Achieving System-Level and Agent-Level Dependability in Multi-Agent Systems

Otavio R. Silva
e-mail: otavio@inf.puc-rio.br

Alessandro F. Garcia
e-mail: afgarcia@inf.puc-rio.br

Carlos J. P. Lucena
e-mail: lucena@inf.puc-rio.br

PUC-RioInf.MCC09/01 May, 2001

Abstract

Development of multi-agent object-oriented systems is complicated by two main factors: the increasing complexity associated with their system-level and agent-level properties and the need to provide their dependability. This paper explores the recent advances of separation-of-concerns techniques to deal with the increasing complexity in developing dependable multi-agent object-oriented software. We propose a software architecture which: (i) uses aspect-oriented design and programming to handle separately each agency property and facilitate the provision of fault tolerance at the agent-level, and (ii) adopts a reflective associative blackboard architecture in order to manipulate separately each property and fault tolerance activities at the system-level. We also demonstrate our multi-agent approach through the Portalware system, a web-based environment for the development of e-commerce portals.
Keywords: Multi-agent systems, tuple spaces, fault tolerance, computational reflection, aspect-oriented programming

Resumo

O desenvolvimento de software multi-agente orientado a objetos é complicado devido não somente à crescente complexidade relativa às suas propriedades de agência e de sistema, mas também à necessidade de prover sua confiabilidade. Este artigo explora os recentes avanços nas técnicas de *separation of concerns* para lidar com as dificuldades intrínsecas ao desenvolvimento de software multi-agente. Propõe-se uma arquitetura que: (i) utiliza projeto e programação orientados a aspectos para tratar separadamente cada uma das propriedades de agência bem como o suporte a tolerância a falhas no nível do agente e (ii) adota uma arquitetura reflexiva de *blackboards* associativos visando tratar separadamente cada propriedade e atividades relativas à confiabilidade no nível do sistema. Nossa aboradagem para sistemas multi-agentes é demonstrada através do Portalware, um ambiente baseado na Web para o desenvolvimento de portais de comércio eletrônico.
Palavras-chave: Sistemas multi-agentes, *tuple spaces*, tolerância a falhas, reflexão computacional programação orientada a aspectos

1. Introduction

Development of multi-agent object-oriented systems is complicated by two main factors: the increasing complexity associated with their system-level and agent-level properties and the need to provide their dependability. Software agents, unlike objects, are driven by a set of internal agency properties, such as interaction, autonomy, adaptation, mobility, collaboration, and learning. Moreover, the different agents within a useful multi-agent application need system-level properties to move them from a host to another, to coordinate their activities, to communicate with each other, and to preserve their state. Each of these internal and system-level properties increases the software complexity, and introduces heterogeneous exceptional situations that must be handled properly. Dependable multi-agent OO software incorporates exception handling and fault tolerance activities to detect exceptional conditions and to restore normal computation. In fact, software fault tolerance is often necessary, but itself can be dangerously error-prone because of the additional effort that must be involved in the construction process. The additional software redundancy may increase the size and complexity and thus adversely affect software dependability [28].

In this context, there is a need for a software architecture that controls the complexity and promotes dependability in building multi-agent OO systems. This software architecture should: (i) support directly both agency properties and system-level properties to produce effective multi-agent software, (ii) describe structured integration of agency properties into the object model, (iii) encourage the separate handling of agent-level dependability and system-level dependability, and (iv) allow the construction of dependable multi-agent OO software so that it is easy to understand, maintain and reuse. Most current proposals typically focus on a reduced number of agency and system-level properties, and do not provide direct support for handling and reusing such properties separately (e.g. [1], [5] and [17]). Moreover, the internal architecture of an agent in such proposals is generally encapsulated as an object. Even though it is desirable for an agent to appear as a single object, this scheme results in agent design and implementation being quite poor, complex and difficult to understand, maintain and reuse in practice. In addition, it often is not easy to design software agents properly as the developers of multi-agent systems have to take into account many agency and system-level properties at the same time. It makes the handling of exceptions and the introduction of fault tolerance activities into the system harder to carry out. Ideally, agent system developers should apply special structuring techniques and disciplined ways of associating the different properties and fault tolerance activities at the system and agent-level.

Separation of concerns is one of the main tenets of software engineering using high-level abstractions to hide software complexity. This paper explores the recent advances of separation-of-concerns techniques [15,28] to deal with the increasing complexity in developing dependable multi-agent OO software. In this sense, this paper proposes a software architecture that promotes separation of agent concerns at two levels: (i) uses aspect-oriented design and programming [15] to handle separately each agency property and facilitate the provision of fault tolerance at the agent-level, and (ii) applies the notion of reflection [28] upon associative blackboard architectures [30,31] in order to manipulate separately each property and fault tolerance activities at the system-level. In our approach, the different agency properties and fault tolerance activities of an agent are encapsulated as aspects; agent developers can handle and reuse these aspects separately in order to compose different types of agents. Moreover, reactions in a reflective associative blackboard architecture are used to program system-level properties and fault tolerance activities. We also demonstrate our multi-agent approach through the Portalware system, a web-based environment for the development of e-commerce portals

The remainder of this paper is organized as follows. Section 2 provides brief definitions of multi-agent systems and software fault tolerance. This section also introduces an example that is used throughout this paper to illustrate our approach. Section 3 introduces aspect-oriented design and programming, and reflective associative blackboard architectures. Section 4 presents our software architecture for designing and implementing dependable multi-agent object-oriented applications. Section 5 discusses related work. Section 6 describes some implementation issues. Finally, Section 7 presents some concluding remarks and directions for future work.

2. Multi-Agent Software and Fault Tolerance

2.1. Agents and Multi-Agent Applications

The state of an agent is formalized by knowledge, and is expressed by mental components such as *beliefs*, *goals*, *capabilities* and *plans* [21, 25]. Beliefs model the external environment with which an agent interacts. A goal may be realized through different plans. Software agents carry out plans to achieve their internal goals, and the selection of plans is based on their beliefs. Plans select an agent's appropriate capabilities that could achieve their stated goal(s). There are different kinds of plans, and they are application-specific [13]. The behavior of an agent is composed of and affected by the incorporated *agency properties*. Agency properties are agent-level behavioral features that an agent can have to achieve its goals. Table 1 summarizes the definitions for the main agency properties. In general, autonomy, interaction and adaptation are considered as fundamental properties of software agents, while learning, mobility and collaboration are neither a necessary nor sufficient condition for *agenthood* [21]. There are several types of software agents, including information agents, user agents, and interface agents. Each agent type has different application-specific capabilities and agency properties. A software agent is not usually found completely alone in an application, but often forming an organization with other agents; this organization is called a *multi-agent application*.

