPH model is a realisation of the goal atomicity semantics of the previous section for BDI agents, based on a *compensation-retry model* for reacting to goal achievement problems. RCPH uses a *compensation-based* repair strategy, and allows reasoning about what goals to persist towards achieving, and at what task level. RCPH may be viewed as a *default strategy* in support of goal atomicity, which may be used by an agent to increase the robustness of its behaviour. In this section we describe the key ideas of the model without discussing BDI-specific [17] realisations; then in the following sections describe our approach for supporting it.

The RCPH model separates two primary aspects of problem-handling, for robust agent behaviour: it separates *detection* and characterization of a problem from the semantics of how to *handle* the different types of detected problems. This allows both types of knowledge to be expressed declaratively, both default and domain-specific knowledge to be represented, and different default models to be employed.

By separating detection from handling, explicit reasoning about *goal persistence* is supported. The ability to reason about *goal achievement* and *persistence* (sometimes termed *commitment strategies*) can be key in supporting robust agent behaviour [24,18]. For robustness it is often necessary to reason about whether an agent should continue to work to achieve a goal. For example, detection of failure state for a goal does not necessarily imply that the goal should be dropped (not further worked towards). Conversely, it may be that a goal has not failed, but should be dropped. Examples are cancelling a trip due to an emergency; or from Sect. 1.1, cancelling the 'carry to location' goal if the

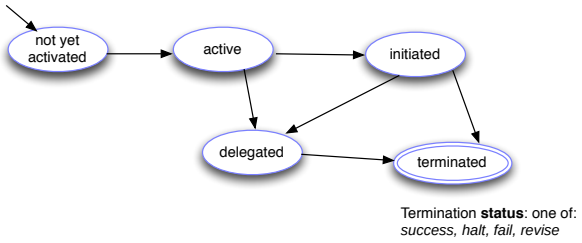

Fig. 3. Goal instance states and their transitions. Work on a given goal *instance* stops when the terminating state is reached. However, the termination may trigger subsequent related problem-handling in RCPH.

doctor moves. Note that there are two aspects of goal persistence. One is whether the agent should persist in trying to achieve a *specific* (sub)goal (perhaps using a different decomposition of that goal). The second is whether the agent should persist in trying to achieve an *ancestor* goal of a subgoal that is dropped.

We define RCPH in the context of use by a BDI-like agent architecture, where agents make use of a plan library specifying alternative ways of decomposing or refining tasks. Thus, for a given *abstract* goal, the agent has been provided a specification of way(s) to decompose or refine that goal into subgoals. Without loss of generality we can view this knowledge as a set of rules. We assume that each such plan refinement rule includes *guard* conditions, which determine eligibility, and a plan body which defines the refinement. If the guard conditions are not met, the rule is not eligible to be selected for application. As the state of the world changes, some decompositions of a goal may no longer be applicable, and others may be newly applicable. If a goal is attempted more than once, the eligible set of decompositions may not be the same each time. When we refer to a *goal retry*, there is no assumption that a previously-attempted decomposition, if still eligible, will or will not be selected for the retry– the model imposes no constraints on how a goal re-decomposition must be performed.

3.1. Goal Instance States

Goal *instances* are the (sub)goal instantiations created by an agent when a plan with specific parameter bindings is selected and instantiated. We alternatively refer to these as *tasks*. To support the RCPH model that we describe below, we require the agent to be able to represent and distinguish between the following goal instance states:

- `not yet activated` - the goal instance is not eligible for consideration for execution or refinement as defined by the agent's deliberation model.
- `active` - execution of the goal instance is not yet initiated, but it is eligible for consideration for execution or refinement as defined by the agent's deliberation model.
- `delegated`- the goal instance is eligible for consideration and has been *delegated* to another agent. Delegation sets up an explicit relationship between the task and a corresponding task of the second agent. (Our model accommodates delegation, which requires a set of communication protocols; however, we do not focus on that aspect of the model in this paper).
- `initiated` - The goal instance moves to this state when execution of the goal instance has begun. If a goal instance has moved to the `initiated` state, then its ancestor goal instances, with respect to the task decomposition hierarchy, move to the `initiated` state if they have not already done so.
- `terminated` - the goal instance is no longer being worked on. If a goal instance is terminated then all of its descendant instances, with respect to the task decomposition hierarchy, also move to the `terminated` state[1].

Figure 3 shows these states and their transitions. A terminated goal instance may not be re-activated. (However, as will be described below, another *version* of that goal parameterization may be instantiated in order to re-attempt achievement of a terminated goal). When a goal instance transitions to the termination state, then to support the RCPH model, it must provide a *termination status*, one of: `success`, `fail`, `halt`, and `revise`. In all cases– by definition– work on the given goal instance terminates regardless of termination status. However, the termination and its status is used by RCPH to reason about subsequent problem-handling, and may trigger initiation of a new goal instance to implement a retry of the goal.

- `success` indicates that the task has succeeded.
- `fail` indicates that failure has been detected for the task.

[1]Note that the implementation of the *initiated* and *terminated* semantics, in a context where subtasks are delegated between agents, requires inter-agent communication.

- `halt` indicates that execution of a task should be stopped, with no associated goal achievement semantics.
- `revise` indicates that even though the goal was considered achieved, its effects now are viewed as unsuccessful and should be addressed. Thus by definition a `revise` status may be generated only for already-*terminated* goal instances that are set to status `success`.

3.2. The RCPH model: problem-handling semantics

In the remainder of this section, we describe the way in which RCPH leverages this representation of goal instances, in conjunction with information about the agent's execution history, to serve as a realisation of the goal atomicity semantics described in Section 2. Subsequent sections will then describe in more detail how the model is supported, both via declarative rules and architecturally.

We first describe RCPH's problem-handling model. By problem-handling, we mean the agent's model for reacting to both goal failures, and situations in which work towards a goal should be halted. The RCPH model has two primary characteristics: a *compensation-retry* model of problem-handling, and explicit reasoning about goal persistence.

RCPH supports the goal event handling semantics outlined in Figure 4. (The concept of *escalation* is described below). In the figure, *repair* indicates an 'undo', or 'cleanup', of the work done towards the problematic goal. As will be discussed below, our realisation of RCPH supports repair via *semantic compensation*; however, other repair models would be possible as well. (Section 4.2 will specify how semantic compensation is defined and supported). In RCPH, *a detected problem always triggers repair*– that is, repair of a task is always required prior to a re-attempt of that task– though the manner of repair (compensation) is situation-dependent (and may be null).

Revision indicates that the goal will be re-attempted after repair, and *cancellation* indicates that the given goal should be halted if active and will not be pursued further. Thus a decision to cancel vs. revise a goal indicates a local decision about *goal persistence*: reasoning about whether to pursue achievement of a specific goal.

The behavioural modes of the figure indicate derived *handling* semantics, not the termination status of the goal instance. For example, *cancellation* mode does not indicate whether the goal instance has terminated in `failure`; but rather indicates that a reattempt on the terminated (and possibly failed) goal should not be pursued. For example, this may be the case if multiple re-attempts of the goal have failed. Conversely, cancellation might be initiated even if the goal has not yet failed; e.g. as determined by detection of exogenous events which obviate the need for goal achievement. Similarly, a *revision* might be triggered either via directly detected failure or via escalation of problem-handling as discussed below. As will be described in Section 4, determining what handling behaviour is applied when, requires both detection of goal status changes, and reasoning about goal persistence.

As indicated by the figure, decoupled *repair* and *retry* (task re-achievement) activities serve as two key building blocks of RCPH. Decoupling repair and re-achievement can support robust problem-handling in a number of ways:

- It allows reasoning about *goal persistence* (sometimes termed *commitment strategies*): explicit reasoning about whether to persist towards goal achievement after (possibly multiple) repair efforts, and at what level of a goal hierarchy. Thus it gives the agent the ability to drop lower-level goal re-attempts but persist towards achieving a higher-level goal.
- It supports the concept of *cancellation* (where no further work should be done to achieve the goal).
- It allows reasoning about handling problems that come up *during* repair.

3.2.1. **Escalation** *of problem-handling and goal persistence*

If an agent cannot repair a problem at task level at which the problem was originally detected, then a useful strategy is to trigger handling at the parent task level: that is, to generate a goal event which triggers repair and retry activities at the parent level. We call this strategy, in which a handling decision is generated at a higher task level based on lower-level activity, *escalation*. Escalation encompasses *exception handling*– i.e., passing up ('throwing') an exception or fault for handling by the parent context.

