PhD Dissertation

International Doctorate School in Information & Communication Technologies

DIT - University of Trento

Testing Techniques for Software Agents

Cu Duy Nguyen

Advisor:

Prof. Anna Perini

FBK-irst, Trento

Co-Advisor:

Prof. Paolo Tonella

FBK-irst, Trento

December 2008

Being deeply loved by someone gives you strength, while loving someone deeply gives you courage. – Lao Tzu –

To my wife, Phuong, and my family.

Acknowledgements

From the moment I knew that I was admitted to the ICT International Doctoral School (University of Trento) and the Fondazione Bruno Kessler (ITC-irst at that time), I knew that I would be on a boat to an exciting, yet challenging, PhD trip. I would like to acknowledge the people who have been guiding, encouraging, supporting me along the journey.

First, I am deeply indebted to my advisors, Prof. Anna Perini and Prof. Paolo Tonella, for embarking with me on this PhD journey. I could not have wished for better collaborators and coaches. Your contributions, detailed comments and insights have been of great value to me.

I wish to say sincere thanks to Prof. John Mylopoulos, Prof. Andrea Omicini, Prof. Mark Harman, and Dr. Jorge J. Goméz Sanz for being the members of my defence committee. I highly appreciate their time to evaluate this thesis and their interesting questions and suggestions in my defence.

I would like to thank people of the Software Engineering research group, Fondazione Bruno Kessler: Alberto Siena, Mirko Morandini, Chiara Di Francescomarino, Leonardo Leiria Fernandes, Nauman Ahmed Qureshi, Emanuela Silvestris, and others. The more I remember your full name, the greater colleagues you become, and the open space where we work turns to be a more wonderful place.

I have had an interesting visit to the "Centre for Research on Evolution, Search and Testing", King's College London, during the PhD programme. I would like to thank Prof. Mark Harman, Prof. Michael Luck, and Dr. Simon Miles for their inspirations and collaborations.

Last but not least, a special acknowledgement goes to my wife, Phuong

Thanh Nguyen, for her endless love and encouragement. Without her unconditional supports I would not be here to realize my dream of understanding the scientific world, and of course, writing this thesis. I would pay my deepest gratitude to my family, my parents, my sisters, and my brothers with whom, I always find happiness and relaxation.

Abstract

Software agents and multiagent systems are a promising technology for today's complex, distributed systems. Methodologies and techniques that address testing and reliability of these systems are increasingly demanded, in particular to support systematic verification/validation and automated test generation and execution.

This work deals with two major research problems: the lack of a structured testing process in engineering software agents and the need of adequate testing techniques to tackle the nature of software agents, e.g., being autonomous, decentralized, collaborative.

To address the first problem, we proposed a goal-oriented testing methodology, aiming at defining a systematic and comprehensive testing process for engineering software agents. It encompasses the development process from the early requirements analysis until the deployment. We investigated how to derive test artefacts, i.e. inputs, scenarios, and so on, from agent requirements specification and design, and use these artefacts to refine the analysis and design in order to detect problems early. More importantly, they are executed afterwards to find defects in the implementation and build confidence in the operation of the agents under development.

Concerning the second problem, the peculiar properties of software agents make testing them troublesome. We developed a number of techniques to generate test cases, automatically or semi-automatically. These include goal-oriented, ontology-based, random, and evolutionary generation tech-

niques. Our experiments have shown that each technique has different strength. For instance, while the random technique is effective in revealing crashes or exceptions, the ontology-based one is strong in detecting communication faults. The combination of these techniques can help to detect different types of fault, making software agents more reliable.

We also investigated approaches to monitoring agent behaviours and evaluating them. All together, the generation, evaluation, and monitoring techniques form a bigger picture: our novel continuous testing method. In this method, test execution can proceed unattendedly and independently of any other human-intensive activity; test cases are generated or evolved continuously using the proposed generation techniques; test results are observed and evaluated by our monitoring and evaluation approaches to give feedbacks to the generation step. The aim of continuous testing is to exercise and stress the agents under test as much as possible, the final goal being the possibility to reveal yet unknown faults.

We applied a case study to illustrate the proposed methodology and performed three experiments to evaluate the performance of the proposed techniques. The obtained results are promising.

Keywords
Software agent testing, goal-oriented testing methodology, multiagent systems, agent-oriented software engineering.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The increasing use of Internet as the backbone for all interconnected services and devices makes software systems highly complex and virtually unlimited in scale. These systems often involve variety of users and heterogeneous platforms. They are evolved continuously to meet the changes of business and technology. In some circumstances, they need to be autonomous and adaptive for dealing with such changes.

Software agents, with their peculiar properties, e.g., (semi-)autonomy, adaptivity, are key technologies to meet modern business needs, e.g., world-wide computing, ubiquitous computing, networked enterprises. They offer also an effective conceptual paradigm to model such complex systems. In fact, research on the development of software agents and MultiAgent System (MAS) has grown into a very active area, and interestingly they are receiving more industrial attention as well.

As these systems are increasingly taking over operations and controls in enterprise management, automated vehicles, and financing systems, assurances that these complex systems operate properly need to be given to their owners and their users. This calls for an investigation of suitable software engineering frameworks, including requirements engineering, architecture, and testing techniques, to provide adequate software development processes and supporting tools.

Testing of software agents and MAS is a challenging task because these systems are distributed, autonomous, and deliberative. They operate in an open world, which requires context awareness. There are issues concerning communication and semantic interoperability, as well as coordination with peers. All these features are known to be hard not only to design and to program (Bergenti et al. 2004), but also to test. In particular, the very specific nature of software agents, which are designed to be autonomous, proactive, collaborative, and ultimately intelligent, makes it difficult to apply existing software testing techniques to them. For instance, agents operate asynchronously and in parallel, which challenges testing and debugging. Agents communicate primarily through message passing instead of method invocation, so existing object-oriented testing approaches are not directly applicable. Agents are autonomous and cooperate with other agents, so they may run correctly by themselves but incorrectly in a community or vice versa. Moreover, agents can be programmed to learn; so successive tests with the same test data may give different results (Rouff 2002).

As a result, testing software agents and MAS seeks for new testing techniques dealing with their peculiar nature. The techniques need to be effective and adequate to evaluate agent's autonomous behaviours and build confidence in them.

From another perspective, while this research field is becoming more mature, there is an emerging need for detailed guidelines during the development process. This is considered a crucial step towards the adoption of Agent-Oriented Software Engineering (AOSE) methodology by industry. A number of methodologies (Perini 2009, Henderson-Sellers and Giorgini 2005) have been proposed so far. While some work considered specification-based formal verification (e.g., *Formal Tropos* (Fuxman et al.

2004) and (Dardenne et al. 1993)), others relied on object-oriented techniques, taking advantage of a mapping of agent-oriented abstractions into object-oriented constructs, UML for instance. However, to the best of our knowledge, none of existing work provides a complete and structured testing process for guiding the testing activities. This is a big gap that we need to bridge in order for agent-oriented methodologies to be widely applicable.

1.1 Research problems

Problem 1: defining a structured testing process for software agents and MAS. Currently, AOSE methodologies have been focusing mainly on requirement analysis, design, and implementation; limited attention was given to validation and verification, as in *Formal Tropos* (Fuxman et al. 2004), and (Dardenne et al. 1993). A structured testing process that complements analysis and design is still absent. This problem is pivotal because without detailed and systematic guidelines, the development cost may raise in terms of effort and productivity.

Problem 2: finding effective testing techniques for software agents. The peculiar properties of software agents and MAS, e.g., being autonomous, distributed, make testing them a troublesome task. Testing traditional software systems, which have reactive (or input-output) style behaviour, is known to be non-trivial, but testing autonomous agents is even more challenging, because they have their own reasons for engaging in proactive behaviours that might differ from an user's concrete expectation, yet are still appropriate; the same test input can give different results in different executions. Moreover, agents communicate primarily through message passing instead of method invocation, so traditional testing approaches are not directly applicable; agents cooperate with other agents, so they may run correctly by themselves but incorrectly in a community or vice versa. Defining adequate and effective techniques to test software agents is, thus, a key problem in agent development.

1.2 Contributions

The contributions of this thesis are summarized as methodological contributions with our goal-oriented testing methodology and testing techniques, and practical contributions with a framework for the specification, generation, and execution of test cases.

Goal-oriented testing methodology to deal with Problem 1.
We propose a testing methodology, called Goal-Oriented Software Testing (GOST), that exploits the link between requirements and test cases. We describe the proposed approach with reference to the Tropos software development methodology (Bresciani et al. 2004, Penserini et al. 2007) and consider MAS as the target implementation technology. The proposed methodology contributes to the existing AOSE methodologies by providing: (*i*) a testing process model, which complements the analysis and design activities by drawing connections between goals (e.g., stakeholder goals) and test cases, and (*ii*) a systematic way for deriving test cases from goal analysis models. A case study has been used extensively to illustrate the methodology.

The benefits that the proposed methodology brings are twofold. First of all, since goal-oriented requirements engineering has been recognized as a powerful and effective approach for building today's complex systems, including MAS, drawing straight connections between goal-oriented construction and goal-oriented testing, like GOST does, can save the development cost and avoid the conceptual gap between analysis and testing. In fact, the common approach to this problem is to transform goal-oriented concepts into object-oriented ones and then use them to create test artefacts. This requires additional transformation effort and may create anomalies as well as a conceptual gap between the two types of concepts. Secondly, as GOST proposes to parallel goal-oriented construction and goal-oriented testing, it also helps to discover problems early, avoiding to implement erroneous specifications. Such benefits are well known (Graham 2002) and have been investigated thoroughly in the test-driven (or test-first) development method (Beck 2002).

Testing techniques to deal with Problem 2.

We propose and study different testing techniques to tackle the challenges in testing software agents. Firstly, for evaluating agent behaviours we propose three different approaches: *constraint-based*, *ontology-based*, and *requirement-based*. Agents are autonomous, but in many cases they must respect constraints, norms, or conventions. Constraint violations are considered as faults. Agents communicate with one another via message passing, the exchanged messages are often prescribed by means of interaction ontologies, so interaction ontologies can be used as test oracles to detect faulty behaviours. Stakeholder's requirements, such as those related to safety or performance, can be used as oracles as well.

Secondly, we investigate four different, yet complementary, approaches to the generation of test cases, partially or fully automated: *goal-oriented*, *ontology-based*, *random*, and *evolutionary*. The goal-oriented approach takes goal analysis diagrams, following the GOST methodology, to generate test case skeletons. Then, the expected input/output behaviour is specified manually. The latter three approaches exploit available interaction ontology, domain data, or existing test cases to automatically generate new test cases. The ultimate goal is to test software agents extensively with diverse and challenging scenarios in order to detect faults.

Lastly, we propose a new testing execution method, called *continuous testing*. This method relies on a *tester agent*, which plays the role of human tester, and a monitoring agent network that monitors the system as a whole to track events, changes, misbehaviours, and so on. The tester agent uses the generation techniques, e.g., ontology-based, evolutionary, to generate and execute new test cases against the agents under test, continuously, while the monitoring agent network guards, reports problems (e.g., violations), or record data for desired measurements. Since the behaviour of an agent can change over time due to the mutual dependencies among agents and to their learning capabilities, a single execution of test cases might be inadequate to reveal faults. Continuous testing allows for an arbitrary extension of the testing time, that can proceed unattended and independently of any other human-intensive activity. Existing test cases are evolved and new test cases can be generated automatically, with the aim of exercising and stressing the application as much as possible. The final goal is the possibility to reveal yet unknown faults.

We have conducted many experiments to evaluate the effectiveness of the proposed testing techniques. The results obtained are very promising in terms of fault detection, coverage, and automated generation.

eCAT: a supporting tool.
To support the methodology and the continuous testing method, we have developed a testing framework, called eCAT (eCAT). The framework consists of tools for test case specification and derivation from goal models, for graphical visualization, for continuous execution, and for fault reporting. eCAT is available online at `http://code.google.com/p/open-ecat/`.

1.3 Terminology

The terms related to software testing used in this dissertation comply with the “Standard Glossary of Terms used in Software Testing V.2.0, Dec, 2nd 2007” (*Standard glossary of terms used in Software Testing* 2007). For convenience, this section presents the most used terms.

behaviour	The response of a component or a system to a set of input values and preconditions.
test input	The data received from an external source by a test object during test execution.
test case	A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify the compliance with a specific requirement.
test suite	A set of test cases for a component or system under test.
test scenario	A document specifying a sequence of actions for the execution of a test. Also known as test script or manual test script.
test execution	The process of running a test on the component or system under test, producing actual (a) result(s)
test objective	A reason or purpose for designing and executing a test.
test oracle	A source to determine expected results to compare with the actual result of the software under test. An oracle may be an existing system (for a benchmark), a user manual, or an individual's specialized knowledge, but should not be the code.
test coverage	The degree, expressed as a percentage, to which a specified coverage item has been exercised.

Regarding the goal concept and its related terms, we adopt the definitions used in (Bresciani et al. 2004):

Actor	models an entity that has strategic goals and intentionality within the system or the organizational setting. An actor represents a physical, social or software agent as well as a role or position.
Goal	represents actors' strategic interests. We distinguish hardgoals from softgoals, the second having no clear-cut definition and/or criteria for deciding whether they are satisfied or not.
Plan	represents, at an abstract level, a way of doing something. The execution of plan can be a means for satisfying a goal or for satisfying a softgoal.
Belief	represents actor knowledge of the world.
Resource	represents a physical or an informational entity.

In addition to these definitions, we give definitions of terms and abbreviations used at the end of this dissertation. The reader can refer to the Glossary chapter at ease.

1.4 Thesis structure

The thesis is organized as in Figure 1.1. Chapter 2 surveys recent work on software testing in general and software agents and MAS testing in particular. Chapters 3 and 4 discuss the GOST methodology and a rich set of testing techniques for software agents, respectively. Then, Chapter 5 introduces eCAT, a supporting framework to facilitate software agent developers in defining and executing tests. In Chapter 6, we present three experiments conducted to evaluate the performance of our newly proposed testing techniques and tools. Finally, Chapter 7 concludes our work and discusses future research directions.

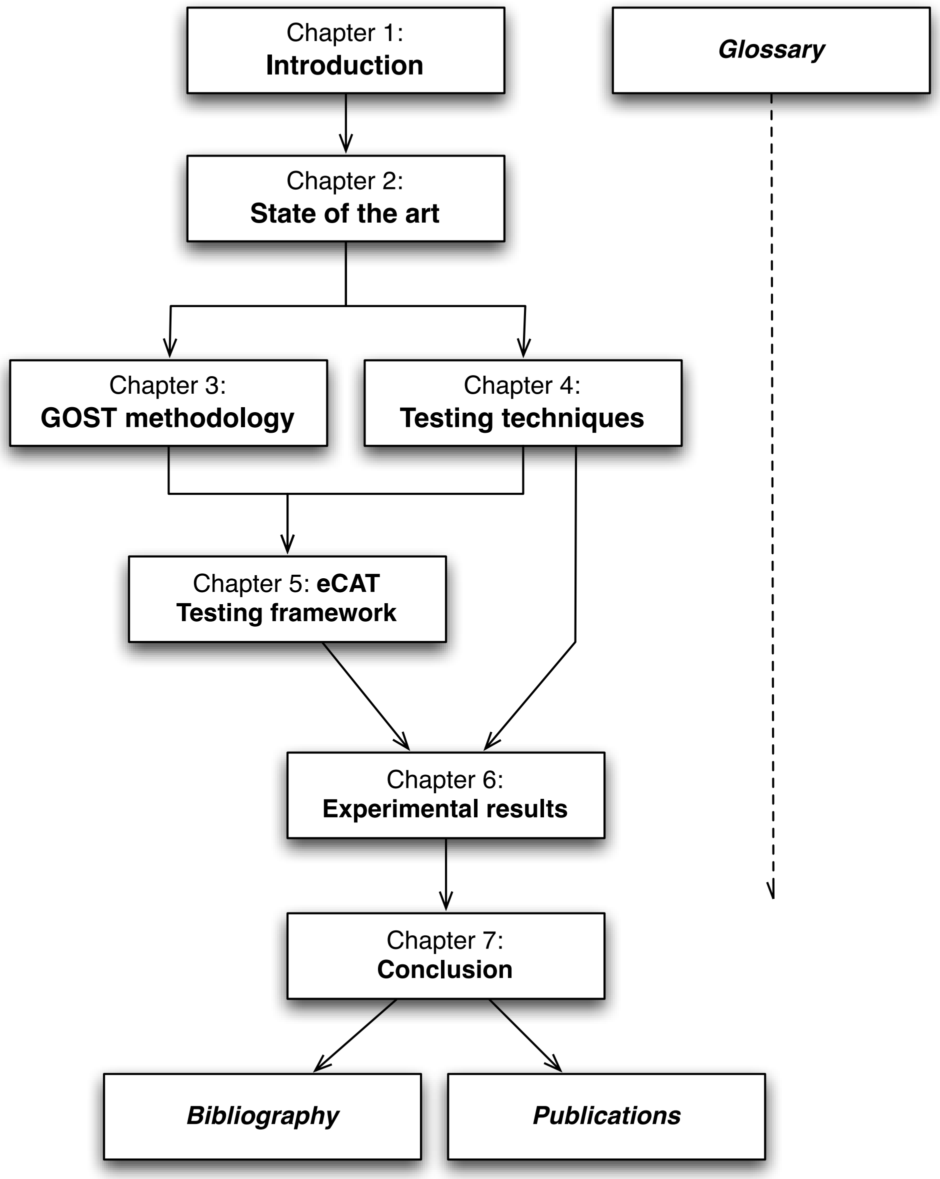


Figure 1.1: Thesis outline

Chapter 2

State of the art

2.1 Software testing

Software testing is a software development activity, aimed at evaluating product quality and improving it by identifying defects and problems. Software testing consists of the dynamic verification of the behaviour of a program on a set of suitably selected test cases (Bourque and Dupuis 2004). Different from static verification activities like formal proofing or model checking, testing involves running specified test cases against the system under test.

Software testing is an important activity that encompasses the whole development and maintenance process (Adrion et al. 1982, Schach 1996). Test design and planning start from the early stage of the requirement process. Testing objective is to find defects in specifications, design artefacts, and implementation. On the other hand, the goal of software testing is also to prevent defects, as it is obviously much better to prevent faults than to detect and correct them because if the bugs are prevented, there is no code to correct. The act of designing tests is known as one of the best bug prevention activities. Tests design can discover and eliminate bugs at every stage in the software construction process (Beizer 1990). Therefore, the idea of "test first, then code" or test-driven is quite widely discussed today (Beck 2002).

To date, several techniques have been defined and used by software developers. One can examine the system without reference to the internal structure of the component or system (black-box testing) or based on an analysis of the internal structure of the component or system (white-box testing). On the other hand, one can design tests for a system based on the analysis of its code (code-based testing) or its specification (specification-based testing) or derive test cases in whole or in part from a model that describes functional aspects of the system (model-based testing). In practice, we often combine different techniques to test a product in order to increase the opportunity of finding defects.

Recently, a new testing technique called Evolutionary testing (ET) (McMinn and Holcombe 2003, Wegener 2005) has been introduced. The technique is inspired by the evolution theory in biology that emphasizes natural selection, inheritance, and variability. Fitter individuals have a higher chance to survive and to reproduce offspring; and special characteristics of individuals are inherited. In ET, we usually encode each test case as an individual; and in order to guide the evolution towards better test suites, a fitness measure is a heuristic approximation of the distance from achieving the testing goal (e.g., covering all statements or all branches in the program). Test cases having better fitness values have a higher chance to be selected in generating new test cases. Moreover, mutation is applied during reproduction in order to generate more diverse test set.

The key step in ET is the transformation from testing objective to search problem, specifically fitness measure. Different testing objective gives rise to different fitness definitions. For example, if the testing objective is to exercise code inside an *if* block, one can define a fitness function that gives lower values (considered as better) to test cases that are closer to make the conditions of the *if* statement to be true; the best value is given to the test cases that make the conditions to be true so that code inside the *if* block will be executed. Once a fitness measure has been defined, different optimization search techniques, such as *local search, genetic algorithm, particle swarm* (McMinn and Holcombe 2003) can be used to generate test cases towards optimizing fitness measure (or testing objective, i.e. finding faults).

2.2 Software agents and MAS testing

Software agents are computational programs that have (among others) the following properties: *Reactivity*, agents are able to sense environmental changes and react accordingly; *Proactivity*, agents are autonomous, in that they are able to choose which actions to take in order to reach their goals in given situations; *Social ability*, that is, agents are interacting entities, which cooperate, share knowledge, or compete for goal achievement.

MAS are systems composed of multiple autonomous agents that interact with one another in an open environment to fulfil their goals, and the goals of the systems as a whole. A MAS is usually a distributed and decentralized system, its agents can be located at geographically-different hosts, and they communicate mainly through message passing. Each host provides a specific environment for the agents located at that host.

Due to those peculiar properties of agents and MAS as a whole, testing them is a challenging task that should address the following issues. (Some of them were stated in (Rouff 2002)):

Distributed/asynchronous. Agents operate in parallel and asynchronously. An agent might have to wait for other agents to fulfil its intended goals. An agent might work correctly when it operates alone but incorrectly when put into a community of agents or vice versa. MAS testing tools must have a global view over all distributed agents besides local knowledge about

individual agents, in order to decide whether the whole system operate accordingly to the specifications. In addition, all the issues related to testing distributed systems are applied in testing software agent and MAS as well, for example problems with controllability and observability (Cacciari and Rafiq 1999).
Autonomous. Agents are autonomous. The same test inputs may result in different behaviours at different runs, since agents might update their knowledge base between two runs, or they may learn from previous inputs, resulting in different decisions made in similar situations.
Message passing. Agents communicate through message passing. Traditional testing techniques, involving method invocation, cannot be directly applied.
Environmental and normative factors. Environment and conventions (norms, rules, laws) are important factors that influence or govern the agents' behaviours. Different environmental settings may affect the test results. Sometimes, an environment provides means for agents to communicate or itself is a test input. One must take into account these factors while dealing with testing.
"Sealed" agents. In some particular cases, agents could be seen as "sealed" in that they provide no or little observable primitives to the outside world, resulting in limited access to the internal agents' state and knowledge. An example could be an open MAS that allows third-party agents to come in and access to the resources of the MAS, how do we assure that the third-party agents with limited knowledge about their intentions behave properly?

Work in testing software agents and MAS can be classified into different testing levels: *unit*, *agent*, *integration*, *system*, and *acceptance*. Here we employ general terminologies rather than using specific ones used in the community, e.g., *group, society*. *Group* and *society*, as called elsewhere, are equivalent to *integration* and *system*, respectively. The testing objectives, subjects to test, and activities of each level are described as follows:

- *Unit.* Test all units that make up an agent, including blocks of code, implementation of agent units like goals, plans, knowledge base, reasoning engine, rules specification, and so forth; make sure that they work as designed.

- *Agent.* Test the integration of the different modules inside an agent; test agents' capabilities to fulfil their goals and to sense and effect the environment.

- *Integration* or *Group.* Test the interaction of agents, communication protocol and semantics, interaction of agents with the environment, integration of agents with shared resources, regulations enforcement; Observe emergent properties, collective behaviours; make sure that a group of agents and environmental resources work correctly together.

- *System* or *Society.* Test the MAS as a system running at the target operating environment; test the expected emergent and macroscopic properties of the system as a whole; test the quality properties that the intended system must reach, such as adaptation, openness, fault-tolerance, performance.

- *Acceptance.* Test the MAS in the customer's execution environment and verify that it meets stakeholder goals, with the participation of stakeholders.

The rest of this section surveys recent and active work on testing software agents and MAS, with respect to these categories. This classification is intended only to help easily understand the research work in the field. It is also worthwhile noticing that this classification is not complete in the sense that some work addresses testing in more than one level, but we put them in the level they mainly focus.

2.2.1 Unit

At the unit level, Zhang et al. (2007) introduced a model based testing framework using the design models of the Prometheus agent development methodology (Padgham and Winikoff 2002). Different from traditional software systems, units in agent systems are more complex in the way that they are triggered and executed. For instance, plans are triggered by events. The framework focuses on testing agent plans (units) and mechanisms for generating suitable test cases and for determining the order in which the units are to be tested. Ekinci et al. (2008) claimed that agent goals are the smallest testable units in MAS and proposed to test these units by means of *test goals*. Each test goal is conceptually decomposed into three sub-goals: *setup*, *goal under test*, and *assert*. The first and last goal prepare pre-conditions and check post-conditions while testing the goal under test, respectively.

Unit testing needs to make sure that all units that are parts of an agent, like goals, plans, knowledge base, reasoning engine, rules specification, and even blocks of code work as designed. Effort has been spent on some particular elements, such as goals, plans. However, fully addressing unit testing in AOSE still opens room for research. An analogy of expected results can be those of unit testing research in the object-oriented development.

2.2.2 Agent

At the agent level, Gómez-Sanz et al. (2008) introduced advances in testing and debugging made in the INGENIAS methodology (Pavón et al. 2005). The meta-model of INGENIAS has been extended to incorporate the declaration of testing, i.e., *tests* and *test packages*. JUnit-based test case and suite skeletons can be generated and it is the developer's task to modify them as needed. The work also provided facilities to access mental states of individual agents to check them at runtime.

Coelho et al. (2006) proposed a framework for unit testing of MAS based on the use of *Mock Agents*. Even though they called it *unit testing* but their work focused on testing roles of agents at agent level according to our classification. Mock agents that simulate real agents in communicating with the agent under test were implemented manually; each corresponds to one agent role. Sharing the inspiration from JUnit (Gamma and Beck 2000) with Coelho et al. (2006), Tiryaki et al. (2006) proposed a test-driven MAS development approach that supported iterative and incremental MAS construction. A testing framework called SUnit, which was built on top of JUnit and Seagent (Dikenelli et al. 2005), was developed to support the approach. The framework allows writing tests for agent behaviours and interactions between agents.

Lam and Barber (2005) proposed a semi-automated process for comprehending software agent behaviours. The approach imitates what a human user, can be a tester, does in software comprehension: building and refining a knowledge base about the behaviours of agents, and using it to verify and explain behaviours of agents at runtime. Although the work did not deal with other problems in testing, like the generation and execution of test cases, the way it evaluates agent behaviours is interesting and relevant for testing software agents.

Núñez et al. (2005) introduced a formal framework to specify the behaviour of autonomous e-commerce agents. The desired behaviours of the agents under test are presented by means of a new formalism, called *utility state machine*, that embodies users' preferences in its states. Two testing methodologies were proposed to check whether an implementation of a specified agent behaves as expected (i.e., conformance testing). In their *active* testing approach, they used for each agent under test a *test* (a special agent) that takes the formal specification of the agent to facilitate it to reach a specific state. The operational trace of the agent is then compared to the specification in order to detect faults. On the other hand, the authors also proposed to use *passive* testing in which the agents under test were observed only, not stimulated like in active testing. Invalid traces, if any, are then identified thanks to the formal specifications of the agents.

2.2.3 Integration

At the integration level, effort has been put in agent interaction to verify dialogue semantics and workflows. The ACLAnalyser (Botía et al. 2004) tool runs on the JADE (Telecom Italia Lab 2000) platform. It intercepts all messages exchanged among agents and stores them in a relational database. This approach exploits clustering techniques to build agent interaction graphs that support the detection of missed communication between agents that are expected to interact, unbalanced execution configurations, overhead data exchanged between agents. This tool has been enhanced with data mining techniques to process results of the execution of large scale MAS (Botía et al. 2006).

Padgham et al. (2005) use design artefacts (e.g., agent interaction protocols and plan specification) to provide automatic identification of the source of errors detected at run-time. A central debugging agent is added to a MAS to monitor the agent conversations. It receives a carbon copy of each message exchanged between agents, during a specific conversation. Interaction protocol specifications corresponding to the conversation are fired and then analyzed to detect automatically erroneous conditions. Ekinci et al. (2008) view integration testing of MAS rather abstract. They considered *system goals* as the source cause for integration and apply the same approach for testing agent goals (unit – according to their view) to test these goals.

Also at the integration level but pursuing a deontic approach, Rodrigues et al. (2005) proposed to exploit social conventions, i.e. norms, rules, that prescribe permissions, obligations, and/or prohibitions of agents in an open MAS to integration test. Information available in the specifications of these conventions gives rise to a number of types of assertions, such as *time to live, role, cardinality*, and so on. During test execution a special agent called *Report Agent* will observe events and messages in order to generate analysis report afterwards.

2.2.4 System and acceptance

At the system level of testing MAS, one has to test the expected emergent and macroscopic properties and/or the expected qualities of the system as a whole. Some initial effort has been devoting to the validation of macroscopic behaviours of MAS. Sudeikat and Renz (2008) proposed to use the system dynamics modelling notions for the validation of MAS. These allow to describe the intended, macroscopic observable behaviours that originate from structures of cyclic causalities. System simulations are then used to measure system state values in order to examine whether causalities are observable.

To the best of our knowledge, there is no work dealing explicitly with testing MAS at the acceptance level, currently. In fact, agent, integration, and system test harnesses can be reused in acceptance test, providing execution facilities. However, as testing objectives of acceptance test differ from those of the lower levels, evaluation metrics at this level, such as metrics for openness, fault-tolerance, adaptivity, demand for further research.

2.2.5 Summary

In summary, most of the contemporary research work on testing software agent and MAS focuses mainly on agent and integration level. Basic issues of testing software agents like *message passing*, *distributed/asynchronous* have been considered; testing frameworks have been proposed to facilitate testing process. However, there is still much room for further investigations, for instance:

- A complete and comprehensive testing process for software agents and MAS.
- Testing MAS at system and acceptance level, how do the developers and the end-users build confidence in autonomous agents?
- Test inputs definition and generation to deal with open and dynamic nature of software agents and MAS.
- Test oracles, how to judge an autonomous behaviour? How to evaluate agents that have their own goals from human tester's subjective perspectives?
- Testing emergent properties at macroscopic system level, how to judge if an emergent property is correct? how to check the mutual relationship between macroscopic and agent behaviours?
- Deriving metrics to assess the qualities of the MAS under test, such as safety, efficiency, and openness.
- Reducing/removing side effects in test execution and monitoring because introducing new entities in the system, e.g., *mock agents tester agents*, and *monitoring agent* as in many approaches, can influence the behaviour of the agents under test and the performance of the system as a whole.

Chapter 3

Goal-oriented testing methodology

3.1 Introduction

The strong connection between requirements engineering and testing is widely recognized (Graham 2002). First, designing test cases early and in parallel with requirements helps discovering problems early, thus avoiding to implement erroneous specifications. Secondly, good requirements produce better tests. Moreover, early test specification produces better requirements because it helps to clarify ambiguities in requirements. The link is so important that considerable effort has been devoted to what is called test-driven (or test-first) development. In such approach, tests are produced from requirements before implementing the requirements themselves (Beck 2002). Software development turns out to be the process of making test cases pass.

Several AOSE methodologies (Henderson-Sellers and Giorgini 2005) have been proposed so far. In terms of testing and verification, while some consider specification-based formal verification (e.g., *Formal Tropos* (Fuxman et al. 2004, Perini et al. 2003) and (Dardenne et al. 1993)), other borrow Object-Oriented (OO) testing techniques, taking advantage of a mapping of agent-oriented abstractions into OO constructs (e.g., *PASSI* (Cossentino 2005) and *INGENIAS* (Pavón et al. 2005)). However, a structured testing process for AOSE methodologies is still absent.

In this chapter, we propose a testing methodology, called GOST, that exploits the link between requirements and test cases, following the V-Model (Development Standards for IT Systems of the Federal Republic of Germany 2005). We describe the proposed approach with reference to the Tropos software development methodology (Bresciani et al. 2004, Penserini et al. 2007) and consider MAS as the target implementation technology. Similar to object-oriented approaches in which test cases are derived from use-case requirements models, we investigate how to derive test cases from goal-oriented Tropos requirements models.

Specifically, the proposed methodology contributes to the existing AOSE methodologies by providing: (*i*) a testing process model, which complements the development methodology by drawing a connection between goals and test cases and (*ii*) a systematic way for deriving test cases from goal analysis.

It is worth noticing that differently from goal-oriented test generation in the context of coverage testing, i.e., generation of test inputs to achieve a coverage goal, such as branch coverage (Gotlieb et al. 2007), the goal-oriented software testing methodology proposed in this chapter aims at exploiting goal analysis to derive systematically test suites and using the achievement of goals, e.g., stakeholder goals, system goals, as criteria for testing. Inversely, the derived test suites provide feedback useful for refining the analysis, design, and code artefacts to detect and solve problems as early as possible.