Agency Property	Definition
Interaction	An agent communicates with the environment and other agents by means of sensors and effectors
Adaptation	An agent should rapidly adapt/modify its state and behavior according to new environmental conditions
Autonomy	An agent is capable of acting without direct external intervention; it has its own control thread and can accept or refuse a request
Mobility	An agent is able to transport itself from one environment in a network to another in order to achieve its goals
Learning	An agent can learn based on previous experience while reacting and interacting with its external environment
Collaboration	An agent can cooperate and negotiate with other agents in order to achieve its goals and the system's goals

Table 1. An Overview of Agency Properties

2.2. Issues in Multi-agent Systems

Many useful agent applications require some system-level features to facilitate their construction. A *multi-agent system* is an infrastructure that provides such features to the development of software agents and multi-agent applications. In this section, we discuss the main system-level features: (i) *mobility*, (ii) *coordination*, (iii) *communication*, and (iv) *persistence*. Note that even though some of these features appear as agent-level properties (e.g. mobility), the system itself should incorporate these features to assist the development of complex multi-agent applications.

Mobility. Providing a transparent protocol regarding to the mobility agency property should also be a system-level feature. Although the agents themselves decide when they want to be transferred from one host to another, it should be a system responsibility, granting their transference in an efficient and dependable way. Since mobile agents roam through different hosts over the network, the application that has created the agents does not have control of their existence. The agents may "die" while executing on a remote host, for example, if the host is powered off. This might not be interesting for the application that created the agents, as their state would be lost forever. The system should then enable some kind of agent

recovering mechanism. Providing access to local resources is also a feature that has to be supported by the system. As mobile agents run on different hosts with different kinds resources, if this feature was not supported it would be impossible to know which resources are available and what are their locations.

Coordination. Some agent types collaborate in order to fulfill their goals (Table 1). To ensure this cooperative behavior the system itself should enable some kind of coordination. Coordination is defined as the process of managing dependencies between activities [33]. Coordination is applied to multi-agent applications in order to avoid duplicate problem solving, and ensure that the actions of different agents are synchronized [34]. It is very difficult to guarantee the global coordination of a multi-agent application without an express global control. By this reason, there is a need for the system to provide support for coordination activities. Mobility makes coordination activities more difficult since mobile agents systems are often intrinsically dynamic. Agents can be created or cloned at runtime, making it difficult to identify the agents that belong to the system; these characteristics make a spatially coupled coordination protocol, where agents must explicitly identify by name any other communication colleague. In addition, it is not easy to predict the schedule and position of the mobile agents due to their dynamic and autonomy characteristics. In this way, using a meeting-oriented coordination protocol [35,36] does not suit well the mobile agent paradigm. In order to interact, this protocol forces agents to be at the same time in the same place (temporal coupling).

Communication. In spite of the collaboration property to be incorporated at the agent-level, the system itself should provide some kind of communication interface between the agents. This interface would act as a protocol to the information exchange among software agents, thus helping agent designers in the multi-agent application development. This communication protocol may be implemented using message handlers [17], blackboards [34], tuple spaces [27] or even direct method call [24]. The system should also provide transparent access to the communication interface, for example supplying some kind of API. The system must guarantee that a single agent will not monopolize broadcast information. In a blackboard communication interface, broadcasting a message means making it available in the space to be accessed by every agent or a group of agents. However, a malicious agent can then try to take the information out of the space, so that the others do not see the information.

Persistence. In a multi-agent system, agents can exist over long periods of time. They can be active or inactive during this time. While inactive, they are "put to sleep" and have their state saved in a persistent form [21]. In this way, the agent's state is conserved and restored once the agent becomes active. The agent itself has autonomy to decide whether it wants to become persistent or not. In the other hand, the system should support the management of the persistent state and ensure that the agent state is preserved and retrieved in a confident way. The system should also handle properly the messages addressed to agents that are sleeping.

2.3. Software Fault Tolerance for Multi-Agent Systems

Software Redundancy. Dependable software systems require supplementary techniques in order to tolerate the manifestations of faults in its components and, consequently, to avoid system failures [26]. These techniques are often based on redundancy. However, data replication is not usually sufficient. The recovery block is one of the most general techniques [37,38] to provide fault tolerance. This technique applies software diversity that is based on an extra diverse application code intended for error detecting and recovering. The recovery block scheme assumes that independent application programmers design a set of redundant versions of the application code. The acceptance test is used to check the correctness of the execution of these versions. This is essentially a dynamic scheme: versions are executed sequentially and if, upon completing a version, the acceptance test is ensured, the recovery block has been executed. Otherwise the system is rolled back and the next version is tried. Software agents have different versions of their internal plans and capabilities in order to overcome residual faults. In addition, the multi-agent system should apply data-replication in order to tolerate system-level faults.

Exception Handling. Exceptions and exception handling provide a desirable framework for structuring fault tolerance activities in dependable systems. Developers of dependable software can refer to faults as exceptions because they are rarely expected to manifest themselves during the system's normal activity. Exception handling systems allow software developers to define exceptions and to structure the exceptional activity of software components by means of handlers. A suitable handler is invoked when an exception is raised during the program execution. In the context of multi-agent systems, exceptions can be classified into two types: (i) *agent-level exceptions*, and (ii) *system-level exceptions*. Agent-level exceptions are associated with the internal capabilities and agency properties of software agents, which should include handlers to deal with them. Similarly, system-level exceptions are associated with system-level properties and should be handled by the system; when the system is not able to deal with these exceptional situations, it should signal them to the software agents.