However, escalation is of broader scope: it can also be used also to address a need for revision of a goal implementation in the context of cancellation. If a currently executing subtask is *cancelled*, then as defined above above, the semantics of the cancellation are that the task effects should be 'cleaned up'– addressed via compensation– but the task should not be persisted:

1. Goal *success*: no further work is needed (the goal is terminated), and no repair is needed.
2. Goal *revision*: The goal instance is terminated. However, the problem should be repaired, and another attempt towards achieving the same goal should be made after repair – that is, the agent should keep working towards that goal.
3. Goal *cancellation*: The goal instance is terminated. Repair should be made. No further work towards that specific goal should be done after repair. Handling should be *escalated* to the lowest ancestor of that goal on the execution path, by subsequently applying the *revision* mode to that goal after the repair.

Fig. 4. Informal description of the semantics of goal termination handling supported by RCPH, in terms of termination *handling modes*. 'Repair' is used to denote compensation or 'cleanup' of the goal instance's effects; as distinct from goal re-achievement efforts. The concept of *escalation* is explained below.

no further attempt to re-achieve the subtask should be made. A subtask cancellation will semantically compensate for the effects of that subgoal as appropriate for the context of the parent decomposition. So, simply continuing with the other subtasks of the parent decomposition will not (necessarily) be correct with respect to producing the results intended for that parent goal, which the agent still intends to achieve.

This means that, in the absence of domain-specific knowledge about the parent task[2], a cancellation of a child task should *also* trigger subsequent revision– compensation and then a retry– at the parent goal level in order to preserve decomposition semantics. Any parent goal re-decomposition will by definition be based on the agent's *current* environment, including the aspects of the state that caused the child goal cancellation, thus the re-decomposition may be different than the prior decomposition. With the 'hospital' example of Sec. 1.1, if a piece of equipment becomes broken, a re-decomposition of the parent task may involve obtaining a replacement. As will be described in the following sections, subgoals of the re-decomposition *may be detected as already achieved*, obviating the need for further work towards them.

More generally, in addition to tasks under current execution, RCPH supports cancellation of the effects of a finished, successfully-performed task. In such a case, the effects of that cancellation can impact the ongoing higher-level activity of which this task was a (finished) subtask. For example, suppose that a credit card charge that was initially thought successful is later marked as fraudulent. This can impact a higher-level "process order" task of which the credit card charge task was a part. To remediate interactions in such cases, a revision– a compensation and retry– is subsequently triggered for the **lowest-level** (nearest) ancestor in the task decomposition hierarchy of the cancelled task, that is on an *execution path*. (Note that if a goal is cancelled before it is terminated by success or failure, the lowest-level ancestor on the execution path will always be its parent, reducing to the scenario above).

E.g., with the credit card card example above, suppose that the lowest-level ancestor of the credit card task is the "process order" task. Post-execution failure of the credit card task will therefore trigger both compensation for the credit card charges, and compensation for and re-decomposition of the parent task (e.g., recall the shipment if possible, then handle the order in light of the new situation by logging it in the company's fraud database).

Escalation does not imply that there has been an explicitly detected failure of the goal instance to which repair is escalated; rather the failure is implicit in the failure to successfully perform lower-level repairs. With escalation, the handling behaviours support a model of goal persistence at some level of the task hierarchy; goals for which a problem has been detected will either be persisted at that level via revision behaviour, or cancelled and then *revision* precipitated at a higher goal level.

Escalation is thus part of the RCPH model, as included in Figure 4; it is employed to ensure that implicit interactions between cancellation of finished tasks, and ongoing work, will be addressed. As discussed in more detail below, default logic about persistence provides guidance about when the agent should persist a local goal, and when to *cancel* that effort and 'push' handling upward via escalation to a higher-level revision. The default knowledge may be overridden by the developer with situation-specific problem-handling knowledge where available.

[2]A BDI agent need not have explicit models of action effects. In fact its models may be primarily implicit, encoded as selection knowledge for plans from libraries.

3.3. Supporting goal atomicity with the RCPH model

The two key characteristics of the RCPH model– its repair/retry model, and its support of goal persistence via escalation– allow it to support and map to the *atomic goal* abstractions of Section 2.

We describe this mapping in terms of APLR, our high-level agent programming language for robustness. APLR supports the *atomic goal* abstractions of Section 2, thus shielding the developer from the details of execution. Atomic goals are by definition transparent to the 'internal' processing that takes place before one of the terminating atomic goal states is reached, and thus may include repair and retry activities. Atomic goals are a *run-time*, not define-time abstraction; the ability to access execution context in specifying domain knowledge, while shielding the developer from details of the problem-handling methodology, is key to supporting robustness in an APL.

Below, we reconcile the APLR-level abstract goal representation with the RCPH methodology by defining how the APLR goal references– which have atomic goal semantics– are mapped to the underlying execution model, and how the *status* values of APLR (atomic) goals map to goal instance information. Using this mapping, **the status of a referenced goal in APLR does not evaluate to `failure` until recovery work on the goal has finished**.

It is worth noting that the APLR-level atomic goal abstractions may be decoupled from the RCPH methodology; other underlying problem-handling models might also be mapped to the APLR-level goal atomicity semantics.

3.3.1. Task trees and RCPH-driven task tree rewriting

We define a *task tree* to be the tree formed by recursively following the task decomposition *child_of* relationships from a root task, created at run-time. An agent may have a 'forest' of such trees, each corresponding to a current root task that it is working on.

Given a partial-order tree in which serial siblings are ordered from left to right, the *execution paths* in a task tree are:

- The rightmost leaves which are in an `initiated`, `delegated`, or `terminated` state; and
- the *ancestors* of those leaf nodes as defined by the task decomposition relationship.

Thus, an execution path can be viewed as describing an 'execution edge' in its task tree. If a tree contains concurrent subgoals, it will have more than one execution path[3]. Figure 5 shows two execution paths in a task tree.

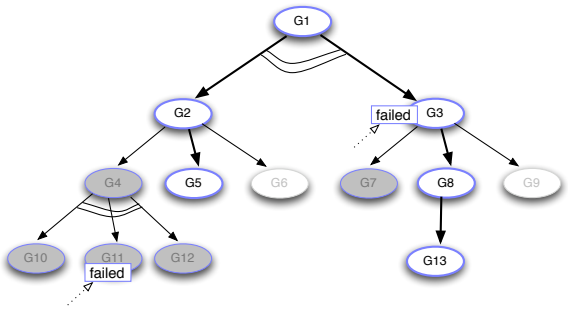

Fig. 5. Task tree execution paths. Concurrently-executing subtasks are denoted by double bars. Terminated task nodes are darkened; task which are not yet initiated are grayed-out. Nodes with heavy borders are in an `initiated` or `delegated` state; the two execution paths are shown by the heavy arrows. In our model, sequentially-ordered subgoals can not become active until their previous siblings have terminated. The figure shows detection of failure for an abstract (non-primitive) subgoal instance (G3) as well as post-facto detection of failure for a task (G11) originally thought to have been successful; both of which are supported by RCPH.

As described above, the RCPH model utilizes task compensation and retry to effect its problem-handling. We extend the task tree model to include RCPH by additionally requiring that:

- the compensation (repair) for a task, optionally followed serially by a retry of the task, *replace* the original task in the task tree. The original task is by definition terminated. (Recall that '*replace*' is defined to mean that the new goal(s) are inline replacements for the goal that got the event, with respect to its task decomposition and sibling relationships. The replaced task and its children are no longer active and are removed from the task tree.)
- `retry_of` and `compensation_of` relationships between tasks (goal instances) are recorded.

We can then define the *versions* of a goal to be the temporally ordered series of goal instances with the same parameterization (argument bindings) in the execution history that are related by `retry_of` replacement relationships[4]. Fig. 6 illustrates a problem-

[3] These definitions do not enforce any particular implementation models; however, many agent architectures maintain non-terminated tasks in a *goal base*.

[4] Our model also supports other types of replacement relationships in relating goal versions; but for simplicity we do not discuss them in this paper.

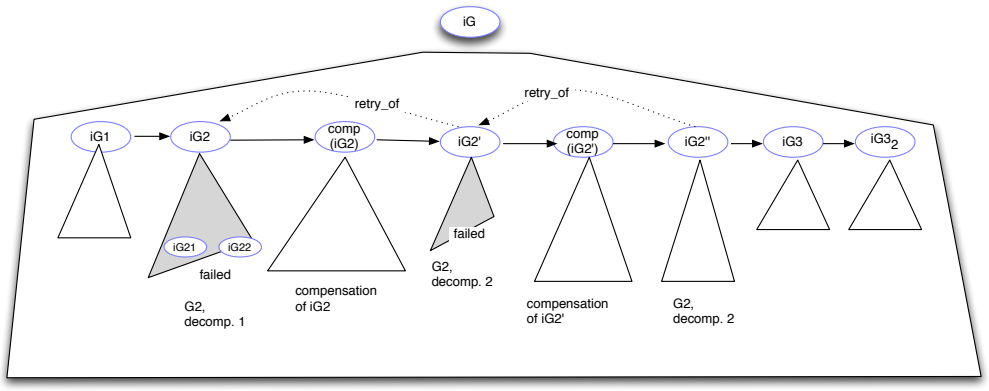

Fig. 6. An example problem-solving history which includes handling of problems in achieving G2. The prefix *i* is used to indicate that the history contains goal instances. Goal G is defined to have one decomposition, which expands to the subgoal sequence **G1; G2; G3**. Goal G2, one of G's subgoals, has two different decompositions defined, and both are utilised in this history (the goal instances *iG2* and *iG2'*). The term *compensation* will be defined in the next section and refers to the work done to compensate for a task's effects. (As will be described, the of the different decomposition instances may themselves be different).

handling execution sequence, or history, which includes multiple versions of a goal.