3.2 Tropos methodology background

Tropos is an agent-oriented software engineering methodology (Bresciani et al. 2004, Penserini et al. 2007) that adopts a requirement-driven approach, that is system requirements are derived from a deep model of the problem domain, called *Early Requirements* model, in which the stakeholders, their goals and the social dependencies among them for goal achievement are made explicit (see Table 3.1). System requirements are then derived from an analysis of the goals that domain stakeholders will delegate to the intended system. This is modelled in the so-called *Late Requirements* model, which is the input of the following design phases. In particular, in the *Architectural Design* phase, candidate system architectures are derived and analyzed against non-functional requirements (or quality factors). In the *Detailed Design* phase the system specification is further detailed, taking into account the target implementation platform. In case of MAS, system actors are defined in terms of agent roles and specifications of agent communication and coordination protocols are given.

The Tropos methodology provides a conceptual modelling language based on the $i*$ framework (Yu 1995), including a diagrammatic notation to build views of the model and goal analysis techniques. Basic constructs of the language are those of actor, goal, plan, softgoal, resource, and capabilities. Dependency links between pairs of actors allow to model the fact that one actor depends on another in order to achieve a goal, execute a plan, or acquire a resource and can be depicted in *actor diagrams*.

Goals are classified into hardgoals and softgoals; the latter has no clearcut definition and/or criteria as to whether they are satisfied. Softgoals are particularly useful to specify non-functional requirements. Goals are analyzed from the owner actor perspective through *AND, OR* decomposition; *means-end* analysis of *plans* and *resources* that provide means for achieving the goal (the end); contribution analysis that points out hardgoals and softgoals that contribute positively or negatively to reaching the goal being analyzed.

A modelling tool is provided to support a model-driven development

Table 3.1: Tropos development process by phases and output artefacts.

Phase	**Description**	**Output artefact**
Early Req. (ER)	The organizational settings where the system-to-be will operate and the relevant stakeholders are identified during this stage.	Domain model (i.e. the organizational setting, as is). Stakeholders are represented as actors while their objectives are represented as goals, specified in terms of ER Actor and Goal Diagram — e.g., Fig. 3.1
Late Req. (LR)	The system-to-be is introduced as a new actor with its new dependencies with existing actors that indicate the obligations of the system towards its environment as well as what the system can expect from existing actors in its environment.	Model of the system-to-be where system requirements are modelled in terms of system goals, by means of LR Actor and Goal Diagrams — e.g., Fig 3.2
Archit. Design (AD)	More system actors are introduced. They are assigned to subgoals or goals and tasks (those assigned to the system as a whole). The implementation platform is chosen, allowing designers to reuse existing design patterns.	System architecture model, specified in terms of a set of interacting software agents in an AD Actor Diagram — e.g., Fig. 3.3
Detailed Design (DD)	System actors are defined in further detail, including specification of communication and coordination protocols. Plans are designed in detail using existing modelling languages like UML or AUML (Odell et al. 2000).	Specification of software agent roles, capabilities, and interactions, by means of Activity and Sequence Diagrams — e.g., Fig. 3.4, Fig. 3.7
Implementation	The Tropos specification, produced during detailed design, is transformed into a MAS code skeleton. This is done through a mapping from the Tropos constructs to those of a target-programming platform, such as JADE (Telecom Italia Lab 2000).	MAS skeleton code and implementation documents.

process (Perini and Susi 2005) in which requirements models are refined into design models, from which agent code skeleton can be automatically derived (Penserini et al. 2007).

The Tropos methodology determines the basic requirements for the GOST approach. Indeed, it uses the notion of agent and all related mentalistic notions, in particular the concept of goal, in all phases of software development, from early analysis down to implementation, providing goal-oriented specification and code. Moreover, Tropos provides a structured, tool-supported process, which is organized along five main phases, each one producing a specific set of modelling artefacts, as recalled in Table 3.1.

3.3 Motivating example

To illustrate the GOST methodology, we introduce a multi-agent system that is composed of several cleaning agents working at an airport. This software could be deployed on a physical platform composed of a set of moving robots. We name this system *Mr. Cleaners. Mr. Cleaners* are in charge of keeping the airport clean; agents in the system have to collaborate to optimize their work and be nice with passengers.

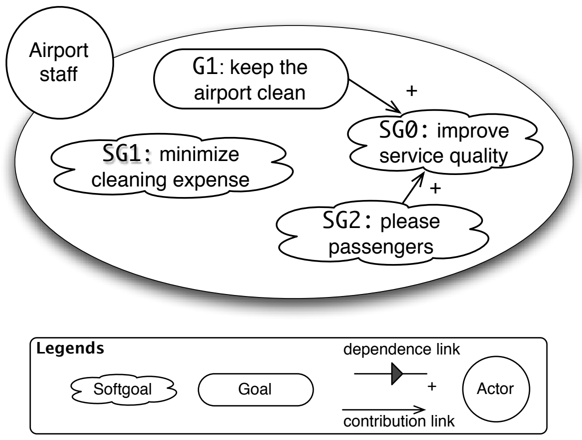


Figure 3.1: Early requirements for Mr. Cleaners

Following the guidelines of Tropos (Bresciani et al. 2004), we do the *early requirements* analysis and identify stakeholders' goals associated with *Mr. Cleaners* (see Figure 3.1) [1]. There are two top softgoals that the airport wants to reach: ***SG1**: minimize-cleaning-expense* and ***SG0**: improve-service-quality*. To reach the latter, two other sub-goals need to be fulfilled: ***G1**: keep-the-airport-clean* and ***SG2**: please-passengers*. There could be more goals that the airport wants to achieve, but we consider only these goals to keep the example simple and understandable.

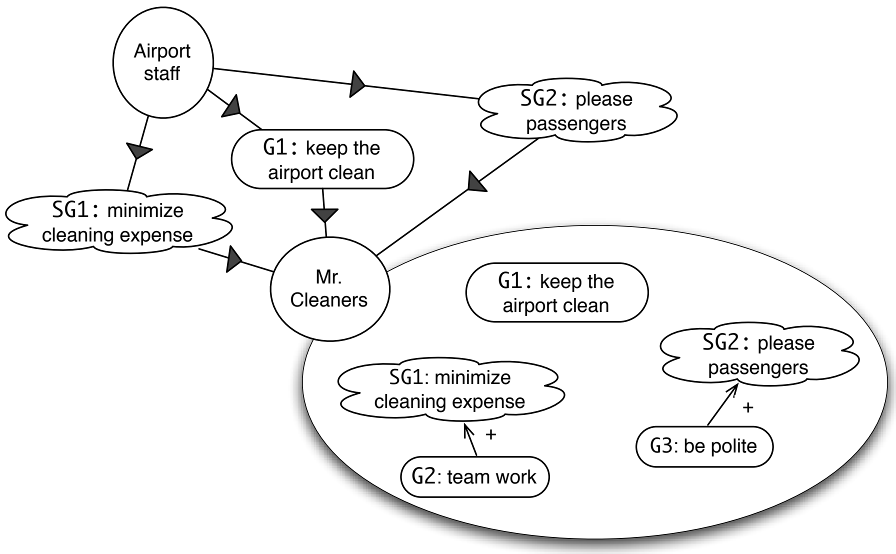

Airport staff
SG2: please passengers
G1: keep the airport clean
SG1: minimize cleaning expense
Mr. Cleaners
G1: keep the airport clean
SG2: please passengers
SG1: minimize cleaning expense
+
G2: team work
+
G3: be polite

Figure 3.2: Late requirements for Mr. Cleaners

Figure 3.2 shows the late requirements analysis for *Mr. Cleaners*. The airport staff delegates three goals SG1, SG2, and G1 to the multi-agent system under construction, *Mr. Cleaners*. At a high-level view, the system adds two hardgoals: ***G2**: team-work* and ***G3**: be-polite* in order to reach SG1, SG2, as required by the airport staff. *Mr. Cleaners* must achieve all the three hardgoals.

[1] An analysis of the alternative ways to fulfilling stakeholder strategic goals is usually done in Tropos Early Requirements. See, for instance, (Perini 2009) for an example of how this step is performed on a cleaning robots scenario.

Moving on from the late requirements analysis, system actors are added in the architectural design of *Mr. Cleaners*. In this example, system actors are the cleaning agents. Goals of the system G1, G3, G2 are delegated to the agents.

Figure 3.3 depicts the architecture system as a whole, showing (for example) three cleaning agents. Notice that at the deployment time the number of agents will be determined by the number of available robots. The mutual goal dependency G2 represents the fact that the peer agents will coordinate to better achieve the system goal SG1 and will reflect into individual agent goals. Moreover, the agents share resources, namely *recharging-stations*, *waste-bins*, *wastes*, and *obstacles*, and knowledge about them. The internal architectural design of the cleaning agent is described in Figure 3.4.

Finally, Figure 3.4 shows the architectural design of the cleaning agent. A number of goals and plans (tasks) are assigned to the agent. At the highest level there are four root goals: *G2: team-work, G4: maintain-battery, G3: be-polite*, and *G1: keep-the-airport-clean.* G1, G2, G3 are delegated from the system, while G4 is the agent own goal to keep the agent alive.

These goals are, then, decomposed into sub-goals. For instance, *G4: maintain-battery* is *AND*-decomposed into two sub-goals *G4.1: query-charging-location*, and *achieve-move-to-a-location. AND* decomposition requires all sub-goals to be achieved to obtain the achievement of their root goal. Plans are lastly added to the design as means to achieve goals. The detailed design of plans can be done following guidelines described in (Bresciani et al. 2004, Penserini et al. 2007). An example is given in Figure 3.7.

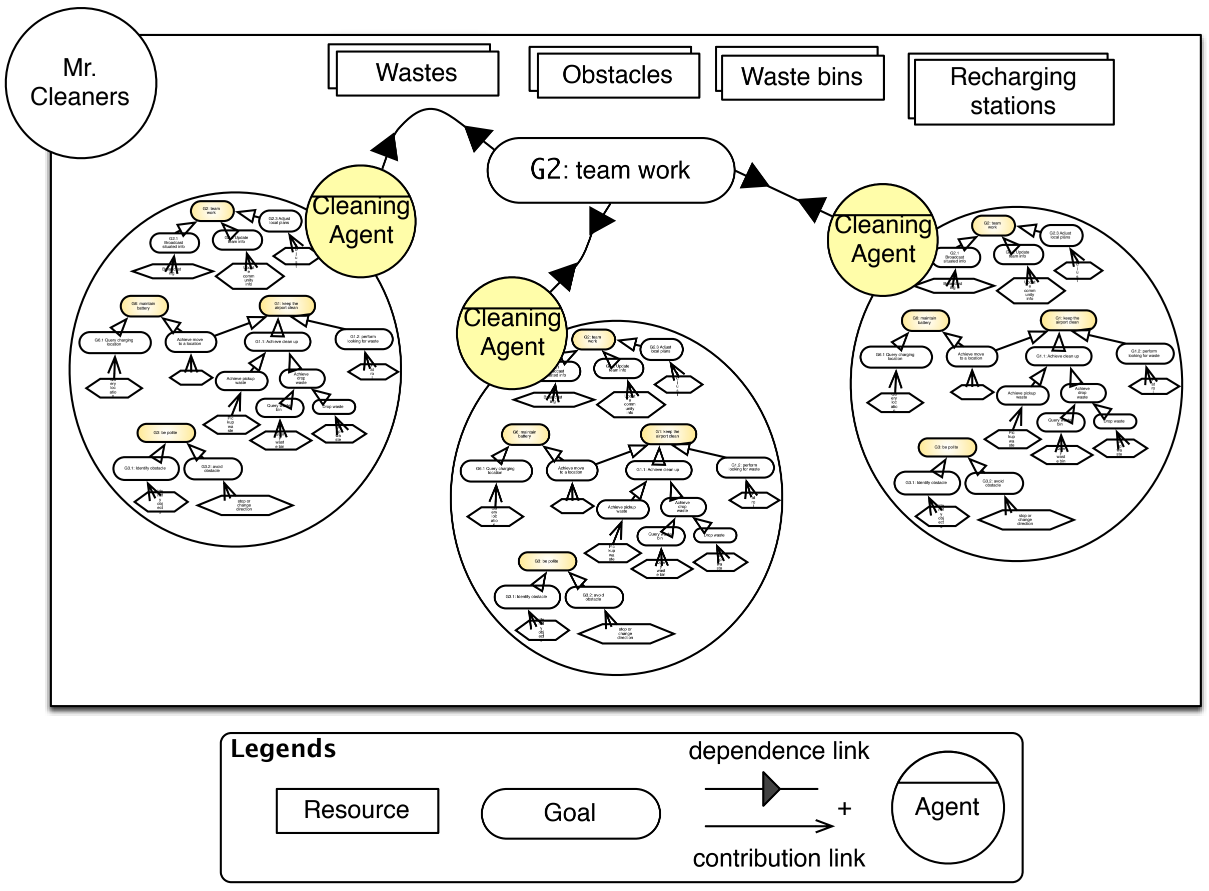


Figure 3.3: Architecture of Mr. Cleaners

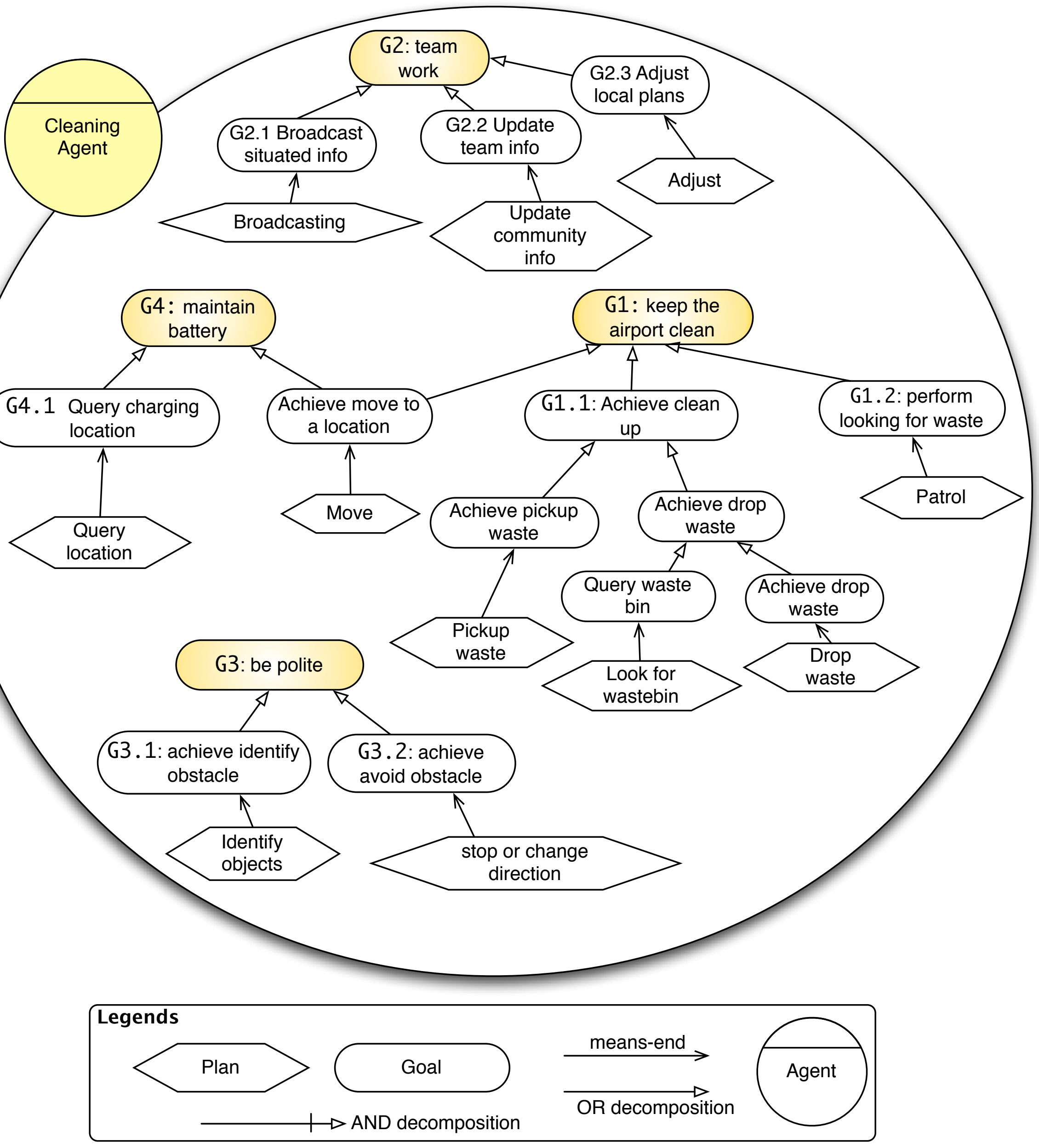


Figure 3.4: Architectural design of the cleaning agent

3.4 Methodology

This section presents the proposed methodology. We discuss different goal types, testing types, a testing process model. The relationships between goal types and testing levels are presented with reference to the process. Finally, we discuss how to derive systematically test cases from goal models.

3.4.1 Goal types

Different perspectives give different goal classifications. For instance, (Dastani et al. 2006) classify agent goals in agent programming into three categories, namely *perform*, *achieve*, and *maintain*, according to the agent's attitude toward them.

We use a general perspective on goals, but not from a specific subject (e.g., agent), to classify them based on the Tropos software engineering process. Goals are classified into the following types according to the different phases of the process:

Type	Descriptions
Stakeholder	goals that represent stakeholder objectives and requirements towards the system to-be. This type of goal is mainly identified at the early requirements phase of Tropos.
System	goals that represent system-level objectives or qualities that the system to-be has to reach or provide. For instance, goals that are related to performance, openness of the system as a whole are system goals. This type of goal is mainly specified at the late requirements phase of Tropos
Collaborative	goals that require the agents of the system to-be to cooperate or share tasks, or goals that are related to emergent properties resulting from interactions. This type of goal can be called also as *group* goal, and they often appear at the architectural design phase of Tropos
Agent	goals that belong to or are assigned to particular agents. This type of goal appears when designing agents.

Let's go back to our motivating example in Section 3.3. Goals shown in

Figure 3.1 (G1, SG0, SG1, SG2) are stakeholder goals while those inside the balloon presented in Figure 3.2 are system goals; these goals capture the strategic and system-level objectives of the airport regarding *Mr. Cleaners*. The goal G2: *teamwork* in Figure 3.3 is a collaborative goal of the cleaning agents. Finally, all goals presented in Figure 3.4 are of agent goal type.

Different goal types are related to different testing scopes and test evaluation methods. [2] The next sections discuss testing types and the mapping between goal types and testing types.

3.4.2 Testing levels

We propose to divide the MAS testing process into different levels to better focus on the specific problems that may occur at each level. The five testing levels being proposed are: *unit, agent, integration, system*, and *acceptance*. Details are as follows:

Level	What to test
Unit	test code units and modules that make up agents like goals, plans, beliefs, sensors, reasoning engine, and so on.
Agent	test the integration of the different modules inside an agent; test agents' capabilities to fulfil their goals and to sense and effect the environment.
Integration	test the interaction of agents, communication protocol and semantics, interaction of agents with the environment, integration of agents with shared resources, regulations enforcement; observe emergent properties; make sure that a group of agents and environmental resources work correctly together.
System	test the MAS as a system running at the target operating environment; test for quality properties that the intended system must reach, such as adaptation, openness, fault-tolerance, performance.
Acceptance	test the MAS in the customer execution environment and verify that it meets the stakeholder goals, with the participation of stakeholders.

[2]Notice that to keep notation simple we do not change the labels of the goals while changing scope, namely actors in Tropos. A more complete notation, for instance for the G1 goal will be the following: ${}^{St}G1$, ${}^{Sys}G1$, ${}^{A}G1$ to refer to G1 as stakeholder, system or agent goal respectively.

3.4.3 A process model for goal-oriented testing

The V-Model (Development Standards for IT Systems of the Federal Republic of Germany 2005) proposes a system development process, which defines a parallel flow of testing activities with respect to construction activities. The upper branch of the **V** (see Figure 3.5, turn this figure on end to see the V) represents the construction activities, and the lower branch of the **V** represents the testing flow where the application is tested against the artefacts defined on the upper-branch. The main trait of the V-model is that it represents explicitly the mutual relationships between construction artefacts and testing artefacts.

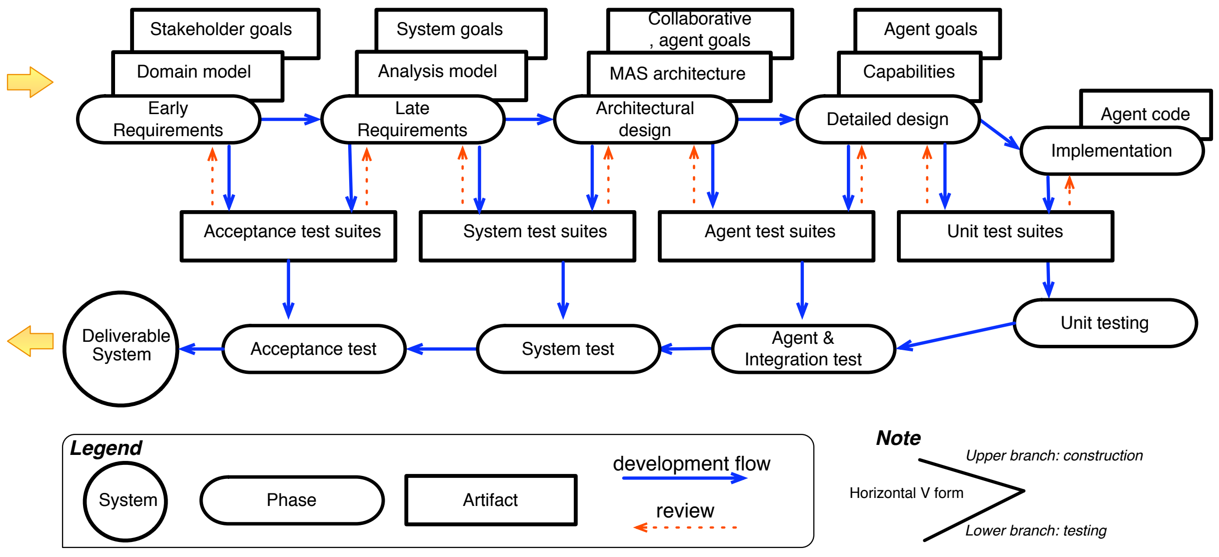


Figure 3.5: V process model for goal-oriented testing

Tropos guides the software engineers in building a conceptual model, which is incrementally refined and extended, from an early requirement model to system design artefacts and then to code, according to the upper branch of the **V** depicted in Figure 3.5. We integrate testing in Tropos by defining the lower branch of the **V** and by providing a systematic way to derive test cases from Tropos modelling artefacts, i.e. from the upper branch of the **V**, in Figure 3.5.

The modelling artefacts produced along the development process are:

Early Requirements model:	a domain model (i.e. the organizational setting, as is)
Late Requirements model:	a model of the system-to-be where system requirements are modelled in terms of system goal graph
Architectural Design model:	a system architecture model, specified in terms of a set of interacting software agents
Detailed Design model:	a specification of software agent roles, capabilities, and interactions
Implementation artefacts:	agent code and implementation documents

With those artefacts come *stakeholder* goals, *system* goals, *collaborative* goals and *agent* goals, respectively. These goals provide valuable testing objectives and input data. For instance, the stakeholder goals in Figure 3.1 are requirements and criteria for the acceptance test of *Mr. Cleaners*: the airport accepts *Mr. Cleaners* only when it reaches three goals G1, SG1, and SG2 (hence achieving SG0)

Figure 3.5 depicts the relationships between different types of goal (also modelling artefacts) and different testing levels as vertical flows from the upper branch of the **V** to its lower branch. In particular, domain model, stakeholder goals, and system goals are used to derive acceptance test suites. Stakeholder goals, analysis model and system goals are used to conduct system test, and so forth. In other words, based on the outputs of the first two phases, developers derive acceptance test suites; using the outputs of the *Late Requirements* and *Architectural Design* phases, developers derive system test suites to test the system as a whole, and so forth.

The derivation of test suites takes place at the same time as the system is constructed, thus helping refine back the system analysis and design to uncover omissions and defects early, in accordance with the test-first development approach (Beck 2002). The benefits of designing test early in software development have been discussed in (Graham 2002). The review flows (dotted bottom-top arrows) in the **V** illustrate these activities. For example, while deriving test cases for an agent at agent level, one might uncover a problem with the agent design that there is no means (no plan) to achieve a goal. Thus, the design has to be revised.

The reason for the use of artefacts of two phases to derive one testing type, for instance *Architectural and Detailed Design* to derive integration test, is that the artefacts of the former phase (e.g., *Architectural Design*) give a broader view to plan the tests, while the latter phase (e.g., *Detailed Design*) provides necessary materials to create test cases (e.g., information about actual test data).

3.4.4 Test suite derivation

This section introduces in details guidelines to derive test suites according to the proposed **V** process model. The guidelines contain four parts, as illustrated in Figure 3.6. First, we discuss how to derive test suites for acceptance test from organizational and system goals. Second, we discuss how system, collaborative, and agent goals are used to create system test suites. Next, as we move on in the development process to the agent interaction and capability design, we show how to exploit collaborative and agent goals to create integration test suites. Finally, we discuss in depth how to create test suites for agent plans, goals, and agents themselves. Examples are given in each part to illustrate the derivation. In addition, we also discuss when the derivations take place, when test suites are executed, and goal-oriented test adequacy at each test level.

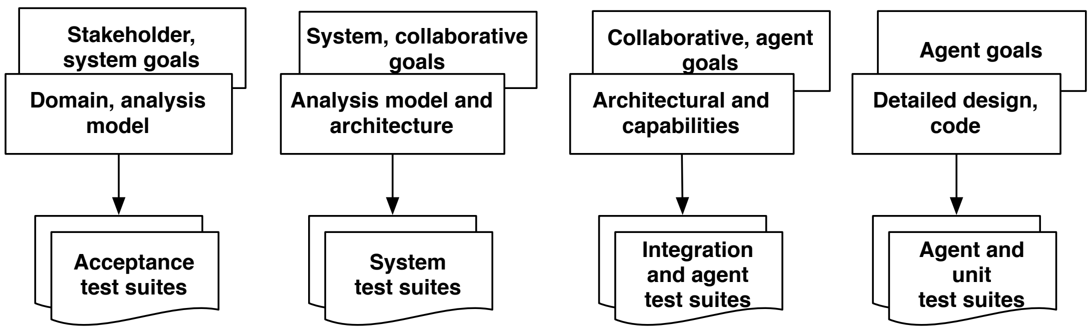

Stakeholder, system goals
Domain, analysis model
Acceptance test suites
System, collaborative goals
Analysis model and architecture
System test suites
Collaborative, agent goals
Architectural and capabilities
Integration and agent test suites
Agent goals
Detailed design, code
Agent and unit test suites

Figure 3.6: Test suite derivation

3.4.4.1 Acceptance test

Acceptance test suite derivation takes place at the *Late Requirements* phase, in parallel with the system analysis. At this stage, we have identified: actors, actors' goals, and dependencies between actors. Actors in the organizational setting include stakeholders, identified at *Early Requirements* phase, and system actors. Stakeholder actors present their intentions to the system actors by goal dependencies: they delegate goals to the system actors. In general, these goals represent users' objectives and intentions with regard to the system-to-be, so the fulfilment of these goals is a pivotal benchmark to the system acceptance. Thus, we will use them as foundations for acceptance test suites.

Acceptance test suite derivation consists of the following steps:

```
1:  for all actor ∈ {stakeholder actors} do
2:    for all $g$ ∈ {actor's goals} do
3:      analyze the goal decomposition/contribution tree of $g$
4:      for all $lg$ ∈ {leaf goals of the decomposition/contribution tree} do
5:        /* create a test suite for $lg$ */
6:        step1: identify operational or usage scenarios related to $lg$
7:        step2: identify fulfilment criteria (oracle) for each scenario
8:        step3: create one test suite with at least one test case for each
          scenario
9:      end for
10:   end for
11: end for
```

The procedure reads: for each stakeholder actor identified in the early and late requirements phases, a set of goals that the actor delegates or depends on the system is identified. (These goals are analyzed by means of decomposition or contribution analysis; and the results are goal decomposition/contribution trees inside system actors.) Then, for each of these goals, we have to read the corresponding analysis goal tree to identify the leaf goals of the tree, and finally to create a test suite for each leaf goal (step 1, 2, 3).

The analysis of each system actor consists of goal decomposition/contribution trees, in that, goals can be decomposed into sub-goals, and sub-goals are means to achieve or to contribute to the goals. According to the introduced steps, we analyze the goal trees and create a test suite for each leaf goal, each test suite contains a set of test cases corresponding to the scenarios identified. The operational and usage scenarios and the oracle depend on the problem domain, but they often need agreements from both sides:

customer and development team. Both work together to define these scenarios. Finally, the fulfilment of *actor*'s goals can be reasoned on the basis of the fulfilment of the leaf goals and the goal decomposition/contribution trees.

Regarding the motivating example, to make it simple we have identified only one stakeholder actor: the *airport staff*; this actor delegates three goals SG1, G1, SG2 (see Figure 3.2) to *Mr. Cleaners*. Based on goal models specified in the first two phases *Early* and *Late Requirements*, we identify three leaf goals that give rise to three acceptance test suites, following the steps described above. Each test suite can have several test cases. Table 3.2 summarizes the descriptions of the test suites.

The test scenarios presented in Table 3.2 are abstract, and we keep them so to make our example simple. In reality they should be specified in much more details. For instance, for the scenario of the test case *ATC1.1*, we could specify it as follows: "The checking area 3 - Terminal 5 - Heathrow airport is used for acceptance test. It is a rectangle of 10 x 20 metres that we consider with gates upfront. At positions (2, 2), (4, 5), (10, 15), (10, 16) (positions are expressed in metres assuming a South-North orientation of the area), we put the following waste: 1 towel and 1 plastic glass at (2, 2); 2 newspapers at (4, 5); 120ml of soft drink at (10, 15); dust (100g) at (10, 16), within a 0.5 meter circle. *Mr. Cleaners* is put at position (1, 1) and is switched on by a staff member of the airport. It is left alone, cleaning the area for half an hour. Then, it is switched off by a staff member of the airport."

The derived acceptance test suites can be used for two distinctive objectives: *(i)* refining the analysis model, and *(ii)* acceptance test. The first objective is realized during acceptance test suite derivation. By using derived suites to review the specification, one could point out problems with the analysis goal model, such as decomposition, unsatisfiability, ambigui-

Table 3.2: Acceptance testing: test suites derived for *Mr. Cleaners*

TS	**Stakeholder Goal**	**TC**	**Scenario**	**Oracle**
ATS1	*G1: keep the airport clean*	ATC1.1	given an actual area of the airport (**A** for short), wastes are placed at specified positions $(p_1, p_2, \ldots, p_n)$, the amount of waste is $(a_1, a_2, \ldots, a_n)$, respectively. *Mr. Cleaners* is in charge of cleaning that area	the area will be cleaned in less than t minutes
		ATC1.2	area **A** has wastes that are repeatedly thrown into it in a random manner	the area is periodically cleaned
		ATC1.3	depending on the time at the airport, area **A** can be more or less dirty: the amount of waste is a function of time and position (e.g., $w = f(t, p)$)	*Mr. Cleaners* adapts its cleaning interval and focal positions
ATS2	*G2: teamwork*	ATC2.1	agents of *Mr. Cleaners* work together in area **A**	the agents do not overlap their cleaning areas
		ATC2.2	there two recharging stations $(X_1, X2)$ in **A**	there is no conflict with regard to the recharging stations
ATS3	*G3: be polite*	ATC3.1	while the cleaning agents are moving or cleaning in area **A**, there are N humans moving in the area along different directions	the cleaning agents stop moving/working and nod their heads to say hello when they meet a human
TS: test suite, **TC**: test case				

ties, implicit assumptions, inconsistencies (e.g., a goal cannot be fulfilled or a hardgoal somehow contributes to a softgoal both positively and negatively), and so forth. Problems pointed out at this stage could substantially reduce development effort, since they can be solved before implementation. On the other hand, the second objective requires the system to be built. At this time, derived test suites are used by the customer to evaluate the delivered system to decide eventually whether the system is ready to be deployed or it needs further improvement.

The basic requirement for the system acceptance (all the derived test suites are passed) entails that all the goals of all the stakeholder actors are achieved or satisfied.

