Abstract

Mobility occurs naturally in many distributed system applications such as telecommunications and electronic commerce. Mobility may reduce bandwidth consumption and coupling and increase flexibility. However, it seems that relatively little work has been done to support quality assurance techniques such as testing and verification of mobile systems.

This thesis describes an approach for checking the conformance of a mobile, distributed application with respect to an executable model at runtime. The approach is based on \textit{kiltera} — a novel, high-level language supporting the description and execution of models of concurrent, mobile, distributed, and timed computation. The approach allows distributed, rather than centralized, monitoring. However, it makes very few assumptions about the platform that the mobile agent system is implemented in.

We have implemented our approach and validated it using four case studies. Two of them are examples of mobile agent systems, the two others are implementations of distributed algorithms. Our approach was able to detect seeded faults in the implementations. To check the effectiveness and the efficiency of our approach more comprehensively a mutation-based evaluation framework has been implemented. In this framework a set of new mutation operators for mobile agent systems has been
identified in order to automatically generate and run a number of mutants programs and then evaluate the ability of our approach to detect these mutants. We found that our approach is very effective and efficient in killing the non-equivalent mutants.
Co-Authors

Chapter 3 and part of Chapter 4 were published in paper co-authored with my supervisor Juergen Dingel and Ernesto Posse in the Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging 2009 (PADTAD 2009) [75]. Parts of Chapter 1 and 2 are based on a paper previously published in the Proceeding of the International Joint Conferences on Computer, Information, System Sciences and Engineering (CISSE’08) [74] and in a Queen’s University technical report [73]. Both of them jointly authored with my supervisor Juergen Dingel.
Statement of Originality

I, Ahmad Saifan, certify that the work presented in this thesis is original unless otherwise noted. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices.
Acknowledgments

First of all, I would like to express my heart-felt and most sincere gratitude to my respected supervisor Juergen Dingel for his guidance, advice, encouragement and extra ordinary patience during this thesis work. Without him, this work would have been impossible.

I would like to thank Luigi Logripopo, Patrick Martin, Jenny Zou, and David Rappaport, the members of my thesis examination committee for their helpful comments and suggestions.

I owe my parents any success I make in my life. Their encouragement, love and support is a key factor in any achievement I have ever made. I am also indebted to my beloved and incomparable wife, Doua, for more than I could ever express, due to her continuous encouragement, patience and support during the course of my Ph.D. program. I also thank the most wonderful thing of my life my children, Qusai and Yousef, for their patience and for the happiness and motivation I always find in their smiles. I want to thank my brothers, Amjad, Laith, and Ammar, and my sisters, for their love and support. I would like to thank all members of the School of Computing at Queens University for their support and friendship. Special thanks to Chanchal Roy and Jeremy Bradbury.
Table of Contents

Abstract

Co-Authors

Statement of Originality

Acknowledgments

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Thesis and Scope of Research

1.2 Contributions

1.3 Thesis Organization

Chapter 2: Background and Related Work

2.1 Systems of Interest

2.1.1 Mobile Agent Systems
Chapter 3: Description of Approach .......................... 32
  3.1 Proposed Approach ........................................ 32
    3.1.1 Construction of HLM ................................. 34
    3.1.2 Construction of KM from HLM ....................... 35
    3.1.3 Instrumentation of IUT and KM using HLM .......... 39
    3.1.4 Execution of IUT and KM ............................ 41
  3.2 Chapter Summary ........................................... 41

Chapter 4: Case Studies ................................. 44
  4.1 Case Study: Online Shopping Example ................. 45
    4.1.1 General Description ................................. 45
    4.1.2 Application of our Approach ...................... 45
    4.1.3 Checking sample properties ....................... 55
    4.1.4 Observations ......................................... 58
List of Tables

2.1 Aglets callback methods .................................................. 23

4.1 Size in lines of code of the IUT and KM before and after the instrumentation ........................................... 58

4.2 Size in lines of code of the IUT and KM before and after the instrumentation of the contract example ................. 73

5.1 Mobile agent mutation operators ....................................... 82

6.1 Overview of the example programs ................................... 109

6.2 The number of mutants generated for the two Aglets programs ................................................................. 111

6.3 Number of program mutants ............................................. 112

6.4 Number of equivalent and killed mutants detected ............. 119

6.5 A comparison between our approach and aspect-oriented approach based on lines of code .................................. 121
List of Figures

2.1 kiltera syntax .................................................. 13
2.2 An example in kiltera .......................................... 14
2.3 Mobility in kiltera ............................................. 16
2.4 Match and timeout statements in kiltera .................. 18
2.5 An event trace of Example1 in Figure 2.2 ................. 18
2.6 An event trace with real time mode ......................... 20
2.7 Typical form of an agent in Aglets ......................... 22

3.1 Steps of our conformance testing approach ............... 33
3.2 A conditional interaction in standard Sequence Diagrams . 34
3.3 A simple Swimlaned Mobility Diagram (SMD) ............... 35
3.4 Message communication in kiltera .......................... 37
3.5 The kiltera model of the corresponding SMD presented in Fig. 3.3 38
3.6 SMD with a check point ....................................... 39
3.7 Location of check point 1 in the IUT of agent P2 ........... 40
3.8 Location of check point 1 in the KM of agent P2 ........... 40
3.9 Code inserted into the IUT of agent P2 at check point 1 .... 40
3.10 Code inserted into KM of agent P2 at check point 1 ....... 41
3.11 The architecture of the monitoring infrastructure ......... 42

xi
4.25 Code inserted into IUT at check point 3 . . . . . . . . . . . . . . . . . . . . . 71
4.26 Code inserted into KM at check point 3 . . . . . . . . . . . . . . . . . . . . . 71
4.27 Checking Property 1 in KM . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
4.28 Checking Property 2 in KM . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
4.29 Checking Property 3 in KM . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
4.30 KM of token ring algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
4.31 Checking entering the CS in token ring algorithm using KM . . . . . . . . 76
4.32 Checking releasing the CS in Lamport algorithm using KM . . . . . . . . 77

6.1 The mutation-based evaluation framework . . . . . . . . . . . . . . . . . 106
6.2 The number of mutants per operator category . . . . . . . . . . . . . . . 113
6.3 The number of mutants per Java file grouped in ranges . . . . . . . . . . 114
6.4 The number of mutants per Java file against lines of code . . . . . . . . 115
6.5 Kiltera output . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
6.6 Part of the analysis report . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118

A.1 The SMD of overall online shopping system . . . . . . . . . . . . . . . . 142
A.2 The SMD of the initialization process . . . . . . . . . . . . . . . . . . . . . 142
A.3 The SMD of the client . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143
A.4 The SMD of the Mall1 with two shops . . . . . . . . . . . . . . . . . . . . . 143
A.5 The SMD of the agent . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
A.6 The KM of the initialization process of the shopping example . . . . . . 144
A.7 The KM of the client . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 145
A.8 The KM of the Mall1 with two shops . . . . . . . . . . . . . . . . . . . . . 145
A.9 The Agent KM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 146
Chapter 1

Introduction

Software quality is an important attribute that all developers of software systems want to achieve. Moreover, the size and complexity of software systems is increasing in general. Mobile Agent Systems (MAS) are no exception.

