Rigorous Development of Ambient Campus Applications that can Recover from Errors

Budi Arief, Alexei Iliasov, and Alexander Romanovsky

School of Computing Science, University of Newcastle upon Tyne,
Newcastle upon Tyne NE1 7RU, England.
{L.B.Arief, Alexei.Iliasov, Alexander.Romanovsky}@newcastle.ac.uk

Abstract. In this paper, we discuss a new method for developing fault-tolerant ambient applications. It supports stepwise rigorous development producing a well structured design and resulting in disciplined integration of error recovery measures into the resulting implementation.

1 Introduction

Ambient campus is a loosely defined term to describe *ambient intelligence* (AmI)[1] systems in an educational (university campus) setting. As such, ambient campus applications are tailored to support activities typically found in the campus domain, including – among others, delivering lectures, organising meetings, and promoting collaborations among researches and students.

This paper presents our work in the development of the ambient campus case study within the RODIN project [1]. This EU-funded project, led by the School of Computing Science of Newcastle University, has an objective to create a methodology and supporting open tool platform for the cost-effective rigorous development of dependable complex software systems and services. In the RODIN project, the ambient campus case study acts as one of the research drivers, where we are investigating how to use formal methods combined with advanced fault-tolerance techniques in developing highly dependable AmI applications. In particular, we are developing modelling and design templates for fault-tolerant, adaptable and reconfigurable software. This case study consists of several working ambient applications (referred to as *scenarios*) for supporting various educational and research activities.

Software developed for AmI applications need to operate in an unstable environment susceptible to various errors and unexpected changes (such as network disconnection and re-connection) as well as delivering context-aware services. These applications tend to rely on the *mobile agent paradigm*, which supports system-structuring using decentralised and distributed entities (*agents*) working together in order to achieve their individual aims. Multi-agent applications pose

[1] Ambient intelligence is a concept developed by the Information Society Technologies Advisory Group (ISTAG) to the European Commission's DG Information Society and the Media, where humans are surrounded by unobtrusive computing and networking technology to assist them in their activities.

many challenges due to their openness, the inherent autonomy of their components (i.e. the agents), the asynchrony and anonymity of the agent communication, and the specific types of faults they need to be resilient to. To address these issues, we developed a framework called CAMA *(Context-Aware Mobile Agents)* [2], which encourages disciplined development of open fault-tolerant mobile agent applications by supporting a set of abstractions ensuring exception handling, system structuring and openness. These abstractions are backed by an effective and easy-to-use middleware allowing high system scalability and guaranteeing agent compatibility. More details on CAMA and its abstractions can be found in [2–4]. The rest of this paper outlines our case study scenarios (Section 2), discusses important fault-tolerance issues in AmI systems (Section 3), and describes our design approach (Section 4).

2 Case Study Scenarios

We have so far implemented two scenarios for our ambient campus case study using the CAMA framework as the core component of the applications. The first scenario (*ambient lecture*) deals with the activities carried out by the teacher and the students during a lecture – such as questions and answers, and group work among the students – through various mobile devices (PDAs and smartphones). The second scenario (*presentation assistant*) covers the activities involved in giving or attending a presentation/seminar. The presenter uses a PDA to control the slides during their presentation and they may receive 'quiet' questions on the topic displayed on the slide from the audience. Each member of the audience will have the current slide displayed on their PDA, which also provides a feature to type in a question relevant to that slide.

We are now working on a more challenging scenario which involves greater agent mobility as well as location specific services. Agents may move physically among multiple locations (rooms), and depending on the location, different services will be provided.

In this scenario – we call it *student induction assistant* scenario – we have new students visiting the university campus for the first time. They need to register to various university departments and services, which are spread on many locations on campus, but they do not want to spend too much time looking for offices and standing in queues. They much prefer spending their time getting to know other students and socialising. So they delegate the registration process to their personalised software agent, which then visits virtual offices of various university departments and institutions, obtains the necessary information for the registration, and makes decisions based on the student's preferences. The agent also records pieces of information collected during this process so that the students can have all the details about their registration.

Unfortunately, not all the registration stages can be handled automatically. Certain steps require personal involvement of the student, for example, signing paperwork in the financial department and manually handling the registration in some of the departments which do not provide fully-featured agent able to

handle the registration automatically. To help the students to go through the rest of registration process, their software agent creates an optimal plan for visiting different university departments and even arranges appointments when needed.