With this model, a task tree can then be viewed as a 'current snapshot' of problem-solving, excluding tasks which terminated unsuccessfully and have been replaced by subsequent versions of repair tasks, but including those subtasks of a current activity for which work is in progress or has not yet begun.

Maintenance of such an execution history, including maintenance of the current task tree and the relationships between goal instances in the history, requires *logging* of (abstracted) execution history. As will be shown in subsequent sections, logging is required for and supports other aspects of our methodology as well.

3.3.2. Mapping task tree information to atomic goal states

Leveraging the relationships between goal instances that are created via the RCPH rewriting (replacement) model above, the execution history– in the form of the task tree and its related terminated tasks– defines and provides a mapping to the *atomic goal* states of Section 2. That is, by enforcing the model above, a BDI agent's current problem-solving runtime state– described in terms of goal instances– can always be expressed abstractly in terms of references to atomic goal states.

The mapping from goal instance history to atomic goal abstractions requires a recursive definition: the compensation tasks within a repair history must themselves be treated as atomic when performing the mapping. Without this semantics– that is, without treating compensation activity as atomic with respect to analysis of the 'forward' problem-handling history– it is not possible to distinguish whether atomic goal achievement has been aborted or is still in progress.

Our high-level language, APLR, allows goal predicates to be used in its definition rule bodies, of the form:
`<goalname>(<arg1>...<argN>)`. For example, goal predicates are used in the definition of decomposition rules, which define rules for implementing a goal by decomposing it into subgoals. (APLR definitional syntax will be described in more detail in the next section). In order to represent and use atomic goal abstractions as concepts, APLR must be able to use atomic goal references– which are run-time abstractions– in its definitions. This requires two things: tying the goal references in the rules to the goal predicates in associated rule definitions; and run-time determination of the status of a goal reference, from execution history. The former is required because in writing rules that include goal status conditions, there must be a way to tie that goal reference to specific information in the agent's execution history. The latter is required because an atomic goal abstraction maps to a series of goal instances in an execution history; the current status of the atomic goal must be derived from that history.

APLR uses the following syntax to allow access to the *status* of referenced runtime atomic goals:

```
status(<goal_ref>, <status_value>)
```

Since a <goal_ref> must be tied to a specific definitional goal predicate in an associated rule, and because a rule might contain more than one predicate with the same goal name and arguments, APLR uses a notation in which references to run-time subgoals are made by specifying the *index position* of the corresponding goal in the decomposition definition body, with the rule body read from left to right. The syntax: child[<i>] is used to indicate the correspondence[5]. A reference to an 'out-of-bounds' index returns undefined.

Once identified in this manner, an APLR goal reference then corresponds to a set of of goal instances in the actual execution history– generated via rule execution– which are *versions* of the referenced goal element in the specification. All tasks instantiated from the same goal element, including retries due to failure, are members of this set. We refer to the temporally newest version of a goal as its most *recent* version. Only references to non-repair goals may be made in this manner.

APLR defines the following status values for a goal reference, reflecting the states of Section 2:

not_yet_active, in_progress, cancelling, succeeded, aborted, cancelled, failed, and failed_escalation

We have introduced new status values failed and failed_escalation, which were not a distinction made in the model of Section 2. These two status values map to the atomic goal terminating state cancelled of Figure 2. In APLR these additional status distinctions provide information about the problem-solving which led to the cancellation, which may be used to make decisions about how to handle the problem.

Figure 7 shows how the APLR (atomic) goal states are defined based on the status of its corresponding goal instances, and the agent's problem-handling history. Note that APLR semantics make no requirements on the implementation of the logging which supports this reasoning, which need not be homogenous across an agent system. The 'associated compensation' for a goal instance refers to a compensation task to repair problems with a given goal instance; the compensation model itself is described in the next section. One important implication of this mapping is that the status of a referenced goal in APLR *does not evaluate to* failure *until recovery work on the goal has finished*. Failure of a goal *instance* does not necessarily indicate atomic goal failure. The example of Fig. 6 illustrates this concept.

Thus, the RCPH semantics described in Figure 4 may be mapped consistently to the atomic goal states of APLR. In the following section, we describe a rule-based specification of RCPH which realizes this semantics.

[5] APLR imposes certain definitional constraints so that such a correspondence is always well-defined.

4. Realisation of the RCPH model: APLR with BDIH

In this section we specify APLR in more detail, and describe how RCPH, and its use by the APLR language, is supported both via a declarative rule model and via agent architectural support.

Our approach is developed for BDI agents, which treat goals procedurally (e.g. dMARS [5], JACK [12], and 3APL [11]). There are two issues with a canonical BDI model from the perspective of agent robustness and the RCPH model.

First, we claim that some form of execution logging is required in general for robust agent behaviour: in order to react robustly to problems it is necessary for the agent to reason about its past actions. However, it is not robust to require the agent developer to support direct execution log management and access. Instead, the agent architecture should support automatic execution logging at the framework level. A developer-level agent programming language, such as APLR, should then provide abstractions of the logged information to support problem-handling and robust behaviour. We name the architectural extension developed in this research *BDIH*, where *H* denotes *'History'*.

Additionally, a procedural representation does not allow reasoning *about* the agent's goals [24] and is thus limited with respect to supporting robust behaviour. By representing domain knowledge about procedural goals (abstract plans) declaratively, the agent can reason about how to revise them in response to problems, and can separate its reasoning algorithms from domain knowledge.

We exploit declarative domain-specific agent knowledge in conjunction with logging to produce more robust agent behaviour. The APLR language allows the developer to organise and specify this declarative information, and leverages the underlying logging trans-

- If its corresponding goal instance has status `not_yet_active`, the APLR goal reference status is `not_yet_active`.
- An APLR goal reference `G` has status `in_progress` when:
  Given `G`'s most recent version `V`:
  - the state of `V` is not `terminated`, or
  - the termination status of `V` is `fail` or `halt`; and the *termination handling mode* for the goal instance `V` is *revision*.
  That is, `G`'s status is `in_progress` when local problem-handling for the referenced goal is still underway. This will be the case if implementation of the the most current version or retry of a subgoal is still in progress.
  (In the example of Figure 6, if `child[1]` were to be referenced in the context of `G`'s decomposition specification– referring to `G2`– then `status(child[1], in_progress)` will hold until `iG″` terminates successfully).
- A goal reference `G` has status `succeeded` if the goal has terminated with success. This is the case when:
  Given `G`'s most recent version `V`:
  - the termination status of `V` is `success`
  (Continuing the example of Figure 6, `status(child[1], succeeded)` will hold once `iG2″` terminates successfully).
- A goal reference `G` has status `aborted` if:
  - For `G`'s most recent version `V`, `V′s` subsequent (atomic) compensation task has terminated with non-success.

The status types `failed`, `failed_escalation`, and `cancelled` are all subclasses of the atomic goal *cancellation* state. These conditions must hold for all of the following:

- the subsequent (atomic) compensation task for `G`'s most recent version `V` has terminated with success.
- the *termination handling mode* for the goal instance `V` is *cancelled*.

The cancellation subclasses are then distinguished as follows:

- An APLR goal reference has status `failed` when additionally:
  - `G`'s most recent version `V` has goal instance termination status `fail`.
  Thus, a goal reference will not return `failed` status until the local recovery effort for that goal has completed.
  (In the example of Figure 6, `status(child[1], failed)`, referenced in the context of `G`'s decomposition specification, will never hold. However, if instead after N retries of `G2`, problem-handling had instead been escalated to `iG` (terminating local goal persistence), `status(child[1], failed)` would then hold).
- An APLR goal reference `G` has status `failed_escalation` if a recovery process for that goal, initiated because of an escalation goal event action, has terminated unsuccessfully. This is the case when additionally:
  Given `G`'s most recent version `V`:
  - the termination status of some version `V′` of `G` was `halt` set via *escalation* from a failure event;
  - the termination status of `G`'s most recent version `V` is `fail`
- An APLR goal reference `G` has status `cancelled` when additionally:
  - `G`'s most recent version `V` has goal instance termination status `halt`.