3.4.4.2 System test

The transition from *Late Requirements* to *Architectural Design* phase consists of identifying agents that realize the specified system actors, assigning system actors' goals (called system goals) to agents goals, and projecting system actors' dependencies to agents dependencies and interactions. At this stage, apart from the artefacts (actors, goal models) obtained from the *Late Requirements* phase, there are agents, their goals, roles, collaborative goals, agents' dependencies for goals, resources, the dependencies between agents and the environment, regulations, constraints, and so forth. System test suites should consider and make use of these artefacts.

System tests suite derivation takes place in parallel with architectural design. Similar to acceptance test suite derivation where we take stakeholder actors' goals as foundation concepts, we use system actors' goals as foundations to create system test suites as they provide the system-level objectives and requirements. When the system as a whole is built so that the system actors' goals (including functional hardgoals and quality softgoals) are fulfilled, it is ready to be passed to the customer for acceptance test.

System test suite derivation consists of the following steps:

```
1: for all $actor \in$ {system actors} do
2:   for all $g \in$ {$actor$'s leaf goals} do
3:     /* create a test suite for $g$ */
4:     step1: identify which agent(s) realize(s) $g$
5:     step2: analyze the goal model of each agent to identify goals
           that represent the achievement of $g$
6:     step3: identify environmental factors, pre-conditions, inputs
           that facilitate or trigger $g$
7:     step4: identify fulfilment criteria (oracle) for $g$
8:     step5: create one test suite with at least one test case for $g$
9:   end for
10: end for
```

The procedure is described as follows: for each system actor, the goal analysis of the actor is analyzed to filter the leaf goals. For each leaf goal g of a system actor, one has to create a test suite to test the achievement of the goal. Five creation steps are: (1) identifying which agent(s) realize(s) the goal g, (2) analyzing the goal model of each agent to identify goals related to the achievement of g, (3) identifying environmental factors, pre-conditions, inputs that facilitate or trigger g, (4) identifying fulfilment criteria for g, (5) creating a test suite having a set of test cases for g that take inputs and oracles identified from previous steps.

Since system actors can have more goals than those delegated to the system by stakeholder, the number of system test suites is usually higher than the number of acceptance test suites. Moreover, at this stage the system is designed, so more detailed information is available. As a consequence, system test suites can reuse information from acceptance test suites, but much more details can be added, such as fulfilment criteria for goals and expected behaviours of agents involved.

Let's consider again the motivating example. To create system test suites for *Mr. Cleaners* we start analyzing the *Mr. Cleaners* actor (Figure 3.2) and figure out that we need to test three goals: G1, G2, G3. Next, based on the architectural design of *Mr. Cleaners* and the cleaning agents, we identify which agent goals to test and which resources of the environment to set up. This identification can be straightforward based on goal identifiers, like in the case of the goal G2, G3, but it may require further analysis, when the transition from system actors' goals to agents' goals is not explicit, as for the goal G1. In this case, an external knowledge about problem domain described in analysis documents has been used. Table 3.3 and 3.4 describe system test suites that we derived for *Mr. Cleaners*. The former shows the goal realization mapping between *Mr. Cleaners* actor and the cleaning agent, while the latter describes some test cases that are created for each system actor's goal, accordingly.

Table 3.3: System testing: test suites derived for *Mr. Cleaners*

TS	**System goal**	**Agent**	**Agent goal**
STS1	*G1: keep the airport clean*	Cleaning Agent	*G1: keep the airport clean*
			G4: maintain battery
			G2: teamwork
STS2	*G2: teamwork*	Cleaning Agent	*G2: teamwork*
STS3	*G3: be polite*	Cleaning Agent	*G3: be polite*

As apparent from Table 3.4, the test case *STC1.3* has an undefined test oracle (G*) with respect to the cleaning agent, because in the design of the cleaning agent, there is no goal or plan that aims at adapting the behaviour of the agent according to the amount of waste and time. This is a clear indication that we have to further refine the design of the cleaning

Table 3.4: System testing: examples of test suite derived for *Mr. Cleaners*

TS	**System goal**	**TC**	**Scenario**	**Oracle**
STS1	*G1: keep the airport clean*	STC1.1	similar to the scenario of the acceptance test case ATC1.1, the testing area **A'** considered is located in the development site	the cleaning agent must fulfil two agent goals G1, G2 and maintain G4 within the required time
		STC1.2	similar to ATC1.2, realized on **A'**	the cleaning agent must fulfil the agent goals G1, G2 in a periodical manner, it has to maintain the goal G4
		STC1.3	similar to ATC1.2, on **A'**	the cleaning agent must achieve the agent goals G1, G2, G4, so as to adapt its cleaning interval depending on the amount of waste. This adaptation can be associated to a goal G*.
STS2	*G2: teamwork*	STC2.1	similar to ATC2.1, on **A'**	the cleaning agent must achieve the agent goal G2
		STC2.2	similar to ATC2.2, on **A'**	the cleaning agent must achieve the agent goal G2
TS: test suite, **TC**: test case				

agent. For example, one can add a goal, namely, *changing workload* to the agent design, decompose it, and so forth. Nevertheless, this example demonstrates that we can detect problems, such as under specifications or implicit specifications, quite early, thanks to test suite derivation. Indeed, system test suites are used first to refine the system design and detect design problems early; and later, to perform system test.

3.4.4.3 Integration test

The aim of integration testing is to make sure that agents work together correctly – sharing tasks and resources – to achieve collaborative or agent goals. To obtain this objective, we consider dependencies between agents for collaborative goals and dependencies between agents and resources. In fact, these dependencies are sources that lead to interactions, i.e. agent-agent and agent-environment interactions. We can use them to derive test suites that exercise these dependencies and then evaluate the result of the interactions.

Integration test suite derivation takes place once we have finished detailed design, so that we can make use of the interaction protocol design. The derivation for collaborative goals consists of the following steps:

```
1: for all $g \in$ {collaborative goals} do
2:   /* create a test suite for $g$ */
3:   step1: identify agents involved
4:   step2: identify interaction scenarios
5:   step3: identify interaction protocols, ontologies
6:   step4: identify fulfilment criteria (oracle) for each scenario
7:   step5: create a test suite for each scenario
8: end for
```

The procedure reads: in the architectural design of the system we identify a set of collaborative goals. For each of these goals we identify agents that are involved, interaction scenarios, protocols, and ontology. Then, we identify fulfilment criteria for the goal. Finally, for each scenario we can define a test suite making use of data identified, i.e. agents, protocols, criteria, and so on.

For example, *G2: teamwork* is a collaborative goal that involves all the cleaning agents. When we go further into the detailed design of the agent, in Figure 3.4, we determine two interaction scenarios: (1) one cleaning agent broadcasts information about its location; and, (2) the agent receives a message broadcast from another cleaning agent. Let's consider scenario (1), Figure 3.7 shows the detailed design of the scenario: first, an agent sends a request to the Directory Facilitator (DF) (FIPA 2004) to get the addresses of other cleaning agents. Once a list of agents is returned, the agent broadcasts a message containing situated information to all the agents in the list. In order to test this scenario, we create the test case described in Table 3.5.

Table 3.5: Integration testing: a test case derived for *G2: teamwork*

Test scenario	1. instantiate two cleaning agents working together
	2. monitor the communication between these two agents and between each of them and the DF (FIPA 2004)
Oracle	1. the two agents register themselves with the DF
	2. the two agents send requests to the DF
	3. the two agents send messages to each other
	4. the content of the messages is valid

Testing the integration of agents with the operating environment consists of testing their perception and affecting capabilities. That is, we need to make sure that the agents under test are able to perceive changes regarding the resources they are interested in. We test whether they can affect such resources properly. The following steps guide us when deriving

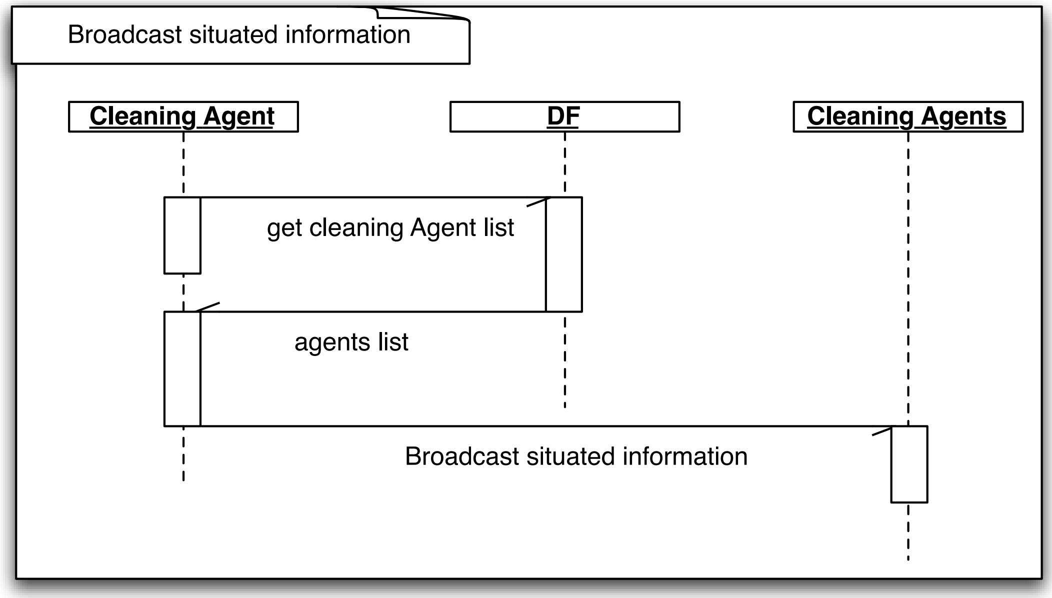


Figure 3.7: Broadcasting situated information protocol

test suites for testing the agent-environment interaction:

```
1: for all agent do
2:    step1: identify related resources
3:    step2: identify integration scenarios
4:    step3: identify access policy, interaction protocol, ontology,
      and other related factors if any.
5:    step4: identify fulfilment criteria (oracle) for each scenario
6:    step5: create one test suite for each scenario
7: end for
```

The procedure is described as follows: for each agent type in the system we identify resources that the agents of the type use. Then, we identify usage or interaction scenarios, access policies, protocols, and other related factors. Finally, we define criteria for each scenario and create a test suite for it, making use of the data identified.

Another aim of integration testing is to observe emergent properties resulting from agent interactions. Testing for emergence consists of making sure that all the involved agents respect predefined rules and that the expected group behaviours or patterns are actually observed. Test suites created for this objective should focus on providing necessary environment, so as to facilitate the agent interaction under test, and on enforcing the rules that govern the behaviour of the agents under test. Moreover, test oracle for emergence involves human observation and common perspective because different observers, having no shared perspective, may see the testing outputs, i.e. emergent properties, differently. So the definition of test oracles needs to take these issues into account.

As with the other testing levels, integration test suites are aimed at two distinctive targets: (i) to refine the interaction design and solve integration problems as early as possible; and, (ii) to test the integration of the implemented agents with one another and with the environment, once these are available. The first target is realized during the *Detailed Design* phase and integration test suite derivation, while the second can be started as soon as an agent or an environmental resource is implemented. Mock agents, which simulate behaviours of agents, can be used during integration testing (with regard to the second target) so that we do not need to wait until all the involved entities are implemented to start integration testing.

3.4.4.4 Unit and agent test

Unit testing consists of verifying agent units, e.g., goals, plans, beliefs, and events, that agents are composed of. In the rest of this section we discuss mainly plan, goal, and agent testing.

i. Plan testing

Zhang et al. (2007) has discussed different aspects related to plans and events testing. Though introduced in the context of Prometheus methodology (Padgham and Winikoff 2004), those aspects apply to our approach as well, because the agent architecture that both Prometheus and Tropos use is the BDI architecture (Rao and Georgeff 1995). In short, plans are means to achieve goals (*ends*), plans are triggered as a result of goals selection. Consequently, to test a plan, we need to create test suites such that they satisfy all the pre-conditions of its *end* goal and pre-conditions of the plan itself. These conditions, among others, contain corresponding events or percepts that eventually trigger the plan. Then, we have to evaluate the execution of the plan, its subsequent tasks.

As for plan testing oracle, plan execution can be evaluated by using the state of its *end* goal. For example, if the state of the end goal is maintained or achieved as a result of the plan execution, one may conclude that the implemented plan passed the test.

Test suite derivation for plans takes place at the *Detailed Design* phase. For each single plan, we need to create a test suite that contains a set of test cases to challenge the plan with different inputs. Let's consider our motivating example, for each plan we create a unit test suite, so there are 11 test suites in total. For instance, for the plan *Move*, the associated test suite is informally described in Table 3.6.

ii. Agent goal testing

Goals are states of affair, and one must do something in order to achieve his/her goals. A very natural way of testing the achievement of a goal is to check one's work or behaviour with respect to the goal. Similarly, to test a goal we have to check what the agent does to fulfil the goal.

Table 3.6: Test suite for plan *Move*

Suite	TS1	
Plan	Move	
Plan class	Move	
Test Case	**Scenario**	**Oracle**
UTC1.1	There is an event that requires the cleaning agent to move from position A(1,1) to position B(3,5), no obstacle is in the middle of the two points	The cleaning agent moves straight from A to B
UTC1.2	Between A(1,1) and B(3,5), there is a static obstacle at point C(2,3)	The cleaning agent moves close to C, identifies the obstacle, avoids C before going to B
UTC1.3	The agent is requested to move from (1,1) to (3,-1)	The cleaning agent moves to the boundary nearest to (3,-1)

When applying the Tropos methodology, we can find out how goals can be fulfilled by looking at their relationships with other goals and with plans. For instance, if there is a *Means-End* relationship between goal g_1 and plan p_1, we say g_1 is fulfilled when p_1 is executed successfully; if goal g_2 contributes positively to softgoal sg_2 (*Contribution+* relationship) then we can say sg_2 is partially satisfied when g_2 is fulfilled. Based on the relationships associated with a goal, we can check the fulfilment of the goal.

Internal design of an agent consists of goal decomposition/contribution trees. For example, Figure 3.4 depicts the design of the cleaning agent, consisting of five trees associated with four root goals: *G2: team-work, G4: maintain-battery, G1: keep-the-airport-clean, G3: be-polite.* The fulfilment of the root goals of the trees is evaluated based on the fulfilment of their sub-goals and the relationships between the root goals and the sub-goals, and so on with the intermediate goals inside the trees. The fulfilment of the leaf goals of the trees is evaluated based on their relationships with the *means* plans. We call these relationships as *elementary relationships*.

The principal *elementary relationships* are depicted in Figure 3.8. These include: (1) *Means-End* between a plan and a hardgoal; (2) *Contribution+* between a plan and a softgoal; (3) *Contribution-* between a plan and a softgoal. In order to test this kind of relationships, the execution of the plan corresponding to a goal is triggered and checked based on assertions and constraints on the expected behaviour. Developers derive test suites from goal diagrams by starting from the relationships associated with each goal. Each relationship gives raise to a corresponding test suite, consisting of a set of test cases that are used to check goal fulfilment (called positive test cases) and counter-fulfilment (called negative test cases). Positive test cases are aimed at verifying the fulfilment capability of an agent with regard to a given goal; negative test cases, on the other hand, are used to ensure an appropriate behaviour of the agent under test when it cannot achieve a given goal.

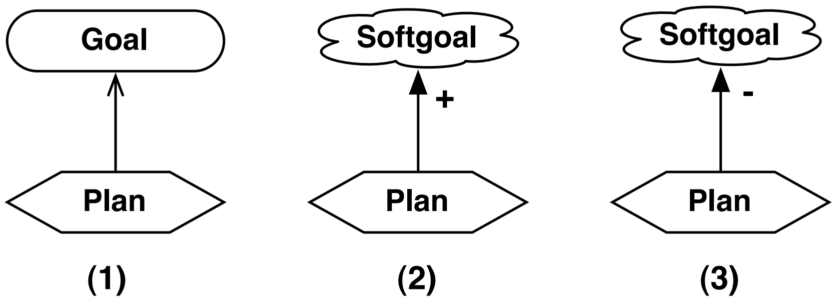


Figure 3.8: Elementary relationships. (1): a *Means-End* plan-hardgoal; (2): a *Contribution+* plan-softgoal; (3): a *Contribution-* plan-softgoal

The derivation steps are as follows:

```
1: for all $g \in$ {leaf goals} do
2:     step1: identify means plans from elementary relationships.
3:     step2: identify the fulfilment criteria of $g$
4:     step3: identify possible plans execution orders or schedules
5:     step4: create one test suite for $g$
6: end for
```

The procedure can be described as follows: for each leaf goal we identify *means* plans from the elementary relationships related to the goal. The fulfilment of the goal and possible plan executions are then defined. Finally, a test suite should be created for the goal in which each test case addresses one possible execution scenario.

For the intermediate goals (i.e. not leaf goal), test suites are derived by inspecting all relationships that lead to the considered goals. This ends up analyzing all elementary relationships and creating/reusing test suites derived for them. Once the results of these test suites are obtained, we can reason about the achievement of the intermediate goals based on the decomposition and/or contribution analysis. For example, to test the goal *G2: team work* of our cleaning agent (Section 3.3), we have to analyze its decomposition into three sub-goals; from there, we have to test three elementary relationships between the sub-goals and their corresponding plans. Since this is simply an AND-decomposition, if three test suites derived for these three elementary relationships are passed, then the goal G2 is passed; otherwise the goal is failed.

For more sophisticated intermediate goals, for example G1, we have to analyze all possible combination scenarios based on the goal analysis and reason about the fulfilment of the goals on the basis of these scenarios and the results of the test suites derived for the related leaf goals.

Let's take the agent goal *G1: keep-the-airport-clean* of the cleaning agent for example. By analyzing the goal decomposition tree, which has G1 as root, 5 elementary relationships are identified: *Move→Achieve-move-to-a-location*, *Achieve-pickup-waste→Pickup-waste*, *Query-waste-bin→Look-for-wastebin*, *Achieve-drop-waste→Drop-waste*, and *Perform-looking-for-waste→Patrol.* Each of them gives rise to a different test suite.

iii. Agent testing

An agent is composed of smaller components, e.g., beliefs, goals, plans, events, reasoning module, and so forth. Testing at the agent level consists of integration testing of agent components, so one has to derive test suites to verify this integration.

Agent-level test suites have a strong relation with test suites created for testing agent goals. Because, first of all, in most cases, testing a goal involves testing one or a number of plans, testing a plan involves events, percepts, and resources. So to some extent, testing a goal triggers some integration of plans, events, and so on. Hence, test suites derived to test agent goals are also effective to test the agent integration.

However, at the agent level, we need to test the integration of goals as well. Some goals have dependencies among them, such as priority or inhibition dependences; others may be maintained or achieved in parallel while sharing a resource. So we have to identify goal integration scenarios, create test suites for each, and look for integration problems such as dependency violations, deadlock, livelock, and the like.

Let's consider our motivating example once more. At the agent level, we have to derive test suites to check if the agent can perform: maintain-battery (G4), be-polite (G3), keep-the-airport-clean (G1), and team-work (G2). Moreover, we have to check the possible conflicts among these goals.

For example, at a given moment in time, the cleaning agent can only move either to a recharging station, or to a waste bin, or to a new position for patrolling. Hence, some goal might be temporarily sacrificed in favour of another one. In addition to that, we have also to check if collaborative goals (e.g., *G2: teamwork*) are achieved in parallel with the other goals.

The basic test adequacy requirement for an agent is that all the agent goals must be tested. The agent should be able to achieve its goals and behave correctly in the cases where its intended goal cannot be achieved. This adequacy requirement may or may not be sufficient to cover the agent components, i.e. plans, events, beliefs, etc. If some are never exercised by the test suites defined to reach the basic adequacy criterion (goal coverage), more test suites have to be defined to complete agent testing.

3.4.5 Test suite structure

The key elements of goal-oriented testing are goals, either organizational, system, collaborative, or agent goals. The underlying objectives at different testing levels consist of tackling goals fulfillment. Thus, derived test suites must be able to specify test target, i.e. goal, and test scenario, including inputs, conditions, and expected behaviour of the agent under test.

To support specifying goal-oriented test suites, we propose the structure illustrated as a UML class diagram in Figure 3.9. It can be read as follows: each *Test Suite* contains a set of *Test Cases*, each *Test Case* contains a test scenario in which *Test Actions* are specified. Each *Test Suite* targets one or more agents, goals, and/or plans. Each *Test Suite* or *Test Case* can contain *Support Actions* (e.g., setup testing environment, tear down when finished). Finally, pre- and post-conditions can be specified for a goal, a plan or a scenario.

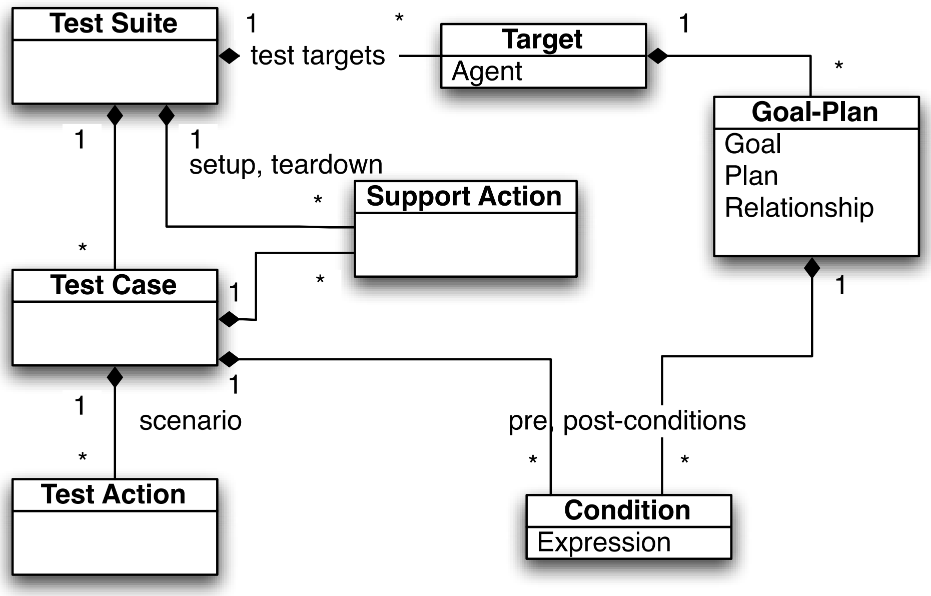


Figure 3.9: Overall structure of test suites

The proposed structure of test suite, test case, and test scenario are designed such that they can be used at different formality levels and with different programming languages. Informally, developers can specify their test cases using descriptive text. This format can be used by human testers to specify manually input data and evaluate the output results. When used formally, the specified test cases can be read by testing tools. To this purpose, the contents of the elements *Test Action, Support Action, Condition* support user-defined data types. Developers can associate their machine-readable data with their own parser and grammar so that test suites can be executed automatically.

In our implementation, we provide the following types for *Test Action* and *Support Action*:

Test Action	
wait	for observation
communication	send or receive a message to/from the agent under test
checkpoint	check received message, constraints or assertions
branch	go to other test actions depending on branch conditions
env-effect	make changes to the environment
Support Action	
start agent	start an agent
kill agent	remove an agent
register agent	register to the DF (FIPA 2004)
deregister agent	remove the agent from the DF
executable	launch supporting code to make changes on the testing environment
not-executable	manual action or description

We have defined this structure in an XML schema, available for download and reference at `http://se.itc.it/dnguyen/xsd/TestSuite.xsd`. We also provide a tool that allows generating test suites from goal models, editing test suites, and executing them. Details are described in Chapter 5.

3.5 Summary

This chapter presented the GOST methodology that took goal-oriented requirements analysis and design artefacts as the core elements for test case derivation. The proposed methodology has been illustrated with respect to the Tropos development process. It provides systematic guidance to generate test suites from modelling artefacts produced along with the development process. These test suites, on the one hand, can be used to refine goal analysis and to detect problems early in the development process. On the other hand, they are executed afterwards to test the achievement of the goals from which they were derived.

The procedures for generating the test suites presented follow the structure of the Tropos goal-oriented modelling artefacts. However, it is worth noticing that the GOST approach is based on a generalizable set of guidelines that complement a goal-oriented requirements analysis and design process with a suitable testing process. Basic steps for customizing GOST to different goal-oriented methodologies are the following:

- Identify of the analysis and design phases supported by the methodology under consideration, and of the corresponding set of artefacts, in order to select the testing levels to be considered among the following: unit, agent, integration, system and acceptance testing.
- Identify of how the different analysis and design artefacts can be combined to derive a specific level test suite (that is, the definition of the test suite derivation schema as the one depicted in Figure 3.6).
- For each specific level, define a derivation procedure, which takes into account how goals are analyzed in the corresponding design modelling artefacts (i.e. follow the goal decomposition and refinement mechanisms supported by the given methodology, according to the modelling language meta-model).

Chapter 4

Testing techniques

Testing can be subdivided into defining or generating test inputs and test scenarios, specifying test oracles to judge testing results, and executing test cases. In this chapter we introduce different techniques to tackle these problems, taking into account agent's properties. Particularly, we investigate automated ways to generating test inputs that can produce enormous number of different and challenging situations to exercise the agents under test. This overcomes the limited human effort for testing. The automated generation, to some extent, helps dealing with the dynamic nature of the environments where the agents under test operate.

In this chapter, we first present three approaches to evaluate behaviours of software agents. As agents are autonomous, saying if an agent exhibits a correct behaviour or not is not as straightforward as traditional programs. We put test evaluation, i.e. to evaluate test results, in the first place as feedbacks from test results give important insights to guide the automated test input generation. Then, we introduce monitoring as a way to collect data about test execution. The monitoring technique can deal with the distributed and asynchronous property of agent-based systems, and provide a global view of what happens during test execution. Finally, we present four test generation and one novel execution techniques. Experimental results, discussed in Chapter 6, will show the ability of these techniques in detecting faults.

4.1 Evaluation of agent behaviours

We consider three types of agent faults: faults related to constraints that restrict agent's behaviours, faults related to interaction semantics that define the semantics of agent interaction, and faults related to user's requirements. Corresponding test oracles are defined to pinpoint these kinds of faults.

4.1.1 Constraint-based oracle

The behaviour of autonomous agents can change over time. This makes the evaluation of test results a non-trivial task. Often, it is impossible to give a fixed verdict to a test case based on the comparison of the returned message with a gold standard, because the returned message may be different, even for the same input, at different times. Similarly, mental states of an agent, e.g., beliefs, can change with respect to the same inputs, specifying some invariants as oracles for these variables can be non-trivial. We propose to use constraints that restrict the behaviours of software agents as test verdicts. Constraint violations are considered as faults.

Behavioural constraints are specified in terms of pre-, post-, and invariant conditions. For low testing levels, i.e. unit and agent, we propose to specify these conditions by using the Object Constraint Language (OCL) (OMG 2006). As most of the contemporary languages used to program software agents are object-oriented, e.g., (Telecom Italia Lab 2000), or employ object-oriented code to operationalize agents' plans, e.g., (Pokahr et al. 2005, Agent Oriented Software Pty. Ltd. n.d., Bordini et al. 2007), the emerging OCL can be used to guarantee that agent code units execute correctly. However, software agents are distributed programs that run at geographically different hosts, handling OCL constraint violations need to take this into account. Our monitoring agent network, introduced in Section 4.2, can deal with this issue.

From OCL constraints, monitoring guards (to check constraint violations) can be generated automatically, using a tool called OCL4Java[1] and its user-defined handler. We specialize this type of violation handler to notify a local monitoring agent during testing whenever a constraint is violated. Local monitoring agent is an agent that runs in the same place with the agents under test. It is in charge of monitoring not only constraint violations but also many more types of events, such as communications, exceptions, belief changes, and so on. Details about the monitoring agent will be introduced shortly.

Following is an example of pre-/post-condition specified in OCL, which requires the *order* attribute to be not null and ensures that after updating the proposed price must be between 0 and 2000:

```
public class ExecuteOrderPlan extends Plan {
  ....
  @Constraint("pre: self.order->notEmpty\n" +
              "post: price > 0 and price < 2000")
  public void body() {
       ....
  }
  ....
}
```

The following code is generated by OCL4Java from the constraint above. The implementation of the method *handleConstraintFailed* is specialized to inform the local monitoring agent whenever the constraint is violated.

```
public class ExecuteOrderPlan extends Plan {
   ....
```

[1]http://www.ocl4java.org

```
   @Constraint("pre: self.order->notEmpty\n" +
               "post: price > 0 and price < 2000")
   public void body() {
       if (!assertPreCondition_6fa583bb_for_method_body()) {
         org.ocl4java.ConstraintFailedHandlerManager.handleConstraintFailed(
         "pre: self.order->notEmpty" ...);

      // body code
      ...

      if (!assertPostCondition_6fa584bb_for_method_body()) {
         org.ocl4java.ConstraintFailedHandlerManager.handleConstraintFailed(
         "post: post: price > 0 and price < 2000" ...);
   }
   ....
}
```

Using the same manner, i.e., specifying constraints and deriving monitoring guards from them, we can specify constraints for higher levels such as group of agents or the system as a whole. However, this needs further elaboration on monitoring data because these types of constraints involve multiple parties. Some research work has investigated this direction, e.g., (Rodrigues et al. 2005).

4.1.2 Ontology-based oracle

4.1.2.1 Agent interaction ontology

In order for a pair of agents to understand each other, a basic requirement is that they speak the same language and talk about the same things. This is usually achieved by means of an ontology, namely, *interaction ontology*. Popular multi-agent platforms like JADE (Telecom Italia Lab 2000), JADEX (Pokahr et al. 2005), widely support the use of ontologies. They provide tools for generating code from ontology documents, thus, reducing the development effort, and for runtime binding of the message contents with concepts defined in an ontology.

A common structure of interaction ontology involves two main concepts (also known as *Classes*): `Concept` and `AgentAction`. Sub-classes of `AgentAction` define actions that can be performed by some agents (e.g., `Propose`), while sub-classes of `Concept` define common concepts understandable by agents that interact (e.g., `Book`).

Let us consider a book-trading multi-agent system in which *Seller* and *Buyer* agents negotiate in order to sell and buy books. There could be multiple sellers and buyers that want to sell or buy the same book at the same time, so the goal of the sellers is to choose a buyer that proposes the highest price whereas the goal of the buyers is to choose the seller with the cheapest price. Let us assume that these agents use the FIPA Contract Net protocol (FIPA 2002b) and the interaction ontology presented in Figure 4.1. The ontology consists of a concept `Book` having two properties *title* and *author* and an agent action `Propose`, to propose a *price* for a *book*.

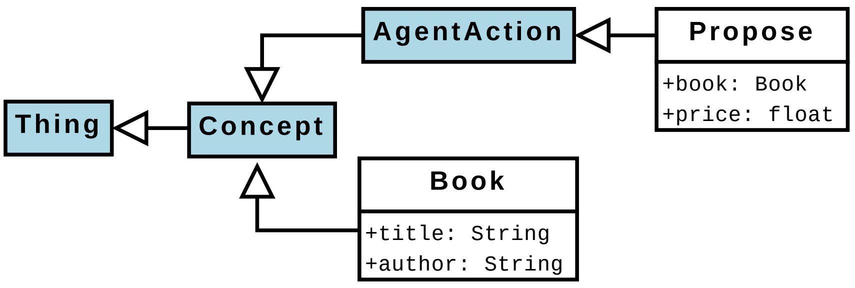


Figure 4.1: Book-trading interaction ontology, specified as UML class diagram

Rules can be added to the ontology properties in order to restrict admitted values. For example, the *price* property in Figure 4.1 may be constrained to be within 0 and 2000. The rule is specified in the Web Ontology Language (OWL) as follows:

```
<owl:Restriction>
   <owl:onProperty rdf:resource="#price"/>
```

```
  <owl:hasValue ...>min 0 and max 2000</owl:hasValue>
</owl:Restriction>
```

A specific agent action can now be built, based on the shared understanding of the concept `Book`. For example, an agent *Buyer* could send an ACL message of the type `REQUEST` to the agent *Seller*, with the following content:

(Propose (Book :title "Testing Agents") :price 135.7)

The message is understood by both agents thanks to the shared interaction ontology.

4.1.2.2 Ontology as oracle

Agent interaction ontologies provide available tools to verify agent communication semantics. This can be used in testing. In fact, the message content sent by the agents under test is expected to respect the rules, datatypes, and structural relationships specified in the ontology. Sending invalid messages with respect to the chosen interaction ontology is a fault.

For instance, when the *Buyer* sends a call for proposal for a book, the *Seller* agent must reply with a message whose content belongs to the `Propose` action and complies with its rules and datatypes. Otherwise, an error is detected. On the other extreme, if the *Buyer* sends something else but not a call for proposal, then it is faulty.