Walking around on the university campus, these new students pass through *ambients* – special locations providing context-sensitive services (see Figure 1). An ambient has sensors detecting the presence of a student and a means of communicating to the student. An ambient gets additional information about students nearby by talking to their software agent. Ambients help students to navigate within the campus, provide information on campus events and activities, and assist them with the registration process. The ambients infrastructure can also be used to guide students to safety in case of emergency, such as fire.

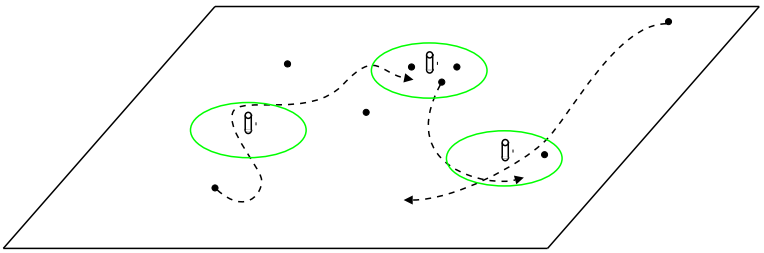

Fig. 1. *Student induction assistant* scenario: the dots represent free roaming student agents; the cylinders are static infrastructure agents (equipped with detection sensors); and the ovals represent *ambients* – areas where roaming agents can get connection and location-specific services.

3 Challenges in Developing Fault-Tolerance Ambient Intelligence Systems

Developing fault-tolerant ambient intelligence systems is not a trivial task. There are many challenging factors to consider; some of the most important ones are:

- *Decentralisation and homogeneity*
  Multi-agent systems are composed of a number of independent computing nodes. However, while traditional distributed systems are *orchestrated* – explicitly, by a dedicated entity, or implicitly, through an implemented algorithm – in order to solve a common task, agents in an ambient system decide *independently* to *collaborate* in order to achieve their individual goals. In other words, ambient systems do not have inherent hierarchical organisation. Typically, individual agents are not linked by any relations and they may not have the same privileges, rights or capabilities.
- *Weak Communication Mechanisms*
  Agent systems commonly employ communication mechanisms which provide

very weak, if any, delivery and ordering guarantees. This is important from the implementation point of view as agent systems are often deployed on wearable computing platforms with limited processing power, and they tend to use unreliable wireless networks for communication means. This makes it difficult to distinguish between a crash of an agent, a delay in a message delivery and other similar problems caused by network delay. Thus, a recovery mechanism should not attempt to make a distinction between network failures and agent crashes unless there is a support for this from the communication mechanism.

- *Autonomy*
  During its lifetime, an agent usually communicates with a large number of other agents, which are developed in a decentralised manner by independent developers. This is very different from the situation in classical distributed system where all the system components are part of a closed system and thus fully trusted. Each agent participating in a multi-agent application tries to achieve its own goal. This may lead to a situation where some agents may have conflicting goals. From recovery viewpoint, this means that no single agent should be given an unfair advantage. Any scenarios where an agent controls or prescribes a recovery process to another agent must be avoided.
- *Anonymity*
  Most agent systems employ anonymous communication where agents do not have to disclose their names or identity to other agents. This has a number of benefits: agents do not have to learn the names of other agents prior to communication; there is no need to create fresh names nor to ensure naming consistency in the presence of migration; and it is easy to implement group communication. Anonymity is also an important security feature - no one can sense an agent's presence until it produces a message or an event. It is also harder to tell which messages are produced by which agent. For a recovery mechanism, anonymity means that we are not able to explicitly address agents which must be involved in the recovery. It may even be impossible to discover the number of agents that must be involved. Even though it is straightforward to implement an exchange for agents names, its impact on agent security and the cost of maintaining consistency usually outweigh the benefits of having named-agents.
- *Message Context*
  In sequential systems, recovery actions are attached to certain regions, objects or classes which define a context for a recovery procedure. There is no obvious counterpart for these structuring units in asynchronously communicating agents. Agent produces messages in a certain order, each being a result of some calculations. When the data sent along with a message cause an exception in an agent, the agent may want to notify the original message producer, for example, by sending an exception. When an exception arrives at the message producer (which is believed to be the source of the problem), it is possible that the agent has proceeded with other calculations and the context in which the message was produced is already destroyed. In addition, an agent can disappear due to migration or termination.