2.4. Portalware Agents: A Case Study

Portalware [9] is a Web-based environment for the construction and management of e-commerce portals. Portalware encompasses three agent types (Figure 1): (i) information agents, (ii) interface agents, and (iii) user agents. In addition, broker agents are included to the system in order to allocate tasks to the Portalware's agents. Each of Portalware's agents has different capabilities and properties, but everyone implements the fundamental aspects defined by agenthood. For the purpose of brevity, we discuss in detail only the Portalware's information agents. For a more complete discussion about this example the reader can refer to [8]. Portalware users often need to search for information that is stored on different hosts. When a user requires a given information search specified by a keyword, a broker agent is responsible for assigning searching tasks to multiple information agents. The broker agent must know which information agents are working for him, enabling further search result requests. Each information agent contains plans for searching information. The search plan determines the agent's searching capability. If an information agent does not find the required keyword in the local host, it should move to another host. So, when an agent finds the keyword, it is important to notify every information agent performing the same search. In other words, the different information agents need to coordinate their activities in order to avoid redundant work. As soon as the information agents finish their tasks, they send the result to the broker agent and are put to sleep until another search request is made.

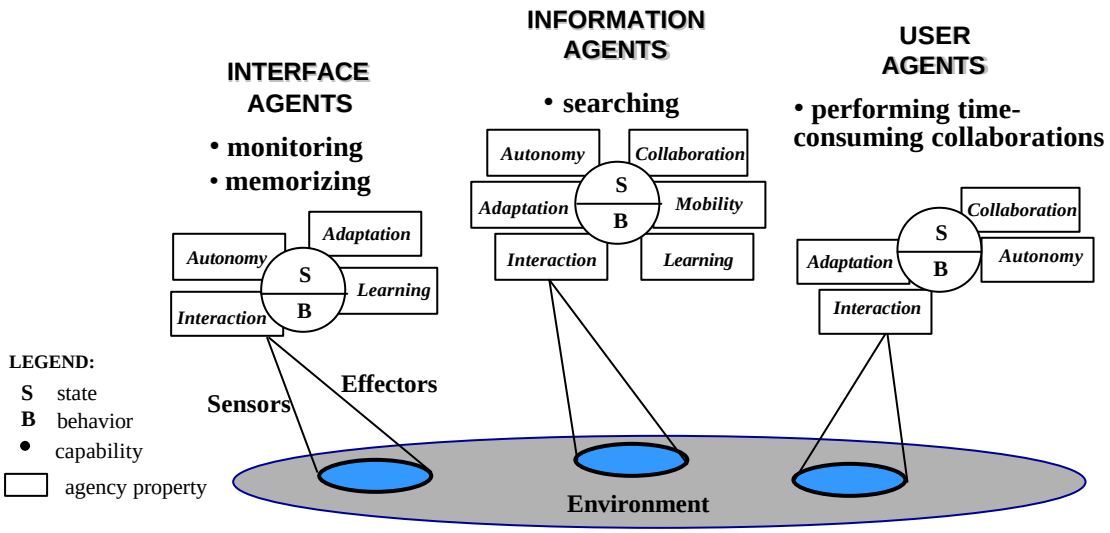

Figure 1. Portalware Agents and their Agency Properties

Table 2 describes some agent-level exceptions that can be raised during this scenario. Software agents include handlers in order to deal with the agent-level exceptions. In addition, different redundant versions of their plans and capabilities are created to overcome design faults. The agent-multi system is responsible for controlling replicated messages and mobile agents to guarantee dependability.

Agency Property	Agency Exceptions
Interaction	UnknownSender
Autonomy	ThreadStartError
Adaptation	EligiblePlanNotFound EligibleGoalNotDefined
Mobility	UnavailableResource
Collaboration	TimingError
Learning	UnknownEvent

Table 2. Some Agent-Level Exceptions

3. Techniques for Software Structuring

3.1. Aspect-Oriented Design and Programming

Aspect-oriented design and programming has been proposed as a technique for improving separation of concerns in software design and implementation. The central idea is that while hierarchical modularity mechanisms of object-oriented design and implementation languages are extremely useful, they are inherently unable to modularize all concerns of interest in complex systems. Aspect-oriented design does for concerns that naturally cut across each other what object-oriented design does for concerns that are naturally hierarchical – it provides mechanisms that explicitly capture the crosscutting structure. Thus, the goal of aspect-oriented design and programming [15] is to support the developer in cleanly separating components (objects) and *aspects* (concerns) from each other, by providing mechanisms that make it possible to abstract and compose them to produce the overall system.

Aspects are defined as system properties that *crosscut* components in a system's design and implementation. Aspects are classified into two kinds: (i) *generic* aspects, and *domain-specific*. Traditionally, aspect-oriented designs have used generic aspects, such as logging, persistence, synchronization, and so on. Like subclasses and superclasses, *subaspects* can extend and inherit elements from their *superaspects*. Central to the process of composing aspects and components is the concept of *join points*, the elements of the component language semantics with which the aspect programs coordinate. Join points are principal points in the dynamic execution of the program (Figure 2). Examples of join points are exception occurrences, method calls and receptions, method executions and field sets and reads.

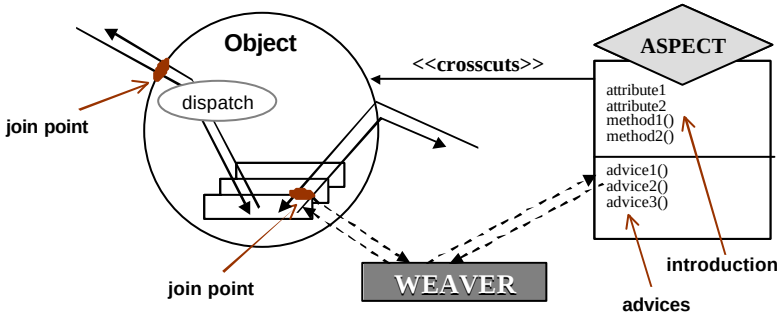

Figure 2. AspectJ Mechanisms for Dealing with Crosscutting Aspects.

AspectJ [16] is a practical aspect-oriented extension to the Java programming language. AspectJ provides support for implementing aspects. Aspects are associated with one or more objects, and are comprised of *pointcuts*, *advices*, and *introduction*. Pointcuts are collections of join points; advice is a special method-like construct that can be attached to pointcuts; introduction is a construct that defines new ordinary Java member declarations to the object to which the aspect is attached (such as, attributes and methods). *Weaver* is the mechanism responsible for deviating the normal control flow to an advice, when program execution point is at a join point (Figure 2).