Our testing framework, which will be introduced in Chapter 5, performs this type of verification automatically. More interestingly, it is able to take the interaction ontology that the agents under test use to generate variety test inputs to challenge them, at nearly no cost.

4.1.3 Requirement-based oracle

Autonomous software agents differ from traditional softwares in that they have their own goals and operate in a self-motivated fashion. External subjects might have little or no control over the behaviours of autonomous agents. As a result, this challenges testing because expected outcomes may not be immediate or the way to define them is non-trivial. For instance, in the same environmental settings (i.e. test inputs), an autonomous agent may decide to do different things due to learning or decision-making. Defining concrete expected outcomes for this agent based on an external perspective, i.e. a human being tester, is hard.

The ultimate goal of building agents with autonomy is to release human beings from some tasks, possibly dangerous ones. However, before letting an agent to perform any task, we need to make sure that she is qualified or she has certain qualities to perform that task. We need to have confidence in their autonomous operations. The agents need to be reliable and trusted before being put to real environments.

We propose to apply the recruitment metaphor to evaluate autonomous software agents. Here, software agents are candidates and stakeholder requirements are used as evaluation criteria. Each agent is given a trial period in which it has to solve a suite of tests with different difficulty. Agents are recruited (trusted) only when they pass the required quality criteria.

In requirements engineering, the importance of stakeholder goals has long been recognized. As such, the concept of *goal* has been considered as central to some goal-oriented requirements engineering (GORE) approaches (Bresciani et al. 2004, Dardenne et al. 1993). In GORE, softgoals play a key role in representing non-functional or "ility" requirements, such as dependability, availability, security, and so forth, which can denote the important criteria for evaluating autonomy. Returning to the recruitment approach to evaluating autonomous agents, we propose to use stakeholder softgoals as criteria for assessing the quality of autonomous agents, since satisfying quality criteria derived from these softgoals is likely to indicate that the agents are reliable. [2]

Relevant softgoals to evaluate agent autonomy are transformed or represented as quality functions (or quality metrics). This transformation is tricky and depends on the nature of the softgoal at hand and also on the problem domain. Ad-hoc metrics can be defined for softgoals using domain expertise.

As an example, Figure 4.2 illustrates the goals of a specific stakeholder in an airport organization, namely the building manager, who decides to assign the goal of airport cleanliness to a cleaner agent. The notation used in the figure is proposed in Tropos (Bresciani et al. 2004). In this example, the agent must operate autonomously, with no human intervention. The agent must be robust and efficient as stated in the two stakeholder's softgoals, depicted as two cloud shapes. Applying the proposed approach, these two softgoals can be used as criteria to evaluate the quality of the cleaner agent. The agent can be built with a given level of autonomy, and robustness and efficiency are two key quality criteria for evaluating it. If the cleaner agent can perform tasks autonomously, but is not robust (for example, it crashes), it is not ready to be deployed.

Regarding the *robustness* softgoal, two sub-goals contribute to *robustness* that are taken into account in this example are *maintaining-battery* and *avoiding-obstacles*. We can define a threshold for the *maintaining-battery* capability (e.g. 10%), and monitor the battery level at runtime. Figure 4.3 shows two scenarios of the battery level: 4.3(a) is an acceptable

[2]As softgoal has no clear-cut criteria for its achievement, the notion "satisficing" has been used in the literature to indicate whether a softgoal is satisficied or not. Jureta et al. (2007) stated that "a softgoal is satisficed when thresholds of some precise criteria are reached"; we share this view in testing software agents

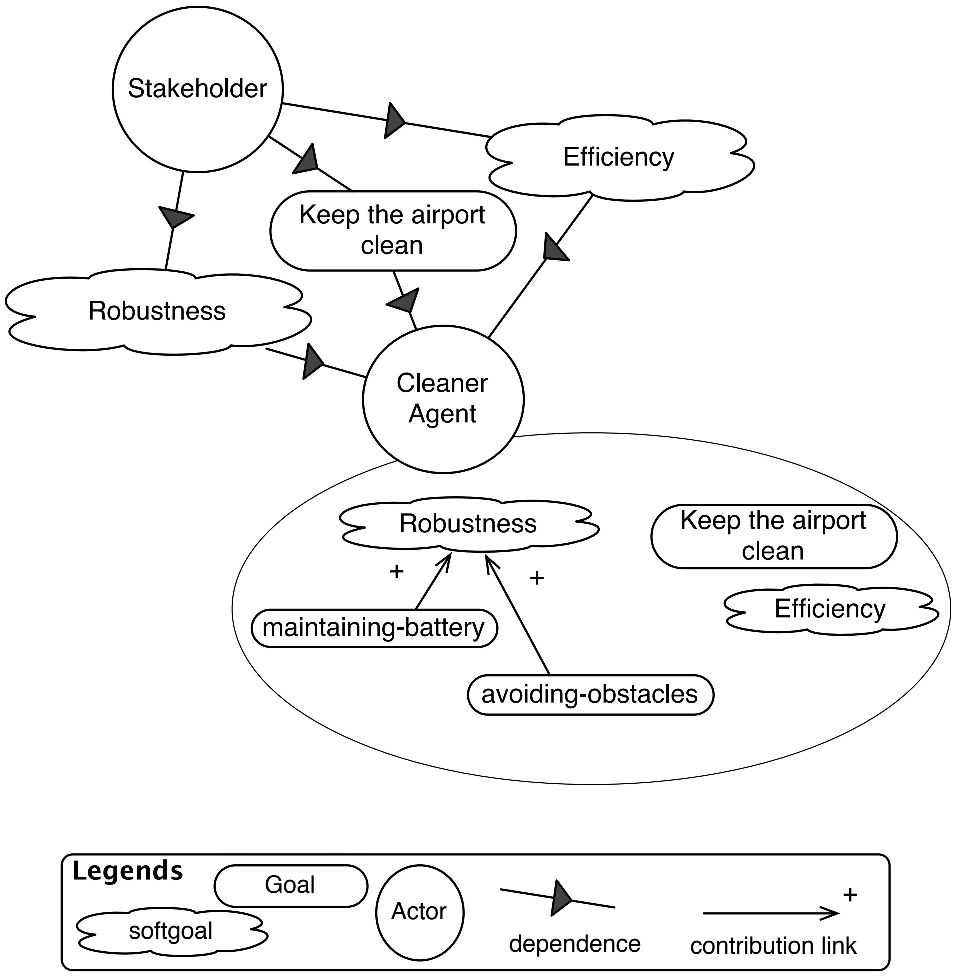
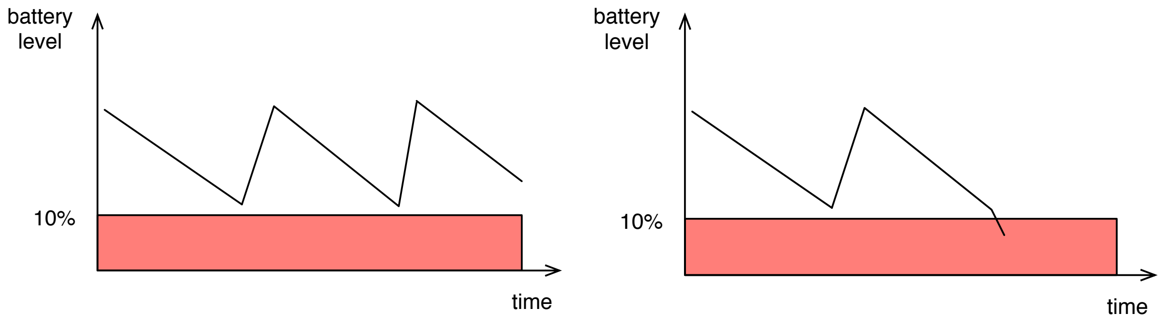


Figure 4.2: Example of stakeholders' softgoals and contribution analysis

scenario where the battery level is maintained at a sufficiently high level within the period considered, while 4.3(b) is an unacceptable scenario in which the battery level drops below 10%.



(a) Acceptable scenario (b) Unacceptable scenario

Figure 4.3: Different scenarios related to the battery level

Similarly, for the softgoal *avoiding-obstacles*, one can define the distance to the closest obstacles during movement as a quality criterion. Correspondingly, a quality threshold ε (distance units) can be defined, and the agent must stay farther from obstacles than this threshold.

In reality, apart from robustness, we can impose many other requirements related to autonomy on the cleaner agent: *stability, efficiency, safety*, for example. Stability demands the agent should avoid dropping its goals too frequently. Efficiency requires the agent to finish cleaning an area after a specific amount of time, or it must bring a quantity of waste (e.g., 10 Kg) to the dustbins per hour. The safety requirement demands that the agent must switch to its 'safe mode' in undesirable circumstances, e.g., arms malfunction.

4.2 Monitoring

In testing software agent and MAS, monitoring plays an important role as it allows us to observe the operation and interaction of the agents under test. It provides necessary data to detect abnormalities in the system, such as constraint violation, communication semantics mismatching, or requirement unsatisfaction.

We propose two reference architectures for monitoring agent locally, Figure 4.4, and globally, Figure 4.5.

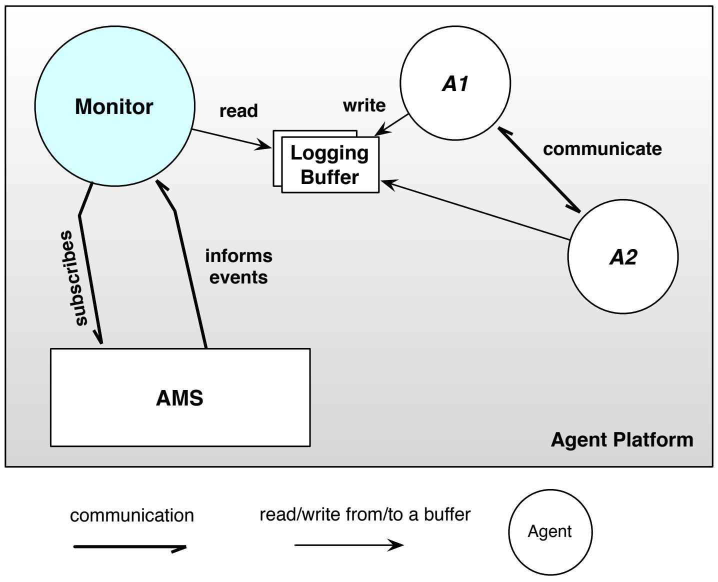


Figure 4.4: Reference architecture for monitoring one single platform

At the local level, in a single platform shown in Figure 4.4, a special agent named *Monitor* (or *Monitoring Agent* in other places) subscribes itself to the Agent Management System (AMS)[3] (FIPA 2004) in order to be notified about all relevant events happen within the platform. These events include: an agent was born, is dead, is frozen, moves, adopts a goal, changes its beliefs, and the like. In particular, the AMS will inform the *Monitor* about any interactions, messages sent or received by the agents under test.

[3]The AMS is responsible for managing the operation of an agent platform, such as the creation of agents, the deletion of agents, deciding whether an agent can dynamically register with the agent platform and overseeing the migration of agents to and from the platform. Registration with the AMS implies authorisation to access the message transport service of the agent platform.

In addition, we propose to use a special component called *Logging buffer*. As the name says, this is a buffer where observed data can be store and read. Information about violations, exceptions or desired data to be observed, such as states of the agent under test, are stored into this buffer. The monitoring agent is in charge of watching this buffer to report any problem occurred. Agent code can be instrumented; aspect programming can be used to inject code for monitoring. In particular cases when allowed, we can ignore the logging buffer. Instead, monitoring code can send messages about problem to the monitoring agent, transparently with the agents under test.

At the global level, since a MAS usually consists of multiple distributed platforms, Figure 4.5, it is important to incorporate information from all of them to provide a complete and full view about the system under test. This can be achieved by means of a network of monitoring agents: the *remote monitoring agents* act the same as the *Monitor* at the local level, mentioned above, each is responsible for monitoring one single platform; all observed data from the distributed platforms are sent to the *Central Monitoring Agent*. Therefore, we obtain a global and synthesised view of the system during test execution.

One possible issue that need attention is possible side effects of using the monitoring agents. That is, the monitoring agents might influence the behaviours or the performance of the agents under test. The monitoring agents need to be implemented or deployed in a way that is as much transparent to the agents under test as possible. Or at least, we need to control the testing environment to dismiss any side-effect problem.

These architectures are implemented in our tool, introduced in Chapter 5. Real-time observed data help not only detecting problems, but also providing useful feedbacks to guide automated test input generation. The next section will discuss test generation techniques.

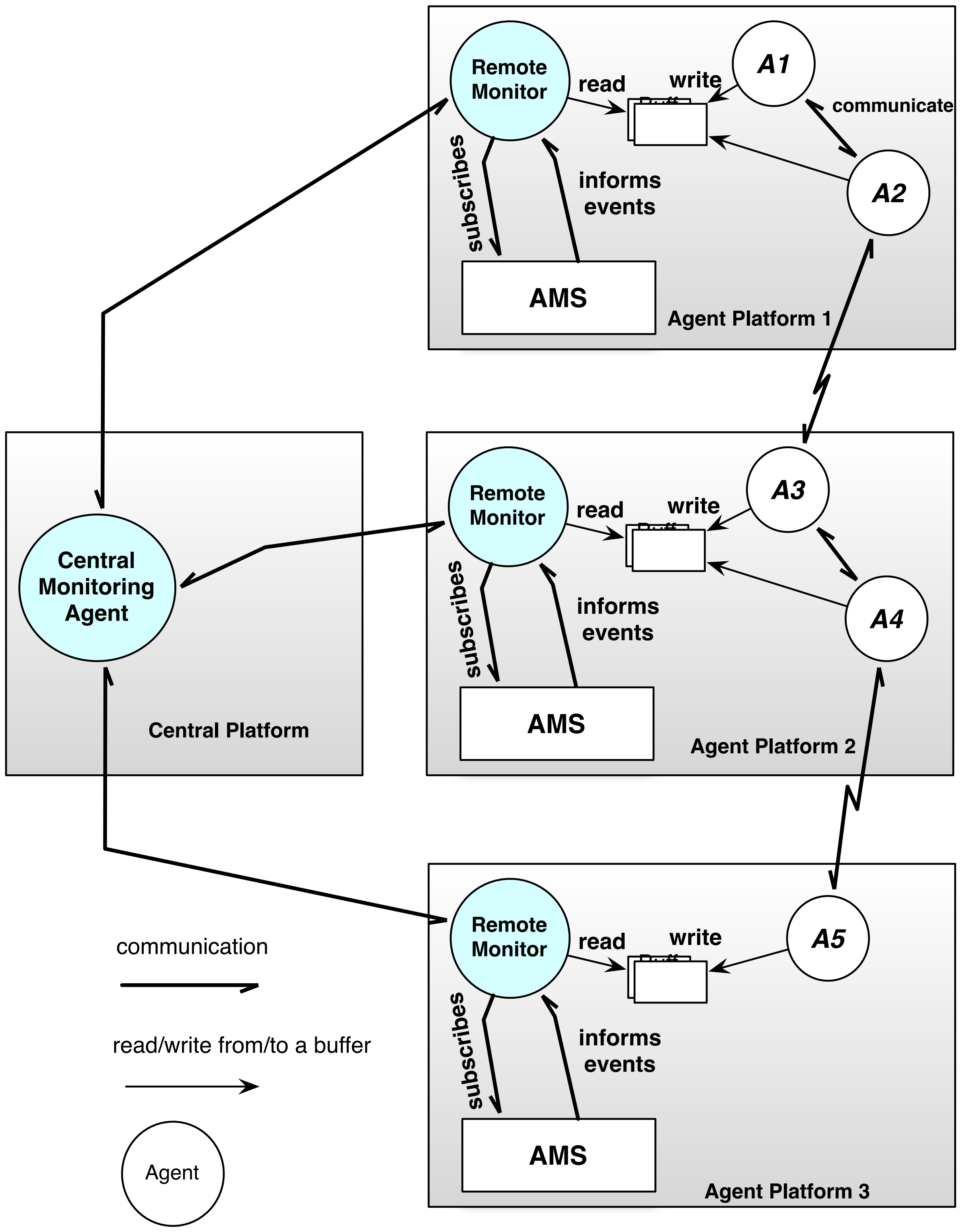


Figure 4.5: Reference architecture for monitoring multiple platforms

4.3 Generation

4.3.1 Test inputs for software agents

Georgeff and Ingrand (1989) presented a minimal design of a reference architecture for BDI agents (Rao and Georgeff 1995), which has been being widely applied to build autonomous agents. In the architecture, agents perceive the outside world (environment) through a set of sensors and make changes to the world through a set of effectors. Recently, Weyns et al. (2007) complemented to that architecture with a reference model for the environment, in which agents access the environment by employing perception (sense and percept), action (make changes to the environment), or communication (send and receive messages).

In terms of test inputs, from the proposed architectures we identify two types of black-box test input for agents: environmental settings and incoming messages. The former type concerns the surrounding world with respect to an agent; changes that are perceived by the agent can lead it to expose different behaviours. For instance, if an obstacle appears on the path that an agent is following, the agent might change its path instead of going straight or try to remove the obstacle. The latter concerns the messages that are sent to agents under test. These messages, once accepted by the agents, may ask the agents to fulfil a task or to reach a goal. More generally, incoming messages can change the behaviour of agents.

Depending on the kind of the agents under test, test inputs can be generated by producing environmental settings upon which the agents under test operate, or by creating messages and submitting them to the agents, or both.

4.3.2 Goal-oriented generation

Goal-oriented test cases generation is a part of a methodology, presented in Chapter 3. It integrates testing into Tropos, providing a systematic way of deriving test cases from Tropos output artefacts. Goal-based specification diagrams are used as inputs to generate test case skeletons to test goal fulfillment. Specific test inputs (i.e. message content), and expected outcome are partially generated from plan design (e.g., UML activity or sequence diagrams) and are then completed manually by the tester according to some test scenarios. These scenarios can be user-defined, or can follow some particular interaction protocols.

4.3.3 Ontology-based generation

This technique concerns generating messages to test software agents.

Agent behaviours are often influenced by messages received. Hence, at the core of test case generation is the ability to build meaningful messages that exercise the agent under test so as to cover most of the possible running conditions. We propose an approach to test generation using agent interaction ontology. The approach exploits ontology that defines the semantics of agent interactions to generate test inputs and guide the exploration of the input space. We develop an ontology-based input generator. It is integrated with our testing framework, introduced in the next chapter.

Valid inputs. The task of the ontology-based test generator consists of completing the message content to send to the agent under test. For each concept to be instantiated in the message, the generator either picks up an existing or creates a new instance of the required concept. No input value is generated by the test generator if the interaction protocol prescribes that a value from a previously exchanged message must remain the same.

Then, the selected instance is encoded according to a proper content codec (for the message content) and is made ready to be executed. As an example, the following excerpt shows an XML-encoded content of a message that contains information about a proposal for a book, including the `Propose` action:

```
<root ... xmlns="jadex.examples.booktrading.ontology"/>
   <Book n:id="2" title="Introduction to MultiAgent Systems"
                  author="Michael Wooldridge"/>
   <Propose n:id="1" price="47.50" r:book="2"/>
</root>
```

When new instances are generated, the test generator selects one from those available in the ontology. The selection is based on the number of usages of each instance, or aims at increasing the diversity of test inputs and exploring the input space more extensively.

In the case when no ontology instances are available, valid test inputs can be still generated by using available information, such as rules and property datatypes, specified in the interaction ontology. For example, based on the rule about the *price*, the generator can generate any value in the range from 0 to 2000 as a valid input value to be processed by the *Seller* or *Buyer* agents.

More generally, for the properties of *Numeric* datatype, we can exploit the boundaries of the datatype, as well as the rules that limit the values of the properties, to generate valid input values. For the properties of *string* datatype, we can only exploit the list of allowed values, if available. Most of the times, meaningful values for properties of string datatype are hardly generated without the help of an ontology. The full list of valid input generation rules is provided in Table 4.1.

Invalid input generation. Invalid input generation is based on rules and datatypes that appear in the interaction ontology. When boundaries are

Table 4.1: Valid input generation rules

Datatype	**Rule**	**Description**
Numeric	**RVN1**	New value that has not been used before from ontology instances
	RVN2	Reused value from ontology instances
	RVN3	Randomly generated value respecting rules in ontology
	RVN4	Default or template value defined in ontology
Boolean	**RVB1**	*true*
	RVB2	*false*
String	**RVS1**	New value that has not been used before from ontology instances
	RVS2	Reused value from ontology instances
	RVS3	Randomly generated value respecting rules in ontology
	RVS4	Default or template value defined in ontology

specified for numeric properties, the generator goes beyond them deliberately. For string properties, the generator produces null (or empty) strings as potentially invalid values. Other options available to the generator are to randomly modify a valid input (taken from the available ontology instances) or to randomly generate a new one in order to try to produce an invalid value. Another generation rule available to the test generator involves the creation of an input value of the wrong datatype (e.g., an alphabetic string where a numeric is expected). The full list of invalid input generation rules is provided in Table 4.2.

The generator aims at producing invalid inputs that are as diverse as possible, in an attempt to test the robustness of the agents under test, making sure that they still behave correctly in most invalid circumstances. According to the book-trading ontology described above, the test generator knows that the property *price* is of datatype *float* and that there is a rule

Table 4.2: Invalid input generation rules

Datatype	**Rule**	**Description**
Numeric	**RIN1**	Value causing overflow (underflow)
	RIN2	Value violating rules in ontology
	RIN3	Value of different datatype
	RIN4	*null* value
Boolean	**RIB1**	Value of different datatype
	RIB2	*null* value
String	**RIS1**	Value violating rules in ontology
	RIS2	Value of different datatype
	RIS3	*null* value
	RIS4	Empty string
	RIS5	Randomly generated string
	RIS6	Randomly mutated valid string

stating that *price* must be between 0 and 2000. The generator may produce the invalid values -1, 2001 to test both sides of the boundaries. Values that are not of type *float* may be also used to exercise the agents under test.

Message generation. When generating the full message, the test generator applies the input combination rules described in Table 4.3. For valid messages, the only possibility is to use only valid input values. For invalid messages, the generator can choose either to have only invalid values, or to have interleaved valid and invalid values, or to have just one invalid value. Rule selection follows the general criteria of maximizing diversity, as explained below.

When a valid message can only be formed with inputs coming from an unique, existing instance, the more restrictive rule **RVC2** must be applied instead of **RVC1**. If input values from different instances can be freely combined, we can use **RVC1**. When **RVC2** must be used, one way to generate invalid inputs is mixing values from different instances, as pre-

Table 4.3: Input combination rules

Message	Rule	Description
Valid message	**RVC1**	All values valid
	RVC2	All values valid and from the same instance
Invalid message	**RIC1**	All values invalid
	RIC2	Invalid and valid values interleaved
	RIC3	Just one invalid value
	RIC4	All values valid but from different instances

scribed by **RIC4**.

Input space exploration. The generator uses coverage information to decide how to explore the input space. The test generator gives priority to classes and instances never selected before. When instances are reused, if possible the generator selects instances with low reuse frequency. When invalid inputs are produced, the generator chooses the so-far least-used invalid input generation rules.

4.3.4 Random generation

Random testing has been proven to be very effective in revealing some types of faults, specially those that result in crashing or raising exceptions (Mills et al. 1987, Thévenod-Fosse and Waeselynck 1993). In dynamic and open environments for MAS, random testing seems to be a natural choice because it can generate unpredictable scenarios, which likely happen in such environments.

We are interested in two types of test inputs: messages and environment settings. The following discusses the random generation of these types of inputs.

4.3.4.1 Random generation of messages

We propose an approach to random testing of software agents, composing of two steps, Figure 4.6. First, a communication protocol is randomly selected among the standard ones provided by the agent platform, e.g., FIPA Request Protocol (FIPA 2002b) and/or those specified in a library by human tester. Then, messages that are required by the protocol are randomly generated and sent to the agents under test. In order to insert meaningful data into the messages, a model of the domain data, coming from the business domain of the MAS under test, must be also supplied. The message format is prescribed by the agent environment of choice (such as the FIPA ACLMessage (FIPA 2002a)), while the content is constrained by a domain data model. Such a model prescribes the range and the structure of the data that are produced randomly, either in terms of generation rules or in the (simpler) form of sets of admissible data that are sampled randomly. The model of domain data can be specified by means of an ontology as well, so ontology-based generation rules can be applied to generate message content.

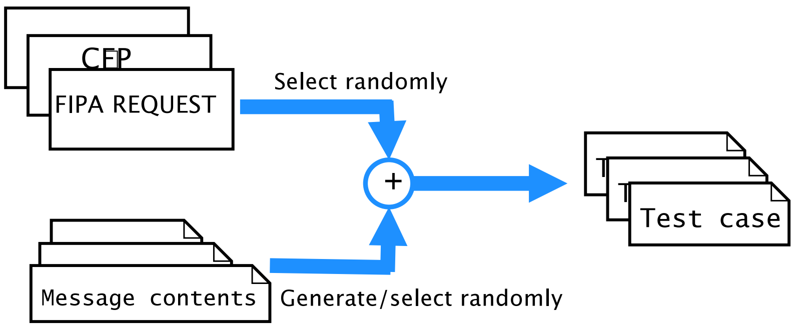


Figure 4.6: Procedure of the random generation technique

Randomly generated messages are then sent to the agents under test and it is the responsibility of our monitoring agent network to observe their responses, i.e., communications, exceptions etc. happening in the agent system. When a deviation from the expected behaviour is found (condition violated or crash), it is reported to the development team.

A limitation of random testing of MAS is that long and meaningful interaction sequences are hardly generated randomly. However, it is often the case that agent interaction protocols need only few trigger messages, like those specified in (FIPA 2002b), or the agent under test needs only one message to trigger its goals. In these cases, random testing is a cheap and efficient technique that can reveal faults. Evidence is provided in the experimental chapter. For the generation of longer sequences that are inherently constructed so as to maximize the likelihood of revealing faults, more sophisticated techniques need to be used, such as manual or evolutionary.

4.3.4.2 Random generation of environment settings

Random testing can also be used to generate random contexts (i.e., environment settings) in which the agents under test operate. As some agents can be programmed to monitor and/or sense the surrounding environment, randomly generated environment settings can lead them to expose different behaviours, yet including faulty ones. Therefore, random generation of environment settings can be effective for agents that have active behaviours with respect to the environment, i.e., sensing, monitoring environmental artefacts.

For example, a cleaning agent has to clean an area in which there can be waste, wastebins, charging stations, and obstacles located at arbitrary locations. By placing these objects randomly, i.e., random generation, there can be some settings where the agent hits obstacles, which is a fault.

This technique can be done by (i) identifying the objects that link to the agent under test, (ii) identifying the attributes of the objects, and (iii) generate randomly values for these attributes. In the example above, we can generate randomly values for the location attribute of waste, wastebins, charging stations, and obstacles.

4.3.5 Evolutionary generation

The specific properties of software agents (autonomous, self-adaptive, learning, and so on) demand for a framework that supports extensive and possibly automated testing. Therefore, we propose to apply ET (Evolutionary Testing) for testing software agents and define two methods to guide the evolution of test cases: mutation guided, and quality function guided.

In this technique, the agents under test are free to evolve during testing, but at the same time their behaviours are observed and used to guide the evolution of test cases, making them more challenging, to run again on the next cycle. Testing objectives, e.g., to see if an agent violates a constraint, are transformed into fitness functions to guide the evolutionary generation of test inputs.

The testing procedure is presented in Figure 4.7. It has the following steps:

1. *Generate initial population.* A set of test cases is called *population.* Each test case is an individual in the population. Initial population can be generated randomly or taken from existing test cases created by testers.

2. *Execution and monitoring.* Test execution means to put the autonomous agents under test into the testing environment so that they can operate, i.e. performing tasks or achieving goals, or to send messages to them. At the same time, a monitoring mechanism is needed to observe the behaviours of the autonomous agents. Relevant observed

data are recorded. Many executions might need to be performed repeatedly (or in parallel) in order to provide statistically sufficient data to measure fitness values in the next step. The agents under test might need a sufficient amount of time to perform their tasks.

3. *Collect observed data and calculate fitness values.* Cumulative data from all executions are used to calculate fitness values of selected test cases. The way of calculating fitness values depends on the stakeholder's softgoal of interest and the problem domain. As calculated fitness values provide insights about the improvement towards the optimal ones, if no improvement is observed after a number of generations, the test procedure will stop. Otherwise step 4 will be invoked.

4. *Reproduction.* Two elite individuals are selected, then crossover operation is used to produce two new offsprings. Finally, mutation is applied with certain probability on one (or both) offsprings. The two offsprings are then put back to the population and the next iteration is triggered, i.e. go back to step 2.

In the following we define two fitness functions and methods to measure them.

4.3.5.1 MUTATION GUIDED

Mutation testing (DeMillo et al. 1978, Hamlet 1977) is a way to assess the adequacy of a test suite and to improve it. Mutation operators are applied to the original program (i.e., an agent under test) in order to artificially introduce known defects. The changed version of the program is called a *mutant*. For example, a mutant could be created by modifying a branch condition, e.g., the following JADE code (Telecom Italia Lab 2000):

if (msg.getPerformative() == ACLMessage.REQUEST)

can be changed into

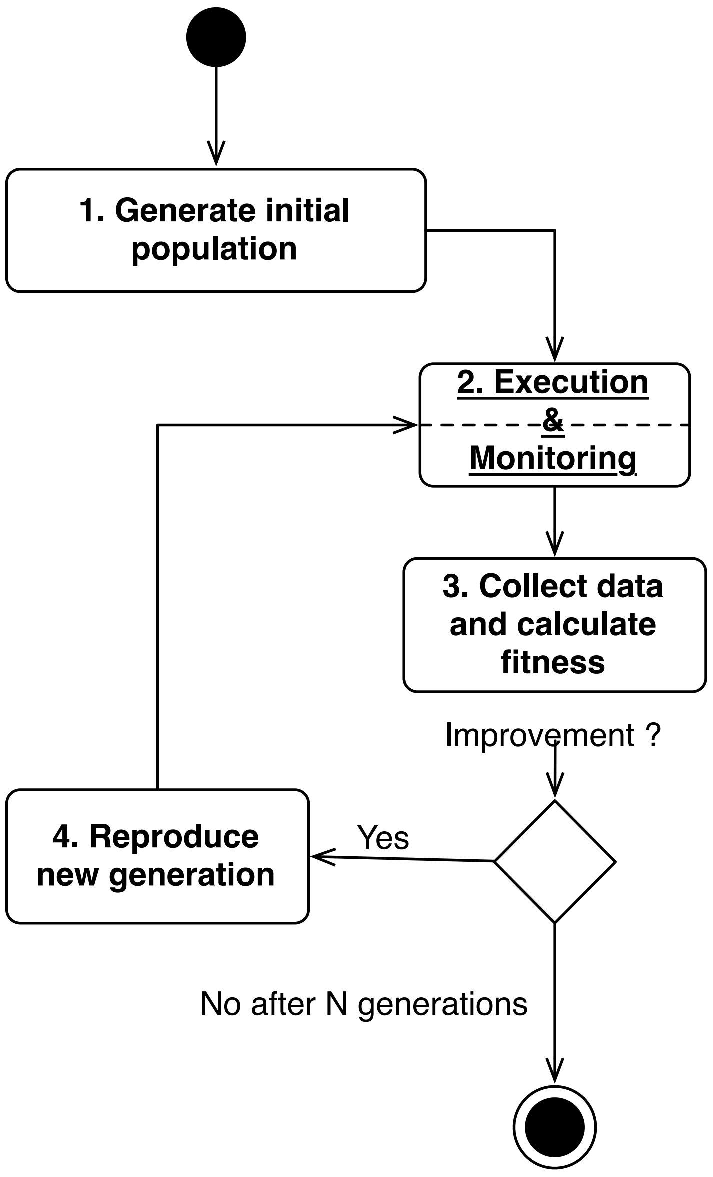

1. Generate initial population
2. Execution & Monitoring
3. Collect data and calculate fitness
Improvement ?
Yes
4. Reproduce new generation
No after N generations

Figure 4.7: Evolutionary testing procedure

if (msg.getPerformative() == ACLMessage.REQUEST_WHEN)
or a mutant can be created by modifying a method invocation (e.g., *receive()* changed into *blockingReceive()*).

A test case is able to reveal the artificial defects seeded into a mutant if the output of its execution deviates from the output of its execution on the original program. In such a case, the mutant is said to have been *killed*. The adequacy of a test suite is measured as the ratio of all the killed mutants over all the mutants generated. When such a ratio is low, the test suite is considered inadequate and more test cases are added to increase its capability of revealing the artificially injected faults, under the assumption that this will lead to revealing also the "true" faults.