– *Message Semantics*
In a distributed system developed in a centralised manner, semantics of values passed between system components is fixed at the time of the system design and implementation. In an open agent system, implementation is decentralised and thus the message semantics must be defined at the stage of a multi-agent application design. If an agent is allowed to send exceptions, the list of exceptions and their semantics must also be defined at the level of an abstract application model. For a recovery mechanism, this means that each agent has to deal only with the exception types it can understand, which usually means having a list of predefined exceptions.

We have to take these issues into account when designing and developing fault-tolerant AmI systems. In the following section, we outline the design approach that can be used for constructing ambient campus case study scenarios.

4 Design Approach

In our previous work, we have developed several case study scenarios [5, 6, 4]. In these scenarios, we focused on the implementation aspects and the general problems of applying formal methods in ambients systems. In our new case study scenario (*student induction assistant* scenario – see Section 2), we shift our focus from implementation to design to validate our formal development approach.

The new scenario will be modelled using Event-B formalism [7]. Our intention is to have a fairly detailed model which covers issues such as communication, networking failures, proactive recovery, liveness, termination, and migration.

4.1 Overview

We are using *design patterns*, *refinement patterns*, and *mobility modelling* in developing the student induction assistant scenario.

Design patterns are described using a natural language and act as guide in formal development. Typically, design patterns are narrowly focused and can be applied only within a given problem domain. We have developed a set of design patterns that are specific for ambient systems [8]. These patterns are inspired by the architecture of the Cama system and help us to formally design a system which can be implemented on top of the Cama framework.

Refinement patterns are rules describing a transformation of an abstract model into a more concrete one. These patterns are specified in an unambiguous form and can be mechanically applied using a special tool, which is a plugin to the RODIN Event-B platform [7, 1]. Some refinement patterns can be seen as a possible implementation of the design patterns. Others are simply useful for the transformation steps, which we believe are common in this kind of systems.

We are planning to apply the Mobility Plugin [9] for verification of dynamic properties, such as liveness, termination, and mobility-related properties. Using this plugin, we are able to extend the Event-B model with process algebraic

description of dynamic behaviour, agent composition and communication scenarios. The plugin handles additional proof-obligation using a built-in model checker. It also includes an animator for visual interactive execution of formal models.

4.2 Application to the Scenario

To proceed any further, we need to agree on same major design principles, identify major challenges and outline the strategy for finding the solution.

To better understand the scenario, we apply the *agent metaphor*. The agent metaphor is a way to reason about systems (not necessarily information systems) by decomposing it into agents and agent subsystems. In this paper, we use term *agent* to refer to a component with independent thread of control and state and the term *agent system* to refer to a system of cooperative agents.

From agent systems' viewpoint, the scenario is composed of the following three major parts: physical university campus, virtual university campus and ambients. In physical university campus, there are students and university employees. Virtual campus is populated with student agents and university agents. Ambients typically have a single controlling agent and a number of visiting agents. These systems are not isolated, they interact in a complex manner and information can flow from one part into another.

However, since we are building a distributed system, it is important to get an implementation as a set of independent but cooperative components (agents). To achieve this, we apply the following design patterns:

agent decomposition During the design, we will gradually introduce more agents by replacing abstract agents with two or more concrete agents.

super agent It is often hard to make a transition from an abstract agent to a set of autonomous agents. What before was a simple centralised algorithm in a set of agents must now be implemented in a distributed manner. To aid this transition, we use *super agent* abstraction, which controls some aspects of the behaviour of the associated agents. Super agent must be gradually removed during refinement as it is unimplementable.

scoping Our system has three clearly distinguishable parts: physical campus, virtual campus and ambients. We want to isolate these subsystems as much as possible. To do this, we use the scoping mechanism, which temporarily isolates cooperating agents. This is a way to achieve the required system decomposition. The isolation properties of the scoping mechanism also make it possible to attempt autonomous recovery of a subsystem.

orthogonal composition As mentioned above, the different parts of our scenario are actually interlinked in a complex manner. To model this connections, we use the *orthogonal composition* pattern. In orthogonal composition, two systems are connected by one or more shared agents. Hence, information from one system into another can flow only through the agent states. We will try to constrain this flow as much as possible in order to obtain to a more robust system.

locations definition To help students and student agents navigate within the physical campus and the virtual campus, we define location as places associated with a particular agent type.

decomposition into roles The end results of system design is a set of agent roles. To obtain role specifications, we decompose scopes into a set of roles.

4.3 Fault-Tolerance Mechanism

We are going to address the fault-tolerance issues at three levels: architectural, modelling and implementation.