3.2. Associative Blackboard Architectures

Blackboard architectures are common memory areas that can be visited by different agents. The agents can write, read and delete data from this area, which can be thus used as an agent communication interface. Apart from communication, these memory areas can be used to write the agent data itself in order to make the agents' state persistent. One specific implementation of blackboard architectures is Linda-like tuple spaces [30,31], also called *associative blackboard architectures*, or just *tuple space architectures*. There are two basic elements in a tuple space: the *Tuple* and the *Space* itself. Unlike traditional messages, a tuple is a data object in its own right, that is, composed of a series of other objects. A tuple can be read, written or removed from a space. Tuples are read from a space in a template associative way, where wildcards can be used in the search. For example, a tuple ("name","silva","otavio") could be read by the operation read("name","silva",String), where String is a wildcard for any array of characters. In tuple spaces there are three basic operations: (i) read – reads a tuple from the tuple space without deleting it, (ii) write – writes a tuple in the space, and (iii) take – reads and remove a tuple from the space. Other derived operations may also exist depending on the tuple space's implementation.

Unlike traditional inter-process (or inter-agent) message passing, tuple spaces provide what is called an uncoupled programming style. When a process creates a tuple and writes it to a space it does not have to know which other processes are going to read that tuple or even when this is going to happen. In fact, tuples exist in a space independently of the processes that have created them or the processes that will perform any other operations on them. There are some commercial tuple space architectures, and the two best known are IBM TSpaces [18] and Java Spaces[23]. These architectures provide support to all basic associative blackboard operations (read, write and take) and implement some others such as blocking process while waiting for tuples, automatic tuple deleting. They can also be accessed from remote hosts and set to be persistent.

Tuple space architectures can be programmed to react to specific stimulus [32] enabling what we call *reflective tuple space architectures*. This can be done by attaching a reflective meta-level layer to the system's tuple space. This layer is responsible for providing behavioral reflectivity to the space. The behavioral reflectivity is the capability of the system to react to particular operations over specific data in order to change its ordinary behavior. For example, the space can be programmed to avoid any tuple removal. The reactions are programmed by using a meta-level tuple space where meta-level tuples are stored. Meta-tuples are defined as a 4-tuple that has the following elements: a *Reaction* (Rct), an *Operation* (Op), an *Agent Identity* (Id), and *TupleID* (TId). These meta-tuples specify which reaction is performed when an agent idetified by Id executes an operation Op over the tuple space, which returns a tuple identified by TId.

The Rct element is an object that is an instance of a class that extends the abstract Reaction class, that has an abstract react method. This method receives a tuple t and returns another tuple. Thus when an operation Op is performed in the space by an agent identified by Id supplying a template tuple, or a tuple to be written, the reflective space executes the following steps: (i) it performs the pattern matching search in the space (read or take operations), receiving a tuple identified by TId, or writes the tuple to the space receiving its following identification TId; (ii) for the received TId it performs a read in the meta-level tuple space using the following template tuple (Reaction,Op,Id,Tid) where Reaction is a wildcard for any reaction object; (iii) if a tuple is returned by the read operation, that is, there is a reaction programmed for

those input values, its reaction is executed over the tuple identified by TId**.** Following this approach, the application designer, its administrator and its agents can program the tuple space simply by inserting reaction tuples in the meta-level layer.

4. A Unified Architecture for Agent-Level and System-Level Dependability

Figure 3 presents our software architecture for dealing with dependability and properties at the agent-level and at the system-level. Aspects are used to modularize the agency properties (*agency aspects*) and the application's general concerns (*generic aspects*), including exception handling, redundancy and persistence. Section 4.1 presents our agent-level approach applying it to Portalware's agents (Section 2.4). A reflective tuple space architecture deals with the system-level properties. Tuple spaces contain tuples that are messages to be exchanged among software agents, and serialized agent objects that are to be moved from one host to another. Different replicas of tuple spaces are available to tolerate faults. When an operation (read, write or take) is performed on tuple spaces, the control flow is deviated to the meta-level tuple spaces where reactions are executed. A *delegator* component is responsible for invoking specific reactions related to coordination, mobility, communication, replication, persistence, and exception handling. This component verifies the performed operation in order to determine which specific reactions should be executed. Section 4.2 discusses how meta-level tuples and reactions are used to support the various system-level properties presenting some scenarios of our case study (Section 2.4).

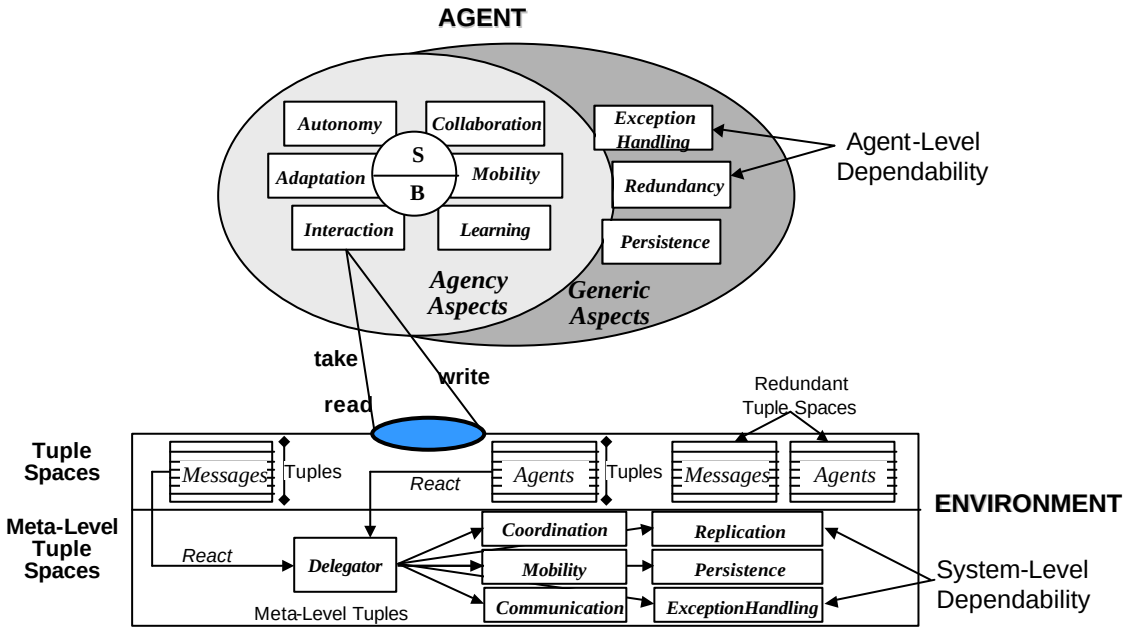

Figure 3. The Proposed Software Architecture

4.1. Agent-Level Dependability

In the following, our approach to agent-level dependability is presented as an aspect-oriented extension of the traditional object model. In particular, our proposal is discussed in terms of: (i) agent types, (ii) agency aspects, (iii) exception handling aspects, and (iv) redundancy aspect. We adopt UML diagrams [2] as the modeling language throughout this paper. The design notation for aspects is based on [14]: aspects are

represented as diamonds, the first part of an aspect represents introductions, and the second one represents advices.