MAS are a special kind of distributed system. In MAS, the code of the agent has the capability to move from one host to another. More precisely, code mobility refers to the “capability to reconfigure dynamically, at run-time, the binding between the software components of the application and their physical location within a computer network” [18]. Mobility occurs naturally in many distributed system applications such as telecommunications and electronic commerce. Moreover, mobility may reduce bandwidth consumption and coupling and increase flexibility [70]. The use of mobility has reached a certain degree of maturity: several different development platforms are available (e.g., Aglets [49, 3], Voyager [2], and Grasshopper [9]) and comparative performance evaluations have been conducted [85]; agent-oriented software engineering (AOSE) has produced tool-supported development methodologies (e.g., Prometheus [68]) and Tropos [15]); promising commercial applications exist [62], and
standardization is being considered [69]. However, it seems that relatively little work has been done to support quality assurance techniques such as testing and verification of mobile systems [26, 89]. There are several characteristics of mobile agent systems that make testing a challenging task. Some of these characteristics include mobility, autonomy, distribution.

1.1 Thesis and Scope of Research

The common way to validate the quality of software systems is testing. The primary goal for this thesis is to provide an approach to improve the quality of mobile agent systems. Our approach will take the benefits of the kiltera high-level programming language in order to build a prototype that allows the validation of these kinds of systems.

Thesis Statement: Using an executable modeling language for mobile, distributed and timed systems such as kiltera, runtime conformance testing of realistic mobile applications can be implemented effectively and efficiently.

In this thesis, we present an approach for checking the conformance of a mobile application with respect to an executable model at runtime. The approach is based on a novel high-level modeling language for mobile, distributed, and timed systems called kiltera [71]. Application of the approach starts with the creation of a high-level model of the system using, e.g., a UML profile; the high-level model is assumed to capture correctly the most relevant aspects of the system behavior such as descriptions of the movement of agents, their interaction with hosts and other agents, and any results
computed. Next, the high-level model is translated into a kiltera model; kiltera’s direct support for many relevant features (e.g., support for concurrency with (a)synchronous message passing, movement of processes, and time- and site-dependent behavior) makes this translation relatively straight-forward. The high-level model is then used to identify suitable “check points” at which conformance between the implementation under test and the kiltera model is to be checked; check points typically occur right before or after the sending or receipt of messages or agent movement; after these check points have been located in the implementation under test and the kiltera model, both are instrumented at these check points to allow relevant information to flow from the implementation under test to the kiltera model. During the last step of our approach, the implementation under test and the kiltera model are executed, both possibly in a distributed fashion. The kiltera model will report any non-conformance that arises during execution.

Our work benefits from the fact that kiltera allows a succinct, accessible expression of many features of mobile and distributed systems. Moreover, model debugging and analysis is possible using kiltera’s simulation environment. Finally, our approach does not assume a central monitoring component; instead, the kiltera model can be arbitrarily distributed, just like the implementation under test, which helps reduce any performance penalty. We have implemented the approach in a prototype and evaluated using a mutation-based evaluation framework.

The scope of our research is constrained by the following:

• In this thesis, we will assume that the mobile agent system has been implemented using the Aglets platform [49, 3]. However, since our approach makes
relatively few assumptions about the agent platform, application of our approach to implementations using other agent languages should be straightforward.

- In our approach, we do not consider test case generation at all. In other words, we assume that test cases are given and provide no way of generating them. While the generation of test cases from the kiltera model is possible, we leave this for future work.

- Our approach will assume the correctness of the model and the instrumentation necessary for monitoring. However, the executability of the models mitigates this assumption by allowing model debugging and analysis.

1.2 Contributions

The contributions of this thesis are the following:

- The development of an approach for automatically checking conformance of a mobile, distributed, timed application with respect to an executable model at runtime. The approach supports distributed monitoring, offers the benefits of formal, yet executable models and is relatively independent of the agent platform used.

- The design, implementation and evaluation of a prototype implementing the approach.

- The development of a set of new mutation operators for mobile agent systems implemented in Aglets. The purpose of these operators is to insert bugs in the
system under test that most of the programmers may make when they implement a mobile agent system. Our experiments confirm that the vast majority of the mutants generated using the approach are non-equivalent.

- The development and implementation of a mutation-based evaluation framework that is used to check the efficiency and the effectiveness of the proposed approach. In this framework the mutants are automatically generated from the system under test using the mutation operators and our analysis prototype is used to determine their conformance to a kiltera model.

### 1.3 Thesis Organization

In this chapter we have motivated and presented the research problem of testing mobile agent systems at runtime. Moreover, we presented the scope and the contribution of this thesis. The remaining chapters of the thesis are organized as follows:

- **Chapter 2**: This chapter presents relevant background and related work.

- **Chapter 3**: This chapter describes in detail the four steps of our approach for checking the conformance of a mobile agent application with respect to an executable model at runtime.

- **Chapter 4**: In this chapter, we apply our proposed approach to four case studies. Two of them are examples of mobile agent systems, the two others are implementations of distributed algorithms.

- **Chapter 5**: This chapter overviews a set of new mutation operators for mobile
agent systems. These mutation operators are used in the mutation-based evaluation framework in order to check the efficiency and the effectiveness of our approach.

- **Chapter 6**: In this chapter, we describe the mutation-based evaluation framework. Specifically we outline the steps of this framework and present the results of experiments using our approach. Moreover, details on the comparison of our runtime conformance checking approach with another monitoring approach are given.

- **Chapter 7**: Finally, this chapter concludes the thesis along with some discussion of the limitations and future work of our runtime conformance checking approach.
Chapter 2

Background and Related Work

This chapter begins with presenting the systems that we are interested in this thesis (Section 2.1). We give a definition of distributed systems and challenges of testing these systems. Moreover, we provide a definition of software agent and mobile agent systems. In Section 2.2, kiltera, the executable high-level language is described in detail. Section 2.3 describes the mobile agent framework Aglets. Solutions that are strongly related to our work are presented in Section 2.4. Section 2.5 summarizes the chapter.