Fig. 7. Mapping between APLR atomic goal states and goal instance handling mode and execution history information. The definition of an *aborted* atomic goal reflects the recursive nature of the mapping.

parently to the developer, supporting a simple, unified algorithm for agent system problem-handling.

We utilise two types of declarative problem-handling knowledge: knowledge about detecting changes in goal status; and information about how to *semantically compensate* a goal. We take the approach that problem-handling knowledge in a BDI context be *organized by goal and goal decomposition structure* to aid robust design. Our default problem-handling method is based on and leverages decomposition knowledge[6]. (An agent *system* need not itself be hierarchically organized to exploit goal decomposition).

In Sec. 3, we described the way in which APLR's reference to atomic goal concepts are mapped– transparently to the user– to the underlying goal instance execution history. In this section, we describe in further detail our agent programming language, called *APLR*, which allows the developer to leverage the vocabulary

[6]By 'decomposition' knowledge, we mean knowledge about how to decompose a goal into subgoals (we include means-end analysis knowledge).

```
<goal_spec> ::= 'goal: ' <head> ':-'
  'decompositions: {' <decomp_list>  '},'
 # goal-level problem-handling knowledge
 <problem_handling_knowledge>

<decomp_list> ::= <decomposition>|
             <decomposition> ‘,’ <decomp_list>

<decomposition> ::=
 '{' <decomposition_spec> ','
  # decomposition-specific problem-handling knowledge
  <problem_handling_knowledge> '}'

<decomposition_spec> ::= <guard> ‘|’ <body> | <base_action>

<problem_handling_knowledge> ::=
 'p:{' <decl_goal_event_rules> ','<compensation_specs> '}'

<decl_goal_event_rules> ::= 'g:{' <dg_list> '}' | 'g:{}'
<dg_list>     ::= <dg_rule> | <dg_rule> ',' <dg_list>
 <dg_rule>  ::= <hguard> '->' <goal_event_action> | <dg_param_setting>
<dg_param_setting> ::= 'retries:' <posint> | 'timeout:' <posint> | 'persist:' <boolean>

<compensation_specs> ::= 'c:{' <cs_list> '}'
<cs_list>     ::= <compens_spec> | <compens_spec> ',' <cs_list>
<compens_spec> ::= <guard> '|' <body> | COMPOSITIONAL | NULL
```

Fig. 8. Fragment of EBNF specification of APLR syntax for specifying decomposition and problem-handling knowledge for a goal, with `head`, `guard`, and `body` as defined by the 3APL BNF spec. [1]. By grouping the decompositions for a goal, all decompositions have the same ‘head’. Thus each decomposition rule includes only the guard and body for that decomposition.

of atomic goal expression to encode high-level agent problem-handling knowledge. APLR explicitly distinguishes goal decomposition semantics from other ‘plan rule’ semantics, such as composite plans and reactive rules (rules of the form *Condition* $\Rightarrow$ *Action*, which may create new goals). In this paper, we describe only the structure of APLR’s goal decomposition information.

Figure 8 shows the high-level structure of goal information in APLR[7]. For a given goal, the user may include both problem-handling knowledge specific to a particular decomposition– *decomposition-level* knowledge– as well as *goal-level* problem-handling knowledge, applicable to all decompositions of a goal. This definitional approach is analogous to the object/method association in OOP languages, and has similar organisational benefits in both definitional clarity and support for refinement. APLR’s decomposition rule model is as in 3APL, where tests on ‘domain events’ (information about the agent’s state) may be expressed in a rule’s guard conditions, determining the rule’s eligibility. Sections 4.1-4.2 will describe the `<problem_handling_knowledge>` component of the specification. The problem-handling knowledge associated with a goal *does not need to be complete for it to be used*.

For effective problem-handling, it is necessary for the agent to consider execution state in terms of goal status. We assume that agents do not necessarily have full access to the details of each other’s history and activities. Thus APLR maintains compatibility with agent modularity by only allowing goal status tests on a goal’s parent and children in the task decomposition hierarchy; this information will always be available regardless of the agent’ delegation model. Thus APLR– as described below–allows access to subgoal execution history in defining information about a goal.

4.1. Goal status rules: Reasoning About Goal Achievement and Persistence

We now describe how the goal status handling modes, which support the problem-handling semantics of Section 3, are determined. Derivation of this information involves reasoning about both changes in goal status, and when and which goals should be persisted.

Knowledge about goal achievement and persistence should be applicable at all levels of an agent’s goal

[7] This specification listing does not include the syntax for specifying user-defined goal-event-handling knowledge as will be described in Sec. 5.1.

decomposition hierarchies, not only for its abstract goals[8], and thus in the context of problem-handling we do not make the distinction that is sometimes drawn between high-level goals and 'abstract plans'. This reasoning capability allows:

- *decoupling goal failure from decomposition failure*, and distinguishing between a goal that has failed based on domain conditions, and one that requires repair because recovery at a lower level was not successful.

- detection of changes in goal status at any level of a goal hierarchy, not only at the point of execution; and deliberative reasoning about which changes to address, if multiple changes are triggered simultaneously by domain events.

Further, the ability to represent and reason about the conditions for success or failure of a goal increases agent robustness with respect to modelling of domain actions: the agent terminates work on a goal when termination conditions are met, rather than simply assuming success when all subgoals are executed or assuming failure if a subgoal fails. The example of Sec. 1.1 illustrated that such assumptions are not always safe.

Note that any persistence logic must be able to access an abstraction of the agent's execution history to be effective: the agent can not make useful decisions about whether to continue to work towards a specific goal without information about previous retry attempts.

To support declarative reasoning about goal status and persistence we must associate several classes of information with each goal instance and define how this information is manipulated. The first type of information is *goal instance termination status*, one of: `success`, `failure`, `halt`, and `revise` as in Figure 3. Second, we define two attributes associated with each goal instance, `persist` and `repair`.

APLR abstracts from this information, which the developer does not manipulate directly, by defining a set of *goal event actions*. Application of an action *terminates* a goal instance, causing a change in goal instance status, which we term a *goal event*. (The handling of generated goal events is described in Sec. 5.1). Fig. 9 lists the set of goal event actions and the status changes (goal events) and attribute changes they cause.

[8]We assume an agent architecture where leaf goal (basic action) success or failure status is generated as part of execution, and that execution can change the agent's environment as well as its internal belief base, thus potentially triggering changes in achievement status of any goal.

APLR then supports specification of *goal status rules*, which encode the conditions under which a goal event action may be applied and thus a goal instance *terminated*. The rules allow expression of the conditions under which a goal achieves success or failure, or work is halted, as well as the conditions under which handling of a goal is *escalated*. By allowing declarative specification of this information, APLR supports explicit detection and reasoning about goal achievement and persistence. Fig. 10 shows goal status rules from the example of Sec. 1.

In general it is difficult to specify all conditions that determine goal failure/success/cancellation in a given domain (determination of failure can be undecidable). It is not robust to require a developer to specify such information on a per-goal basis. Thus default goal status rules are required so that that the developer may add goal status knowledge incrementally, with sensible behaviour resulting. APLR defines a system-wide default rule for goal success, included in Fig. 10, and supports system-wide overrideable default logic for goal escalations and goal timeouts. (Default failure-detection rules are not required: default reaction to failure is subsumed by our problem-handling model, as will be described in Sec. 5). The developer then may add more specific goal status rules, which override default behaviour where applicable. As will be described in Sec. 5, the goal status rules, and the goal events they generate, then allow the agent to support the problem-handling status modes and behaviour of Section 3.2.

Fig. 11 shows examples of goal status rules defined in APLR. Goal-level knowledge, defined for all decompositions of a goal, is applied only if decomposition-level knowledge has not been defined.

4.2. Semantic Compensation

The second type of declarative domain-dependent problem-handling knowledge specified in APLR is *semantic compensation* knowledge. Semantic compensation is used as the *repair* component of RCPH. The example of Section 1.1 showed that some form of 'undo', or cleanup, is often key to successful recovery. In a system of situated agents, interacting with their environment, most actions 'commit' immediately . Thus in an agent system, a cleanup must usually be approached via semantic compensation, as rollback is not feasible [21]. A semantic-compensation-based approach can be viewed as a (default) search heuristic for replanning: often, an effective way to fix a problem is to 'reset' and then re-address the problematic goal.

