ARTS: Agent-oriented Robust Transactional System *

Mingzhong Wang Amy Unruh Kotagiri Ramamohanarao
Dept. of Computer Science and Software Engineering
The University of Melbourne
VIC 3010, Australia
{minwang, unruh, rao}@csse.unimelb.edu.au

ABSTRACT

This paper presents the ARTS (Agent-oriented Robust Transactional System) model, which applies transaction concepts to provide agent developers with high-level support for agent system robustness and reliability. ARTS abstractly considers agents as executors of encapsulated task entities which comply with a set of execution constraints on both normative execution and compensation (repair) semantics. ARTS then defines the task interface in terms of predictable terminating states to support a contract-like interaction among agents. In conjunction with this encapsulation of task semantics, ARTS defines a model for specifying scoped compensation and exception-handling plans for a given task, and for systematically selecting and executing these plans — triggered by subtask events — so that the enclosing task semantics are enforced. These capabilities together define a model that reduces design complexity while increasing system robustness, by allowing an agent developer to compose recursively-defined, atomically-handled tasks.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*multiagent systems, languages and structures*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Programming by contract, Reliability*; D.2.11 [**Software Engineering**]: Software Architectures—*Languages*

General Terms

Design, Languages, Reliability, Theory

Keywords

Agent programming language, transaction, robustness

1. INTRODUCTION

Although current research in agent architectures and programming languages has made great strides, the objective of a powerful yet easy-to-use mechanism for maintaining system robustness and reliability, with respect to the correct execution of agent actions even in the presence of abnormalities, remains unfulfilled. As a result, while some aspects of an agent language may be high-level, developers still must consider low-level details of disturbances, failure, or uncontrolled interactions between the agents in a system.

Addressing the demands from agent developers, we propose the Agent-oriented Robust Transactional System (ARTS) model [6], which defines a concise hierarchical model for the composition of distributed agents. By encapsulating compensation semantics, defining contract-like cooperations and hierarchically organizing exceptions, ARTS allows an agent developer to compose recursively-defined, atomically-treated tasks, and to consider exception-handling at a higher-level of abstraction. It ensures modular agent system design and reduces the coupling among agents. Thus, the complexity of dealing with combining possible termination states of all participating agents is greatly reduced.

*We dedicate this paper to Jim Gray for his pioneering work in the field of transaction processing.

AAMAS'07 May 14–18 2007, Honolulu, Hawai'i, USA.

2. ABSTRACT AGENT MODEL IN ARTS

For the purposes of the approach described here, we can assume without loss of generality a task decomposition model where each agent is programmed to achieve a task T, with T's subtasks delegated to sub-agents. Thus, we use *Task* and *Agent* interchangeably.

2.1 Modelling a Single Agent

We first describe our agent model from the perspective of the interface it exposes to the other agents in the system. That is, we describe its behaviour as an encapsulated entity — a *single* agent — whose implementation details are not visible.

Fig. 1 shows the interface of a single agent. To achieve a certain goal, the agent has a set of specifications in the form of constraints labelled *P*, *M* and *Q*.

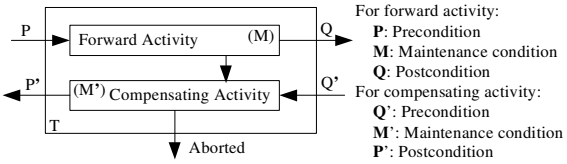


Figure 1: Interface of Single Agent

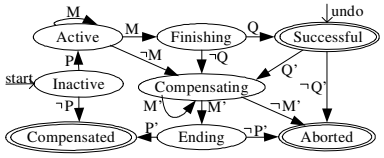
Exogenous events, unanticipated interactions between agents, or non-deterministic action results, may require plan repair. If each task in the system can encapsulate a definition of what a successful "undo" / "compensation" [4] of the task entails, then this knowledge provides a powerful means for enforcing and supporting task *failure atomicity*. Thus, we also define a set of constraints on T for compensation, called Q', M' and P'.

978-81-904262-7-5 (RPS)

Formally, T can be denoted by $T =< B, \sigma, C, \delta, I, D >$.