2.1 Systems of Interest

2.1.1 Mobile Agent Systems

Because mobile agent systems are a special kind of distributed system, we first discuss these systems and the challenges of testing this kind of system.

There are several definitions of a distributed system. For example, Coulouris
et al [25] define a distributed system as “a system in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing”. Tanenbaum and Steen [82] define it as “a collection of independent computers that appear to the users of the system as a single computer”.

Distributed systems consist of a number of independent components running concurrently on different machines that interact with each other through communication networks to meet a common goal. In other words, in distributed systems the components are autonomous, i.e, they possess full control over their parts at all times. The components, however, have to take into account that they are being used by other components and have to react properly to requests. There are multiple points of failure in a distributed system. Distributed systems could fail because a component of the system has failed. Moreover, network communication is not always successful (transmission errors) and sometimes messages do not arrive on time. In real-time systems, for example, if the deadlines of the operations are not met, serious consequences may occur. Often, fault tolerance mechanisms must be used to ensure that these problems do not result in failure of the overall system. Moreover, when many network messages are transmitted over a particular network, the performance of the communication may deteriorate. In summary, all of these aspects severely complicate the correct design and implementation of distributed systems.

Testing distributed systems is a difficult and challenging task. It is much more difficult to test a distributed system than to test a sequential program. For sequential programs, we can say that the program is correct when its inputs produce correct outputs according to its specification. However, in distributed systems correctness of the input-output relationship of a single process alone does not imply the correctness
of the entire system, because components have an ongoing interaction and the sys-
tem could enter an improper state even if each process has the correct input-output
relationship.

Apart from the potential for non determinism and failure due to this ongoing inter-
action, there are many other characteristics of distributed systems that make testing
of this kind of system difficult. Typically, distributed systems are heterogeneous in
terms of communication networks, operating systems, hardware platforms and also
the programming language used to develop individual components. This means that
the system should be able to run over a wide variety of different platforms and access
different kinds of interfaces. Moreover, the size and complexity of distributed systems
is growing dramatically which complicates testing further. Nonetheless, because dis-
tributed systems are used in many critical applications in banks, hospitals, businesses,
ofices, etc, finding effective and efficient testing techniques for distributed systems
are more important than ever.

Software Agents

From the software engineering point of view a software agent is: “a software entity
which functions continuously and autonomously in a particular environment, often
inhabited by other agents and processes” [80]. There are different forms or types of
software agents [63, 81, 8]: autonomous (agents are dynamic entities and can make a
decision on their own), proactive (agents do not just follow the rules but it also de-
pends on the internal state of the agent), intelligent (agents have the ability to learn
and adapt according to the situations based on previous experience), cooperative
(collaborate with other agents), distributed (agents can be executed as independent
threads and on distributed processors) or mobile (the agent can move from one platform to another carrying its state, data, and code).

**Mobile Agent Technology**

Mobile Agent Systems (MASs) are a special kind of distributed system and “have been introduced as a key enabler of distributed computing” [8]. In MAS, the code of the agent has the capability to move from one host to another. Code mobility refers to the “capability to reconfigure dynamically, at runtime, the binding between the software components of the application and their physical location within a computer network” [18]. Mobile agents have been developed to extend or to enhance the functionality and the operations of the client-server paradigm [42, 72, 36]. Mobility may reduce bandwidth consumption and coupling and increase flexibility [70]. Mobility occurs naturally in many distributed system applications such as telecommunications and electronic commerce. In [5], Far states that “nowadays, an increasing number of software projects are being revised and restructured in terms of multi-agent systems”.

In addition to the potential for failures in distributed systems, the following additional failures may occur in mobile agent systems. The agent could be killed incorrectly by other agent. The agent could be lost if for example it is moved to an incorrect address. Furthermore, sometimes the agent loses the connection with other agents or it is moved under the wrong circumstances. To conclude, testing MASs is a complex and challenging task. Weyns and Georgeff [87] mentioned in a recent paper (January 2010) that “as with any distributed system, testing a multiagent system is challenging. Decentralization and development in an open environment add to the complexity”.
2.2 An Executable Modeling Language: kiltera

kiltera developed by Poss [71] is a language used to describe the behavior of timed, concurrent, interacting processes which may be distributed over several sites. It provides operators to compose processes in parallel, to describe communication via events, or equivalently via message-passing over channels, to limit the scope of events, to delay processes and to observe the passage of time, as well as to move processes to remote sites.

The core of the language is a process algebra which it is called $\pi_{klt}$, an extension of the $\pi$-calculus [59]. Unlike other basic process algebras, kiltera provides some higher-level constructs to facilitate development. In particular, kiltera allows the use of complex expressions and data-structures in messages, and uses pattern-matching as a mechanism to extract information from data.

The language has a formal semantics and a meta-theory (see [71]) which serve as the basis for the formal analysis of models. Furthermore, it has been implemented, supporting both uniprocessor and truly distributed simulation.

In the following we introduce informally a significant subset of the language and provide a brief description of the simulator.

2.2.1 Overview

A model or specification in kiltera consists of one or more modules, the smallest “movable” processing unit. Each module has the syntax:

\[
\text{module } A[\tilde{x}](\tilde{y}) : P \quad \text{or} \quad \text{module } A[\tilde{x}] : \text{sites } \tilde{s} \text{ in } P
\]
Here $P$ ranges over process terms, defined below. We use $x, x_i, \ldots$ for port/channel/event names, and $A, B, \ldots$ for process/module names, $s, s_i$ for site names, and $y, y_i, \ldots$ for any other variable name. The notation $\tilde{x}$ denotes a list of names or values $x_1, \ldots, x_n$. In the definition of a module, the names $\tilde{x}$ represent the interface of the module, that is, its ports, or equivalently, the events which it can use to communicate with other modules. The names $\tilde{y}$ represent local state variables and the (optional) $\tilde{s}$ represents the names of sites known by this module. The process body $P$ is a process term which describes the structure and behavior of the module.

The syntax for process terms $P$ is shown in Figure 2.1 on page 13. Here $E$ ranges over expressions, $F$ ranges over patterns, $op \in \{+, -, *, /, \text{mod}, \text{and}, \text{or}, \text{not}, <, >, =, \leq, \geq, != \}$, $n$ ranges over floating point numbers, $s$ ranges over strings, $x$ ranges over variable names, and $f$ ranges over function names, with function definitions having the form: function $f(\tilde{x}) : E$.