Agent Types. The Agent class specifies the core state and behavior of an agent, and should be instantiated in order to create the application's agents. Since an agent is described in terms of its goals, beliefs, and plans, the attributes of an Agent object should hold references to objects that represent these elements, namely Belief, Goal and Plan objects. Methods of the Agent class are used to update these attributes and implement the agent's capabilities. Our approach proposes the use of inheritance in order to create different agent types. Different types of agents are organized hierarchically as subclasses that derive from the root Agent class. The methods of these subclasses implement the capabilities of each agent type. Figure 4 illustrates the subclasses representing the different kinds of agents of our case study (Section 2.4). For example, the method search(keyword) of the InformationAgent class implements the capability of information agents searching for information according to a specified keyword.

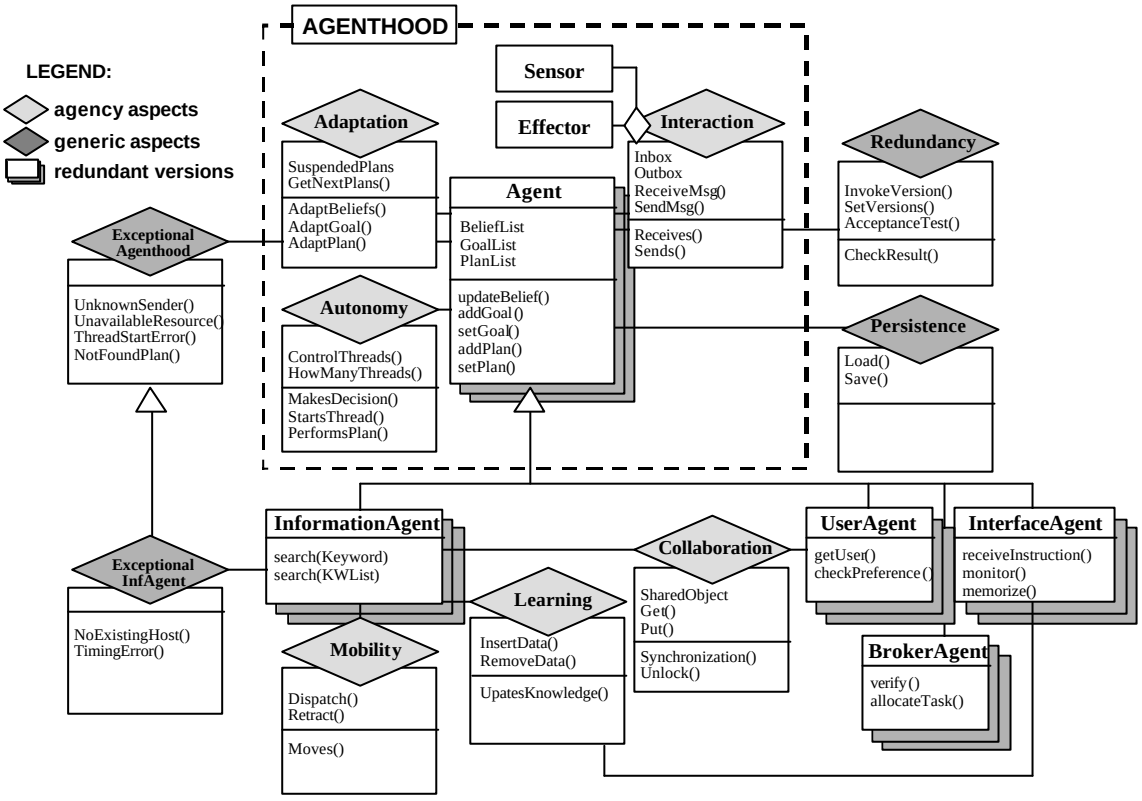

Figure 4. Our Aspect-based Approach for the Portalware Agents

Agency Aspects. Agency aspects are used to implement the agency properties an agent incorporates. Each agency aspect is responsible for providing the appropriate behavior for an agent's property. An agency aspect introduces an interface related to the agent's property, and implements the advices that crosscut the core agent's functionality. Figure 4 depicts the aspects that define essential agency properties for agenthood: interaction, adaptation, and autonomy (Section 2.1). For example, when the Interaction aspect is associated with the Agent class, it makes any Agent instance interactive. In other words, the Interaction aspect extends the Agent class's behavior to send and receive messages. This aspect updates messages and senses changes in the environment by means of sensors and effectors. The Adaptation aspect makes an Agent object adaptive; it adapts an agent's state and behavior according to new environmental conditions. Its advices are responsible for updating beliefs, goals, and plans, respectively. The Autonomy

aspect makes an Agent object autonomous, it encapsulates and manages one or more independent threads of control, implements the acceptance or refusal of a capability request and for acting without direct external intervention. The agency aspects that are specific to each agent type are associated with the corresponding subclasses (Figure 4). Note that the different types of software agents inherit the agency aspects attached to the Agent superclass. As a consequence, the information agent inherits the agenthood features and only defines its specific capabilities, plans and aspects. Thus, the InformationAgent class is additionally associated with the Mobility, Collaboration, and Learning aspects.