- B stands for the beliefs which reflect the perception of T about its working environment. B evolves with the execution of T and can be denoted as a sequence of state changes.
- σ is the possible execution states of T. $\sigma \in \{$*Inactive, Active, Finishing, Successful, Compensating, Ending, Compensated, Aborted*$\}$. Predicate $\sigma(T)$ is true if the state of T is σ.
- C stands for the constraints on T, denoting P, Q, M and P', Q', M'. Each kind of constraint needs to be satisfied by beliefs during the corresponding execution state.
- $\delta : \sigma \times C \times B \rightarrow \sigma$ is the state transition function of T.
- I stands for the interface to control the behaviour of T. It can require T to either *start* or *undo*.
- D is a set of dependencies specifying connections between each participating agent. For a single agent, D is empty.

δ is a key part of the definition above. Fig. 2 defines the state transition graph which enforces the agent's interface in Fig. 1. With the transition predicates, we make the closed-world assumption with negation as failure.



Inactive: has not been required to execute
Active: is executing its normative work
Successful: has achieved Q
Compensated: has achieved P'

Compensating: is performing a semantic compensation of its work
Ending: has finished its compensating work, but has not yet verified that it has achieved the compensation conditions P'
Finishing: has done all its work, but has not yet verified that it has achieved the conditions Q
Aborted: does not have an allowable action to take next, but has not achieved Q or P'

Figure 2: State Transition Graph for a Single Agent

2.2 Modelling Agent Composition

Agents in a distributed system often cooperate with each other to achieve a goal. Often this is done by breaking down a higher-level goal into subgoals, creating a hierarchical task structure. Here, we primarily discuss the relationship between the parent and its children in such a hierarchy.

In viewing an agent as a building block, we can construct hierarchical structures to model complex tasks. A composite parent agent shares the same form as a single agent with three extensions.

State Vector: Because the composite agent contains more than one child agent, and each child has its own state, we use σ_p to represent the state of the parent, and $\sigma_{c1}, \sigma_{c2}, \ldots$ to represent the states of its children respectively. Thus, instead of σ, $\Sigma =< \sigma_p, \sigma_{c1}, \sigma_{c2}, \ldots >$ is a vector recording states of all participating agents.

State Transition Function: Since Σ is a vector, we redefine δ as $\delta : \sigma_p \times C \times B \rightarrow \sigma_p$ to keep the state transition graph unchanged for the parent T. Then we model the relationship between the children and parent by adding the required outcome conditions of the children to the Maintenance Conditions of T. Thus, the complexity of extending δ according to Σ is eliminated, and both the composite agent and atomic agent have unified state transition settings.

Dependency Set D: The dependencies effectively encode the task execution paths, as defined by the task's business logic and exception handling knowledge in conjunction with ARTS' methodology for organizing and using this knowledge. There has been much previous research towards task dependency management [1, 3]. However, our proposal merges dependencies into a concise programming setting and provides them with an operational semantics to guarantee robustness and reliability.

Reconciliation of Single and Composite Agent Perspectives. Although two kinds of agent models, a *single agent* and a *composite agent*, have been discussed above, they map to the same underlying model. Whether an agent is viewed as single, or composed of other agents, depends upon the abstraction level at which it is considered. If we need to access an agent's internal organization, we model it as compositional. If we just need to use its functionality as a building block, we model it as a single agent.

3. THE ARTS MODEL

ARTS has two primary aspects: task knowledge — *procedures*— specified by the agent developer, and the execution platform supporting them. This section first describes the specification of the procedures, then gives a brief overview of the dependency management provided by the platform.

3.1 Exception-Handling Knowledge in ARTS

The programs in ARTS are a set of procedures indicating how to react to the events produced by each agent. As will be detailed below, the procedures have nested scope, so that each plan to handle an exception can be associated with its own set of exception-handling plans specific to that context. Exception-handling plans may encode both *compensation* and *retry* semantics: the developer determines the semantics appropriate to the situation.

The operating concepts in ARTS are defined using EBNF-like notation. Besides standard connectives from first-order logics, three new connectives are used: " ; " for Sequential composition, " || " for Parallel composition and " , " for Selective composition.