The process done simply terminates. The term “trigger $x$ with $E$” triggers an event $x$ and associates this event with the value of expression $E$. Alternatively, one can say that it sends the message $E$ through channel $x$ (a channel and an event are synonymous). The expression $E$ is optional. This process performs communication by unicasting: if there are multiple listeners, only one of them accepts the message, and the choice is non-deterministic.

\footnote{In the presentation of the syntax we use braces \{} and \{\} to denote syntactic nesting for the par and seq operators, but in the actual implementation and the examples we use indentation-based nesting.}
$P ::= \text{done} \mid \text{trigger } x \text{ with } E \mid \text{when } \beta_1 \rightarrow P_1 \mid \cdots \mid \beta_n \rightarrow P_n \mid \text{event } \tilde{x} \text{ in } P \mid \text{wait } E \rightarrow P \mid \text{par } \{P_1, \ldots, P_n\} \mid \text{process } A[\tilde{x}](\tilde{y}) : P_1 \text{ in } P_2 \mid A[\tilde{x}](\tilde{E}) \mid \text{move } A[\tilde{x}](\tilde{y}) \text{ to } s \mid \text{here } s \text{ in } P \mid \text{dchannel } \tilde{x} \text{ in } P$

$\beta ::= x \text{ with } F \text{ after } y$

$E ::= n \mid \text{true} \mid \text{false} \mid \text{"s"} \mid x \mid \text{op } E \mid E_1 \text{ op } E_2 \mid f(E_1, \ldots, E_m) \mid (E_1, \ldots, E_m)$

$F ::= n \mid \text{true} \mid \text{false} \mid \text{"s"} \mid x \mid (F_1, \ldots, F_m)$

Figure 2.1: \textit{kiltera} syntax

**Example 1:**

Figure 2.2 represents an example in \textit{kiltera} where the name of the module is Example1$^2$. Two processes are defined within this module: \textit{Sender} and \textit{Receiver}. Each of these two processes has one port $x\_ch$ and $y\_ch$ respectively. The definitions of these two processes are valid within the scope that is defined in lines 10-14 using the word \textit{in}. In this scope, we create two instances of the two processes that are executing in parallel and are connected through the channel (event) $a\_ch$. Lines 3-4 represent the body of the \textit{Sender} process and lines 7-9 represent the body of \textit{Receiver} process.

A process can trigger events and can react to events. In line 3, the \textit{Sender} process triggers an event $x\_ch$ and sends ‘‘message’’ as data along with this event. Then, the process terminates (line 4). In line 7, the \textit{Receiver} process listens to the occurrence of the event (through event $y\_ch$) and binds the received value to the variable.

$^2$Note that this module does not have any ports that are used to communicate with other modules.
data. When the Receiver process receives the message, it prints the received value (line 8) and terminates (line 9).

```plaintext
module Example1[]:
  process Sender[x.ch]:
    trigger x.ch with ‘message’
    done
  process Receiver[y.ch]:
    when y.ch with data ->
      print data
    done
  in channel a.ch in
    par
      Sender[a.ch]
      Receiver[a.ch]
```

Figure 2.2: An example in kiltera

Channel mobility is achieved in the same way as in the π-calculus since event/channel names are expressions, and so they can be sent to other processes as messages. The process “when $\beta_1 \rightarrow P_1 \mid \cdots \mid \beta_n \rightarrow P_n$” is a listener, consisting of a list of alternative input guarded processes $\beta_i \rightarrow P_i$. Each input guard $\beta_i$ is of the form “$x_i \mathbin{\text{with}} F_i \mathbin{\text{after}} y_i$”, where $x_i$ is an event/channel name, $F_i$ is a pattern, and $y_i$ is a variable (the suffixes “with $F$” and “after $y$” are optional). This process listens to all events (channels) $x_i$, and when $x_i$ is triggered with a value $v$ that matches the pattern $F_i$, the corresponding process $P_i$ is executed with $y_i$ bound to the amount of time that the listener waited, and the alternatives are discarded \(^3\). A listener process represents, thus, a process in a state with external choice. Pattern-matching of inputs means that the input value must have the same “shape” as the pattern, and if successful, the free names in the pattern are bound to the corresponding values of the

\(^3\)Note that to enable an input guard it is not enough for the event to be triggered: the event’s value must match the guard’s pattern as well.
CHAPTER 2. BACKGROUND AND RELATED WORK

input. For example, the value \((3, \text{true}, 7)\) matches the pattern \((3, x, y)\) with the resulting binding \(\{x \mapsto \text{true}, y \mapsto 7\}\). The scope of these bindings is the corresponding \(P_i\). The process “\(\text{event } \tilde{x} \text{ in } P\)”, also written “\(\text{channel } \tilde{x} \text{ in } P\)”, hides the names \(\tilde{x}\) from the environment, so that they are private to \(P\). Alternatively, we can say that it creates new events/channels \(\tilde{x}\) whose scope is \(P\). The process “\(\text{wait } E \to P\)” delays the execution of process \(P\) by an amount of time equal to the value of the expression \(E\). The process “\(\text{par } \{P_1, ..., P_n\}\)” is the parallel composition of \(P_1, ..., P_n\).

The process “\(\text{process } A[\tilde{x}](\tilde{y}) : P_1 \text{ in } P_2\)” declares a new process definition \(A\) with ports \(\tilde{x}\), (optional) state variables \(\tilde{y}\) and body \(P_1\). The scope of this definition is the process \(P_2\). The term “\(A[\tilde{x}](\tilde{E})\)” creates a new instance of a process (or module) named \(A\), whose definition is in the current scope, where the ports \(\tilde{x}\) and variables \(\tilde{y}\) of the definitions are substituted in the body of \(A\) by the events or channels \(\tilde{x}\) and the values of \(\tilde{E}\) respectively. The process “\(\text{move } A[\tilde{x}](\tilde{y}) \text{ to } s\)” creates an instance of the process defined by module \(A\) in site \(s\). The process “\(\text{here } s \text{ in } P\)” binds the name of the local site to \(s\) in \(P\). Finally, the process “\(\text{dchannel } \tilde{x} \text{ in } P\)” creates a channel to communicate with modules on remote sites. Note that modules are essentially the same as process definitions, except that they do not have a surrounding lexical context and therefore are self-contained, since their only external references are its ports. This is why we only allow modules to be moved to other sites.

**Example 2:**

Figure 2.3 represents an example in kiltera where the name of the module is Example2. This module has one port \(\text{module\_port\_connection}\) used to connect this module with other modules. Moreover, this module has two processes \(P_1\) and \(P_2\).