Exception Handling Aspects. Exception handling aspects are associated with classes and implement the handlers for the different agent-level exceptions (Figure 4). As a consequence, the exceptional behavior of an agent is separated explicitly from its core functionality. The Agent class is attached to the Exceptiona Agenthood aspect that contains the handlers for: (i) exceptions raised during the execution of the Agent class' methods, and (ii) exceptions associated with the agenthood aspects (Table 2). An additional exception handling aspect is assigned to each agent type and includes only the handlers specific for a given agent type. For example, the ExceptionalInfAgent aspect extends the ExceptionalAgenthood aspect and only needs to include the handlers for exception conditions raised during the execution of capabilities and plans of an information agent. In addition, this aspect includes the handlers for their specific agency aspects. It is not necessary to implement again the handlers for agenthood aspects. In this way, the use of exception handling aspects allows application designers to compose an exception handling aspect hierarchy that is orthogonal to the agent type hierarchy. The exception handling aspects are organized hierarchically so that the resultant hierarchy is orthogonal to the agent classes.

Redundancy Aspect. A redundancy aspect is defined to control the redundant versions for the agent's capabilities and plans. A number of redundant classes must be created to implement different versions for the agent's capabilities and plans. Version objects are known only to the Redundancy aspect and are called by it. The Redundancy aspect implements the acceptance test and is responsible for invoking transparently a new version. When the executions of capabilities and plans are completed, the CheckResult() advice gets the results, and invokes the acceptance test. If the output result is not correct according to the acceptance test, the Redundancy aspect rolls back the state of the associated Agent object and calls a new redundant version. The Agent class code is not intermingled with calls to the redundancy services.

4.2. System-Level Dependability

Our proposal uses a uniform technology approach based on a reflective tuple space architecture for multi-agent systems. The proposed architecture uses reflective tuple spaces to deal with the main system-level features (Section 2.2).

Mobility. We propose using the tuple space architecture itself to implement the agent's mobility protocol. In agent-based OO software, no matter what technique is used to model the agent [3, 8, 13], it will always be implemented as an object or a set of objects. Implementing agents as objects allow them to be elements of tuples that can be written, read or taken from a tuple space. Since tuple spaces can be accessed from remote hosts, these hosts can take tuples containing agents from the tuple spaces enabling the agents' actual mobility.

In our case study, as agents learn based on previous experience while roaming through different hosts, then keeping them alive is very important to the application, in order to preserve its evolution. Using the reflective tuple space model is useful in this situation since the dispatching of an agent can be programmed with a reaction that writes an agent's clone in a redundant tuple space. If the agent does not come back to its original environment after a period of time, specified by the designer, its clone becomes active.

This architecture can also be used to deal with agents that retrograde after their ride through the network. To implement this issue the agents must provide a comparison heuristic. This heuristic is used to decide

which between two agents is more interesting to the system. When a mobile agent returns to its natural host, the reflective tuple space can react eliminating the active agent or its clone. The comparison heuristic is then used to keep "alive" the best among them.

The tuple spaces are also used to provide information and access to system local resources. The system's administrator inserts Tuples containing information about which resources are available in the system level tuple space. The reactive model is then used to grant access to local resources only to authorized agents. This is done by programming the tuple spaces to react to any resource information reading, returning the information only to the agents that are allowed to receive it.

Coordination. Reflective tuple spaces provide an uncoupled (spatial and temporal) programming style (section 3.1). This well suits mobile and multi-agent applications as in a wide and dynamic environment a complete and updated knowledge of execution environments and of other agents may be difficult or even impossible to acquire. As agents would somehow require pattern-matching mechanisms to deal with uncertainty, dynamism and heterogeneity, it is worthwhile integrating these mechanisms directly in the coordination model to simplify agent programming. In our case study coordination is addressed to preventing duplicate keyword searching when multiple information agents are working in a single search task. In this situation tuple spaces can be programmed in order to automatically inform that another agent has already done the search in a specific host, avoiding redundant work.

Communication. Blackboards have already been used as communication interfaces for many multi-agent applications [41,34]. These interfaces suit well the agents' autonomy as they can decide whether they want to look for information or not, thus preserving their autonomous behavior. The use of tuple spaces grants an even better communication platform as the associative search mechanism can provide complex information searches in a simple manner.

As discussed previously, a malicious agent can then try to take a broadcast information out of the space, so that the other agents do not see the information. This problem can be prevented by adding to the meta-level tuple space a reaction that acts over any take operation performed over tuples that contain broadcast information. This kind of reaction, instead of taking the tuple out of the space, will only read it. Communications reactions can also be used to inform an agent that sent a message that it was read by the receiver or even that other agents have been looking for information on the same subject. In our case study this issue is applied to inform the broker agent which information agents are working on its behalf.

Persistence. Commercial tuple spaces [18,23] may also be set to be persistent. As agents themselves can be elements of tuples, they can be used as the multi-agent system persistent storage structure. We believe that using tuple spaces to implement the agent's persistence is better than using the file system as the associative search mechanism provide more sophisticated manners to retrieve the persistent agents. For example, an agent can be retrieved by its capabilities rather than by its identity.

Handling messages addressed to agents while they are "asleep" is still an issue to be thought about in many multi-agent applications. If an agent has to be awakened with each incoming message to see whether it is relevant or not, storing the agent to save running many in-memory copies would be counterproductive. Using reflective tuple spaces optimizes this process, as they can be programmed to react to relevant messages, awaking persistent agents. This is specially addressed by our case study when broker agents assign search tasks to information agents that are asleep.

Exception Handling and Replication. Different system-level exceptions can be raised and should be handled by the system. In our approach, reactions are used to implement the handlers for system-level exceptions. When an appropriate handler is not found, the delegator signals a failure exception to the software agent using the tuple space architecture. In order to implement a dependable application, redundancy of data is often necessary. The use of tuple spaces enables the creation of redundant spaces in a simple way. The use of redundant spaces in addition to reflective tuple spaces enables automatic replication of redundant data. For example, in order to provide a dependable communication system

means warranting that any information sent from one agent to another will always arrive at its destination. Reactions are programmed to replicate messages and mobile agents themselves to guarantee dependability in a way that is transparent to agents' programmers.

5. Implementation Issues

Our aspect-based approach (Section 4.1) was implemented for the case study application (Section 2.4) using AspectJ [16]. The implementation consists of 91 classes, and 16 aspects (including subaspects). In order to implement the dependency relationship between these different aspects, we used the "dominates" construct of AspectJ. We previously constructed a software architecture for implementing agent-based object-oriented applications without applying the structuring principles of our aspect-based approach (Section 4). We have found our approach allows building agent software with fewer lines of code than building agent software by using our previously-constructed software architecture [8].