The goal event attributes used by RCPH:

- `repair` flag: boolean
- `persist` attribute: one of `false`, `local`, or `escalate`.
- (optional) goal event `MODE` information

The **goal event actions** defined by RCPH, which utilize the attributes above:

- `success` action: place **`success` event** + `repair=false` + `persist=false`. May be applied only to tasks on an execution path.
- `fail` action:
  if task is on execution path: place **`fail` event** + `repair=true` + `persist=local`
  if task is successfully finished: place **`revise` event** + `repair=true` + `persist=escalate`
- `cancel` action:
  if task is on execution path: place **`halt` event** if an event is not already placed + `repair=true` + `persist=escalate`
  if task is successfully finished: place **`revise` event** + `repair=true` + `persist=escalate`
- `escalated` action: place **`halt` event** + `repair=true` + `persist=local`. May be applied only to tasks on an execution path.
- `halt` action: place **`halt` event** + `repair=false` + `persist=false`. May be applied only to tasks on an execution path.
- The following two actions, which do not set a goal event, are only applicable if the goal instance has already terminated via a previous goal event.
  * `set_escalate` action: `persist=escalate`
  * `unset_persist` action: `persist=false`

Fig. 9. The set of goal event actions which support RCPH, and their associated event attributes. Most of the goal event actions generate goal status events, moving the goal instance to the *terminated* state. Two actions only modify the goal event attributes; not generating a new event but changing attribute value(s), thus affecting the event's handling. All actions may include optional specification of termination *mode* information associated with the event, but for simplicity we do not include this above.

- **Cancel** `carry_to(Equip, Loc)` when it is a subgoal of `get_equip_to(Equip_type, Doctor)` if the agent learns that the `Doctor` is no longer at that `Loc`.
- Detect **success** of `get_equip_to(Equip_type, Doctor)` if the `Doctor` obtains equipment of that type, possibly by other means.
- **Cancel** `carry(Equip, Dir)` if `Equip` is broken.
- **Cancel** a `carry_to` goal if child goal `carry` has been cancelled.
- **Retry** a `get_equip_to` goal at most 3 times.
- **Timeout** on `locate_doctor` if the goal is not achieved after 1 hour.

Default **success**: if all subgoals are successful *and* more specific rules are not defined.

Fig. 10. Goal status rules from the example of Section 1.1, expressed in pseudo-code. Note that the rules require access to the agent's execution history; both for reasoning about retries and escalation, and to access the status of previously-executed subgoals of a given goal. As the example shows, the rules may also access the *status* of subgoals for a given goal, where the subgoal status semantics is *atomic*.

There are several robustness benefits to using semantic compensation as a component of a default problem-handling method: it can restore scarce resources; and keep problems from propagating. Further, it can 'reset' an agent and system to match implicit assumptions made in application design: a planner is more likely to have knowledge applicable to states that were considered during development. Semantic compensations can be used to address anticipated as well as as unexpected exceptions, and may perform 'forward' repair as well as cleanup.

A key aspect of using semantic compensation in agent problem-handling is that it can be used to address partially- as well as fully-achieved goals, and address both goal cancellation and failure situations in a consistent way, supporting a **unified repair/persistence model**. Compensation can also apply to tasks that have *already completed*, but whose effects later need to be cancelled.

```
goal: carry_to(Equip, Loc) :-
 decompositions: {
  #decomposition 1
  {east_of(Agent, Loc) |
    carry(Equip, west); carry_to(Equip, Loc),
   p:{}}, # no decomposition-level problem-handling knowledge
  # decomposition 2
    {north_of(Agent,Loc) |
      carry(Equip, south); carry_to(Equip, Loc), p:{}}
    ... },
 #goal-level problem-handling knowledge
 p: {g:{status((child[0], cancelled) -> cancelled(self) },
  c:{...}   }
```

```
goal: get_equip_to(Equip_type, Doctor) :-
 decompositions: {
 {obtain(Equip_type); locate(Doctor);
 (if ((isa(Equip, Equip_type)) && holding(Agent, Equip)
   && location(Doctor, Loc)))
    then carry_to(Equip, Loc)),
  p:{} }},
 #goal-level goal status rules
 p: {g:{ retries:3, timeout:1hr,
   ~location(Doctor,Loc) -> cancelled(child[2]),
   holding(Doctor, Equip) -> success(self) },
   c: { (T) | COMPOSITIONAL }   }
```

Fig. 11. Defining declarative goal status knowledge in APLR. The goal is referenced with `'self'`. `'p'`: is the problem-handling component of the specification and `'g'` refers to its goal status knowledge component. 'Agent' refers to the agent executing the rules. (The decomposition body includes an 'if' composite which will not be further discussed here). 'Head' goal and decomposition variable bindings are accessed within the problem-handling component of the definition. Subgoals are referenced via the '`child[N]`' construct, where 'N' is Nth order of subgoal specification in the forward decomposition rule body, and the `status` of such a goal reference has atomic goal semantics. The `'c:'` (compensation) specification is described below.

```
goal: obtain_equip(Equip_type): {
decompositions: {
 # decomposition 1
 {avail(Equip_type, Storeroom) | fetch(Equip_type,
   Storeroom),
   p:{g:{},
     c: {obtained(Equip, Equip_type) | store_nearby(Equip)}}},
 # decomposition 2
 {~avail(Equip_type, ?) | order(Equip_type),
   p:{g: {...}
    c: {(T) | comp(child[0])}}
   },
  #no goal-level compensation knowledge
   p: {g: { ... }, c: {}    }}
```

Fig. 12. Specification of compensation knowledge for a goal and its decompositions in APLR. As with the example of Section 4.1, variable bindings in the forward decompositions may be accessed in defining the compensation specifications.

4.2.1. Definition of Compensation Knowledge

RCPH initiates compensation of a goal instance during problem-handling via introduction of explicit `'comp(<goalref>)'` goals into the agent's goal base, which then trigger compensation activity. (The RCPH algorithm for task tree modification will be described in further detail in Sec. 5). In this section we describe how the APLR goal compensation definitions determine the compensation activity that occurs when a `comp()` goal is selected and expanded.

In APLR, a developer specifies compensation knowledge for a goal in the context of the definition for that goal. This approach is roughly analogous to defining associated object 'end' methods in OO languages. As with goal status knowledge, compensations may be defined for all decompositions of a goal, or for a specific decomposition of that goal (thus compensating for a particular *way* of achieving that goal). For each context, e.g. a certain goal decomposition, there may be multiple compensations defined. The guard conditions of a given definition determine its applicability within its context.

The compensation definitions are decomposition rules themselves. The implicit head of each rule is a *compensation goal* `comp(<goalref>)`, where the argument refers to the goal of the enclosing definition, and the body specifies the goals that should be achieved in order to affect the compensation. (A compensation rule head is always a compensation goal, and thus the head is always implicit in the specification). During execution these rules will match against `comp(...)` goals in the goal base).

A compensation decomposition rule body may include *compensation goals with a subgoal reference as the argument*, e.g. `'comp(<subgoal_ref>)'`. The construct indicates that the compensation for the given subgoal should be initiated as part of the parent compensation. APLR provides a shorthand for defining a compensation in terms of the compensations of its subgoals (executed in reverse order if the 'forward' goals were sequential), called '`COMPOSITIONAL`'. If all but leaf goal compensations are defined compositionally, this is analogous to an *open nested transaction* model [8]. Non-compositional compensations require additional domain knowledge, but tailor a repair to a situation, and thus can better reflect application semantics.

Because compensation definitions are specified in terms of a guard condition and a rule body, as are 'forward' task decomposition rules, they allow context-sensitive expansion of compensation goals at run-time in the same manner. That is, compensations are not predefined plans, but rather are expanded when the compensation goal is selected for execution, according to state and to which decomposition– of the goal being compensated– was executed.

However, the difference between compensation and regular 'forward' decomposition rules is in how they are *selected* and *constructed*: in RCPH, compensation definitions allow **use of subgoal execution history** in both the selection and run-time instantiation of the compensation rules. Reference to (atomic) subgoal status is allowed in compensation definition guard conditions. Further– as with `COMPOSITIONAL` compensation– the actual child subgoal execution order for the goal being compensated may be used at run-time to determine the specific compensation. The compensation definition is an abstraction. Thus compensations are defined in a *modular, recursive*, and *history- and context-dependent* manner: a compensation can be defined in terms of subgoal compensations without requiring access to the specifications of those compensations. The specification and creation of a compensation are disjoint from its implementation, which is determined by context. This results in recursively-defined compensations, which terminate when a compensation is *not* defined in terms of subgoal compensations.