\(^4\text{This is essentially the same as pattern-matching in functional languages like ML or Haskell.}\)
module Example2 [module_port_connection]
sites A, B
process P1 [x_ch]:
  channel response_ch in
  wait 50 ->
  seq
    trigger x_ch with ("create", response_ch).
    when response_ch with result ->
    // Do something with result
when x_ch with (data, response_ch) ->
  par
    move Agent [module_port_connection] to B
    trigger response_ch with "created".
in
channel z_ch in
par
  P1 [z_ch]
  P2 [z_ch]

Figure 2.3: Mobility in kiltera

Lines 4-9 represent the body of process P1 and lines 12-15 represent the body of process P2. There are two sites known by this module A and B (line 2). These sites are used to identify the location that a process is in since each process executes in a site. In process P1, a new channel response_ch has been created (line 4). The scope of this channel are the lines 5-9. Line 5 makes the process P1 to wait for 50 seconds. After that the lines 6-9 will be executed. The sequential composition operator seq in line 6 executes the statements in lines 7-8 sequentially. Line 7 represents an example of channel mobility, where the tuple ("create", response_ch) is sent as a message through the channel x_ch. Here, response_ch is a mobile channel and is sent as a message. In line 12, process P2 will receive the message through channel x_ch because the tuple ("create", response_ch) matches the pattern (data, response_ch). The parallel composition operator par in line 13 executes
the statements in lines 14-15 concurrently. In line 14, we create an instance of module Agent at site B and allow communication with module Example2 through the channel module_port_connection. In line 15, process P2 sends a response to process P1 with the message “created” through the mobile channel response_ch.

There are several derived process terms, such as sequential composition, timeouts, conditionals, etc. Here we only mention a few of them. The process term “\(\text{when } \beta_1 \rightarrow P_1 | \cdots | \beta_n \rightarrow P_n \text{ timeout } E \rightarrow P\)” associates a timeout with a listener. If after an amount of time determined by the value of the expression \(E\) none of the events have been triggered, control passes to \(P\). The process “\(\text{match } E \text{ with } F_1 \rightarrow P_1 | \cdots | F_n \rightarrow P_n\)” evaluates the expression \(E\) and attempts to match it with each pattern \(F_i\). If a pattern \(F_i\) matches then the corresponding process \(P_i\) is executed. If more than one pattern matches the choice is non-deterministic. The process “\(\text{if } E \text{ then } P \text{ else } Q\)” is shorthand for “\(\text{match } E \text{ with } \text{true} \rightarrow P | \text{false} \rightarrow Q\).” Finally, we also have terms of the form “\(\text{let } y = E \text{ in } P\)” which define local names in a process \(P\).

Example 3:

Figure 2.4 shows two examples in kiltera. The first one shows the match statement and the other shows the timeout statement. In Figure 2.4 (a) we are using the match statement to check whether the Queue is empty or not. If the Queue is empty (line 2) the process will terminate (line 3). Otherwise, the head of the Queue will be assigned to the variable head and the remainder of it will be assigned to the variable remainder. In Figure 2.4 (b) the process waits for an event a_ch for an amount of time equal to the value of \(t\). If the event occurs before this amount of time, “\(\text{data received}\)” (line 2) will be printed and the statement print “\(\text{‘TimeOut’}\),
is discarded. If the event does not occur in this amount of time, the statement in line 2 is discarded and TimeOut will be printed.

(a): match

(b): timeout

---

<table>
<thead>
<tr>
<th>time</th>
<th>location</th>
<th>action</th>
<th>port</th>
<th>event</th>
<th>data</th>
<th>position</th>
<th>site</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000</td>
<td>Example1.Sender</td>
<td>trigger</td>
<td>x_ch</td>
<td>a_ch</td>
<td>message</td>
<td>(2,4)</td>
<td>local</td>
<td>1</td>
</tr>
<tr>
<td>0.000</td>
<td>Example1.Receiver</td>
<td>reaction</td>
<td>y_ch</td>
<td>a_ch</td>
<td>message</td>
<td>(-1,-1)</td>
<td>local</td>
<td>1</td>
</tr>
<tr>
<td>0.000</td>
<td>Example1.Receiver</td>
<td>print</td>
<td>None</td>
<td>None</td>
<td>&quot;message&quot;</td>
<td>(-1,-1)</td>
<td>local</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2.5: An event trace of Example1 in Figure 2.2

The event-scheduler contains a queue of simulation events (terms) to be executed,
but rather than store them all in a single linear queue, \textit{kiltera} divides them into \textit{time-slots}, i.e., sequences of all simulation events to be executed at a given instant in time. Hence the global event queue is a time-ordered queue of time-slots, each of which is a queue of terms. Execution proceeds by taking the first time-slot in the queue, taking the first term in the time-slot and performing its action. Once the first time-slot becomes empty, the simulator proceeds to the next time-slot.

Each action executed depends on the specific construct. The \texttt{par} construct for example, simply adds the subterms to the current time-slot. The action for \texttt{event} creates a new communication event object in the heap. Interaction is done by means of the observer design pattern. The action for a \texttt{when} creates a listener for the appropriate events and registers them with the corresponding event objects. The \texttt{trigger} construct notifies the communication event object, which then selects one of its listeners and executes the corresponding continuation (while discarding other branches of the original listener). The delay construct \texttt{wait} simply adds the term in the appropriate time-slot of the global queue.

The \textit{kiltera} simulator supports two modes: real-time and logical time. In real time mode, once the current time slot becomes empty, the simulator actually pauses and waits until the time associated with the next time slot has been reached. In logical time mode, once the current time slot becomes empty, a logical global clock is set to the time of the next time slot and execution resumes immediately. Figure 2.6 shows an event trace of an updated version of Example1 in Figure 2.2 executed in real time mode by adding the statement “\texttt{wait 5-}” between lines 2 and 3).

Distributed simulation in \textit{kiltera} is achieved using the TimeWarp algorithm [43]. Briefly, this is an optimistic simulation algorithm, where multiple event-schedulers
(such as the one described above) run on different sites, but rather than blocking to
wait for external messages, each simulator proceeds as fast as it can, and whenever an
external message arrives, its time-stamp is compared with the local clock (the time
of the first time-slot). If the external message has a future time-stamp relative to the
local time, it is simply scheduled. If it is in the past, then the simulator rolls back
to a time before the message’s time. This algorithm is guaranteed to yield the same
behavior of a single global event-scheduler.

In the case of kiltera whenever we move a module to a remote site we send a copy
of the module to that site (the IP address for the site is specified in a configuration
file). The site must be running a daemon which, whenever it receives a module,
starts a new simulator locally which runs the newly arrived module. Each simulator
executes the TimeWarp algorithm, and has a “dchannel” manager to handle remote
communications through the dchannels created locally.