In our case study, different instances of information agents should have varying characteristics. They may be collaborative or non-collaborative, they may be static or mobile, they may or may not learn. So it is desirable to build personalized information agents to attach different aspects to distinct instances of information agents. Our proposed model supports this feature. However, the current version of AspectJ does not provide direct support for associating different aspects with different class instances. This feature is currently supported by some meta-object protocols such as Guaraná [22].

We have also used IBM TSpaces [18], a blackboard architecture for network communication with database capabilities, to implement our reflective tuple space architecture. TSpaces provides group communication services, database services and event notification services. It is implemented in the Java programming language and thus it automatically possesses network ubiquity through platform independence, as well as standard type representation for all datatypes. In order to implement the programmable tuple spaces using IBM TSpaces, we extend its `TupleSpace` class, overriding its basic operations (`read`, `write` and `take`) to perform the search for reactions in the meta-level tuple space and performing them over system's tuples, as addressed in Section 3.2.

6. Discussion and Related Work

Simpler Evolution. Kendall *et al*. [13] proposes the layered agent architectural pattern, which separates different layers of an agent, such as sensory layer, collaboration layer, and so on. This approach is interesting because it allows software engineers to deal with agent layers separately. However, this proposal causes object schizophrenia [12], that is the agent state and behavior, which are intended to appear as a single object, are actually distributed over multiple objects. In addition, we believe agent design evolution is cumbersome since adding or removing any of these layers is not trivial; it requires invasive adaptation of the adjacent layers. In our approach, agency aspects can be added to or removed from classes in a plug-and-play way. The remaining behavior of the agent is then kept. For example, the `Mobility` aspect can be detached from the `InformationAgent` class without requiring any invasive adaptation for the other agent's components.

Reduced Complexity in Handling Exceptional Conditions. Moreover, when the state and behavior of software agents are distributed over multiple objects, exception handling is harder since the exceptional behavior will also be spread over them. Our approach encourages agent system developers to think about exceptions associated with agency properties in a systematic way, since the agency properties are explicitly separated.

Improved Reusability of Exceptional Behavior. Handlers are usually implemented using `try … catch` blocks supported by exception handling mechanisms in object-oriented programming languages such as Java and C++. However, it promotes the production of dependable OO systems that are difficult to read and reuse [28]. Hierarchies of exception handling aspects allow exception handling subaspects to inherit

handlers from their superaspects and, consequently, they allow exceptional code reuse. When reuse is not desired, the handler can be redefined at the subaspects.

Experience with Both Domain-Specific Aspects and Generic Aspects. Research in aspect-oriented software engineering has concentrated on the implementation phase. A few works have presented aspect-based design solutions. In addition, since aspect-oriented programming is still in its infancy, little experience with employing this paradigm is currently available. To date, aspect-oriented programming has been typically used to implement generic aspects such as persistence, error detection/handling, logging, tracing, caching, and synchronization. However, each of these papers is generally dedicated to only one of these generic aspects. In this work, we provide an aspect-based approach that: (i) handles both agency-specific aspects as well as generic aspects (e.g. exception handling), and (ii) encompasses a number of different aspects and their relationships.

Better System-Level Dependability. Reflective tuple spaces have already been used in the MARS System [32] and in the TuCSon [19] model for coordination purposes. However, these systems do not explore the capabilities of the reflective architecture to implement other system-level features. There are many frameworks to deal with mobile agents applications such as Aglets [17] and Concordia [20]. Although these frameworks deal appropriately with many agent-level and system-level issues, they do not focus on the system's dependability. Our approach uses reflective tuple space architectures in order to provide a uniform technology to deal with main system-level features. We believe that our model improves exception handling and fault tolerance as system-level exceptional behavior is also handled in a uniform manner. In this case, tuples are the structuring units that serve as natural areas of error containment and error recovery.

7. Conclusions and Future Work

With agent-based object-oriented (OO) software growing in size and complexity, introducing dependability while satisfying quality requirements, such as maintainability and reusability, are still deep concerns to engineers of multi-agent OO software. In this paper, we focus on fault tolerance and exception handling as one of the chief means for guaranteeing system dependability. This paper presented a software architecture meant to be simple enough to enable the use of exception handling and fault tolerance techniques in the development of dependable multi-agent OO software.

In particular, our architecture supports agency and system-level properties separately in order to produce effective multi-agent software. It also describes structured integration of agency properties into the object model, and encourages the separate handling of agent-level and system-level dependability. As a consequence, we believe that our approach allows the construction of dependable multi-agent systems that are easy to understand, maintain and reuse. The proposed software architecture has the potential to bring exception handling and fault tolerance to complex multi-agent applications due to its simplicity and ease of implementation. Designers of dependable multi-agent software can concentrate on the complexity inherent to their systems rather than on the intricacies of the exception handling and fault tolerance schemes, easing the task of building multi-agent applications with high reliability.

As future work we intend to conclude the implementation of our architecture in two separate phases. The first one consists of implementing and validating our system-level reflective tuple spaces architecture for some of the main system level properties: coordination, communication, persistence. The second one approaches the implementation and validation of redundancy and exception handling generic aspects at the agent and system level. We also intend to apply our system-level architecture to other testbed applications such as the iDeal Market [41]. We plan to implement agent communication based on high level languages such as KQML [4] and FIPA ACL [7] using our tuple space architecture. This will be done joining XML [10] translations for these languages [6] and the TSpaces facilities for XML and XQL[11]. These implementations will help us to perform some experiments related to performance measurement and scalability, comparing it with other multi-agent architectures.

References

[1] J. Bigus and J. Bigus. "Constructing Intelligent Agents with Java – A Programmer's Guide to Smarter Applications". Wiley, 1998.

[2] G. Booch, and J. Rumbaugh. "Unified Modeling Language – User Guide". Addison-Wesley, 1999.

[3] J. Bradshaw et al. "KaoS: Toward an Industrial-Strength Generic Agent Architecture". In J. Bradshaw (Ed.), Software Agents. Cambridge: AAAI/MIT Press, 1996.