We combine mutation and evolutionary testing for the automated generation of the test cases. We name this technique Evol-Mutation. In particular, we use the mutation adequacy score as a fitness measure to guide evolution, under the hypothesis that test suites that are better at killing mutants are also likely to be better at revealing real faults.

Given the agent under test A, we apply mutation operators to A to produce a set of mutants $\{A_1, A_2, \ldots, A_N\}$. Each contains one fault.

After a test case T is executed against A and its mutants $\{A_1, A_2, \ldots, A_N\}$, fitness function of T_i is calculated as $F(TC_i) = \frac{K_i}{N}$, where K_i is the number of mutants killed by T_i. To increase performance, the executions of T_i on the mutants should be performed in parallel (e.g., on a cluster of computers, with one mutant per node).

4.3.5.2 QUALITY FUNCTION GUIDED

As discussed in Section 4.1.3, stakeholders' softgoals can be used to derive quality functions to judge the quality of the agents under test. The agents are faulty or unreliable if quality functions are not as expected.

We propose an evaluation methodology consisting of two main steps:

1. *Representing stakeholder softgoals as quality functions.* Relevant softgoals that need to be used to evaluate agent autonomy are transformed or represented as quality functions for measuring stakeholder satisfaction. This transformation is domain specific and depends on the nature of the softgoal as well as on the problem domain.
2. *Evolutionary testing.* In order to generate varied tests with increasing level of difficulty, we advocate the use of meta-heuristic search algorithms that have been used in other work on Search Based Software Engineering (Harman 2007), and, more specifically, we advocate the use of evolutionary algorithms. The quality functions of interest are used as objective functions to guide the search towards generating more challenging test cases.

Let's consider again the cleaner agents in Section 4.1.3, we can use the closest distance to obstacles as fitness function to guide the generation of test cases. The evolutionary algorithm will then optimize this function; smaller is better, meaning that the later test cases have higher probability of pushing the cleaner agents to hit obstacles, which is considered as fault.

4.4 Continuous execution

Throughout the chapter we have studied techniques to evaluate and observe agent's behaviour, and techniques to generate test cases automatically. This section discusses how to put them in action together in a method called continuous testing. It is a test execution process in which automated input generation, evaluation, and evolution make a closed loop. This method can proceed without human intervention.

Testing software agents can be achieved very naturally by means of a dedicated Tester Agent (TA) which continuously interacts with agents under test, and of a monitoring agent network which checks those agents states. Since agents communicate primarily through message passing, the TA can send messages to other agents to stimulate behaviours that can potentially lead to fault discovery. The messages sent by the TA are those encoded in the test suites, which can in turn be manually derived from goal diagrams --following the GOST methodology 3-- or automatically generated. It is then the monitoring agents' responsibility to observe the reactions to the messages sent by the TA and, in case these are not compliant with the expected behaviour (post-conditions violated) or crashes happen, to inform the development team that a fault was revealed.

Furthermore, the TA can also use the random and the quality-function-guided evolutionary techniques to generate environment settings. This can be applied to test agents that depend on the environment.

Since the behaviour of an agent can change over time, due to the mutual dependencies among agents and to their learning capabilities, a single execution of test suites might be inadequate to reveal faults. The use of the TA allows for an arbitrary extension of the testing time, that can proceed unattended and independently of any other human-intensive activity. The TA is empowered with generation techniques, described previously, to evolve existing test suites and to generate new ones, with the aim of exercising and stressing the application as much as possible, the final goal being the possibility to reveal yet unknown faults.

The continuous testing process is shown, as an UML activity diagram, in Figure 4.8. The human tester has to start the process and check the final results; other activities are performed by the TA and the monitoring agents. Notice that at the *generating / evolving test cases* stage, we can apply not only evolutionary generation techniques but also ontology-based or random ones as well. In some cases, even with test cases remaining unchanged, continuous process is still effective because of the peculiar properties (learning, self-adaptivity) of the agents under test.

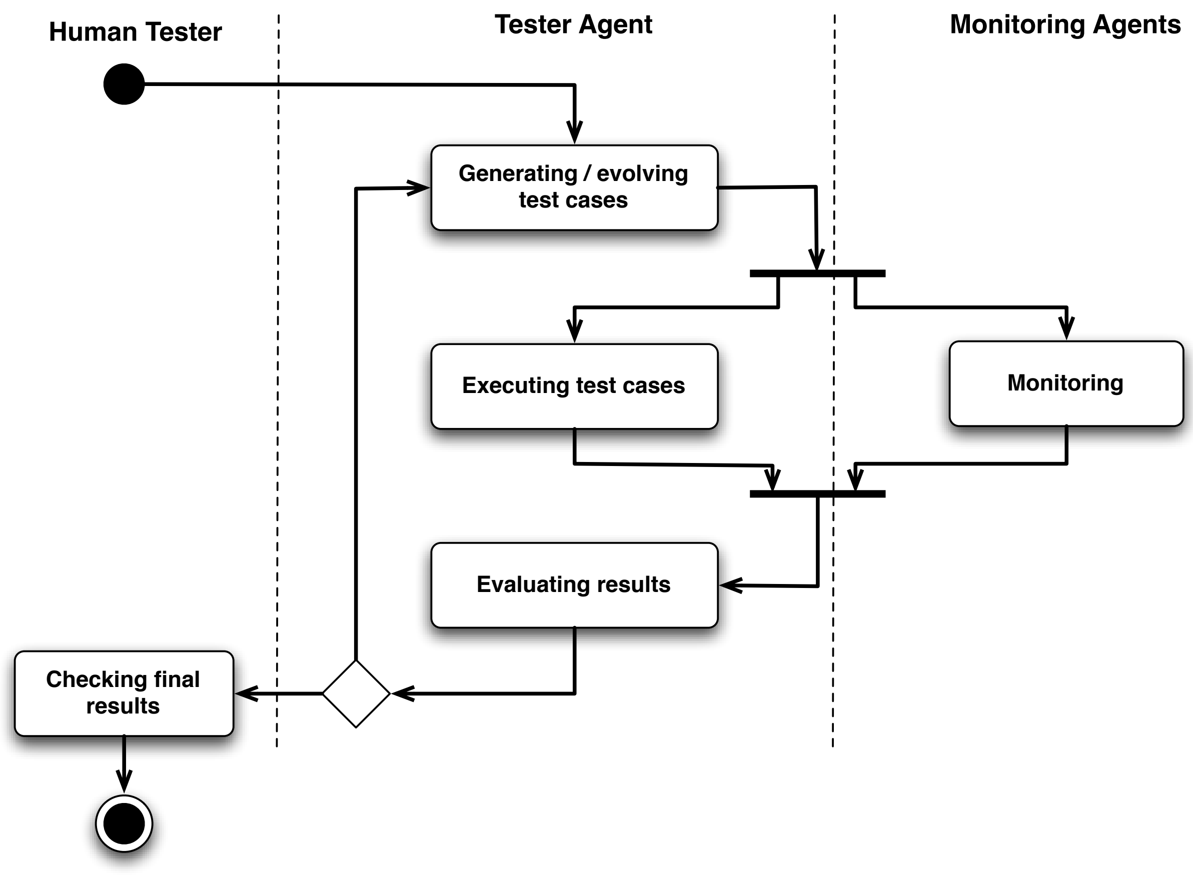


Figure 4.8: Continuous testing process

4.5 Summary

In summary, this chapter has discussed a novel testing method for software agents: continuous testing. It consists of automated test input generation, evaluation, monitoring techniques, and eventually automated execution.

Four generation techniques have been investigated. The goal-oriented one takes Tropos analysis diagrams, produced by using TAOM4E [4], to derive test suites to test for goal fulfillments. The ontology-based and random technique exploit available agent interaction ontology, interaction protocols, and domain data to generate messages ready to be submitted to the agents under test. The random technique can also be used to generate environmental settings. The advanced evolutionary technique implements the evolution algorithm to evolve existing test cases to produce new and more challenging ones based on runtime feedbacks. These feedbacks include the number of mutants killed (mutation-guided) or the distances to quality thresholds (quality-function-guided).

The following table, Table 4.4, summarizes the types of test input that these techniques generate so far:

Table 4.4: Testing techniques and test input types

Technique	**Messaging type**	**Environment type**
Goal-oriented	Yes	Yes
Ontology-based	Yes	No
Random	Yes	Yes
Mutation-guided evolutionary	Yes	No
Quality-function-guided evolutionary	No	Yes

For what concerns the evaluation of agent behaviours, we proposed to use constraints such as norms to detect faulty behaviours that violate these constraints. We proposed to use ontology to check if messages sending from

[4] http://sra.fbk.eu/tools/taom4e

agents are semantically and syntactically correct, and use requirements to judge if the agents under test are reliable given their autonomy. Monitoring technique is, then, used to observe, guard, and provide instant feedback information for test input generation.

Special agents including the TA and the monitoring agents are equipped with these techniques to make continuous and automated testing possible. The TA continuously generates or evolves test cases, using random, ontology-based, or evolutionary approach, and then executes them, while the monitoring agents monitor the behaviours of the agents under test, report faults, and provide desired information for evolution. We will discuss these agents in Chapter 5.

Chapter 5

eCAT testing framework

We build a testing frame work called eCAT (stand for Environment for Continuous Agent Testing) to support the GOST methodology presented in Chapter 3 and different testing techniques presented in Chapter 4. The framework consists of the TA, monitoring agent network, and tools for test case specification, graphical visualization, continuous execution, and fault reporting. eCAT is available online at `http://code.google.com/p/open-ecat/`.

The architecture of eCAT is presented in Figure 5.1. It consists of three main components: *Test Suite Editor*, allowing human testers to derive test cases from goal analysis diagrams; TA, capable to generate automatically new test cases and to execute them on a MAS; and *Monitoring Agents*, that monitor communication among agents, including the TA, and all events happening in the execution environments in order to trace and report errors. *Remote monitoring agents* are deployed with the environments of the agents under test, transparently to them, in order to avoid possible side effects. All the *remote monitoring agents* are under the control of the *Central monitoring agent*, which is located at the same host as the TA. The monitoring agents overhear agent interactions, events, and constraint violations taking place in the environments, providing a global view of what is going on during testing and helping the TA evaluate test results.

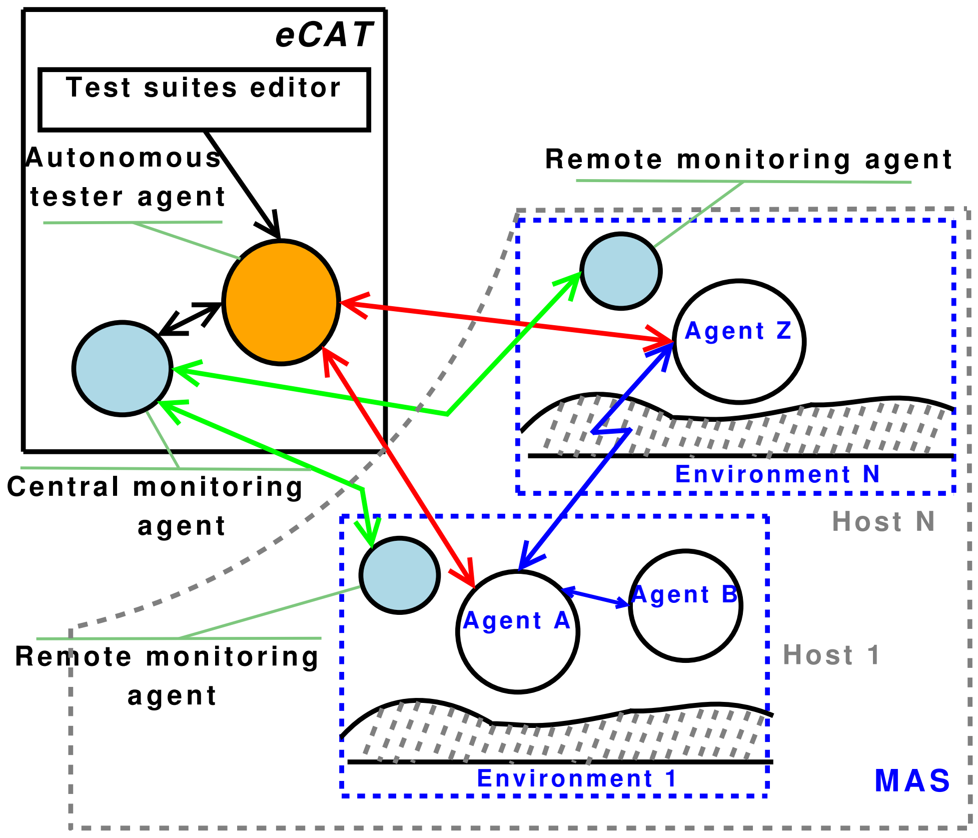


Figure 5.1: eCAT framework

eCAT features (i) *specification tool* that allows generating test case skeletons from goal analysis diagrams produced using TAOM4E (`http://sra.fbk.eu/tools/taom4e`), and editing them graphically; (ii) *generation and execution tool* that can generate and evolve test cases, and execute them continuously; (iii) *monitoring tool* to help observing and reporting faults. Details are introduced in the following sections.

5.1 Specification tool

Test suite structure has been discussed in Section 3.4.5. A test suite contains a set of test cases and suite supporting actions, e.g., set-up and tear-down. Each test case consists of one test scenario and specific supporting actions. A test scenario is composed of a list of test actions (such as sending a message, receiving a message, and so on) and their subsequent links.

Graphically we propose to represent a test scenario as finite state machine in which test actions and order links are represented as states and transitions, respectively. As an example, Figure 5.2 depicts a scenario that is used to test an agent that communicate using FIPA REQUEST protocol (FIPA 2002b). The explanation of the scenario is described in Figure 5.3. First, the TA sends a request to the agent under test, then the TA waits for a return message. In case the reply is "accepted" then the TA checks the next "INFORM" message and then finishes, otherwise it does nothing and stops.

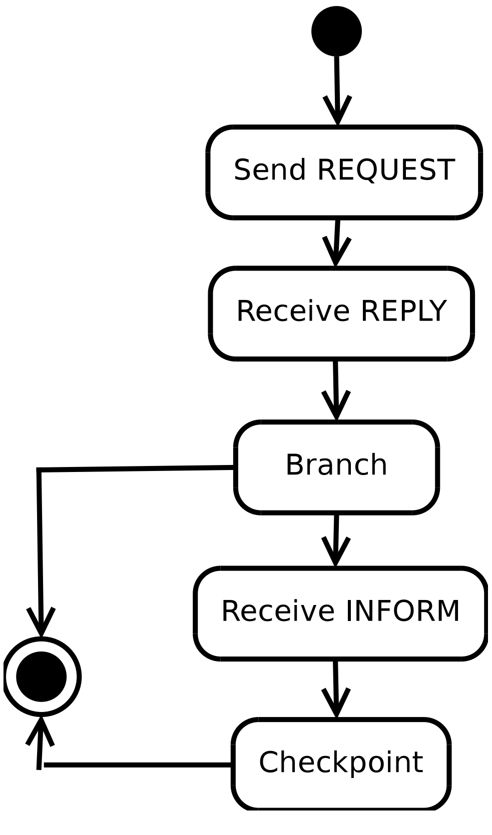


Figure 5.2: An example of FSM presentation of a test scenario

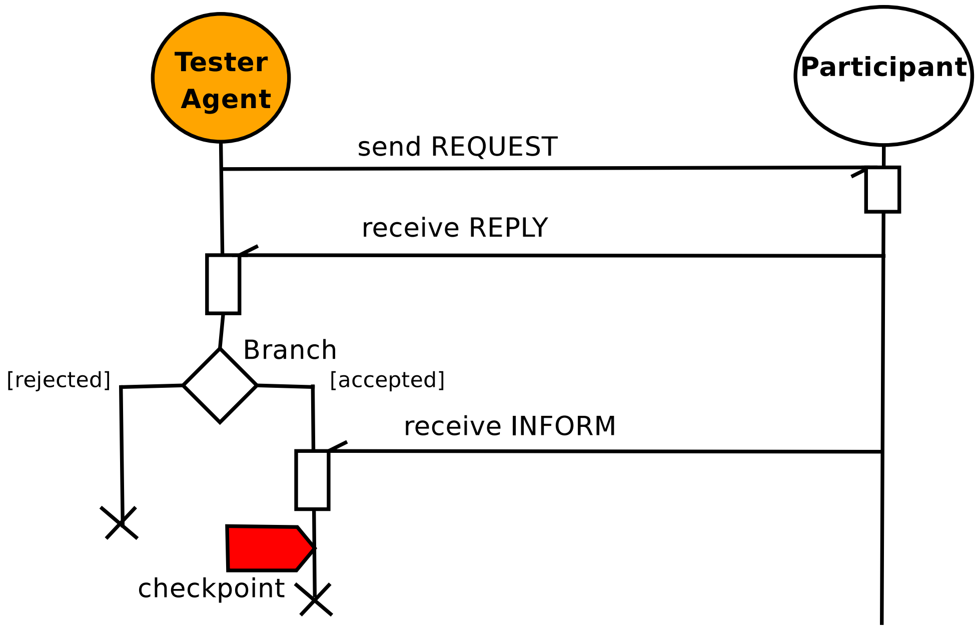


Figure 5.3: Description of the test scenario in Figure 5.2

The *Test Suite Editor* of eCAT is built on top of the Eclipse Graphical Modeling Framework (GMF) (*The Eclipse Graphical Modeling Framework (GMF)* n.d.). It has three main features:

1. Generate test suite automatically from Tropos goal analysis diagrams, edited in TAOM4E.

2. Provide wizards to create test suites from FIPA standard (or user-defined) interaction protocol (FIPA 2002b).

3. Allow editing test suite graphically, in a drag-drop-like fashion. Test scenarios are presented as finite state machines, in each state user can specify specific testing action such as communication, checking, branching, and test data such as message content, oracle.

Figure 5.4 depicts the graphical editing environment where end-user can easily visualize and edit a test suite, which is composed of test cases, the specific agent to be tested, its goals, and so on. On the right side of the

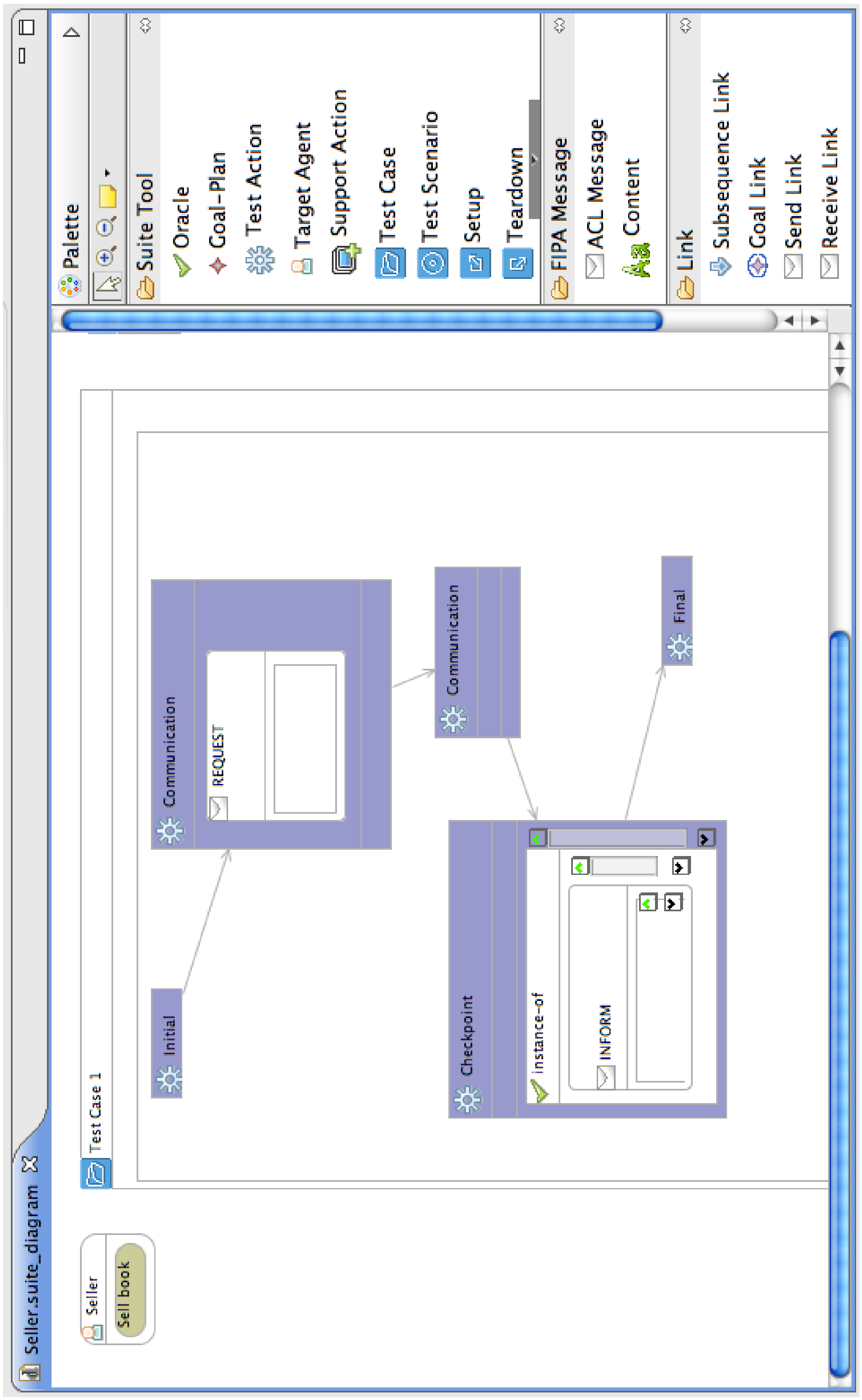
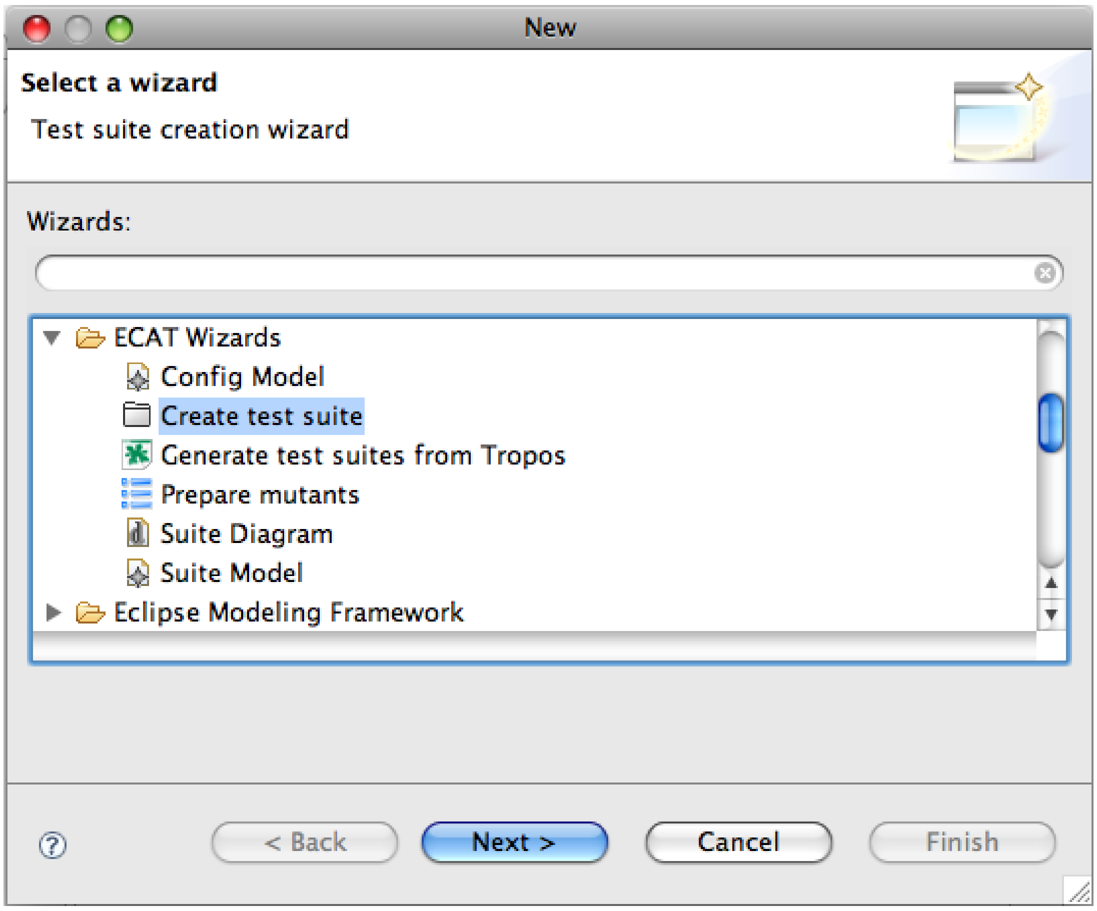


Figure 5.4: eCAT test suite editor

figure there is a drawing palette, it contains artefacts being ready for use to drag-drop into the suite. Detailed information, such as outgoing message content, expected communication act can be specified easily as well.



Select a wizard

Test suite creation wizard

Wizards:

- ECAT Wizards
  - Config Model
  - Create test suite
  - Generate test suites from Tropos
  - Prepare mutants
  - Suite Diagram
  - Suite Model
- Eclipse Modeling Framework

Figure 5.5: eCAT generation wizards

eCAT provides a number of wizards, see Figure 5.5. Using them one can generate test suites from Tropos diagrams, create a test suite based on existing interaction protocol, prepare mutation testing environments, and so forth.

5.2 Generation and execution tool

Four test cases generation techniques are equipped to eCAT: *Goal-oriented*, *Ontology-based*, *Random*, and *Evolutionary*. Details are presented in Chapter 4 while their brief description follows.

GOAL-ORIENTED. Goal-oriented test cases generation is a part of the GOST methodology presented in 3 that integrates testing into Tropos, providing a systematic way of deriving test cases from Tropos output artefacts. eCAT can take these artefacts as inputs to generate test case skeletons that are aimed at testing goal fulfillment. Specific test inputs (i.e. message content), and expected outcome are partially generated from plan design (e.g., UML activity or sequence diagrams) and are then completed manually by testers.

ONTOLOGY-BASED. eCAT takes advantage of agent interaction ontologies, which define the semantics of agent interactions, in order to generate automatically both valid and invalid test inputs, to provide guidance in the exploration of the input space, and to obtain a test oracle against which to validate the test outputs.

RANDOM. eCAT is capable of generating random test cases. First, the TA selects a communication protocol among those provided by the agents platform, e.g., FIPA Interaction Protocol (FIPA 2002b). Then, messages are randomly generated and sent to the agents under test. The message format is that prescribed by the agent environment of choice (such as the FIPA ACLMessage (FIPA 2002a)), while the content is constrained by a domain data model. Such a model prescribes the range and the structure of the data that are produced randomly, either in terms of generation rules or in the (simpler) form of sets of admissible data that are sampled randomly.

EVOLUTIONARY. Evolutionary algorithms guided by mutation or quality-function-based fitness are implemented in eCAT, allowing it to evolve test cases during test execution. Based on monitoring data from the current execution, the TA can evolve the existing test cases (current population) to be more challenging ones for the next execution.

All the above-mentioned techniques can be used in the continuous test execution mechanism of eCAT. Testing process is seen as a loop of generating, executing and monitoring, evaluating, evolving (only in evolutionary technique), then go back to generating. This continuous process makes it possible to test software agents extensively and automatically.

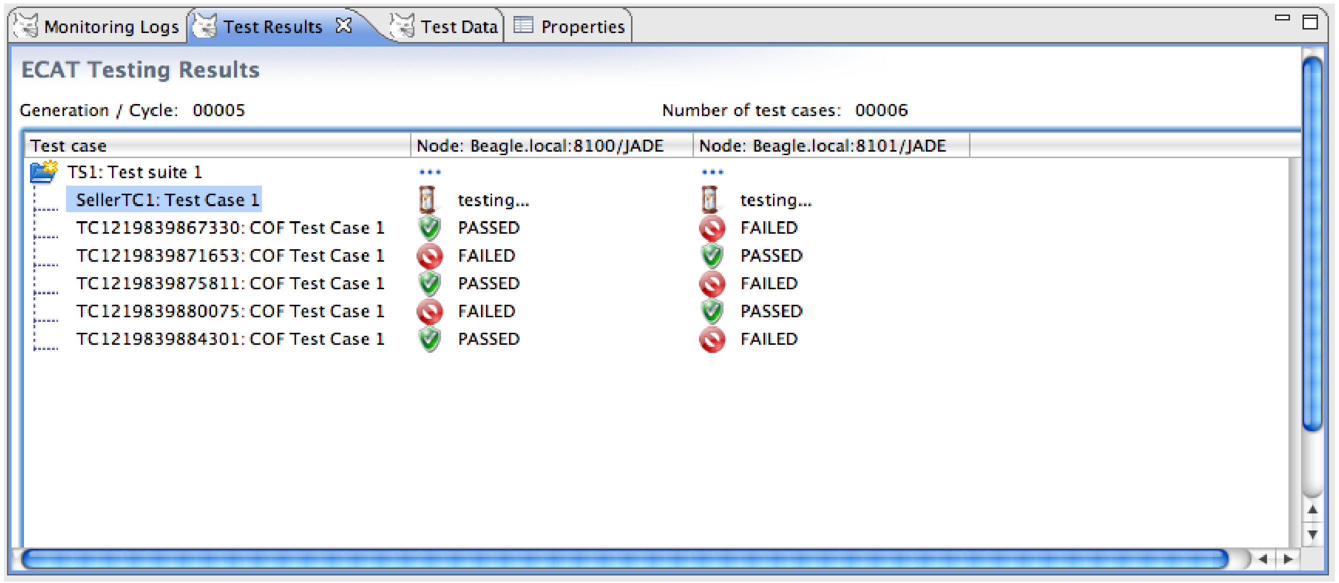
eCAT provides runtime view of testing process and results. Example is shown in Figure 5.6. At the same point we can see testing results of a number of agent platforms in parallel, the number of generations/cycles has been passed so far, and the number of test cases exercised. Use can select a concrete test case to see the failure, if any, in order to speculate the cause of the failure.

Concrete test data, can be generated automatically or manually defined, of each test case can be viewed through data view, Figure 5.7. In that figure, detailed test actions are encoded using XML.

5.3 Monitoring tool

eCAT contains a network of monitoring agents: the remote monitoring agents that side in agent platforms guard for events, violations, interactions happened at platform level during testing, while the central agent incorporates monitoring data from all the remote agents, makes the available for evaluating test results and reporting. Multiple agent platforms that are used for testing can be located at a same host (i.e. computer) or at geographically different hosts thank to the monitoring network.