2.3 Aglets

The Aglets Software Development Kit (ASDK) [49, 3] is a framework and environment
for developing and running mobile agents. It was originally developed at the IBM
Tokyo Research Laboratory. The Aglets system is one of the most popular Java-based open source mobile agent systems [34, 50, 85]. The ASDK includes Aglets API packages, documentation, sample Aglets, and the Tahiti Server. Tahiti is a Java application that allows a user to receive, manage, and send aglets to other computers that are running Tahiti [67].

The execution environment for Aglets is called context. An Aglet is created, exists, works, sleeps, and dies in such a context. An Aglet is a Java agent able to autonomously and spontaneously move from one context to another. In Aglets, the agent starts its work by executing its run() method. Furthermore, it can double itself by using the clone() method. It can migrate from one context to another using the dispatch() method. The retract() method requests a destination agent system to send the specified agent back to the sender. Also, the agent can destroy itself using the dispose() method. In Aglets, agents can communicate with each other by exchanging messages using the sendMessage() method. When a message is sent to an agent, its handleMessage() method is called so that the agent receives the message. Figure 2.7 shows the general form of an agent in Aglets.

Before any major event in an agent’s life, a callback method is invoked to allow the agent to prepare for the event. Aglets support several of these callback methods. These callbacks can be overridden in order to customize the behavior of an agent. The Aglet’s callback methods are summarized in Table 2.1.

For more information, see [49, 3]. Based on these methods, Aglets support three kinds of event. These events are discussed in [30]. As presented in the Aglets manual [30]:


declare
Aglets supports an event/event listener model, where an agent can register event listeners to particular kind of events, thus it can handle events. There are mainly three kind of events, tied to different scenarios of the agent life cycle: cloning, mobility and persistency.

Aglets has been used to implement different agent-based systems, for example: Shakshuki and Abu-Draz [79] use Aglets to implement a multi-agent system of an online trading system. This system is designed to help users to buy products from distributed resources based on their interests and preferences.

Ong and Sun [67] use Aglets to develop a web-based real-time monitoring system. In this system, there are different kinds of distributed machines that are connected with the central controller. While these machines are running they can request a specific monitoring program from the central controller. This process is performed by a request agent. When the system controller receives this request through an agent,
Table 2.1: Aglets callback methods

<table>
<thead>
<tr>
<th>Callback method</th>
<th>when it is called</th>
</tr>
</thead>
<tbody>
<tr>
<td>onCreation</td>
<td>the first time an aglet springs to life. Used to initialize a new agent. This method is called only once in the life cycle of an agent</td>
</tr>
<tr>
<td>onCloning</td>
<td>about to be cloned</td>
</tr>
<tr>
<td>onClone</td>
<td>clone is actually created</td>
</tr>
<tr>
<td>onCloned</td>
<td>after the clone was created</td>
</tr>
<tr>
<td>onDispatching</td>
<td>before an agent is actually dispatched</td>
</tr>
<tr>
<td>onReverting</td>
<td>about to be retracted</td>
</tr>
<tr>
<td>onArrival</td>
<td>after arrived at the destination</td>
</tr>
<tr>
<td>onDeactivating</td>
<td>about to be deactivated</td>
</tr>
<tr>
<td>onActivation</td>
<td>after activated</td>
</tr>
</tbody>
</table>

this agent returns a suitable monitoring program to the request machine so that it can be monitored locally.

Chen and Tu [21] also use Aglets to develop a mobile communication system for exchanging textual and graphical information. Their system is a handwriting communication system that employs Aglets for information exchange and for enhancing collaborative learning on the internet.

2.4 Related Work

We identify four important research topics that are closely related to this thesis. The following subsections describe these topics.

2.4.1 Model-Based Testing

Model-based testing (MBT) consist of four main phases:

1. Build the model,
2. Generate test cases,

3. Execute test cases, and

4. Check the conformance.

During the software testing process the testers use a model that is built from the requirements of the system in order to describe the application behaviors. Moreover, sometimes they use the model to automatically generate test cases. As described in [4, 31], this model can be presented in terms of, e.g., the input sequences accepted by the system, the actions, and the outputs performed by the system. The second phase is to generate test cases. In MBT, test cases can be generated online or offline. There are several coverage criteria in the literature that help the tester control the test generation process [86] (e.g., structural model coverage criteria, data coverage criteria, random and stochastic criteria, fault-based criteria, etc). In the third phase, if test cases are generated from the model we need to translate them into an executable form. Since these executable test cases can be used for as long as the software needs more testing, they should be written in a very efficient way. After that we apply these executable test cases to the implementation under test (IUT) to produce the actual outputs of the system. After executing the test cases and getting the actual outputs of the system, we have to evaluate and analyze the results. In checking conformance, a comparison between the actual outputs of the IUT with the expected outputs is done (phase four).

MBT has been used to test the behavior of different kinds of distributed systems. E.g., Bochmann and Petrenko [12] review existing protocol testing methods including methods using Finite State Machines (FSM) and Labeled Transition Systems (LTS) to derive the system specification, its implementation, conformance relations and to
generate test sequences. Furthermore, MBT is also used to improve different quality attributes in distributed systems such as performance [29], security [11, 88], deadlock [22], and reliability [37, 48]. Several different MBT tools are available, e.g., Spec Explorer [17], TorX [83], Conformiq Qtroniq [41], etc. Brinksma and Tretmans [16] present an annotated bibliography of testing labeled transition systems which points out the relevant work in this area. More details about using MBT to test distributed systems and comparing between different MBT tools can be found in [73, 74].

Our approach has several similarities with MBT. First, we also build the model from the requirements that represents the behavior of the implementation under test. Moreover, in our approach we execute the test cases to get the actual outputs, as in MBT. Furthermore, we have a conformance check step that is used to check the conformance between the expected outputs and the actual outputs. However, our approach differs from the standard MBT procedure described above in which MBT test cases are generated from the model. In our approach, we do not consider test case generation at all. In other words, we assume that test cases are given and provide no way of generating them. While the generation of test cases from the kiltera model is possible, we leave this for future work. All of our examples do not require user input. Therefore, no test cases need to be created to be able to use our approach.

2.4.2 Runtime Monitoring

Monitoring can be used to check the completeness of the test suite with respect to the specification. For example, Amyot and Logrippo [7] use a probe insertion technique to measure the structure coverage of a LOTOS specification. In their approach they insert probes in a specification to measure the coverage of all the instances of events.
The purpose of this approach is to detect incomplete test suites, check the consistency between the specification and its test suite, and check for unreachable parts of the specification with respect to the requirements.