At the architectural level, we use the agent metaphor to introduce fault-tolerance properties such as redundancy (spawning an agent copy to survive an agent crash) and diversity (having the same service provided by independently implemented agents).

At the modelling level, we apply the assumptions mechanism [10] along with FT-specific design patterns. The assumptions mechanism helps us to build more robust agent applications and it has two different styles. In the first one, it is assumed that certain undesirable events are not going to happen during some activity. If such an event happens, the whole activity is aborted. In the second style, through negotiations, agents agree to temporarily restrict their behaviour and cooperate in a simpler environment. Design patterns help developers to introduce some common fault-tolerance techniques when modelling an agent system. These techniques range from abstract system-level patterns to very specific agent-level patterns dealing with specific faults.

Early on, we have extended the blackboard communication pattern [11] with nested scopes and exception propagation [12]. These two extensions are essentially the implementation techniques for recovery actions introduced during the modelling stage. We also rely extensively on reactive agent architecture. This has two immediate benefits: its implementation style matches the modelling style of Event-B, and the recovery of multi-threaded agents becomes similar to that of the asynchronous reactive architecture.

5 Conclusion

This paper provides an outline of the work that we have carried out in developing fault-tolerant ambient applications. We use design patterns, refinement patterns, and mobility modelling during the design process. By using these techniques, we aim to validate the formal development approach that we take in developing more robust and fault-tolerant ambient applications.

We plan to demonstrate our approach through an ambient campus *student induction assistant* scenario. We are currently developing an agent-based system for this scenario, using the Cama framework and middleware [2] that we have previously developed as the centre piece of the system. This will be augmented with sensors for providing location specific services, as well as fault-tolerance mechanism at the architectural, modelling and implementation levels.

6 Acknowledgements

This work is supported by the IST RODIN Project [1]. A. Iliasov is partially supported by the ORS award (UK).

References

1. Rodin: Rigorous Open Development Environment for Complex Systems. IST FP6 STREP project, http://rodin.cs.ncl.ac.uk/ (Last accessed: 17 May 2007)
2. Arief, B., Iliasov, A., Romanovsky, A.: On Developing Open Mobile Fault Tolerant Agent Systems. In Choren, R., et al., eds.: SELMAS 2006, LNCS 4408. Springer-Verlag (2007) 21–40
3. Iliasov, A.: Implementation of Cama Middleware. http://sourceforge.net/projects/cama (Last accessed: 17 May 2007)
4. Iliasov, A., Romanovsky, A., Arief, B., Laibinis, L., Troubitsyna, E.: On Rigorous Design and Implementation of Fault Tolerant Ambient Systems. Technical report, CS-TR-993, School of Computing Science, Newcastle University (Dec 2006)
5. Arief, B., Coleman, J., Hall, A., Hilton, A., Iliasov, A., Johnson, I., Jones, C., Laibinis, L., Leppanen, S., Oliver, I., Romanovsky, A., Snook, C., Troubitsyna, E., Ziegler, J.: Rodin Deliverable D4: Traceable Requirements Document for Case Studies. Technical report, Project IST-511599, School of Computing Science, University of Newcastle (2005)
6. Troubitsyna, E., ed.: Rodin Deliverable D8: Initial Report on Case Study Development. Project IST-511599, School of Computing Science, University of Newcastle (2005)
7. Metayer, C., Abrial, J.R., Voisin, L.: Rodin Deliverable 3.2: Event-B Language. Technical report, Project IST-511599, School of Computing Science, University of Newcastle (2005)
8. Laibinis, L., Iliasov, A., Troubitsyna, E., Romanovsky, A.: Formal Approach to Ensuring Interoperability of Mobile Agents. Technical report, CS-TR-989, School of Computing Science, Newcastle University, UK. October (2006)
9. Butler, M., ed.: Rodin Deliverable D11: Definition of Plug-in Tools. Project IST-511599, School of Computing Science, University of Newcastle (2005)
10. Iliasov, A., Romanovsky, A.: Choosing Application Structuring and Fault Tolerance Using Assumptions, To be presented at DSN 2007 (2007)
11. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System Of Patterns. West Sussex, England: John Wiley & Sons Ltd. (1996)
12. Iliasov, A., Romanovsky, A.: Structured Coordination Spaces for Fault Tolerant Mobile Agents. In Dony, C., Knudsen, J.L., Romanovsky, A., Tripathi, A., eds.: LNCS 4119. (2006) 181–199