Fig. 12 continues the example of Section 1.1. The example described ways an agent could "clean up" problems encountered while working towards its goals; a subset of these activities are translated to compensation specifications in APLR. The `obtain_equip` goal– a subgoal of `get_equip_to`– has two decompositions: one for obtaining the equipment locally from a storeroom ('Decomp1'), and another for ordering an instance of that equipment from another hospital ('Decomp2'). The compensation for Decomp2 is defined in terms of the compensation of its subgoal. The compensation for Decomp1 is different: it does not make sense to 'undo' the `fetch` goal in the Decomp1 context. Instead, Decomp1's compensation is defined as a goal to *store the equipment in the nearest storeroom*. The `get_equip_to` goal, shown in Fig. 11, has one only decomposition. The compensation for that decomposition is defined *compositionally*: on compensation, the '`comp()`' goals for its subgoals will be invoked. The compensation of the `obtain_equip` subgoal will be determined by which decomposition of `obtain_equip` was performed.

At run-time, a 'goal-level' compensation definition will only be used if no 'decomposition-level' definition is applicable. A compensation may be defined to be NULL (take no action). Defaults support sensible behaviour where compensations are not specified: NULL compensation is the default at a leaf goal (thus supporting retry on leaf failure), and compositional compensation is the default for abstract goals. This allows the system to *'bootstrap' repair knowledge* by defaulting to a recursively compositional model.

Compensation of partially-executed goals via execution history. An agent's ability to sensibly compensate partially-executed goals is key to the utility of compensation. Our model supports this in two ways. The agent's history maintenance allows it to determine how much of the forward decomposition has been executed prior to compensation initiation. If a compensation is compositional, then only the compensations for *the child goals for which work has begun*, are invoked; and if a compensation for a child goal has just been performed prior to addressing a problem at the parent level (as described below), the child compensation will not be re-invoked.

Additionally, an agent's declarative goal status detection knowledge allows it to identify which subgoals of a compensation *have already been achieved*. Such a goal instance will trigger termination with status success and thus will not be be worked on. (The agent employs this approach for any domain goal, not just goals that are part of a compensation.) For example, in the hospital scenario of Sec. 1.1, the compensation definition for the `carry` goal might include a subgoal: `area_non-hazardous`. If a `carry` goal instance is cancelled because of problems with the robot carrying the equipment, then there may be cleanup to perform in order for this subgoal to be achieved. However, if the subgoal's conditions for successful termination are already achieved, no cleanup activity will be initiated.

5. A Matrix of Problem-Handling Behaviors

The declarative domain knowledge described in Sec. 4, in conjunction with the use of logging and repair as compensation, allows the agent to support a default problem-handling method that increases in sophistication as the developer adds domain-dependent knowledge. Fig. 13 shows the spectrum of behaviours produced by this method.

Without user-defined compensations, the system does only goal retry on failure– with no 'cleanup' prior to retry– thus defaulting to a failure-handling model used in many BDI systems. With no domain-specific knowledge about detecting changes in abstract goal status, the system implements bottom-up exception-handling. Failures are triggered (only) by execution problems, with the failure propagated to the parent af-

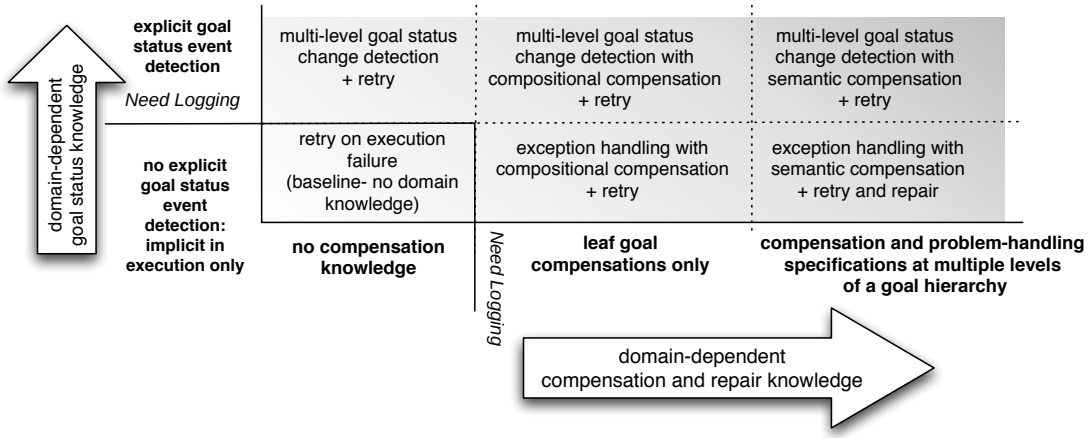

Fig. 13. The spectrum of problem-handling behaviour that is generated as domain knowledge is introduced along two dimensions. The framework-level algorithm remains the same; emergent behaviour becomes more sophisticated as the user adds more information.

ter after N re-attempts (where N is defined by the system's defaults.)

If compensation knowledge is associated only with leaf goals, the resultant compensation behaviour defaults to *compositional*. That is, the compensation of a task is defined in terms of the compensation of its subtasks. For example, a 'travel planning' task could be compensated by initiating the compensation for each subtask ('book flight', 'arrange hotel', etc.).

Compositional compensation can often be useful, but in some situations it may be more effective to consider overall task semantics in defining the compensation. For example, a 'cooking' task might involve obtaining, preparing, and mixing different ingredients. Once the food is mixed, a compositional approach to compensation makes little sense. A sensible compensation of these activities addresses the task as a whole and depends upon context and application goals. If the task was cancelled for external reasons (a catering job cancelled), then the prepared food might be used elsewhere. If the task has failed due to inability to obtain an ingredient, then it may be possible to use the already-mixed ingredients as a base for another dish. The greater the extent to which compensation definitions for non-leaf goals are defined– where these definitions may be conditional– the greater the use of *semantic* compensation in that context.

There are three aspects to supporting this matrix of problem-handling behaviours: defining a semantics for *handling goal status events*, defining a set of domain-independent rules that generate new goal status events under various circumstances, and defining a set of rules to manage run-time expansion of compensation goals. We first outline the approach with respect to the first two aspects, then sketch how it is realised as sets of plan rules employed by a 3APL-like agent. Fig. 14 summarises the information flow that supports our methodology.

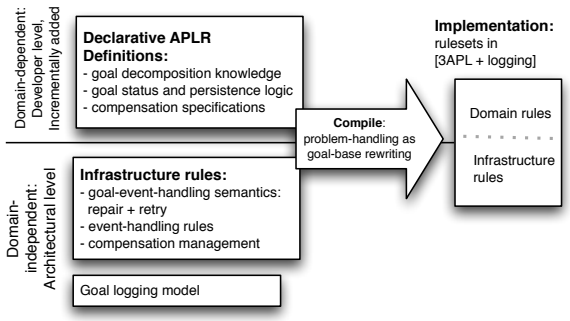

Fig. 14. An overview of the information flow for our problem-handling methodology. Additions to the APLR specification trigger domain rule recompilation.

5.1. Goal Status Events

Section 4.1 described the declarative information APLR associates with a goal instance: status and persistence information, encoded via *goal event actions*. We define a *goal status event* as a change in status information for a goal instance[9]. Success or failure

[9]Note that this use of 'event' is different than that of e.g. a dMars event, since here these are not domain triggering events (which are

1. **Terminate work on abstract goals (only) in response to goal status events**. Termination of an abstract goal recursively terminates its children, halting those that are under execution.

2. For a **selected** goal status event E on task G:

```
If there is user-defined knowledge for handling E in that context
 then { invoke the defined task tree modification }
   else: {
    if repair==true for G then {
     if G was current then {
      replace G with an instance of
      a compensation goal for that task: comp(G). }
     else { // task was already terminated
      create a new (root) tree for the compensation comp(G)
      }
    }
 if persist==local for G then {
    create a new instance of the goal,
    inserted serially after comp(G).  }
 }
```

Fig. 15. The goal-event-handling algorithm which supports RCPH. '*Replace*' means: the new goal(s) are 'inline' replacements for the goal that got the event, with respect to its task decomposition and sibling relationships. The replaced task (and its children) are no longer active and are removed from the task tree. The compensation goal `comp()` goal was defined in Section 4.2.

events are generated for leaf goals as the result of execution. Additionally, via goal status rules, goal status events may be generated for current goal instances at *any level in the task tree*, as well as for *previously-executed* goals that are part of a current high-level task but are no longer current. (A completed subgoal may later be detected as failed due to new information; logging is required to detect such situations). Thus goal status rules are a form of *monitoring*.

We define a goal status event semantics that specifies how the agent responds to a given event, using *remove, repair*, and *retry* responses, shown in Fig. 15. 'Remove' is removal from the goal base. 'Repair' is realised as semantic compensation, allowing response to partially- as well as fully-achieved goals, and both cancellation and failure. 'Retry' means to re-attempt a new instance of the goal (perhaps via a different decomposition). While not covered in this paper, APLR as well as its underlying event-handling algorithm allow the definition of user-defined handling rules to override the application of the RCPH model. Thus the repair+retry model is not invoked if more specialized event-handling knowledge overrides it for a specific situation. Figure 16 shows an example task tree modification process (temporal relationships between sibling tasks are not shown). Goal event *selection* is discussed below. The algorithm of Figure 15 may be applied in a uniform manner to all parts of the goal tree, *allowing repair activities within the context of a compensation*.