Definition 1. An atomic event is signalled directly by the changes of T's state. ϵ stands for the *null* event, which always evaluates to be true. T_i is the ID of a child agent.

$$
\begin{aligned}
< atomicEvent > ::= \ & \text{“}\epsilon\text{”} \mid \text{“}\neg M\text{”} \mid \text{“}\neg M'\text{”} \mid \text{“}\neg Q\text{”} \mid \\
& \text{“}Successful(T_i)\text{”} \mid \text{“}Compensated(T_i)\text{”} \mid \\
& \text{“}Aborted(T_i)\text{”} \\
< event > ::= \ & < atomicEvent > \mid \\
& < event > \text{ “}\wedge\text{” } < event > \\
< Event > ::= \ & < event > \mid \text{“}start(T)\text{”} \mid \text{“}undo(T)\text{”}
\end{aligned}
$$

In response to a given event, the parent will compose its children to form reactive plans that address the event. Sequential composition means that each component will be executed one after the other. Parallel composition means that all components will be launched together, and selective composition means that alternative choices will be tried one after the other until one succeeds.

It is important to note that from the view of T, "$start(T)$" and "$undo(T)$" are Events because they are generated externally, while "$start(T_i)$" and "$undo(T_i)$" are atomic tasks because they are used by T to compose task structures.

Definition 2. Task composition serves to organize a group of single agents according to their functionalities for achieving a certain goal (task Postcondition). The abbreviated notation T_i is used for $start(T_i)$, which means to make the agent T_i run.

$$
\begin{aligned}
< atomicTask > ::= \ & \text{“}\epsilon\text{”} \mid \text{“}T_i\text{”} \mid \text{“}undo(T_i)\text{”} \\
< Task > ::= \ & < atomicTask > \mid \\
& < sequenceComposition > \mid \\
& < parallelComposition > \mid \\
& < selectionComposition > \\
< sequenceComposition > ::= \ & < Task > \text{ “;” } < Task > \\
< parallelComposition > ::= \ & < Task > \text{ “||” } < Task > \\
< selectionComposition > ::= \ & < Task > \text{ “,” } < Task >
\end{aligned}
$$

ϵ is a special agent in the system whose pre-, maintenance and post- conditions are always satisfied. Although it does nothing and

always succeeds, it acts as a bridge to connect unrelated branches in Parallel or Selection composition. In practice, we assign it a name with the same form of T_i without treating it differently.

A plan describes the process of how to react to a certain Event, according to domain knowledge. The keyword "continue" means to resume the parent task execution.

Definition 3. A plan specifies a composition of a group of agents to react to an event.

$$< Plan > ::= < Event > \text{ “} \leftarrow \text{ ”} < Task > [\text{“;”“}continue\text{”}]$$

As we show, agents are organized in a hierarchical structure inside a plan to handle a given event. In practice, we need different plans to deal with events situated in different contexts. Thus, we define a hierarchy to encode the context of event handling and confine the effective scope of each plan.

Definition 4. A procedure contains a plan, as well as a set of amendment procedures to deal with exceptions within the scope of the plan.

$$< Procedure > ::= (\text{“}\epsilon\text{”} \mid < Plan >) \text{“.”} < amendmentProcedures >$$

$$< amendmentProcedures > ::= \text{“\{”} < Procedure > (\text{“ :: ”} < Procedure >)^{*} \text{“\}”}$$

Each procedure consists of two parts: a *plan part* to define the purpose and behaviour of a procedure, and an *amendment part* to deal with exceptions that might occur in its plan part. This definition allows organization of the plans into a tree structure: a plan is only effective in the scope of its parent plan, which provides the execution context for it. The concept of *plan tree* decomposes and organizes the flat structure among normative and compensation executions for T into different levels, and provides the benefits of modularity, manageability and scalability.

In Fig. 3, we use a scenario for a Surgery Manager, whose responsibility is to manage and coordinate all activities related to a patient's surgery in hospital, to illustrate the ARTS model.