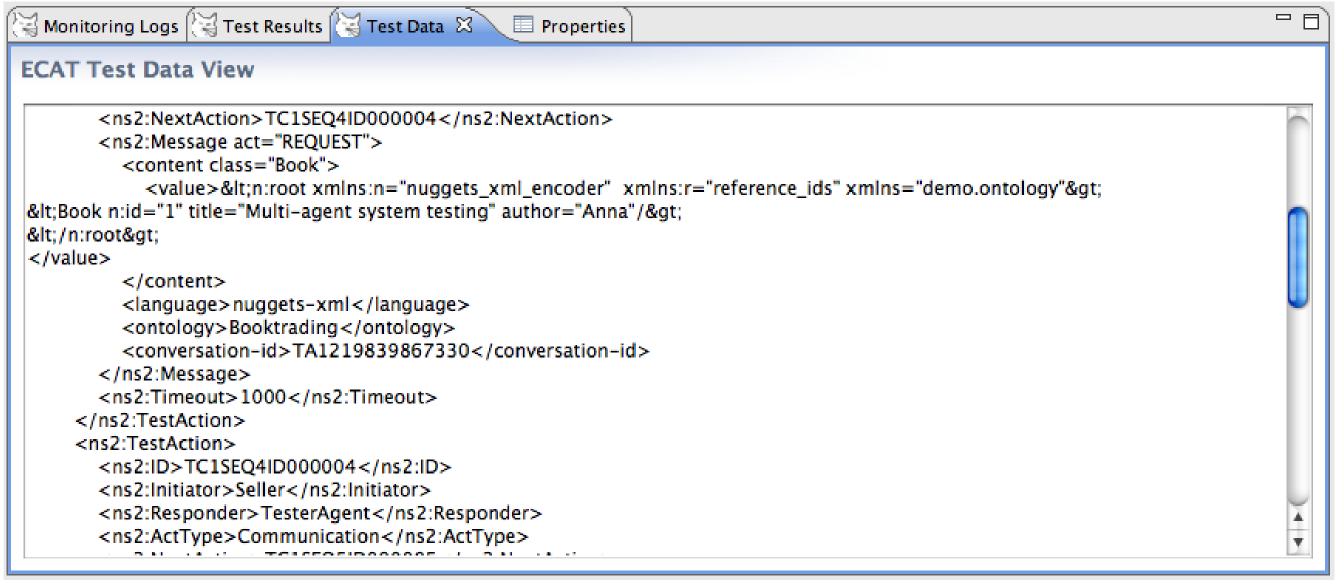
For example, Figure 5.9 shows two remote monitoring agents running



Generation / Cycle: 00005 Number of test cases: 00006

Test case	Node: Beagle.local:8100/JADE	Node: Beagle.local:8101/JADE
TS1: Test suite 1	•••	•••
SellerTC1: Test Case 1	testing...	testing...
TC1219839867330: COF Test Case 1	✓ PASSED	🚫 FAILED
TC1219839871653: COF Test Case 1	🚫 FAILED	✓ PASSED
TC1219839875811: COF Test Case 1	✓ PASSED	🚫 FAILED
TC1219839880075: COF Test Case 1	🚫 FAILED	✓ PASSED
TC1219839884301: COF Test Case 1	✓ PASSED	🚫 FAILED

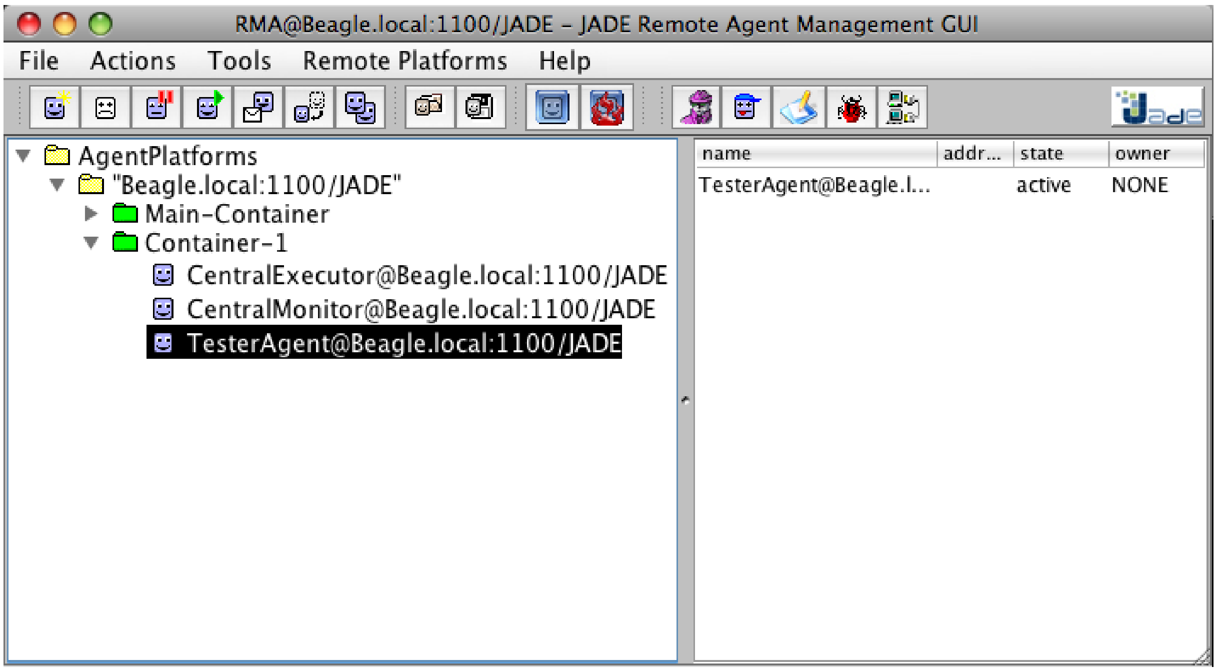
Figure 5.6: eCAT test result view



```
        <ns2:NextAction>TC1SEQ4ID000004</ns2:NextAction>
        <ns2:Message act="REQUEST">
          <content class="Book">
            <value>&lt;n:root xmlns:n="nuggets_xml_encoder"  xmlns:r="reference_ids" xmlns="demo.ontology"&gt;
&lt;Book n:id="1" title="Multi-agent system testing" author="Anna"/&gt;
&lt;/n:root&gt;
</value>
          </content>
          <language>nuggets-xml</language>
          <ontology>Booktrading</ontology>
          <conversation-id>TA1219839867330</conversation-id>
        </ns2:Message>
        <ns2:Timeout>1000</ns2:Timeout>
      </ns2:TestAction>
      <ns2:TestAction>
        <ns2:ID>TC1SEQ4ID000004</ns2:ID>
        <ns2:Initiator>Seller</ns2:Initiator>
        <ns2:Responder>TesterAgent</ns2:Responder>
        <ns2:ActType>Communication</ns2:ActType>
```

Figure 5.7: eCAT test data view

at two different platforms. The two JADEX/JADE (Pokahr et al. 2005, Telecom Italia Lab 2000) platforms run at a same host but with two different communication ports. The two remote monitoring agents collaborate with the central monitoring agent, shown in Figure 5.8, during test execution to provide traces, reports, and desired events to observe. The central monitoring agent runs in *Container-1* in the figure.


RMA@Beagle.local:1100/JADE – JADE Remote Agent Management GUI
File
Actions
Tools
Remote Platforms
Help

```
▼ AgentPlatforms
  ▼ "Beagle.local:1100/JADE"
    ▶ Main-Container
    ▼ Container-1
        CentralExecutor@Beagle.local:1100/JADE
        CentralMonitor@Beagle.local:1100/JADE
        TesterAgent@Beagle.local:1100/JADE
```

name	addr...	state	owner
TesterAgent@Beagle.l...		active	NONE

Figure 5.8: eCAT: the central monitoring agent in action

Similar to test result view, eCAT also provides runtime view of monitoring traces, including interactions, events, constraint violations, desired guards. This gives the developer a global view of what happens during testing and helps locating problems. In Figure 5.10, we can easily observe an interaction event, from the TA to an agent called *Seller*; the content of the interaction is partially presented in the *Event details* section. In Figure 5.11, we can see a constraint violation taking place in the body of the *MakeProposalPlan* plan; details of the violation says that the constraint: "$acceptable\_price > 0$ and $acceptable\_price < 2000$" has been violated.

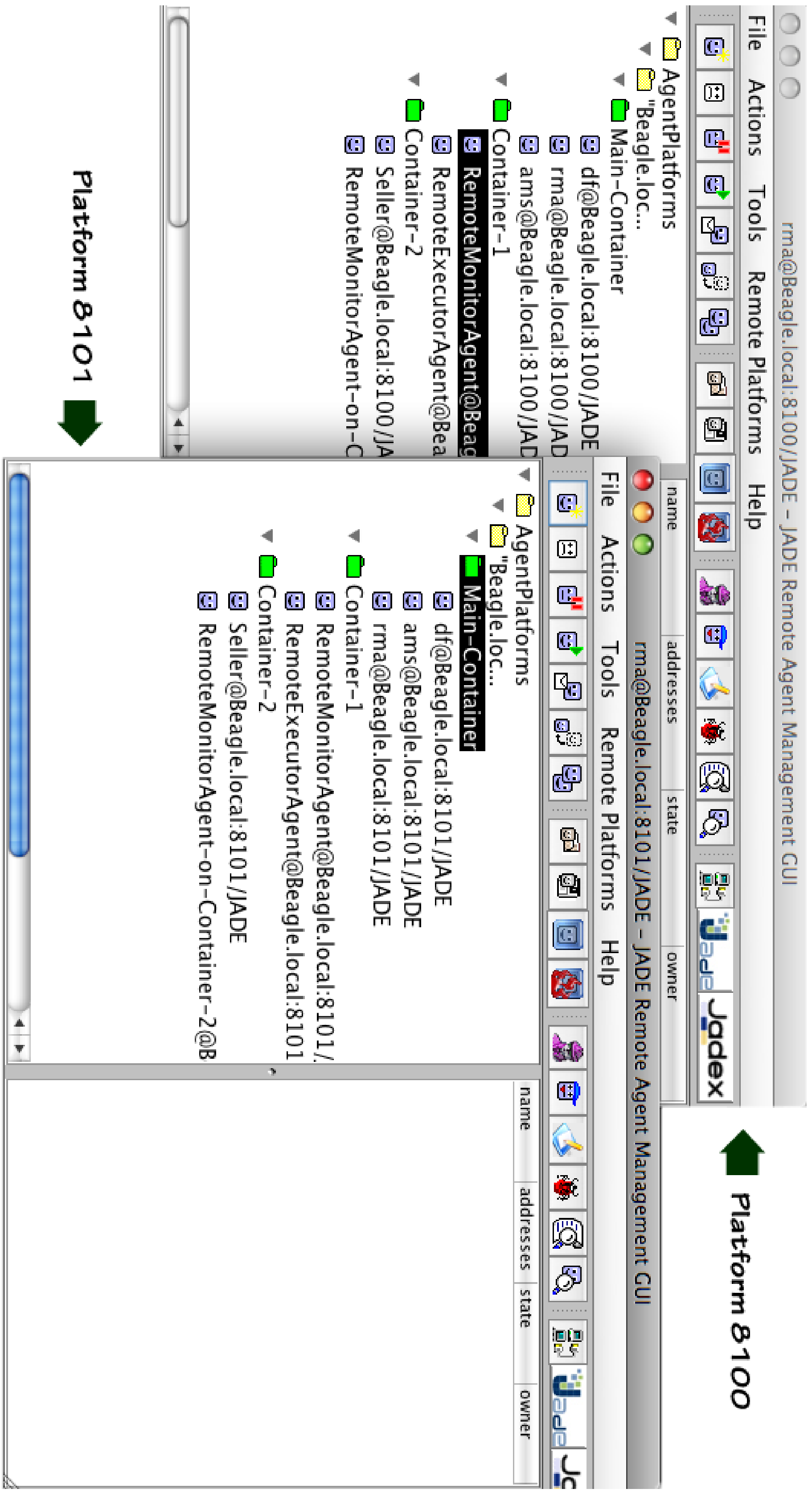
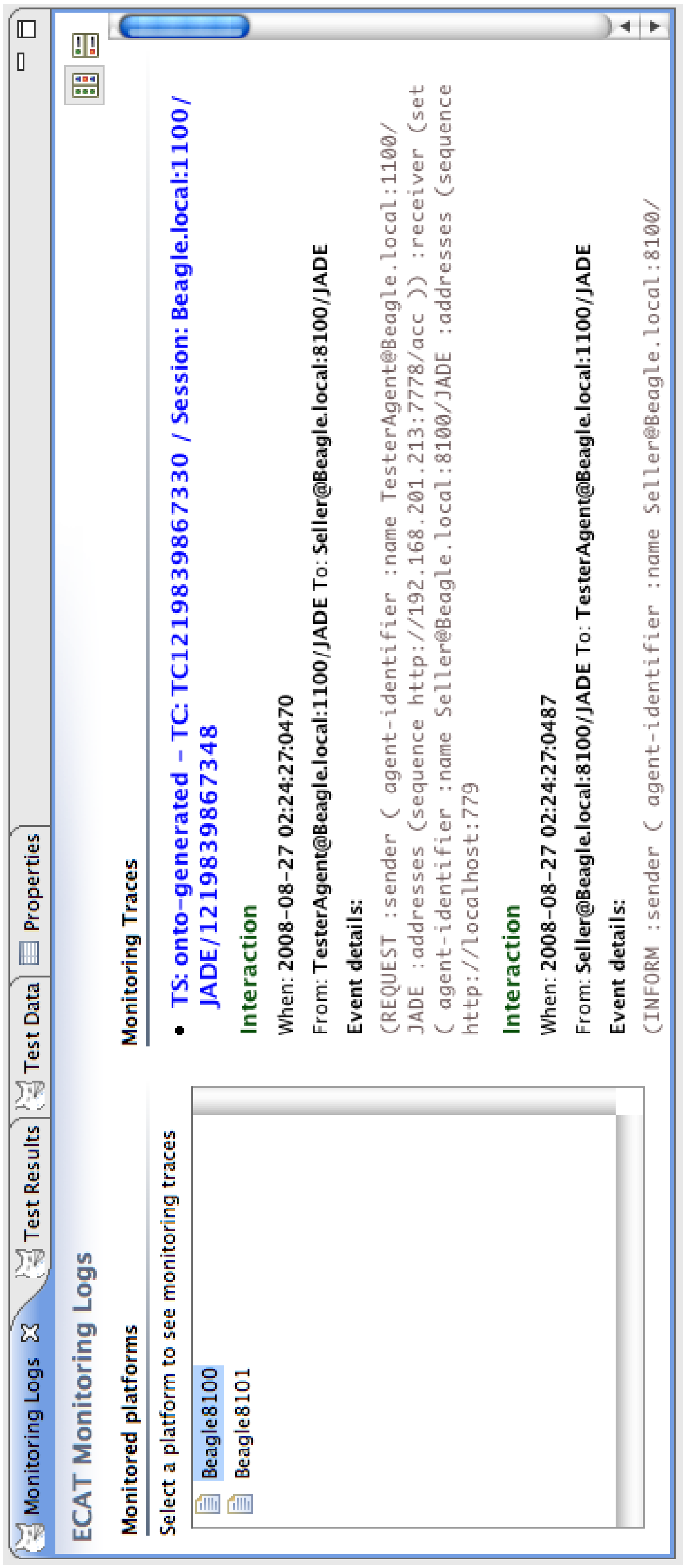

rma@Beagle.local:8100/JADE - JADE Remote Agent Management GUI
File Actions Tools Remote Platforms Help
AgentPlatforms
"Beagle.loc...
Main-Container
df@Beagle.local:8100/JADE
rma@Beagle.local:8100/JAD
ams@Beagle.local:8100/JAD
Container-1
RemoteMonitorAgent@Beag
RemoteExecutorAgent@Bea
Container-2
Seller@Beagle.local:8100/JA
RemoteMonitorAgent-on-C
name addresses state owner
Platform 8100
rma@Beagle.local:8101/JADE - JADE Remote Agent Management GUI
File Actions Tools Remote Platforms Help
AgentPlatforms
"Beagle.loc...
Main-Container
df@Beagle.local:8101/JADE
ams@Beagle.local:8101/JADE
rma@Beagle.local:8101/JADE
Container-1
RemoteMonitorAgent@Beagle.local:8101/
RemoteExecutorAgent@Beagle.local:8101
Container-2
Seller@Beagle.local:8101/JADE
RemoteMonitorAgent-on-Container-2@B
name addresses state owner
Platform 8101

Figure 5.9: eCAT: the remote monitoring agents in action



ECAT Monitoring Logs

Monitored platforms

Select a platform to see monitoring traces

Beagle8100

Beagle8101

Monitoring Traces

- TS: onto-generated - TC: TC1219839867330 / Session: Beagle.local:1100/JADE/1219839867348

Interaction

When: 2008-08-27 02:24:27:0470

From: TesterAgent@Beagle.local:1100/JADE To: Seller@Beagle.local:8100/JADE

Event details:

```
(REQUEST :sender ( agent-identifier :name TesterAgent@Beagle.local:1100/
JADE :addresses (sequence http://192.168.201.213:7778/acc )) :receiver (set
( agent-identifier :name Seller@Beagle.local:8100/JADE :addresses (sequence
http://localhost:779
```

Interaction

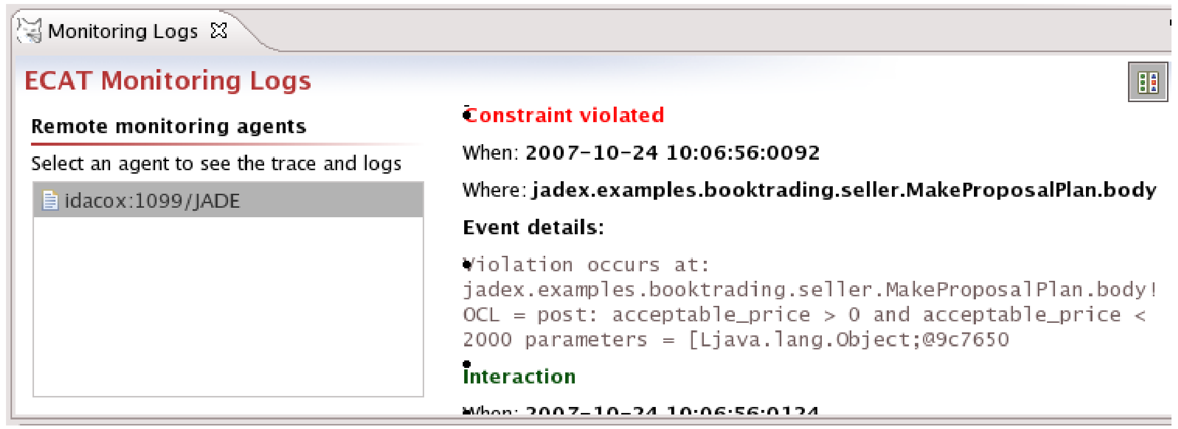
When: 2008-08-27 02:24:27:0487

From: Seller@Beagle.local:8100/JADE To: TesterAgent@Beagle.local:1100/JADE

Event details:

```
(INFORM :sender ( agent-identifier :name Seller@Beagle.local:8100/
```

Figure 5.10: eCAT monitoring: interaction view



Constraint violated

When: 2007-10-24 10:06:56:0092

Where: jadex.examples.booktrading.seller.MakeProposalPlan.body

Event details:

```
Violation occurs at:
jadex.examples.booktrading.seller.MakeProposalPlan.body!
OCL = post: acceptable_price > 0 and acceptable_price <
2000 parameters = [Ljava.lang.Object;@9c7650
```

Interaction

When: 2007-10-24 10:06:56:0124

Figure 5.11: eCAT monitoring: violation view

Chapter 6

Experiments and results

We have conducted many experiments to evaluate our proposed approaches to the automated generation of test cases and continuous execution, presented in Chapter 4. In this chapter, we present three experiments, their objectives and results.

Table 6.1 summarizes the techniques used in the experiments and the links to download available code and materials. In the first experiment, Section 6.1, we build a MAS called *BibFinder* to study the performance of eCAT in continuous testing and the effectiveness of the random and the Evol-Mutation generation technique. In the second experiment, two ontologies and two MAS systems of different size have been used to evaluate the ability of eCAT in generating test inputs based on agent interaction ontology. Finally, in the last experiment, we investigate the use of quality-based fitness function in generating environment settings to test autonomous cleaner agents.

Some of the results of these experiments have been presented in (Nguyen et al. 2008c,b,a, 2007), others are under review at the time of writing this chapter.

Table 6.1: List of experiments

No	**Techniques**		**Downloadable packages**
	Generation	*Evaluation*	
1	mutation-guided evolutionary, goal-oriented, random	constraint-based	http://se.itc.it/dnguyen/bibfinder
2	ontology-based		http://se.itc.it/dnguyen/thesis/supports/BibFinderBDI.tgz http://se.itc.it/dnguyen/thesis/supports/BookTrader.tgz
3	quality-function-guided evolutionary, random	requirement-based	http://se.itc.it/dnguyen/thesis/supports/MrCleaners.tgz

6.1 Continuous testing

This section describes the experimental results obtained when we used eCAT to test *BibFinder*. First, we introduce *BibFinder*, the MAS under test, its features and architectural design. Then, the different testing techniques applied to *BibFinder*, the testing results, and our evaluation are presented.

BibFinder is a MAS for the retrieval and exchange of bibliographic information in BibTeX format[1]. *BibFinder* is capable of scanning the local drivers of the host machine, where it runs, to search for bibliographic data in the format of BibTeX. It consolidates databases spread over multiple devices into a unique one, where the queried item can be quickly searched. *BibFinder* can also exchange bibliographic information with other agents, in a peer-to-peer manner, thus augmenting its search capability with those provided by other peer agents. Moreover, *BibFinder* performs searches on

[1]http://www.ecst.csuchico.edu/~jacobsd/bib

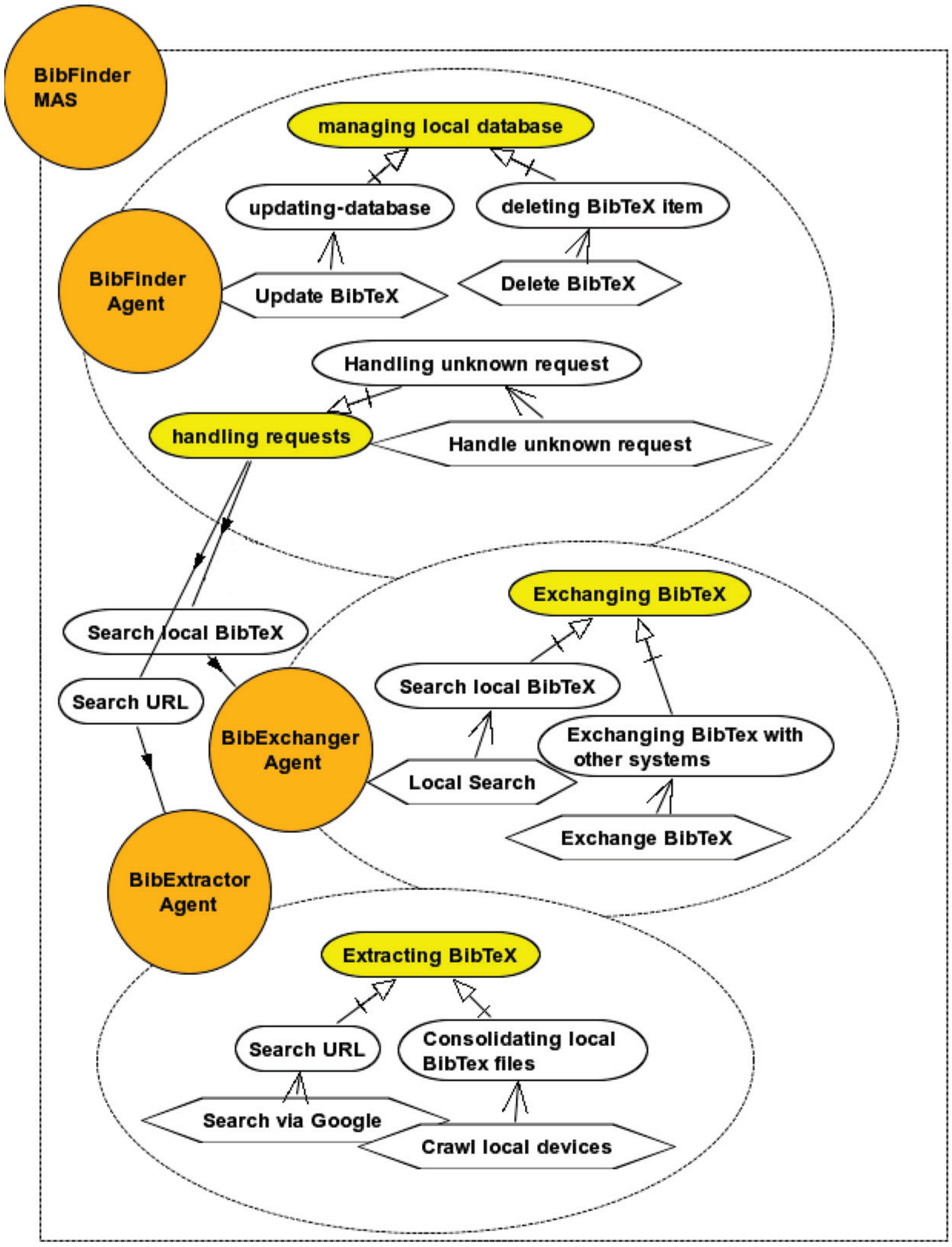


Figure 6.1: Architectural design of *BibFinder* in TAOM4E

and extracts BibTeX data from the Scientific Literature Digital Library[2], exploiting the Google search Web service[3].

Figure 6.1 depicts the architectural design of *BibFinder* in Tropos. The system contains three agents: *BibFinderAgent*, *BibExchangerAgent*, and *BibExtractorAgent*. Roles of each agent are briefly described as follows: *BibFinderAgent* maintains the local BibTeX database and coordinates the operation of the system as a whole; *BibExchangerAgent* is in charge of querying the local database and exchanging data with external agents (e.g., with other instances of *BibFinder*); *BibExtractorAgent* crawls on local storage devices looking for BibTeX files, and performs searches on and extracts BibTeX items from the Internet.

Each agent in *BibFinder* is responsible for some goals and depends on the other agents for fulfilling some other goals. Inside each agent, a given goal can be decomposed into sub-goals, resulting in a tree of goals, in which each leaf goal has a specific plan as means to achieve the goal. For instance, *BibFinderAgent* has two root goals *Managing-local-database* and *Handling-requests*; the former is decomposed into *Updating-database* and *Deleting-BibTeX-item* goal. The plan *Update-BibTeX*, for adding new items or updating existing items in the database, acts as means to achieve the goal *Updating-database*. When serving external requests, *BibFinderAgent* depends on *BibExtractorAgent* for seeking URLs and on *BibExchangerAgent* for querying the local database. Similarly, *BibExtractorAgent* and *BibExchangerAgent* also have goal decompositions and plans specified to fulfil their goals.

This version of *BibFinder* is implemented in JADE (Telecom Italia Lab 2000).

[2]http://citeseer.ist.psu.edu

[3]http://code.google.com/apis/soapsearch

6.1.1 Testing BibFinder

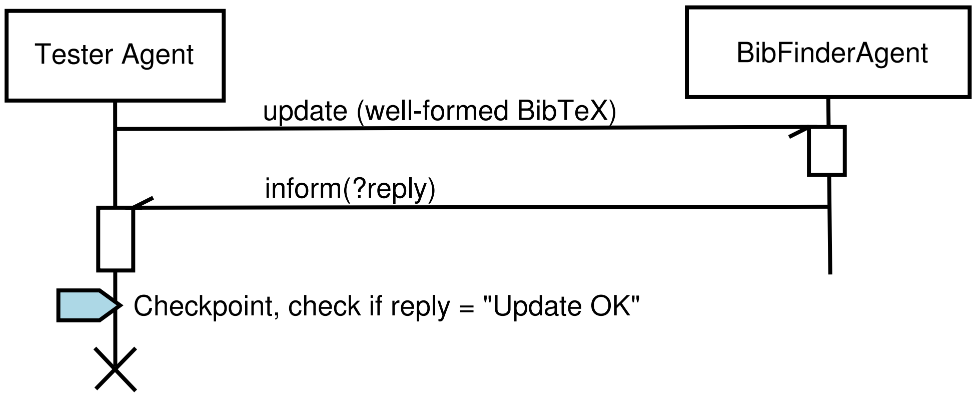
We applied three testing techniques included in eCAT when testing *BibFinder*: (1) random testing, which mainly uncovered bugs that make *BibFinder* crashed; (2) goal-oriented testing, aimed at verifying if the agents in *BibFinder* can fulfil their goals; and (3) Evol-Mutation testing, aimed at revealing more bugs thanks to the possibility of continuous execution. Detailed descriptions of these techniques are provided in Chapter 4.

6.1.1.1 Goal-oriented testing

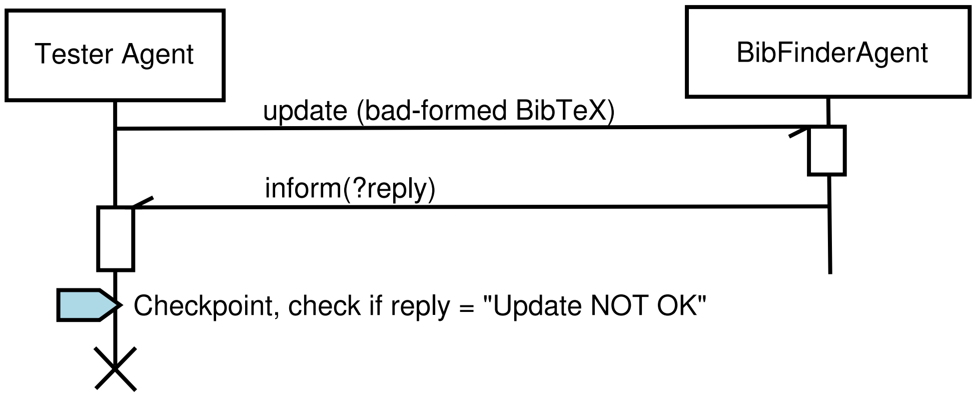
Based on the 6 *Means-End* relationships in *BibFinder*'s architectural design (there are 7 relationships in total, but we excluded the one related to the goal *crawl-local-devices* because it was not fully implemented), we derived 6 test suites to test the fulfilment of the associated goals. This derivation follows the goal-oriented software testing methodology discussed in Chapter 3. These test suites contain 12 test cases specifying 12 different test scenarios. For example, test suite *TS3* was derived to test the fulfilment of the goal *Updating-database* by the plan *Update-BibTeX*. Two test scenarios are presented in Figure 6.2. The scenario 6.2(a) reads: the TA sends a request to *BibFinderAgent*, it then waits for a reply. If the replied message contains the content "Update OK" then the test scenario passes, in cases when timeout occurs or message content differs from "Update OK", the test scenario is considered as failed.

6.1.1.2 Goal-oriented testing enhanced by coverage

Given a test suite, such as the one derived through goal-oriented testing, statement coverage can be measured and used to make sure that all the code has been exercised by at least one test case (excluding any unreachable code). We enhanced goal-oriented testing by manually adding 3 new test


Tester Agent
BibFinderAgent
update (well-formed BibTeX)
inform(?reply)
Checkpoint, check if reply = "Update OK"

(a) Simple scenario with valid data


Tester Agent
BibFinderAgent
update (bad-formed BibTeX)
inform(?reply)
Checkpoint, check if reply = "Update NOT OK"

(b) Simple scenario with invalid data

Figure 6.2: Test suite *TS1*, used to test the goal *Updating-database*

cases, in order to reach 100% statement coverage of the main packages. In other words, we complement black-box testing with white-box testing: by analyzing the coverage rate by means of the tool GroboCodeCoverage[4], we figured out the uncovered code and added test cases able to increase the coverage level, up to 100% coverage.

6.1.1.3 Random testing

In order to apply the random test case generation technique during continuous testing, we built a library of interaction protocols and a repository of domain data. The interaction protocols include the five FIPA protocols *Propose, Request, Request-When, Subscribe*, and *Query* (FIPA 2002b), and twenty-one (simple) protocols, which are created from twenty-one different FIPA communication performatives, such as *AGREE, REQUEST*, etc. Domain data have been collected from the test suites derived from the goal model and have been manually augmented with additional possible input values. The TA generates test cases by selecting domain data randomly and combining them with interaction protocols. The TA continuously generates test cases and executes them against *BibFinder*. The *Monitoring Agents* are in charge of observing the whole system, i.e. *BibFinder* and the JADE platform (Telecom Italia Lab 2000). Based on the intercepted information, it can recognize the situations in which bugs are revealed (e.g., some agents crash).

6.1.1.4 Evol-mutation testing

The preparation step of Evol-Mutation testing consists of creating initial test cases, as initial individuals, and creating mutants of the original *BibFinder* system. The initial population contains the 12 test cases derived from the goal-oriented testing technique. Since *BibFinder* agents are

[4]http://groboutils.sourceforge.net/codecoverage

implemented in JADE, a pure Java platform, we can apply existing object-oriented mutation operators for Java on them in order to create mutants. It would be better to have agent-oriented specific mutation operators, but unfortunately, to the knowledge of the authors, no work has investigated this issue yet. We consider it as a future work.

We adapted the tool MuClipse[5], built on top of μJava (Yu-Seung Ma and Kwon 2005), to create mutants from the source code of three agents: *BibFinderAgent*, *BibExchangerAgent*, *BibExtractorAgent*.
The source code of the supporting classes was left untouched. 24 class-level and 15 statement-level mutation operators (Yu-Seung Ma and Kwon 2005) were applied on those agents. After combining the results, we obtained 178 mutants of *BibFinder* to be used in Evol-Mutation testing.