The goal-event-handling model enforces *persistence of a goal until explicitly dropped*, and enforces *repair prior to a reattempt*[10]. Realisation of goal-event-handling in a multi-agent context requires the use of interaction protocols similar to those in [20].

5.1.1. Goal-Event-Driven Problem-handling

Leaf goal execution provides 'baseline' goal event generation. When a leaf goal executes with success, '`persist=false`' is set for it atomically with its execution. If a leaf goal executes with failure, '`persist=local`' is set. We build on the goal-event-handling semantics to define a domain-independent set of default rules, shown in Fig. 17, that describe when to 'escalate' problems by generating new goal events. These rules can match against the log, and do not distinguish goal events generated *during* repair from those generated during 'normal' problem-solving (thus repair in the context of a compensation goal is treated in a unified way), nor between current and 'finished' goals (logging supports compensation of terminated tasks).

The default rules of Fig. 17 thus provide the architectural-level foundation for the basic problem-handling behaviours of Fig. 4. Then, user-defined domain-dependent knowledge can *refine* this behaviour on a goal or goal-decomposition basis by: specifying conditions under which to *cancel a current or already-executed goal*; under which a goal *succeeds or fails*;

supported in the APLR language via rule guard conditions) but a canonical set of architectural-level goal events.

[10] While it is beyond the scope of this paper, the persistence model allows modelling of *maintenance* goals as well).

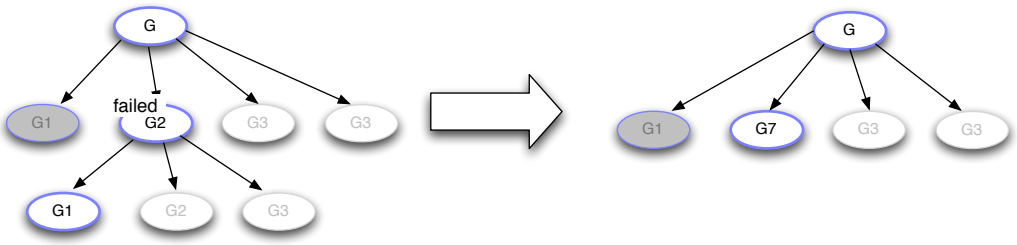

Fig. 16. Example of goal-event-triggered task tree modification in RCPH.

- *Default success*: IF all child tasks of a parent task P have achieved success, ⇒ apply the `success` action to the task P.

- *Failure by timeout*: IF a task T is still under execution (still current) I time units after initiation, ⇒ apply the `fail` action to T.

- *Escalation*: IF a task T has persist==escalate, and compensation for T has finished, ⇒ apply the `escalation` action to `LCA(T)`.

Determine escalation after multiple attempts at achievement:
- IF `persist==local` for a task T, and have a history of N retries with previous instances of that goal, and T failed, and if `escalate_p` is true, ⇒ apply the `set_escalate` action to T.
- IF `persist==local` for a task T, and have a history of N retries with previous instances of that goal, and T has failed, and if `escalate_p` is false, ⇒ apply the `unset_persist` action to T.

Fig. 17. The set of goal status event rules, in pseudo-code, which support RCPH.

and under which to *'push' handling of a goal to its parent goal*; and defining goal *timeout intervals*.

Domain knowledge determines the goal level at which a goal event originally occurs, and the nature of its repair. As a developer adds more knowledge about goal status detection and compensations, the problem-handling increases in sophistication along the dimensions shown in Fig. 13.

5.2. *BDIH Implementation: Goal Base Revision with Logging*

In this section we outline how the problem-handling methodology above, in conjunction with APLR-specified domain knowledge, is mapped to sets of 3APL-like *plan rules* [11]. (3APL plan rules have the form: `<head> :- <guard>| <body>`, where the head of the rule is an expression that matches a goal base pattern, the guard determines rule eligibility, and the body replaces the head in the goal base). The mapping results in a 'compilation' of both domain-dependent and -independent rules, as was suggested in Fig. 14. The resultant plan sets provide a semantics for the APLR syntax and the methodology. Space does not permit a detailed description, but we sketch our approach.

First, to support our problem-handling methodology, the agent must log information about goals and their relationships. For example, it must record '*compensation_of*' and '*retry_of*' relationships between goals, and log goal start and end times. To do this, the agent must support two capabilities. It must be able to represent *goal instances* and test for them in rule guard conditions (goal instances are implicit in APLR, but must be made explicit in the plan rules that implement it); and goal base revision and action execution must occur *atomically* with maintenance of execution *history*, or logging of that activity. As mentioned previously, we refer to the architecture that supports this extended set of behaviours as a *'BDIH'* architecture, to emphasize the role played by maintenance of execution history, for this general class of robustness algorithms.

Goal status rules are modelled as *reactive* rules, while selection of goal status event(s) to handle is modelled as a *deliberative*, not reactive activity. This allows us to cast goal status event handling as *domain-independent goal base revision*. Repair rule 'heads' match against goal patterns (both current and previously-executed), and the guard conditions of these rules

test for goal status events. The rule bodies revise the goal base according to the matched event. For example, a failure event makes eligible a rule that removes the failed goal from the goal base, then adds a `comp(GID)` goal (where GID refers to the failed goal instance), followed by a new instance of the failed goal (a retry).

We have extended the 3APL plan rule syntax and execution semantics to support these capabilities. The rules perform *logging actions* atomically with goal base modification, and provide support for matching against and managing goal instances. Our BDIH implementation is written in the Jess rule language [7].

We claim that the extensions required to support our methodology result in a more robust agent architecture by providing a consistent infrastructure-level mechanism for execution logging, necessary for many classes of agent recovery models.

The implementation supports concurrent subgoal execution, but for simplicity our examples have included only sequential decomposition. The agent's deliberation cycle, shown in Fig. 18, extends a 'canonical' cycle with additional rule precedence ordering and an explicit distinction of of meta-goals. RCPH models meta-goals specifically for *goal status event selection*, but this capability more generally allows any type of meta-reasoning to be made explicit.

Meta-reasoning about goal status event selection is required to support RCPH, particularly in a multi-agent context with delegated tasks. Goal status event rules– both the RCPH ruleset and the rules defined by the developer via APLR– monitor the status of specific goal instances, with events generated on status changes. It is possible for events to be detected on more than one goal instance. Thus, given multiple goal events, the agent must decide which subset of these events it wants to *select* to handle first as in Fig. 15. There is no way to ensure that detection of the goal events among which the agent should choose, will all occur in the same deliberation cycle. For example, consider a multi-agent situation where message latency is a factor. Thus, event selection itself must be a deliberative process, for which a goal is posted. This goal is a *meta-goal*, which is given priority in the deliberation cycle over domain application goals. When goal event selection is thus treated deliberatively, this allows the agent to e.g., reason about expected windows of time in which related goal events will be reported.

In a thread separate to the deliberation cycle, the agent asynchronously senses changes in its environment, and the implementation supports non-monotonicity in goal status events based on environmental changes. The method and implementation have met our expectations of utility in several test domains, including a 'hospital' domain like that described here.

6. Related Work

Earlier versions of 3APL supported the concept of failure rules [10], a rule type with high precedence in the deliberation cycle. With these domain-specific rules, goal status changes were not represented declaratively, and all failure rules were applied reactively and 'at once' in the deliberation cycle. We take a different approach by treating goal status events declaratively. This supports more robust behaviour by allowing deliberative selection of which events to handle, and by factoring domain-independent goal base modification rules from compensation knowledge.

Section 5 discussed how our problem-handling approach can be viewed as subsuming an agent exception-handling behaviour. Other approaches to exception-handling encode handler logic within separate monitoring/sentinel agents, e.g. [14]. In our approach, while we decouple the problem-handling model from the agent's application-level knowledge, its domain logic is leveraged to implement repair.

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 [3] 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. [19] take a related approach in an agent context. However, our approach doesn't require explicit definition of separate handler methods, can generate and handle events for goals not under current execution, and operates at a different level of granularity.