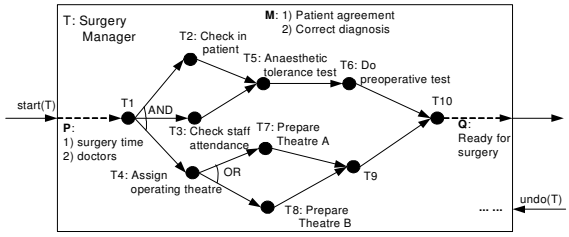


Figure 3: A Surgery Manager Agent in a Hospital Environment

On the scheduled day, the Surgery Manager will ask participating agents to check in the patient (T_2) and check for the attendance of the assigned staff (T_3) before the specified time for preparation. It will also ask for one available operating theatre prepared before the surgery time (T_4, T_7 and T_8). There are also some other tasks required, such as checking tolerance for anaesthetic drugs (T_5) and other preoperative tests (T_6). The process above works well if the surgery manager can fully control each participant, but contingencies may occur at any point due to unpredictable sub-system behaviour. Thus, we define the procedure in Fig. 4 to express part of the knowledge of the Surgery Manager. We introduce three new agents. T_{11} will inform the system administrator that something uncontrollable has happened; T_{12} is the staff scheduler which assigns a new surgeon; and T_{15} deals with financial affairs.

3.2 Execution of Plans

A set of inference rules can be derived from the state transition graph (Fig. 2). They guide and constrain the automatic execution

```
start(T) ← T1; ( ( (T2 || T3); T5; T6 ) || ( T4; (T7 , T8); T9 ) ); T10. {
    Compensated(T6) ← undo(T2) || undo(T3) || undo(T4). {
        Aborted(T4) ← T11. { }
    } ::
    Compensated(T3) ← T12; continue. {
        Aborted(T12) ← undo(T2) || undo(T4). { }
    }
}
undo(T) ← ( undo(T4) || undo(T2) || undo(T3) ) ; start(T15).
```

Figure 4: A Procedure for the Surgery Manager Agent

and compensation of the agents. Within this framework, we add two additional types of inference rules to handle the context switch caused by termination of a plan.

If the termination is caused by an exception during the plan execution, a corresponding child plan should be triggered to run. If no child plan is found, the parent transits to the *Aborted* state. The execution state will become *Compensating* if the identified plan has no "continue" tag.

If the termination is caused by the completion of a root *Plan* of a given plan tree, the agent will check if the goal is achieved and transit to $Successful$ or $Compensated$ accordingly. If not, it looks for other root plans to continue, or enters $Compensating$ because of unachievable goals. Otherwise, the agent will pass back control to its parent if the plan ends with "continue"; or transit to the *Ending* state for further verification of goals.

4. DISCUSSION AND CONCLUSION

ARTS extends the previous work [2] which applied the *nested transaction model* with a more flexible and well-organized agent recovery structure from a programming and software engineering perspective. Compared with [5], which describes a compensation-based problem-handling methodology based on logging, ARTS focuses on agents composition as well their interrelationships. Its hierarchical organization model for agents and their exception handling knowledge provides developers with a powerful abstraction for modeling exception events and for encoding knowledge of exception handling plans and the way that these plans relate to each other. Together with other featured abstraction provided in ARTS, such as contract-like interactions and semantic compensation, an agent developer can compose recursively-defined, atomically-handled tasks to build complex systems with reduced design complexity and increased system robustness and reliability.

5. REFERENCES

[1] P. C. Attie, M. P. Singh, A. P. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In R. Agrawal, S. Baker, and D. A. Bell, editors, *VLDB*, pages 134–145. Morgan Kaufmann, 1993.

[2] P. Busetta, J. Bailey, and K. Ramamohanarao. A reliable computational model for BDI agents. In *1st International Workshop on Safe Agents, held in conjunction with AAMAS'03*, 2003.

[3] P. K. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Trans. Database Syst.*, 19(3):450–491, 1994.

[4] A. Unruh, J. Bailey, and K. Ramamohanarao. Managing semantic compensation in a multi-agent system. In *COOPIS '04 (LNCS 3290), 2004.*, pages 245–263, Agia Napa, 2004.

[5] A. Unruh, J. Bailey, and K. Ramamohanarao. A logging-based approach for building more robust multi-agent systems. In *IAT '06*, pages 342–349, Washington, DC, USA, 2006. IEEE Computer Society.

[6] A more detailed description of ARTS can be found at `http://www.cs.mu.oz.au/~minwang/publications`.