6.1.2 Results

We conducted testing experiments with the goal-oriented (G), coverage-enhanced goal-oriented (G^+), and random (R) techniques on a computer equipped with 2G RAM, processor Core 2 Duo 1.86GHz (named Host in the following). The last technique, Evol-Mutation testing, was used with the original version of *BibFinder* running on the Host and 15 mutants running on 3 cluster machines (4GB RAM, 4 CPUs Xeon 3GHz). These experiments were repeated 10 times for each technique in order to measure the average time and the ability to discover faults. Each execution time is composed of execution cycles, in which test cases are run on *BibFinder* and its mutants. Test cases executed in each cycle can be the same in the goal-oriented and coverage-enhanced goal-oriented techniques; but they are different in random testing. In the Evol-Mutation testing, the test cases executed in a cycle are those from the previous cycle plus one or two new test cases generated by evolution.

[5]http://muclipse.sourceforge.net

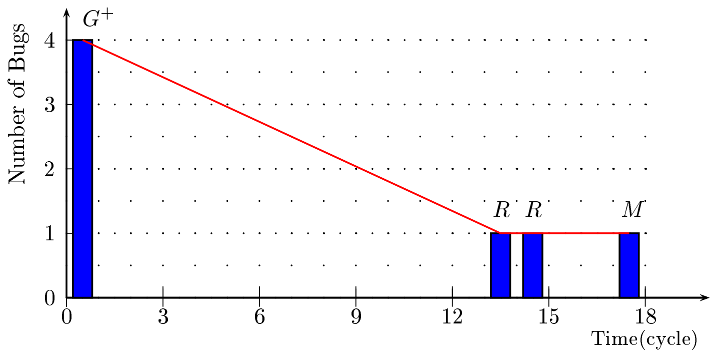


Figure 6.3: Real bugs revealed by cycle

To assess the performance of eCAT we considered real bugs of *BibFinder* that were detected during its development and artificial faults inserted into the code according to the fault seeding method (Harrold et al. 1997). The real faults of *BibFinder* detected by eCAT are presented in Table 6.2. Fault No. 1 says that the *BibFinderAgent* crashed when it was asked to parse a BibTeX; fault No. 2 says that JADE does not support creating a new thread within *BibExtractorAgent*; fault No. 3 shows that the *BibFinderAgent* fails to forward messages to the *BibExchangerAgent* when those messages come from a different JADE platform; etc. Faults are classified by the severity level (i.e. *Fatal* faults make agents die, *Moderate* faults are associated with discrepancies between implementation and specification). In the **Cycle / generation** column, we can find the average cycle (generation in the case of Evol-Mutation technique) when bugs were uncovered. One cycle of random testing costs less time than one of goal-oriented testing and Evol-Mutation. Figure 6.3 depicts the number of bugs uncovered per cycle.

In Table 6.2 and Figure 6.3 we can notice that Random testing is quite effective in detecting fatal bugs. It actually revealed two real fatal bugs and one of them was not detected by any other technique. Goal-oriented testing revealed moderate bugs, showing that the implemented agents fail

Table 6.2: Results of continuous testing on BibFinder

No	Bug	Bug type	Cycle / generation	Technique
Real bugs				
1	BibTeX parsing	Fatal	14	R
2	Using thread in *BibExtractorAgent*	Moderate	1	G, G^+, M
3	Forward message error	Moderate	1	G, G^+, M
4	No reply to incorrect requests	Moderate	1	G^+, M
5	Lack a required data field	Moderate	1	G, G^+, M
6	Update wrong BibTeX	Fatal	G:1, R:15, M:1	G, G^+, M, R
7	Add new wrong BibTeX	Fatal	18	M
Artificial bugs				
8	Index out of bound in *BibExtractorAgent*	Fatal	G^+:1, R:9	G^+, R
9	Always reply null	Moderate	1	G^+
10	No answer to any request	Moderate	1	G^+
11	Index out of bound in *BibExchangerAgent*	Fatal	G^+:1, R:9	G^+, R
12	Return incorrect BibTeX	Moderate	1	G^+
13	Null exception to an array	Fatal	1	G^+
14	Reply wrong performative	Moderate	1	G^+
15	Handle invalid request error	Moderate	1	G^+
16	Infinite loop	Fatal	16	R
17	Null reference from *BibFinderAgent* to *BibExtractorAgent*	Fatal	1	G^+
18	Null reference from *BibFinderAgent* to *BibExchangerAgent*	Fatal	1	G^+
R: Random (0.1 minutes / cycle) , G: Goal-oriented (0.13 minutes / cycle with 12 test cases), G^+: Coverage-enhanced Goal-oriented (0.16 minutes / cycle with 15 test cases), M: evol-Mutation (3.9 minutes / cycle with 15 initial test cases)				

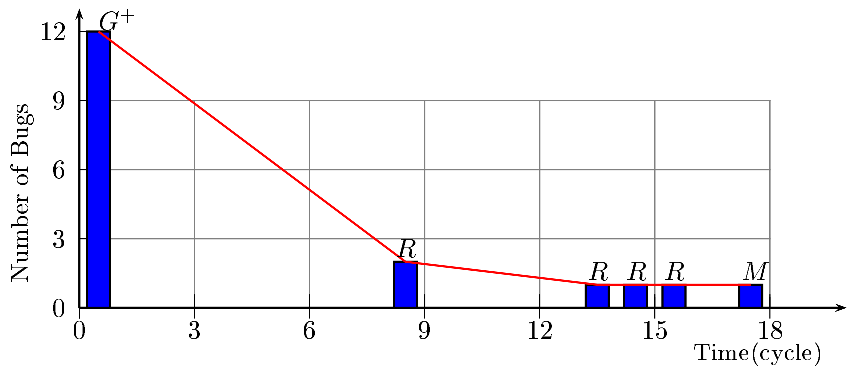


Figure 6.4: Total (real and artificial) bugs revealed by cycle

to fulfil their goals. These moderate bugs were uncovered easily, right at the first cycle, because the agents of *BibFinder* exhibited only reactive behaviours. Since proactive agents could behave differently at different cycles, more test cycles may be necessary to bring proactive agents to a state that reveals faults. However, more experiments are needed to prove this. Finally, looking at the results, we can see that Evol-Mutation testing reveals the bugs uncovered by goal-oriented testing (this is expected since Evol-Mutation takes the test cases used by goal-oriented technique as initial inputs), but, more importantly, Evol-Mutation revealed also bug No. 7, which was not detected by any other technique. This bug was uncovered by mutating a message and enriching its content with data taken from the dynamically constructed database.

To further evaluate the performance of eCAT, we used also the fault seeding method (Harrold et al. 1997). We involved 3 PhD students with a lot of skill and experience in MAS development and asked them to insert realistic bugs (i.e., bugs regarded as similar to real bugs as possible) into *BibFinder*. We obtained 15 copies of *BibFinder*, each containing one bug. First, we ran Coverage-enhanced goal-oriented and Random techniques to find bugs on these copies. Then, we ran Evol-Mutation on the copies left, i.e. containing bugs that could not be found by the other two techniques.

Because Evol-mutation uses test cases of Coverage-enhanced goal-oriented technique as initial inputs, bugs found by the later is a subset of bugs found by the former, so we only need to run Evol-mutation to find the bugs that are left.

Eventually, 11 of these bugs were uncovered by one or more of the testing techniques under study. eCAT could not detect 4 bugs pertaining to *BibExtractorAgent*, even with Evol-mutation technique. These bugs were inserted into the crawling functionality of *BibExtractorAgent* (related to the goal *crawl-local-devices*), by which the agent is able to scan and monitor changes in local directories, in order to search for BibTeX files. These directories can be considered as an environment to *BibFinder* and those bugs can be revealed only by changing this environment. No test cases have been created to test this goal, thus none of these 4 bugs can be revealed.

Table 6.3: Mean time between failures

Technique	G	R	R	R	R	M
Time (m)	0.13	0.9	1.4	1.5	1.6	70.2
Numb. of Bugs	12	2	1	1	1	1
MTBF (m)	0.01	0.39	0.5	0.1	0.1	68.6

The summary of bugs found by each technique is shown in Figure 6.4, where they are plotted against the testing cycles. The mean time between failure (MTBF) is depicted in the log-log plot in Figure 6.5. The detailed MTBF values in minute are presented in Table 6.3. We can see that the mean time between two bugs found at the beginning of testing is very small. Then, it tends to increase, although not always monotonically. Since the number of remaining bugs decreases, it becomes harder and harder to reveal them. After the last bug found (around 1 hour from the beginning of testing) no more bug is revealed by eCAT. In a real development scenario, eCAT can be left running continuously, so as to try to reveal also those

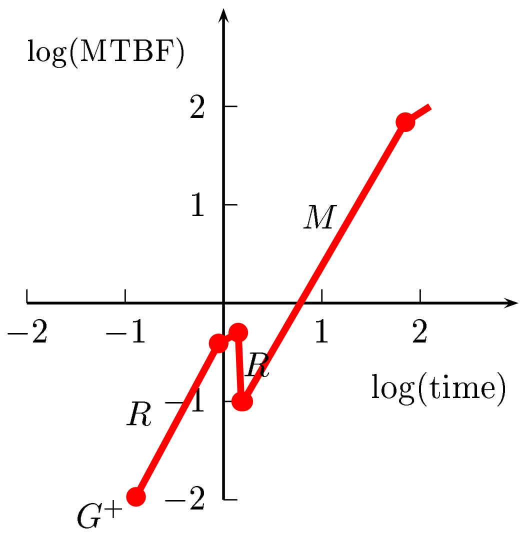


Figure 6.5: Log-log plot of mean time between failures

bugs that are associated with a very long mean time between failures and are thus extremely hard (or impossible) to reveal in traditional testing sessions. Going back to Figure 6.4, we can notice that the goal-oriented and the random techniques are quite effective in the initial testing cycles, when bugs can be revealed by simple and short message sequences and the selection of the input data is not critical to expose them (i.e., there exist large equivalence classes of input data that can be used interchangeably to reveal a given fault). When remaining bugs become hard to find (last testing cycles) goal-oriented and random testing become ineffective and it is only through Evol-Mutation that additional faults can be revealed.

6.2 Ontology-based generation

We have evaluated the performance of the ontology-based test generation, discussed in Section 4.3.3, as well as its capability of revealing faults on two case studies. The first case study (*BookTrader*) is a book-trading MAS. This system was implemented as a set of BDI agents (Rao and Georgeff

1995) in JADEX (Pokahr et al. 2005). We extended it to support ontology-based interaction. After modeling the interaction ontology (see Figure 4.1) using Protégé[6], we generated ontology-supporting code, and modified the implementation of *Seller* and *Buyer* agents accordingly. Moreover, we added OCL constraints (e.g., the price must be between 0 and 2000). The size of this MAS is 1312 line of code (LOC).

On the first assessment on the performance of test generation for *Book-Trader*, we were able to obtain three ontologies with instances of books, comprising respectively 10, 20 and 100 instances. Generation rules introduced in Chapter 4 are used. We applied valid input generation rules **RVS1, RVS2, RVS3** for the *Book* properties *author* and *title*; **RVN1, RVN2, RVN3** for *Proposal*'s *price*; **RVC1** for the combination. We generated invalid messages using **RIS4, RIS5** for *author* and *title*; **RIN2** for *price*; **RIC1, RIC2, RIC3** for the combination.

Table 6.4 shows the total number of possible valid and invalid test inputs that could be generated by the test generator from three different input ontologies, *Onto1, Onto2, Onto3*. *Onto1* contains 10 instances of `Book`. Since `Book` has two properties (*title* and *author*) of type string, if we assume that a dictionary with 3 valid values is available for each property, the test generator can produce in total $(10 + 3)^2 = 169$ valid and $(10+4)*2-1 = 27$ invalid inputs. The `Propose` also has two properties, one of type `Book` and the other one, *price* of type float. Since *price* has a constraint on it value from 0 to 2000, the generator can generate randomly at least 3 valid values within the range and 5 invalid values: *null, overflow, underflow, lower than 0, greater than 2000*. So `Propose` can have $10 * 3 = 30$ immediate valid and $10 * 5 = 50$ immediate invalid inputs; immediate in the sense that it takes directly 10 instances of `Book`. However, based on the fact that `Book`

[6]Protégé is a free, open source ontology editor and knowledge-base framework. Available at `http://protege.stanford.edu`

can have 169 valid and 27 invalid values, `Propose` can therefore have up to 507 valid and 980 invalid values. This is a considerable amount with respect to just 10 initial instances.

Table 6.4: Number of possible inputs from the book-trading ontology

Ontology	Concept	Number of instances	Valid inputs	Invalid inputs
Onto 1	Book	10	169	27
	Propose	-	30→507	50→980
Onto 2	Book	20	529	47
	Propose	-	60→1587	100→2880
Onto 3	Book	100	10609	207
	Propose	-	300→31827	500→∼ 100000

Table 6.5 (a) shows the total number of test cases (divided into valid and invalid test cases) that were generated by executing continuous testing. Test case generation for the *Seller* required the creation of 3 test case templates, while only one template was needed for the *Buyer*. The small number of templates indicates that little manual effort is required by our approach. In fact, the template definition is the only step that requires the human involvement.

The two classes in the *BookTrader* ontology were fully covered by the automatically generated test cases. Moreover, two deviations from the expected behaviour (faults) were observed. Manual testing of the same application was conducted by applying the goal-oriented test case derivation methodology. Results are provided in Table 6.5 (b) and show that 6 test cases were manually defined for each agent under test. They cover the same number of classes in the ontology and reveal the same faults as the automatically generated test cases. Although in this example ontology-based test case generation exhibits no superior performance in terms of ontology coverage or fault detection, it increases the confidence in the correctness of the application, in that it allows exploring a much larger portion of the input space (more than one order of magnitude) at no additional cost.

The second case study *BibFinder* is a MAS that aims at facilitating bibliographic search. Differ from the case study discussed in Section 6.1, this *BibFinder* is a true BDI agent (Rao and Georgeff 1995). A special assistant agent called *BibFinder* helps searching and building references for a specific topic, sharing bibliographic data with other *BibFinder*. In particular, *BibFinder* has the capability to:

1. Consolidate bibliographic data automatically, even when they are scattered geographically;
2. Perform searches on and extract bibliographic data from the Scientific Literature Digital Library[7], exploiting the Google search service[8];
3. Rank publications automatically based on the usage history;
4. Form communities of *BibFinder* automatically in order to share bibliographic and ranking data of similar topics of interest;
5. Join an existing or create a new community based on the interests of the *BibFinder*'s owner;
6. Recommend a list of "must-read" papers to the owner.

Similar to *BookTrader*, *BibFinder* is implemented as a BDI agent (Rao and Georgeff 1995) in JADEX (Pokahr et al. 2005). The size of this MAS is 8484 LOC. The main differences between *BookTrader* and *BibFinder* are that the former has been evolved together with the several versions of JADEX (Pokahr et al. 2005), hence it is likely to contain less faults than the latter, which was implemented recently from scratch. Moreover, the interaction ontology of *BibFinder* is much larger than *BookTrader*'s one.

[7]http://citeseer.ist.psu.edu
[8]http://code.google.com/apis/soapsearch

Notice that the version of *BibFinder* in this experiment is implemented in JADEX, which is different from the previous version of *BibFinder* used in the previous experiment.

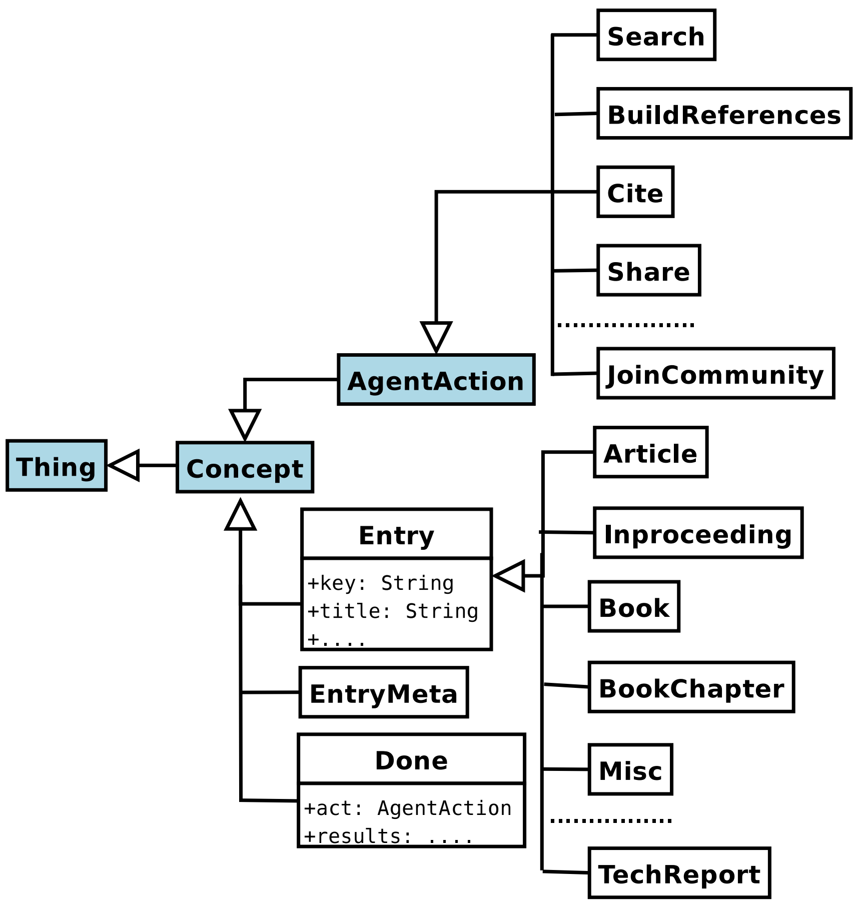


Figure 6.6: Interaction ontology of *BibFinder*

A portion of the interaction ontology of *BibFinder* is presented in Figure 6.6. The complete ontology is quite big, because of the number of properties (e.g., `Entry` has 42 properties) and classes not shown in Figure 6.6 for space reasons (a lot of sub-classes of `Entry` and `AgentAction` are not shown in the figure).

We used a set of BibTeX files, comprising a large number of BibTeX entries, to create a domain-specific ontology that specifies and contains BibTeX data. It was then aligned with the *BibFinder* ontology (shown in

Table 6.5: Faults and coverage evaluation (N/A means Not Applicable). Table 6.5 (a) presents results with automatically-generated test cases, while Table 6.5 (b) presents results with manually-derived test cases.

MAS	**Agent under test**	**Numb of templates**	**Time limit**	**Number of test cases**			**Ontology coverage**	**Numb of faults**
				Valid	**Invalid**	**Total**		
BookTrader	*Seller*	3	30'	209	131	340	2/2	2
	Buyer	1	10'	56	38	94	2/2	2
BibFinder	*BibFinder*	5	1h	853	423	1267	27/27	6

(a) Automatically generated test cases

MAS	**Agent under test**	**Numb of templates**	**Time limit**	**Valid**	**Invalid**	**Total**	**Ontology coverage**	**Numb of faults**
BookTrader	*Seller*	N/A	N/A	3	3	6	2/2	2
	Buyer	N/A	N/A	3	3	6	2/2	2
BibFinder	*BibFinder*	N/A	N/A	7	7	14	13/27	4

(b) Manually derived test cases

Figure 6.6). In total, we obtained 983 ontology instances.

Table 6.5 (a) shows the total number of test cases generated for *BibFinder*, ontology coverage and revealed faults. Compared to the manually derived test cases (Table 6.5, (b)), the continuous and ontology-based testing allowed a much wider exploration of the input space, with a higher ontology coverage and fault revealing capability. Reliability of *BibFinder* is substantially improved after the execution of continuous and ontology-based testing.

6.3 Requirement-based evolutionary generation

In this experiment we further analyze the cleaner agent, introduced briefly in Section 4.1.3, and build a simulation of an agent system composed of an artificial environment and the cleaner agent to evaluate the proposed approach. We describe, in detail, the functionalities of the agent and the way we use softgoals to guide test generation and ultimately evaluate the quality of the agent.

As mentioned in Section 4.1.3, we choose two softgoals: *robustness* and *efficiency* to evaluate the quality of the cleaner agent. By analyzing *robustness*, two further goals decomposed from it are *maintaining-battery* and *avoiding-obstacles*. Similarly, *efficiency* can be decomposed into sub-goals as well. All of them must be taken into account while evaluating the quality of the cleaner agent. Each softgoal gives rise to a fitness function that can guide the generation of test cases. In this section we investigate only the goal *avoiding-obstacles*, using a fitness function derived from the goal to guide the generation of test inputs. The testing objective is to make sure that the agent does not hit any obstacles.

6.3.1 Application

The artificial environment is a square area, A. In the area A there can be obstacles, dustbins, waste, and charging stations located randomly. We define an *environmental setting* as a particular configuration of A, in which numbers of obstacles, dustbins, waste, and charging stations are located at particular locations. Different settings pose different levels of difficulty in which the cleaner agent must operate.

The cleaner agent is in charge of keeping that area clean. In particular, it needs to perform the following tasks autonomously:

1. Explore location of important objects;
2. Look for waste and bring it to the closest bin;
3. Maintain battery charge, with sufficient re-charging;
4. Avoid obstacles by changing course when necessary;
5. Exhibit alacrity by finding the shortest path to reach a specific location, while avoiding obstacles on the way.
6. Exhibit safely by stopping gracefully when movement becomes impossible or battery charge level is too low.

These are all requirements related to autonomy; the way in which the agent achieves them differs depending on the context in which it finds itself. Each functionality gives rise to a goal that the agent needs to achieve or maintain, and multiple goals are active simultaneously during operation. For instance, while exploring the area, the agent needs to avoid obstacles and maintain its battery.

The simulation environment is implemented in JADEX (Pokahr et al. 2005), extending an existing example of JADEX with a sophisticated capability to avoid obstacles. The cleaner agent contains a belief base where information about current location, visited locations, obstacles, dustbins, charging stations, and so on are stored. In addition, the agent has a number of goals and associated plans, with goal deliberation based on goal conditions, such as creation, adoption and inhibition conditions. At runtime, goals are adopted autonomously on the basis of goal deliberation.

6.3.2 Preparation

6.3.2.1 Encoding test inputs

In this case study, an environmental setting (or a test case) is composed of the quantity and location of obstacles, dustbins, waste, and charging stations. Each of these factors is encoded as a single gene, as follows (See also Figure 6.7):

- Divide the area A into RxR cells, R is called *resolution*.
- Place objects (i.e. obstacles, waste, bins, charging stations) into cells. A cell containing an object is denoted by 1, while a content–free cell is denoted by 0.

The resolutions of the environmental factors can be different and their quantity can be controlled in evolutionary testing. For instance, we can

1	0	1	0	1	0
0	1	1	0	1	0
0	1	1	0	1	0
1	1	1	0	1	1
0	1	0	1	0	0
1	0	1	0	0	0

Figure 6.7: Encoding test inputs: an example of 6x6 cells

choose the number of dustbins and charging stations to be as small as than they are in reality, while the amount of waste and number of obstacles can be chosen to be much higher.

During evolution, genes are crossed over and/or mutated, resulting in new environments that combine previous environments or in which objects change their locations.

6.3.2.2 Fitness computation

We define a fitness function f based on the distance to obstacles encountered during the operation of the agent. Real-time observations of the distance of the cleaner agent to all obstacles are performed to measure f. Moreover, since the test outcomes are different even for the same test input, we need to repeat the execution of each test case several times to measure statistical data representing the test outcomes. This section determines a reasonable value for this repetition.

In the same initial environmental setting, different executions can result in different trajectories of the agent. This is due to the random targets that the agent chooses to reach, while exploring the environment. As a result, the agent can find itself in trouble if the randomly-selected target is close to obstacles, or if the path to the target is obstructed by obstacles so that the probability of hitting obstacles becomes high. On the other hand, if by chance, all the selected targets happen to be far away from obstacles, then the probability of hitting obstacles would be low. Figure 6.8 plots the closest distance of the cleaner agent to obstacles over time in two different executions of the same test case.

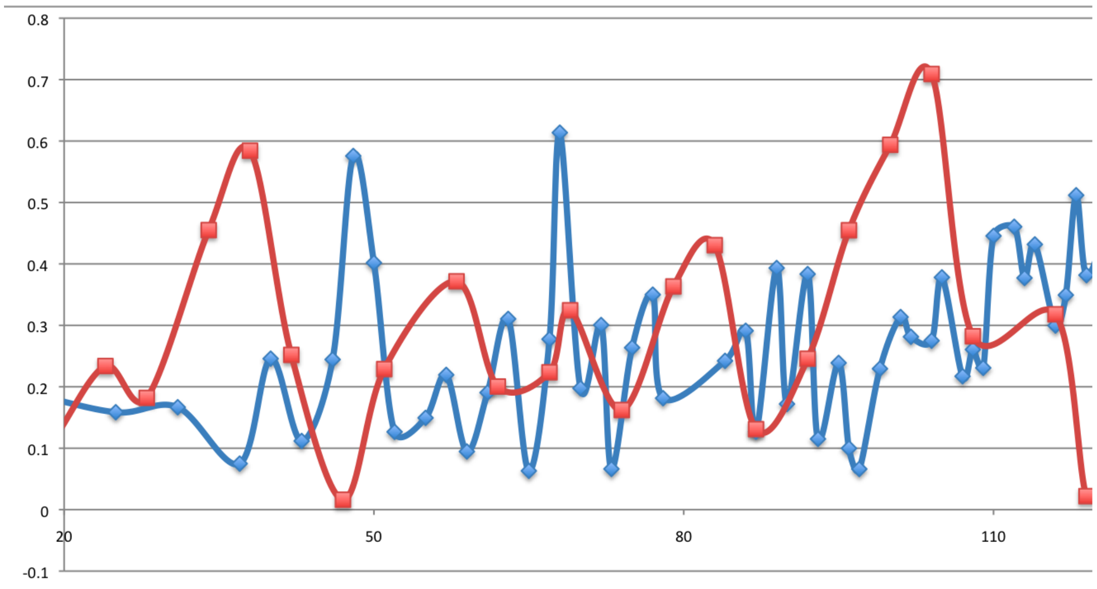


Figure 6.8: Plots of the closest distances of the agent to obstacles over time for two executions of the same test case

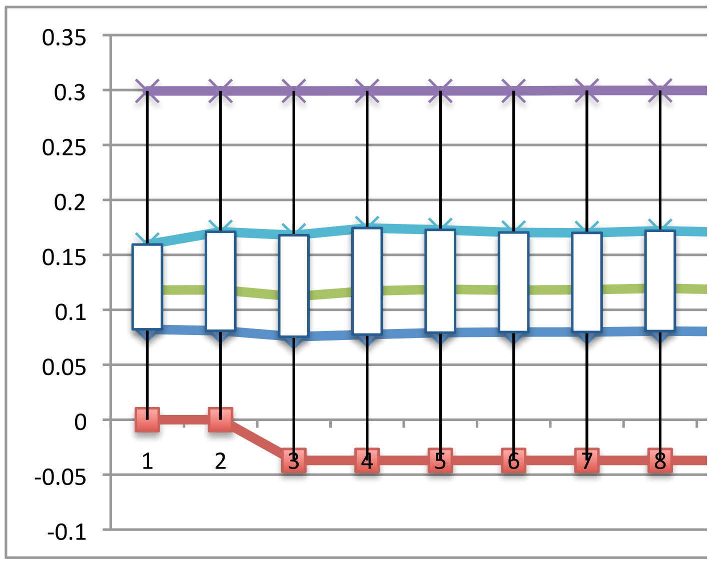
In order to find an effective environment where the probability of encountering obstacles is high, we must run each test case several times in order to reduce the influence of those non-deterministic factors in agent decision-making. In the following, we determine (1) how many executions of a test case is sufficient to evaluate the effect of it, and (2) how much time is needed for each run so that the agent has enough time to exhibit its behaviour. For the second question, a duration of 40 to 60 seconds is determined to be sufficient for each execution, since within that amount of time, the agent can visit all cells in the testing area several times.

To answer the first question, we randomly generate a number of test cases and execute them repeatedly many times. Figures 6.9(a) and 6.9(b) show cumulative box-plots of the closest distances to obstacles over the number of executions of two test cases *TC1* and *TC2*. In each execution we measured the distance of the agent to the closest obstacles in real-time. We use a box-plot presentation because it shows, not only the closest and furthest distance of the whole operation time, but also the 'hardness' of a test case. That is, the quartiles of 25% and 75% of the distances form a range that provides the typical dispersion of distances. If the range is close to 0, the probability of encountering obstacles is high.

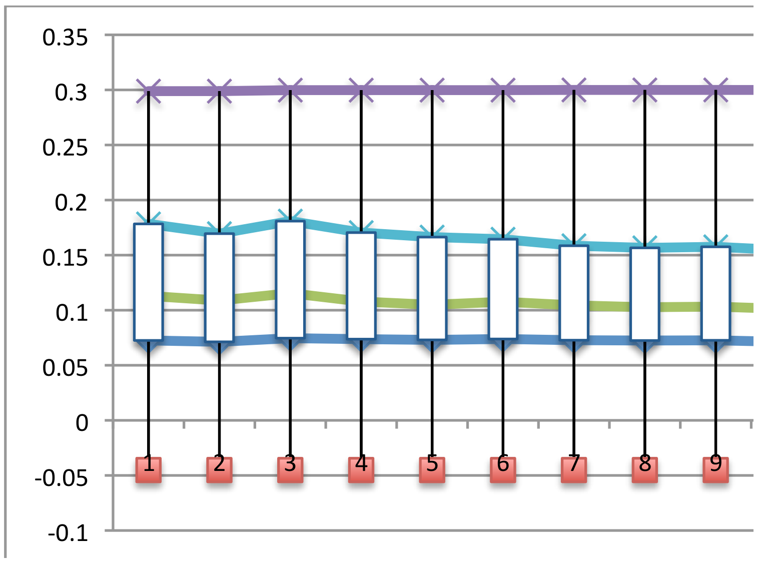
In general, we can observe, from Figure 6.9, that the boxes in each figure tend to converge in terms of size and position. We perform a pilot experiment with a large number of test cases, each has been executed a number of times. The fitness value is non-deterministic, because of the non deterministic behaviour of the agent under test. However, the average fitness value converges toward its final value after 4.6 executions (on average), as apparent from the cumulative box plots. Hence, we use 5 test case executions in our experiments to determine the fitness value associated with a test case.

6.3.2.3 Fitness function

Let D be the vector of all the closest distances to obstacles observed in all executions, and ε be the smallest distance allowed (user-defined threshold).



(a) Cumulative box-plots for *TC1*



(b) Cumulative box-plots for *TC2*

Figure 6.9: Cumulative box-plots for two test cases

Then, the fitness function is defined as follows:

$$f = \begin{cases} min(D) + w_1 * quartile1(D) + w_3 * quartile3(D) \\ \qquad\qquad\qquad \text{if } min(D) > \varepsilon, \\ min(D) - \varepsilon \qquad \text{if } min(D) \leq \varepsilon, \\ +\infty \text{ if the agent cannot move and suspend safely.} \end{cases}$$

where $min(D)$ is the smallest value of the vector D, $quartile1(D)$ is the quartile of 25% of D, and $quartile3(D)$ is the quartile of 75% of D. The weight of the two last values w_1, w_3 must be close to 0 as they are less important than the $min(D)$ value with respect to f. In fact, the reason for using the quartile values is that, among a set of test cases, we favour those that have distance dispersion (i.e. box-plots) close to 0 if they have the same $min(D)$. We have also performed pilot experiments without taking into account the distance dispersion, and the obtained results show that f is less effective when the dispersion is discarded.

The search objective is to bring the box-plots close to the threshold ε whenever $min(D)$ is still greater than the threshold ε. Otherwise, only the value $min(D)$ is relevant because it represents an error (the agent violates the threshold), in these cases, the algorithm searches for $min(D)$ as close to 0 as possible. In the case when the agent cannot move because of surrounding obstacles and it suspends safely, then the value of f is ∞; that is, the value of f can itself guide the search to skip the obvious cases when the agent is surrounded by obstacles.

6.3.3 Evolutionary robustness testing

Our testing objective is to assess the robustness of the cleaner agent. In particular, we test only for the capability of the agent to avoid obstacles by using the fitness function f, defined in the previous section. A genetic algorithm (GA) is used to generate test cases that minimize f, that is to find the test cases that lead the agent to breach the threshold ε (or $f \leq 0$), which is considered as fault.

The experiments were performed on three computers with Intel processors, Core 2 Duo (1.86Ghz), Pentium D (3Gz), and Xeon (4x3Ghz), each has more than 2Gb RAM. Each test case was executed on 5 simulation platforms (i.e., 5 executions per test case) in parallel. The observed data from the platforms was combined to calculate *f*. In evolutionary testing we choose $\varepsilon = 0.05$, and $w_1 = w_3 = 1/3$.