The SPARK [15] 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 recovery, their action model and task expressions are complementary to the approach described here. Eiter et al.

```
1. Apply all instantiated goal creation ('empty head') rules.  These will create new goals.

2. Apply all applicable reactive rules, including those for detecting goal status events and
   propagating goal state change.
   [Implementation must handle the nonmonotonicity of goal instance status, which may
    change over time]

3. Find eligible deliberative rule instantiations.

4. Select one instantiated rule to apply.
   Prefer rules that match a meta-goal. (Event-selection goals are meta-goals.)
   Next, prefer goal-event-handling rules. [These rules only apply to SELECTED goal events].
   Of these rules, prefer user-defined rules to RCPH default rules.
   Lastly, prefer plan refinement and action execution rules.
   (A modification of this algorithm prefers compensation goal refinements over
    'normative' refinements.)

5. Apply the selected rule.  Execution of a basic goal action terminates the matched
   goal instance with status information.
```

Fig. 18. A deliberation engine model to support RCPH. The agent's deliberation cycle extends a 'canonical' cycle with additional rule precedence ordering and explicitly distinguishes meta-goals. While this paper has not discussed the definition of user-defined event-handling knowledge (which may override the RCPH model), this deliberation cycle accommodates its use. The agent's *sensing* cycle occurs asynchronously. (Sensing incorporates information about domain events, including messages, into the agent's belief base).

[6] 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. 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 problem-handling model we describe in this paper can be viewed as falling into the same class of 'plan repair' approaches: the use of compensation/reversal is employed as a search control heuristic over the plan repair space. In Nagi et al. [16], 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.

Work in the programming language community has examined more sophisticated strategies than what we propose for the default retry, which carries out N attempts. An example of a 'smarter' strategy is work in [9], which attempts a retry only after some change has occurred to the information in question. There is also a growing body from the Web service community (e.g. [13,2]) also addressing this aspect, incoporating it within an infrastructure. One problem with this kind of 'some-change' approach is that it assumes that the agent can explicitly detect and model those aspects of the environment that caused the failure, which not will not always be the case for agents that interact with the environment. A simple-counter example is - an agent has trouble making a phone or network connection. It may not be able to detect that the network is back up, independently from trying again to connect and seeing if it is successful. On the other hand, there are scenarios where the relevant change in environment is internally reflected and such an approach ('wait until something happens') would make more sense.

7. Discussion and Conclusions

A theme underlying the methodology of this paper has been to leverage the use of logging, and composition of modularly-defined task-decomposition-structured knowledge, in support of run-time recovery for robust behaviour. The foundations that support RCHP can also support other aspects of robustness.

Of course logging is only one aspect of robustness and recovery and in some situations it may be be preferable to do more than the equivalent of a 'roll-back'. Following this direction, in [22] we suggest that criteria for crash recovery to an *acceptable* rather than consistent state sometimes has more utility in an agent context, and describe 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.

Because faults can lead to an agent entering a dangerous or incoherent state, it is often useful that recovery first focuses on steps to *stabilize* the system by moving it to a known, safe state. In this paper, we did not discuss explicit *stabilization* as a part of the RCPH recovery model distinct from compensation. Stabilization can often be productively viewed as a local activity, applied recursively in a bottom-up manner as tasks are halted upon receipt of a termination event, prior to (possibly top-down) compensation. Thus, RCPH's task-decomposition-structured recovery model can support this approach to stabilization in an integrated manner. To do so, APLR is extended to allow specification of stabilization as well as compensation knowledge.

A powerful aspect of the approach described in this paper is that the execution log allows the agent to *address problems with completed tasks* whose effects later need to be compensated for. Goals introduced to compensate for a problem are logged as any other goal, and compensations of compensations are supported transparently. That is, a compensation goal has distinct heuristics used in generating its subgoals, but its execution generates a trace which can be treated as any other decomposition. In its ability to compensate– and to refine compensations– after goal execution, the model addresses some of the organisational aspects of an agent system. For example, a bank may realise that it has overcharged customers, later requiring reimbursement.

The repair and compensation model of RCPH, as well as APLR and its prototype implementation, support concurrent as well as serial task decomposition. The model described in this paper does not address framework-level semantics for concurrency management. That is, concurrent execution is supported by RCPH, but the knowledge to avoid problematic interactions between concurrently-executing tasks must be added programmatically.

However, our current research explores this important aspect of robustness. In particular, we can leverage the modularly-defined, task-decomposition-structured nature of RCPH, and the logging required to support it, to support an isolation semantics integrated with the RCPH problem-handling model. (Sub)task isolation characteristics may be defined compositionally and dynamically, where each task decomposition specification defines the isolation characteristics of the subtasks in that decomposition and creates its isolation context. Compensation task isolation semantics are based on the 'forward' task being compensated; compensations are performed within the isolation context of the forward task, and for compositional compensations, compensation subtask isolation semantics depends upon forward subtask isolation characteristics.

As described in [23], in related research, we are also developing the foundational concepts underlying RCPH in the context of a more 'closed' agent system model. This model, called ARTS, utilises a shared execution context, where problem-handling knowledge is supported by specification of nested exception handlers, and the task interface is made more formal by explicitly requiring specification of post-conditions and maintenance conditions as well as guard (pre-) conditions.

In this paper we have described how infrastructure-level logging and the provision of declarative information about goals can support agent recovery from run-time problems in BDI agents. We have argued that the use of an execution history is crucial in building robust agents, and that it should be supported at the architectural level. Thus we claim that the *BDIH* extensions required to support our methodology result in a more robust agent architecture independent of RCPH by providing a consistent infrastructure-level mechanism for execution logging, necessary for many classes of agent recovery models.

We have defined a developer-level language, APLR, to support specification of declarative problem-handling information, and insulate and constrain the developer from the infrastructure-level reasoning. Knowledge can be added incrementally to APLR, with the agents' problem-handling behaviour increasing in sophistication as it is added. We map the information encoded in APLR, in conjunction with the 'framework-level' problem-handling rules, to an implementation in the form of 3APL-like plan rule sets. Our implementation extends plan revision rules to support atomic logging actions and representation of goal instances in support of robustness.

The approach we describe is a default method that may be overridden by domain-specific 'plan patching' knowledge if available. Its use of compensation allows it to treat repair for both failure and cancellation in a unified way, and allows higher-level compensations to leverage lower-level ones. Using compensation, many run-time problems can be stabilised without explicit (inter-)action models or identification of the specific cause of a problem.

Our approach, with its restriction on modular specification of problem-handling information, can always be applied in an open agent system without making assumptions about visibility of information across agents. While not discussed here, problem-handling interaction protocols are additionally required to realise the method in a distributed multi-agent scenario [20]. Future work includes the integration of protocols for multi-agent support with our 3APL-based implementation. Other plans include incorporation of underlying transactional support for the logging mechanisms (via Jess' interface with a RDBMS); development of a simulation testbed for comparisons in further application domains; integration of semantic compensation with the use of task rollbacks when possible; and further development of a model which integrates run-time and crash recovery.

References

[1] 3APL group. 3APL BNF specification. `http://www.cs.uu.nl/3apl/bnf.html`.

[2] Rosa Gutierrez andMichael Huhns. On building robust web service-based applications. In *Utilizing Web Services in an Agent Based Transaction Model*, pages 293–310. Springer, 2004.

[3] 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.

[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] Mark D'Inverno, Michael Luck, Michael Georgeff, David Kinny, and Michael Wooldridge. The dMARS architecture: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems*, 9(1 - 2):5–53, 2004.

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

[7] Ernest Friedman-Hill. *Jess in Action*. Manning Publications Company, 2003.

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

[9] Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, 2005.

[10] Koen Hindriks, Frank de Boer, Wiebe van der Hoek, and John-Jules Meyer. Failure, monitoring and recovery in the agent language 3APL. In De Giacomo Giuseppe, editor, *AAAI 1998 Fall Symposium on Cognitive Robotics*, pages 68–75, 1998.

[11] Koen V. Hindriks, Frank S. De Boer, Wiebe Van der Hoek, and John-Jules Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[12] Nick Howden, Ralph Ronnquist, Andrew Hodgson, and Andrew Lucas. Jack $\tilde{\text{N}}$ summary of an agent infrastructure. In *5th International Conference on Autonomous Agents*, 2001.

[13] Tao Jin and Steve Goschnick. Utilizing web services in an agent based transaction model. In *Extending Web Services Technologies*, pages 273–291. Springer, 2004.

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

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

[16] 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.

[17] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In Third International Conference on Principles of Knowledge Representation and Reasoning. Morgan Kaufmann, 1992.

[18] Birna van Riemsdijk, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming in dribble: from beliefs to goals using plans. In *AAMAS*, pages 393–400, 2003.

[19] 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.

[20] 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.

[21] 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.*, 2005.

[22] A. Unruh, H. Harjadi, J. Bailey, and K. Ramamohanarao. Compensation-based recovery management in multi-agent systems. In *Second IEEE Symposium on Multi-Agent Security and Survivability*, Philadelpha, 2005.

[23] Mingzhong Wang, Amy Unruh, and Kotagiri Ramamohanarao. ARTS: Agent-oriented robust transactional system. In *AAMAS-07*, 2007.

[24] M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative and procedural goals in intelligent agent systems. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, 2002.