Moreover, runtime monitoring is the activity of checking the conformance between the target program (the implementation) and the requirements specification of that program at runtime. It is used to detect violations in a system that cannot be detected by using, e.g., testing or model checking. For example, due to the complexity of the systems and to the large numbers of system behaviors, testing cannot be used to guarantee that the system is correct. Moreover, due to the large number of states, model checking may be intractable. Many approaches and tools for runtime monitoring exist (e.g., Java MaC [45, 44], Java PathExplorer [40], Java Runtime Timing-constraint Monitor [60], decentralized monitoring [77], and Java Monitoring-Oriented Programming [19]). All these approaches are based on the same idea: The monitored code is instrumented (possibly automatically) such that it emits sequences of events during execution; event sequences are analyzed by a central analysis component for specification violations. Approaches differ with respect to the application domain (e.g., real-time systems [60, 44], distributed systems [77], and sequential and concurrent systems [45, 19]), the degree of automation of the instrumentation phase (e.g., automatic instrumentation from specifications [45, 77], automatic instrumentation using scripts [40], and manual instrumentation [60]), and the specification formalism supported (e.g., temporal logic [45], timing constraints [60], and rewrite logic [40]). However, our approach presented in this thesis is unique in that it supports the possibly distributed analysis of mobile code. Moreover, specifications are
expressed using an executable process algebra; compared to more declarative specifications (e.g., using temporal logic), this kind of operational specification appears more suitable for the comprehensive description of agent interactions without sacrificing mathematical rigor. Finally many proposed runtime monitoring approaches are based on aspect-oriented programming; which our approach does not use at all. In Section 6.4 a detailed comparison of our approach with aspect-oriented runtime monitoring is given. We will see that for our case studies our approach requires less instrumentation. Moreover, we observe that standard mutation-based testing is not applicable to aspect-oriented monitoring.

Mani et al [57] propose an approach for monitoring the behavior of Multi-Agent Systems (MulAS)\(^5\) to detect resource and communication deadlock. They use UML 2.0 Sequence Diagrams to represent the behavioral model of the MulAS. Then they instrument the MulAS source code by adding two detection techniques: one to detect the resource deadlock and the other to detect the communication deadlock. The monitoring module runs the MulAS system and uses the model to detect the deadlocks. This approach focuses only on how to detect deadlock in MulAS. However, our approach is used to monitor several properties including the correct movement of mobile agents as well as the proper realization of behavioral and timing constraints of mobile, distributed and time applications.

Zhang et al [90] develop a new runtime monitoring tool for distributed systems called FiLM. They use this tool to monitor at runtime the execution of distributed systems against a finite automaton generated from the LTL specification. FiLM is

\(^5\)MulAS is a system composed of several agents, collectively capable of achieving goals that are difficult to achieve by an individual agent or monolithic system. Note that the agents are not necessarily mobile.
used to monitor the calling of user-defined functions or system-defined APIs of distributed Java systems. The tool first instruments the function or the APIs. Then this instrumentation is used to generate the corresponding state information of the function or the APIs that have been executed. After that, this state information is taken by the tool D^3S to generate a sequence of consistent global snapshots with global timestamps\(^6\). The tool then takes the sequence of snapshots as an input to decide whether the property specified by the LTL formula has been violated in the finite trace of the distributed application or not. In this approach FiLM is used to monitor distributed systems. Our approach, on the other hand, is used to monitor both mobile agent systems and distributed systems. Moreover, FiLM is a centralized monitoring tool. However, our approach does not assume a central monitoring component; instead, the kiltera model can be arbitrarily distributed, just like the implementation under test, which helps reduce any performance penalty.

Monitoring can also be performed by using aspect-oriented programming. In this approach, the monitor code is not contained in a separate process, but rather in the unit under test itself. Aspect languages such as AspectJ [33] have been proposed to implement this approach where the monitor code is put into aspects and woven into the system under test at the appropriate places prior to execution [56, 20]. A detailed comparison between our approach and aspect-oriented monitoring will be given in detail in Section 6.4.

\(^6\)D^3S [51] is a checker that allows developers to specify predicates on distributed properties of a deployed system, and that checks these predicates while the system is running.
2.4.3 Testing Mobile Code and Agent Systems

A more limited amount of work exists devoted to the testing of mobile code. Delamaro et al [26], present a framework that is used to support testing of mobile Java code with respect to the standard code coverage criteria (e.g., statement coverage, branch coverage, etc). Figueiredo et al [32], show how test patterns are obtained from design patterns for mobile agents. The specification of the design patterns is used to generate several test cases for any implementation that uses these patterns. To check the efficiency and the reliability of the approach, one of the patterns has been applied to three different sample applications.

A formal framework for conformance testing of mobile code is presented by Marche and Quemener [58]. The work extends the conformance testing theory for distributed systems developed by Tretmans [84] uses labeled transition systems. Our approach, on the other hand, is based on an extension of the $\pi$-calculus, which appears much more suitable, because of the explicit support for mobility. However, our work currently lacks a formal definition of conformance.

In the context of testing agent systems, agents feature not only autonomy and mobility, but also some form of planning based on, e.g., the “beliefs”, “desires” and “intentions” (BDI) model. Our work is not concerned with planning at all. However, it is possible to add planning to the $\text{kiltera}$ model and thus also make it subject to the conformance check.

Agent-oriented software engineering (AOSE) is concerned with supporting the effective construction of reliable agent systems. Most AOSE methodologies (such as Prometheus [68]) advocate the use of models (e.g., sequence diagrams and state machines) in early stages of development. Several papers suggest leveraging these models
for test case generation [78, 92, 91, 61]. The work in [6] discusses conformance testing and thus is closer to ours: agents are monitored with respect to “interaction constraints” that capture properties of interactions between agents; constraint checking is implemented using constraint logic programming; timing constraints are supported, but support for mobility and distributed monitoring appears to be missing.