6.3.3.1 Experiment 1

In this first experiment, we encode the locations and quantity of waste, obstacles, dustbins and charging stations by four genes: one gene for each kind of object. Their resolutions are chosen as follows:

Kind	**Resolution**	**Max quantity**
Waste	12x12	144
Obstacle	8x8	64
Dustbin	2x2	4
Charging station	2x2	4

This experiment was performed on the early beta version of the agent that does not implement the capability to find the shortest path to reach a specific location, and there is no interaction between the two goals: *avoiding-obstacle* and *maintaining-battery*. The latter can inhibit the former while the agent goes to a charging station.

Evolutionary testing is executed with three different configurations: 60, 90 and 120 generations. The best results of all configuration give the optimal value f = -0.05 (or the distance to obstacles is 0). This reveals that the agent is faulty, because it hits obstacles. Our testing technique reveals two faults in the implementation of the cleaner agent:

Fault	**Description**
F1	*F1* occurs when the cleaner has two competing goals active at the same time: *maintaining-battery* and *avoiding-obstacles*. The agent favours the goal *maintaining-battery* regardless of the latter goal, so it hits obstacles on the way to a charging station. The value of f corresponding to this fault is very close or equal to the optimal value (-0.05).
F2	*F2* is that the agent gets too close to an obstacle before the goal *maintaining-battery* is triggered. The value of f corresponding to this fault is smaller than 0, but far away from the optimal value.

To evaluate the performance of evolutionary testing, we also performed random testing on the cleaner agent. For random testing, test cases were represented in exactly the same way that they were for the evolutionary testing approach, but they were generated entirely randomly. All the settings, such as the resolutions of the objects, the values of ε, w_1, w_3, the starting point of the agent, and the fitness function f were the same as for evolutionary testing, to ensure a fair comparison of results. Three experiments of 60, 90 and 120 random test cases were performed. The evolutionary approach used in this thesis is what is known in the literature as a 'steady state genetic algorithm' (Vavak and Fogarty 1996), in which only one new individual is produced at each generation. This means that, at each generation, there is only one new fitness evaluation. We choose settings for random test input generation to ensure that both the random and evolutionary approaches are provided with the same budget of the fitness evaluations. In this case, that means choosing the number of random tests to be equal to the number of generations of the evolutionary algorithm. This ensures a fair comparison of the two approaches — evolutionary and random.

Comparing the results obtained from randomly-generated test cases to evolutionary-generated ones, we observe that the fitness values of evolutionary testing are smaller (meaning better) than those of random testing within a similar testing time. In fact, all of the best values of f of evolutionary testing are optimal ($f = -0.05$) while none of the experiments with random testing achieves this value. In addition, the dispersion of distances of evolutionary testing is more compact, and closer to the optimal value than that of random testing. This implies that evolutionary testing generates more challenging test cases to test the cleaner agent than random testing, though both of them can detect the faults.

6.3.3.2 Experiment 2

The objective of this experiment is to further compare the performance of evolutionary testing to random testing. In this experiment, we fix the locations of 2 charging stations, 2 dustbins, and 6 obstacles (see Figure 6.10). The obstacles are placed in the corners so that once the agent goes to these corners, it is difficult for it to get out. In particular, we place three obstacles in the top-right corner, forming a waste–rich potential 'honey pot trap' from which the agent has only one way to get in and out and could drain its battery there. In this experiment only waste is placed randomly in random testing, or with the guidance of the fitness function in ET.

This experiment was performed on a revised version of the cleaner agent. It has the capability to find the shortest trajectory to reach a specific location, avoiding obstacles on the way. Moreover, we change the implementation of the agent to make testing more challenging, by making the first fault *F1* harder to detect. Now in the goal deliberation mechanism of the cleaner agent, the goal *avoiding-obstacles* can inhibit the goal *maintaining-battery* if the battery level is still greater than 5%. The fault has a chance to reveal only the battery level goes below 5%, not 20% like in the previous experiment.

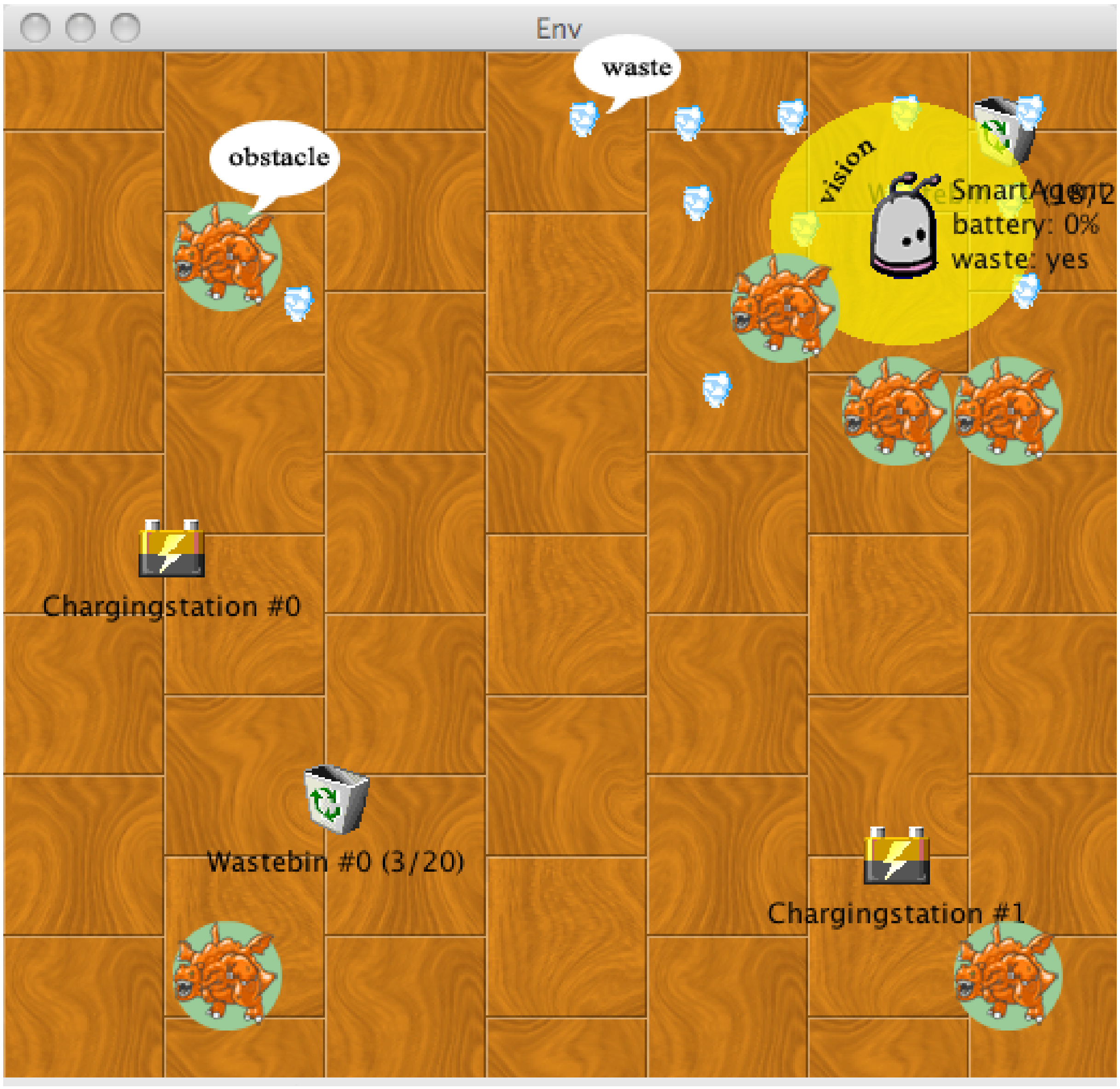


Figure 6.10: A special scenario to test the cleaner agent

The final results of detecting the two faults *F1* and *F2* of this experiment are described as follows. In three runs with 90, 120, or 200 generations, the evolutionary technique detects both faults; while with comparable numbers of random test cases: 90, 120, 200, the random technique can detect only the easy fault *F2*.

Overall, evolutionary testing, guided by fitness functions derived from softgoals, outperforms random testing under the same execution cost and time.

The significance of the test results above is that evolutionary testing, following our approach, tests an agent in a greater range of contexts, thereby accounting for its autonomy to act differently in each. Testing an autonomous agent using a more standard approach can only work if the range of contexts that influence the agent's behaviour is sufficiently limited that the developer can predict them all. However, when considering systems of any substantial complexity, of which a multi-agent system is certainly included, such a limited range is unlikely to occur. We can therefore argue that automated, search-based testing is essential to ensure the robustness of complex systems and, as our tests show, evolutionary testing is an excellent candidate.

6.4 Summary

This chapter has presented the results of three experiments that have been conducted to study the proposed generation, evaluation techniques.

The results obtained from the first experiment, Section 6.1, indicated that continuous testing has a big potential to complement the manual testing activity. In fact, for faults involving long message sequences and specific input data, continuous testing seems particularly suited to explore those states that can potentially lead to them. Evol-Mutation can contribute to the discovery of the hard to reveal faults, which would go probably unnoticed under goal-oriented and random testing.

The results of the second study, Section 6.2, showed that whenever the interaction ontology has a non trivial size, the ontology-based generation method achieved a higher coverage of the ontology classes than manual test case derivation. It also overcomes manual derivation in terms of the number of faults revealed and the portion of input space explored during testing. The level of automation achieved by our tool eCAT allows for the automatic test case generation at negligible extra costs.

Finally, in the third experiment with the autonomous cleaner agents, Section 6.3, we evaluated the systematic way of evaluating the quality of autonomous agents, presented in Section 4.1.3 and Section 4.3.5.2. Briefly, stakeholder requirements were represented as quality measures, and corresponding thresholds were used as testing criteria; autonomous agents needed to meet these criteria in order to be reliable. Then, fitness functions that represent testing objectives were defined accordingly. They guided our evolutionary test generation technique to produce test cases automatically. The longer time for evolution is, the more challenging the evolved test cases are. Thus the autonomous agent is tested more and more extensively. We developed the simulation of the cleaning agent system to evaluate the approach. The observed results, which we reported in Section 6.3, demonstrated that the evolutionary testing was effective. Indeed, our approach has great potential in evaluating complex software entities like autonomous agents.

Chapter 7

Conclusion

7.1 Conclusion

The increasing use of Internet as the backbone for all interconnected services and devices makes software systems highly complex and virtually unlimited in scale. These systems often involve variety of users and heterogeneous platforms. They are evolved continuously in order to meet the changes of business and technology. In some circumstances, they need to be autonomous and adaptive for dealing with such changes. Software agents and MAS are considered as key enabling technologies for building such open, dynamic, and complex systems.

Now, as software agents with built-in autonomy are increasingly taking over control and management activities, such as in automated vehicles or e-commerce systems, testing these systems to make sure that they behave properly becomes crucial. This calls for an investigation of suitable software engineering frameworks, testing in particular, to build high quality and dependable software agents and MAS.

Testing software agents and MAS has been receiving much effort from several active research groups. However, there are still many open issues for research. A complete and comprehensive testing process for software agents and MAS is absent. We need adequate approaches to judge autonomous behaviours, to evaluate agents that have their own goals. We need methods to test emergent properties of MAS as a whole. These opportunities, among others, motivate this PhD work.

This work has contributed to advance the state of the art in different aspects. Firstly, the proposed GOST methodology takes goal-oriented requirements analysis and design artefacts as the core elements for test case derivation. It provides systematic guidance to generate test suites from modelling artefacts produced along with the development process. These test suites, on the one hand, are used to refine goal analysis and to detect problems early. On the other hand, they are executed to test the achievement of the goals from which they were derived. Moreover, the proposed methodology gives a complete classification of testing levels in software agent and MAS testing, and a complete testing process following the standard **V**.

Secondly, we have proposed a number of techniques to tackle the challenges in testing software agents: (i) for evaluating agent's behaviours we have proposed three different approaches, i.e., *constraint-based*, *ontology-based*, and *requirement-based*; (ii) for the generation of test cases (partially or fully automated) we have proposed *goal-oriented*, *ontology-based*, *random*, and *evolutionary* techniques; (iii) then we combined them in a novel testing execution method, called *continuous testing*. This method relies on a *tester agent*, which plays the role of human tester, and a monitoring agent network, which monitors the system as a whole to record events, changes, misbehaviours, and so on. The tester agent uses the generation techniques, e.g., ontology-based, evolutionary, to generate and execute new test cases against the agents under test, continuously; while the monitoring agent network guards, reports problems (e.g., violations), or record data for desired measurements.

Finally, to support the methodology and the continuous testing method, we have been developing a testing framework, called eCAT. The framework consists of tools for test case specification and derivation from goal models, for graphical visualization, for continuous execution, and for fault reporting. eCAT is available online at `http://code.google.com/p/open-ecat/`.

The results obtained, discussed in Chapter 6, are very promising. First of all, they showed that continuous testing had a big potential to complement the manual testing activity. In fact, especially for faults involving long message sequences and specific input data, continuous testing particularly suits to explore those states that can potentially lead to revealing these faults. Whenever high reliability (i.e., long mean time between failures) is the aim, Evol-Mutation can contribute to the discovery of the hard to reveal faults, which would go probably unnoticed under other techniques. Secondly, the ontology-based generation method achieved a higher coverage of the ontology concepts than the manual method. It also overcomes manual derivation in terms of the number of faults revealed and the portion of input space explored during testing. The level of automation achieved by eCAT allows for test case generation at negligible extra costs. Finally, the results obtained in the experiment with the autonomous cleaner agents have demonstrated that the evolutionary testing technique and the use of quality functions derived from requirements are effective. Indeed, our approach has great potential in validating complex software entities like autonomous agents.

For what concerns the GOST methodology, the benefits that it brings are twofold. First of all, since goal-oriented requirements engineering has been recognized as a powerful and effective approach for building software agents and MAS, drawing straight connections between goal-oriented analysis and goal-oriented testing can help to make the concepts used in the development consistent and to save the development cost. Secondly, as GOST proposes to parallel goal-oriented construction and goal-oriented testing, this helps to discover problems early, thus avoiding to implement erroneous specifications.

7.2 Future work

In the future work, we will extend the derivation method to exploit detailed design of plans, which may include interaction or operation design. This extension will enrich the derived test suites with detailed information and will further automate MAS testing. Moreover, we will investigate metrics to evaluate goal-oriented testing coverage of the implementation, as they can give insights to the developer about the testing effort and the confidence level in the implemented MAS. More importantly, we will evaluate thoroughly, by means of an empirical study, the methodology in terms of usability, productivity, and more generally the benefits the methodology could bring.

We will further investigate the pre- and post-conditions that can be checked by the monitoring agents. This can potentially contribute to guide evol-mutation to reveal faults that violate the conditions specified. In addition, we plan to extend our framework to deal with remaining MAS testing issues, such as “sealed” agents and the environment factors, so that the TA can detect faults related to specific environment configurations. Moreover, we will consider how to provide the tester agent with heuristics to analyze the interactions among the agents under test, in order to guide evol-mutation testing towards the generation of test cases that are more likely to reveal faults.

In evolutionary testing of agents, some research issues remain open. In our future work, we will consider multiple sets of simultaneous conflicting and competing requirements. For instance, in the cleaner agent one may want to evaluate robustness in terms of maintaining battery and avoiding obstacles. Since each requirement related to autonomy can give rise to a fitness function (or search objective), multiple requirements call for a multi-objective search technique. The multi-objective versions of evolutionary algorithms are probably suitable to deal with such situations.

Bibliography

Adrion, W. R., Branstad, M. A. and Cherniavsky, J. C.: 1982, Validation, verification, and testing of computer software, *ACM Comput. Surv.* **14**(2), 159–192.

Agent Oriented Software Pty. Ltd.: n.d., JACK Agent Language, http://agent-software.com.au/jack.html.

Beck, K.: 2002, *Test Driven Development: By Example*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Beizer, B.: 1990, *Software Testing Techniques (2nd ed.)*, Van Nostrand Reinhold Co., New York, NY, USA.

Bergenti, F., Gleizes, M.-P. and Zambonelli, F. (eds): 2004, *Methodologies and Software Engineering for Agent Systems : The Agent-Oriented Software Engineering Handbook*, Springer.

Bordini, R. H., Wooldridge, M. and Hübner, J. F.: 2007, *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*, John Wiley & Sons.

Botía, J. A., Gómez-Sanz, J. J. and Pavón, J.: 2006, Intelligent Data Analysis for the Verification of Multi-Agent Systems Interactions, *Intelligent Data Engineering and Automated Learning - IDEAL 2006, 7th International Conference, Burgos, Spain, September 20-23, 2006, Proceedings*, pp. 1207–1214.

Botía, J. A., López-Acosta, A. and Gómez-Skarmeta, A. F.: 2004, ACLAnalyser: A tool for debugging multi-agent systems, *Proc. 16th European Conference on Artificial Intelligence*, pp. 967–968.

Bourque, P. and Dupuis, R. (eds): 2004, *Guide to the Software Engineering Body of Knowledge: 2004 Edition*, IEEE.
URL: *http://www.swebok.org*

Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J. and Perini, A.: 2004, Tropos: An Agent-Oriented Software Development Methodology, *Autonomous Agents and Multi-Agent Systems* **8**(3), 203–236.

Cacciari, L. and Rafiq, O.: 1999, Controllability and observability in distributed testing, *Information and Software Technology* **41**(11-12), 767–780.
URL: *http://www.sciencedirect.com/science/article/B6V0B-3X3TD3J-7/2/be2860fa5fae5350c8b9d85a1a4b0c84*

Coelho, R., Kulesza, U., von Staa, A. and Lucena, C.: 2006, Unit testing in multi-agent systems using mock agents and aspects, *SELMAS '06: Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems*, ACM Press, New York, NY, USA, pp. 83–90.

Cossentino, M.: 2005, From requirements to code with the passi methodology, *in* (Henderson-Sellers and Giorgini 2005).

Dardenne, A., van Lamsweerde, A. and Fickas, S.: 1993, Goal-directed requirements acquisition, *Science of Computer Programming* **20**(1-2), 3–50.

Dastani, M., van Riemsdijk, M. B. and Meyer, J.-J. C.: 2006, Goal types in agent programming, *Proc. 17th European Conference on Artificial Intelligence*, pp. 220–224.

DeMillo, R. A., Lipton, R. J. and Sayward, F. G.: 1978, Hints on test data selection: Help for the practicing programmer, *IEEE Computer* **11**(4), 34–41.

Development Standards for IT Systems of the Federal Republic of Germany: 2005, The V-Model.
URL: *http://www.v-modell-xt.de*

Dikenelli, O., Erdur, R. C. and Gumus, O.: 2005, Seagent: a platform for developing semantic web based multi agent systems, *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, ACM Press, New York, NY, USA, pp. 1271–1272.

Ekinci, E. E., Tiryaki, A. M., Cetin, O. and Dikenelli, O.: 2008, Goal-Oriented Agent Testing Revisited, *Proc. of the 9th Int. Workshop on Agent-Oriented Software Engineering*, pp. 85–96.

FIPA: 2002a, ACL Message Structure Specification, http://www.fipa.org/specs/fipa00061.

FIPA: 2002b, FIPA Interaction Protocols Specifications, http://www.fipa.org/repository/ips.php3.

FIPA: 2004, FIPA Agent Management Specification, http://www.fipa.org/specs/fipa00023/.

Fuxman, A., Liu, L., Mylopoulos, J., Pistore, M., Roveri, M. and Traverso, P.: 2004, Specifying and analyzing early requirements in tropos, *Requirements Engineering* **9**(2), 132–150.

Gamma, E. and Beck, K.: 2000, JUnit: A Regression Testing Framework, http://www.junit.org.

Georgeff, M. P. and Ingrand, F. F.: 1989, Decision-making in an embedded reasoning system, *International Joint Conference on Artificial Intelligence*, pp. 972–978.

Gómez-Sanz, J. J., Botía, J., Serrano, E. and Pavón, J.: 2008, Testing and Debugging of MAS Interactions with INGENIAS, *Proc. of the 9th Int. Workshop on Agent-Oriented Software Engineering.*

Gotlieb, A., Denmat, T. and Botella, B.: 2007, Goal-oriented Test Data Generation for Pointer Programs, *Information and Software Technology* **49**(9-10), 1030–1044.

Graham, D. R.: 2002, Requirements and testing: Seven missing-link myths, *IEEE Software* **19**(5), 15–17.

Hamlet, R. G.: 1977, Testing programs with the aid of a compiler, *IEEE Transactions on Software Engineering* **3**(4), 279–290.

Harman, M.: 2007, The current state and future of search based software engineering, *in* L. Briand and A. Wolf (eds), *IEEE International Conference on Software Engineering (ICSE 2007), Future of Software Engineering*, IEEE Computer Society Press, Los Alamitos, California, USA, pp. 342–357.

Harrold, M. J., Offutt, A. J. and Tewary, K.: 1997, An approach to fault modeling and fault seeding using the program dependence graph., *Journal of Systems and Software* **36**(3), 273–295.

Henderson-Sellers, B. and Giorgini, P. (eds): 2005, *Agent-Oriented Methodologies*, Idea Group Inc.

Jureta, I., Faulkner, S. and Schobbens, P.-Y.: 2007, Achieving, satisficing, and excelling, *Advances in Conceptual Modeling –Foundations and Applications* pp. 286–295.

Lam, D. N. and Barber, K. S.: 2005, *Programming Multi-Agent Systems*, Springer Berlin / Heidelberg, chapter Debugging Agent Behavior in an Implemented Agent System, pp. 104–125.

McMinn, P. and Holcombe, M.: 2003, The state problem for evolutionary testing, *Proceedings of the Genetic and Evolutionary Computation Conference.*

Mills, H. D., Dyer, M. D. and Linger, R. C.: 1987, Cleanroom software engineering, *IEEE Software* **4**(5), 19–25.

Nguyen, C. D., Perini, A. and Tonella, P.: 2007, Automated continuous testing of multi-agent systems, *The fifth European Workshop on Multi-Agent Systems.*

Nguyen, C. D., Perini, A. and Tonella, P.: 2008a, Constraint-based evolutionary testing of autonomous distributed systems, *Proc. of the first Intl. Workshop on Search-Based Software Testing.*

Nguyen, C. D., Perini, A. and Tonella, P.: 2008b, Experimental evaluation of ontology-based test generation for multi-agent systems, *9th International Workshop on Agent-Oriented Software Engineering*, pp. 165–176.

Nguyen, C. D., Perini, A. and Tonella, P.: 2008c, Ontology-based Test Generation for Multi Agent Systems, *Proc. of the International Conference on Autonomous Agents and Multiagent Systems.*

Núñez, M., Rodríguez, I. and Rubio, F.: 2005, Specification and testing of autonomous agents in e-commerce systems, *Software Testing, Verification and Reliability* **15**(4), 211–233.

Odell, J., Parunak, H. V. D. and Bauer, B.: 2000, Extending UML for Agents, *in* G. Wagner, Y. Lesperance and E. Yu (eds), *Proc. of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, Austin, TX, pp. 3–17.

OMG: 2006, Object Constraint Language Specification, OMG Specification. version 2.0.

Padgham, L. and Winikoff, M.: 2002, Prometheus: A pragmatic methodology for engineering intelligent agents, *Proc. Workshop on Agent Oriented Methodologies (OOPSLA 2002)*.

Padgham, L. and Winikoff, M.: 2004, *Developing Intelligent Agent Systems: A Practical Guide*, John Wiley and Sons. ISBN 0-470-86120-7.

Padgham, L., Winikoff, M. and Poutakidis, D.: 2005, Adding debugging support to the Prometheus methodology, *Engineering Applications of Artificial Intelligence* **18**(2), 173–190.

Pavón, J., Gómez-Sanz, J. J. and Fuentes-Fernández, R.: 2005, The INGENIAS Methodology and Tools, *in* (Henderson-Sellers and Giorgini 2005).

Penserini, L., Perini, A., Susi, A. and Mylopoulos, J.: 2007, High variability design for software agents: Extending tropos, *ACM Transactions on Autonomous and Adaptive Systems* **2**(4), 16.

Perini, A.: 2009, *Wiley Encyclopedia of Computer Science and Engineering*, Hoboken: John Wiley & Sons, Inc., chapter Agent-Oriented Software Engineering. dx.doi.org/10.1002/9780470050118.ecse006.

Perini, A., Pistore, M., Roveri, M. and Susi, A.: 2003, Agent-oriented modeling by interleaving formal and informal specification, *Agent-Oriented Software Engineering IV, 4th International Workshop*, Melbourne, Australia, pp. 36–52.

Perini, A. and Susi, A.: 2005, Agent-Oriented Visual Modeling and Model Validation for Engineering Distributed Systems, *Computer Systems Science & Engineering* **20**(4), 319–329.

Pokahr, A., Braubach, L. and Lamersdorf, W.: 2005, *Jadex: A BDI Reasoning Engine*, Kluwer Book, chapter Multi-Agent Programming.
URL: *http://vsis-www.informatik.uni-hamburg.de/projects/jadex*

Rao, A. S. and Georgeff, M. P.: 1995, BDI-agents: from theory to practice, *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco.
URL: *citeseer.ist.psu.edu/rao95bdi.html*

Rodrigues, L. F., de Carvalho, G. R., de Barros Paes, R. and de Lucena, C. J. P.: 2005, Towards an integration test architecture for open mas, *1st Workshop on Software Engineering for Agent-oriented Systems / SBES*.

Rouff, C.: 2002, A test agent for testing agents and their communities, *Aerospace Conference Proceedings*, Vol. 5.

Schach, S. R.: 1996, Testing: Principles and practice, *ACM Comput. Surv.* **28**(1), 277–279.

Standard glossary of terms used in Software Testing: 2007.
URL: *http://www.istqb.org/downloads/glossary-current.pdf*

Sudeikat, J. and Renz, W.: 2008, A Systemic Approach to the Validation of Self-Organizing Dymanics within MAS, *Proc. of the 9th Int. Workshop on Agent-Oriented Software Engineering*, pp. 237–248.

Telecom Italia Lab: 2000, Java Agent DEvelopment Framework.
URL: *http://jade.tilab.com*

The Eclipse Graphical Modeling Framework (GMF): n.d., http://www.eclipse.org/modeling/gmf/.

Thévenod-Fosse, P. and Waeselynck, H.: 1993, Statemate: Applied to statistical software testing, *Proc. of the Int. Symposium on Software Testing and Analysis (ISSTA)*, pp. 78–81.

Tiryaki, A. M., Oztuna, S., Dikenelli, O. and Erdur, R.: 2006, Sunit: A unit testing framework for test driven development of multi-agent systems, *7th International Workshop on Agent Oriented Software Engineering*.

Vavak, F. and Fogarty, T. C.: 1996, Comparison of steady state and generational genetic algorithms for use in nonstationary environments, *In IEEE International Conference on Evolutionary Computation (ICEC*, IEEE Publishing, Inc, pp. 192–195.

Wegener, J.: 2005, *Stochastic Algorithms: Foundations and Applications*, Springer Berlin / Heidelberg, chapter Evolutionary Testing Techniques, pp. 82–94.

Weyns, D., Omicini, A. and Odell, J.: 2007, Environment as a first class abstraction in multiagent systems, *Autonomous Agents and Multi-Agent Systems* **14**(1), 5–30.

Yu, E.: 1995, *Modelling Strategic Relationships for Process Reengineering*, PhD thesis, University of Toronto, Department of Computer Science, University of Toronto.

Yu-Seung Ma, J. O. and Kwon, Y. R.: 2005, Mujava : An automated class mutation system, *Software Testing, Verification and Reliability* **15(2)**, 97–133.

Zhang, Z., Thangarajah, J. and Padgham, L.: 2007, Automated unit testing for agent systems, *2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE-07)*, Barcelona, Spain.

Glossary

ACLAnalyser	A tool that intercepts agent interactions and detects communication problems, such as miscommunicating, unbalanced, or overhead, 17
AMS	Agent Management System, a special agent in FIPA compliant architecture that is responsible to manage the agent platform, 67
AOSE	Agent-Oriented Software Engineering, 2–4, 16, 21, 22
DF	Directory Facilitator, a special agent in FIPA compliant architecture that is centralized registry of entries which associate service descriptions to agent identifiers, 44, 53
eCAT	A Environment for the Continuous Testing of Software Agents, 6, 7, 87, 88, 90, 92–94, 97, 101, 104, 108, 111, 112, 130, 134, 135

Evolutionary testing	Testing technique that transforms testing objective to a search problem and uses metaheuristic search techniques to generate test inputs, 12, 78
Evol-Mutation	Evolutionary-Mutation test generation technique, it combines evolutionary testing and mutation testing, 81, 101, 104, 107–109, 111, 112, 130, 135
FIPA	Foundation for Intelligent Physical Agents is an IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies, 61
GMF	The Eclipse Graphical Modeling Framework, 90
GOST	Goal-Oriented Software Testing, 4, 5, 7, 22, 25, 54, 82, 87, 93, 134, 135
INGENIAS	An AOSE methodology, 21
LOC	Line Of Code, 113, 116
MAS	MultiAgent System, 1–4, 7, 13–19, 22, 23, 31, 66, 68, 75–77, 101, 102, 111, 113, 116, 133–136

Mock agent	Program that simulates behaviour of real agent in communication, 16
Mr. Cleaners	A simulation of a multiagent system that composes of cleaning agents working at an airport, 25
MTBF	Mean Time Between Failure, 112
OCL	Object Constraint Language, 58, 59, 113
OO	Object-Oriented, 21
OWL	Web Ontology Language, a family of knowledge representation languages for authoring ontologies, 61
PASSI	An AOSE methodology, 21
Prometheus	An AOSE methodology, 15
Software testing	Software testing is a software development activity, aimed at evaluating product quality and improving it by identifying defects and problems, 11
SUnit	A testing framework built for testing agents developed with Seagent, 16
Tester Agent	Software agent that acts like human tester to test other agents, 82, 83, 86, 87, 89, 93, 97, 105, 107, 136

Tropos	An AOSE methodology, v, 4, 22, 23, 25, 30–32, 47, 54, 70, 85, 90, 92, 93, 104

Appendix A

Publications

Journal

[1] Cu D. Nguyen, Anna Perini and Paolo Tonella. *Goal-Oriented Testing for MAS*. Int. Journal of Agent-Oriented Software Engineering, (under review).

2009

[2] Cu D. Nguyen, Simon Miles, Anna Perini, Paolo Tonella, Mark Harman, Michael Luck. *Evolutionary Testing of Autonomous Software Agents.* The 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'09) (accepted as full paper).

2008

[3] Cu D. Nguyen, Anna Perini and Paolo Tonella. *Ontology-based Test Generation for Multi-Agent Systems.* In the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08), Estoril, Portugal, May 12-16, 2008, Volume 3, pages 1315-1320, 2008.

[4] Cu D. Nguyen, Anna Perini and Paolo Tonella. *eCAT: a Tool for Automating Test Cases Generation and Execution in Testing Multi-Agent Systems (Demo Paper).* In the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08), Estoril, Portugal, May 12-16, 2008, Demo Proceedings, pages 1669-1670, 2008.

[5] Cu D. Nguyen, Anna Perini and Paolo Tonella. *Experimental Evaluation of Ontology-based Test Generation for Multi-Agent Systems.* In the 9th International Workshop on Agent-Oriented Software Engineering, at the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08). Springer, 2008.

[6] Cu D. Nguyen, Anna Perini and Paolo Tonella. *Automated Continuous Testing of Autonomous Distributed Systems.* In the 1st International Workshop on Search-Based Software Testing, in conjunction with the IEEE International Conference on Software Testing, Verification and Validation, IEEE, 2008. (**Best paper award**).

2007

[7] Cu D. Nguyen, Anna Perini and Paolo Tonella. *Automated Continuous Testing of Multi-Agent Systems.* In the fifth European Workshop on Multi-Agent Systems (EUMAS), December 2007.

[8] M. Morandini, C. D. Nguyen, A. Perini, A. Siena and A. Susi. *Tool-supported Development with Tropos: the Conference Management System Case Study.* In Agent-Oriented Software Engineering VIII, 8th International Workshop, AOSE 2007, Honolulu, HI, USA, May 14, 2007, Revised Selected Papers, volume 4951 of Lecture Notes in Computer Science, pages 182-196, 2008.Science, pages 182-196, 2008.

[9] Cu D. Nguyen, Anna Perini and Paolo Tonella. *A Goal-Oriented Software Testing Methodology.* In Agent-Oriented Software Engineering VIII, 8th International Workshop, AOSE 2007, Honolulu, HI, USA, May 14, 20volume 4951 of Lecture Notes in Computer Science, pages 58-72, 2008.