[4] T. Fiinn , R. Fritzson, D. McKay, R. McEntire. "KQML as an Agent Communication Language". The Proceedings of the Third International Conferenceon Information and Knowledge Management, ACM Press 1994

[5] D. Brugali and K. Sycara. "A Model for Reusable Agent Systems". In: *Implementing Application Frameworks – Object-Oriented Frameworks at Work*, M. Fayad et al. (Ed.), John Wiley & Sons, 1999.

[6] B. Grosof, Y. Labrou. "An Approach to using XML and a Rule-based Content Language with an Agent Communication Language". Proc. IJCAI-99 Workshop on Agent Communication Languages, 1999

[7] Foundation for Intelligent and Physical Agents (FIPA), "FIPA 97 Part 2 Version 2.0: Agent Communication Language Specification", http://www.fipa.org/specs/fipa00003/

[8] A. Garcia, C. Lucena, D. Cowan. "Agents in Object-Oriented Software Enginnering". Technical Report CS-2001-07, Computer Science Department, University of Waterloo, Waterloo, Canada, March 2001.

[9] A. Garcia, M. Cortés, C. Lucena. "A Web Environment for the Development and Maintenance of E-Commerce Portals Based on Groupware Approach". Proceedings of the IRMA'2001, Toronto, May 2001.

[10] World Wide Web Consortium Recommendation, "Extensible Markup Language (XML) 1.0", http://www.w3c.org/xml

[11] J. Robie, J. Lapp, D. Schach, "XML Query Language (XQL)" 1998 http://www.w3.org/TandS/QL/QL98/pp/xql.html

[12] E. Kendall. "Agent Roles and Aspects". ECOOP Workshop on Aspect Oriented Programming, July, 1998.

[13] E. Kendall et al. "A Framework for Agent Systems". In: *Implementing Application Frameworks – Object-Oriented Frameworks at Work*, M. Fayad et al. (Ed.), John Wiley & Sons, 1999.

[14] M. Kersten and G. Murphy. "Atlas: A Case Study in Building a Web-Based Learning Environment using Aspect-Oriented Programming". Proceedings of the OOPSLA'99, Denver, USA, ACM Press, pp. 340-352, 1999.

[15] G. Kiczales et al. "Aspect-Oriented Programming". In Proceedings of the ECOOP'97, Finland. Springer-Verlag LNCS 1241. June 1997.

[16] G. Kiczales et al. "An Overview of AspectJ". In Proceedings of the ECOOP'2001, Budapest, 2001 (to appear).

[17] D. Lange and M. Oshima. "Programming and Developing Java Mobile Agents with Aglets." Addison-Wesley, August 1998.

[18] T. Lehman, S. McLaughry, and P. Wyckoff. "TSpaces: The Next Wave". Hawaii International Conference on System Sciences (HICSS-32), January 1999.

[19] A. Omicini, F. Zambonelli, "Coordination of Mobile Information Agents in TuCSoN", Internet Research, Vol. 8, No. 5, pp. 400-413, 1998

[20] Mitsubishi Electric ITA. "Concordia: An Infrastructure for Collaborating Mobile Agents", www.meitca.com:HSL/Projects/Condordia/

[21] Object Management Group – Agent Platform Special Interest Group. "Agent Technology – Green Paper". Version 1.0, September 2000.

[22] A. Oliva and L. Buzato. "The Design and Implementation of Guaraná". 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99), May 3-7, 1999, San Diego, CA, USA.

[23] JavaSpaces™ Service Specification Version 1.1, Sun Microsystems Specification October 2000 http://www.sun.com/jini/specs/jini1.1html/js-title.html

[24] P. Ripper, M. Fontoura, A. Neto, and C. Lucena. "V-Market: A Framework for e-Commerce Agent Systems." World Wide Web, Baltzer Science Publishers, 3(1), 2000.

[25] Y. Shoham. "Agent-Oriented Programming". Artificial Intelligence, 60(1993): 24-29, 1993.

[26] P. Lee and T. Anderson. "Fault Tolerance: Principles and Practice". Springer-Verlag, 2nd edition, Wien, Austria, January 1990.

[27] O. Silva, D. Orlean, F. Ferreira and C. Lucena. "A Shared-Memory Agent-Based Framework for Business-to-Business Applications". Proceedings of the 2001 Information Resources Management Association International Conference IRMA'2001) - Web Engineering for E-Commerce Applications, May 2001

[28] A. Garcia, C. Rubira, A. Romanovsky and J. Xu. "A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software". Journal of Systems and Software, Elsevier, 2001. (To appear)

[29] P. Maes. Concepts and Experiments in Computacional Reflection. ACM SIGPLAN Notices, 22(12):147-155, 1987

[30] D. Galernter, "Generative Communication in Linda" ACM Transactions on Programming Languages and Systems, vol. 7, No.1, (1985), pp 80-112

[31] N. Carriero, D. Galernter, "Linda in Context", Communications of the ACM, vol. 32, No. 4, (1989), pp 444-458

[32] G. Cabri, L. Leonardi, F. Zambornelli, "Reactive Tuple Spaces for Mobile Agents Coordination", Proceedings of the 2nd International Workshop on Mobile Agents (MA 98), Stuttgart (D), September 1998, Springer Verlag LNCS N. 1477

[33] T. Malone, K. Crowston, "The Interdisciplinary Study of Coordination", ACM Computing Surveys, Vol 26(1), pp. 87-119, 1994

[34] M. Huhns, L. Stephens, "Multiagent Systems and Societies of Agents", In: "Multiagents Systems – A Modern Approach to Distributed Artificial Intelligence", edited by G. Weiss, MIT Press, pp. 79-120

[35] J. White, "Mobile Agents", In: *Software Agents*, J. Bradshaw (ed.), American Association for Artificial Intelligence/MIT Press, 1997.

[36] H. Peine, T. Stolpmann , "The Architecure of the Ara Platform for Mobile Agents", Procedings of the 1st International Workshop on Mobile Agents, Berlin, LNCS, No. 1219, Springer-Verlag, pp 50-61, April 1997

[37] B. Randell. "System Structure for Software Fault Tolerance". *IEEE Transactions on Software Engineering*, SE-1(2): 220-32, 1975.

[38] A. Avizienis. "The N-version Approach to Fault-Tolerant Software". *IEEE Transactions on Software Engineering*, SE-11(12): 1491-1501, 1985.