2.4.4 Program Mutation

We use program mutation [28, 39, 64] to generate a set of mutants in order to check the effectiveness and the efficiency of our proposed approach. Mutation operators specify a single, small, local change to a program and are used to generate a set of mutants from the system under test. A mutant is a slight version of the original program that arises through the application of a mutation operator. Each mutant is executed and when it is detected that the mutant has produced an incorrect result we say that the mutant is killed. However, it is possible that the mutant produces the same outputs as the original program and cannot be killed. In this case we say that the mutant is equivalent. These mutants are not useful for testing or for the analysis. Mutation can be used to evaluate test suites or to generate test cases, but in our approach we used mutation just to check the effectiveness and the efficiency of our proposed approach. Several works have been proposed for using mutation for testing. For example, in [14] a new experimental mutation analysis framework has been proposed for concurrent Java called ExMAn. The ExMAn framework uses 25 mutation operators to test concurrent Java programs. These mutation operators are classified into five categories: modify parameters of concurrent methods, modify the occurrence of concurrency method calls, modify keywords, switch concurrent objects, and modify critical regions.
The MuJava tool [54] uses two types of mutation operators for Java programs: one for method level operator [54] (29 mutation operators) and the second for class level operator [53] (29 mutation operators). Mothra [27, 46] is another mutation tool that uses 22 mutation operators for Fortran-77 programs.

### 2.5 Chapter Summary

This chapter provides the reader with the required background knowledge on runtime monitoring of distributed and mobile agent systems. Section 2.1 describes the systems that we interested in this thesis by defining the term *distributed systems, software agent and mobile agent*. Section 2.2 introduces informally a significant subset of the executable modeling language *kiltera* and briefly describes the simulator. Section 2.3 describes briefly the Aglets mobile agent framework. Section 2.4 surveys the related work to this thesis.
Chapter 3

Description of Approach

In this chapter, we present our approach for checking the conformance of a mobile, distributed application with respect to an executable model at runtime. The approach is based on a novel high-level modeling language for mobile, distributed, and timed systems called \textit{kiltera} presented in Chapter 2. Section 3.1 states the main idea of our approach: it briefly describes a UML profile used to express the high-level model, it gives an informal description of \textit{kiltera}, and sketches how to go from the high-level model to a \textit{kiltera} model. Moreover, this section explains how we instrument the high-level model and \textit{kiltera} model, and finally it describes how the instrumented implementation under test and the instrumented \textit{kiltera} model are executed in parallel. Section 3.2 summarizes the chapter.

3.1 Proposed Approach

The approach represents a new technique for checking the conformance of a mobile, distributed application with respect to an executable model at runtime. The approach
is based on kiltera as described in Chapter 2. The approach consists of the following four steps:

1. Construction of the high-level model (HLM) of the system,

2. Translation of the HLM into a kiltera model (KM),

3. Instrumentation of the implementation under test (IUT) and KM using the HLM,

4. Execution of the instrumented IUT and the instrumented KM in parallel.

Figure 3.1 shows the steps of our approach assuming the IUT is implemented using Aglets. We will now describe each of these steps in detail.

Figure 3.1: Steps of our conformance testing approach
3.1.1 Construction of HLM

Application of the approach starts with the creation of a high-level model (HLM) of the system using, e.g., a UML profile. The HLM is assumed to capture the most relevant aspects of the system behavior such as descriptions of the movement of agents, their interaction with hosts and other agents, and any results computed. In order to describe mobile agent systems we use a UML profile introduced in [47] as our high-level modelling language. This profile extends the UML with four new types of Sequence Diagrams. Here we use only one of these types, called “Swimlaned Mobility Diagrams” (SMDs for short). These diagrams are intended to represent agent location, agent creation and agent movement. Note that UML2 sequence diagrams and therefore also SMDs support the standard control flow constructs such as branching, loops, and parallel and sequential composition. For instance, Figure 3.2 on page 34 shows how to specify a request from X to Y and to check whether the result that X got from Y is equal to “ok” or not. If yes, then X will do A, otherwise, do B. An SMD

![Figure 3.2: A conditional interaction in standard Sequence Diagrams](image)

consists of one or more swimlanes representing nodes (a.k.a. hosts or locations). Each
swimlane is visually represented by a column labelled with the name of the node. Within each swimlane there is a Sequence Diagram with a life-line for each agent in that node. In addition to the standard message arrows for Sequence Diagrams, SMDs can have two new types of arrows between agent life-lines: 1) arrows that represent agent creation, labelled \textit{new}, and 2) arrows that represent agent movement between nodes, labelled \textit{move}. Message arrows between life-lines in different swimlanes represent remote communication. Figure 3.3 on page 35 shows a small example where an agent P1 located at a node A creates, in the same node, a new agent P2 which subsequently migrates to a node B.

![Figure 3.3: A simple Swimlaned Mobility Diagram (SMD)](image)

3.1.2 Construction of KM from HLM

In the second step of our approach, the HLMs are translated into kiltera models (KMs). In this step, the translation from HLM to KM is assumed to be manual, but since kiltera is Turing complete, i.e., every computable algorithm can be expressed in kiltera, this translation is guaranteed to always be possible. Moreover, kiltera offers direct support
for many relevant features (e.g., support for concurrency with (a)synchronous message passing, movement of processes, time, and site-dependent behaviour). Consequently, expressing these concepts in KM is quite straightforward. Partial automation of this translation is possible. However, it is outside the scope of this thesis.

Now we sketch how to go from high-level models specified as SMDs to executable models specified as kiltera models. First we discuss briefly how standard Sequence Diagrams can be represented in kiltera. Then we describe how SMDs are represented.

**Emulating standard Sequence Diagrams in kiltera**

In a normal Sequence Diagram we have several life-lines for different active objects. Arrows between life-lines represent message passing. In kiltera, active objects are processes: a process definition corresponds to the class of an active object and object creation is achieved by process instantiation. The ports in a process definition determine the messages that a process can send or receive. The parallel composition operator par is used to spawn parallel life-lines. Message-passing is achieved with trigger (to send a message) and when to wait for a message. Communication in kiltera is asynchronous. Synchronous messages can be modeled by means of an acknowledgment/response protocol. This is typically done as follows: the sender of a message creates a local “response” channel and sends this channel together with the message, and then waits for the answer on this new local channel. The receiver uses this private channel to send an acknowledgment, or the response to the message. Figure 3.4 shows such an example.
CHAPTER 3. DESCRIPTION OF APPROACH

```plaintext
process Sender[x_ch]:
  channel response_ch in
  seq
    trigger x_ch with ('message', response_ch)
    when response_ch with result ->
      // Do something with result

process Receiver[x_ch]:
  when x_ch with (data, response_ch) ->
    // Do something with data
  trigger response_ch with answer

channel a_ch in
par
  Sender[a_ch]
  Receiver[a_ch]
```

Figure 3.4: Message communication in kiltera

Emulating SMDs in kiltera

In kiltera, sites play the role of nodes or hosts. Within a kiltera model, sites have symbolic names, introduced by the sites keyword. These symbolic names can be associated with actual IP addresses in a separate configuration file. Agents are represented by modules.

Before an agent is able to create agents in a remote site or move to a remote site, it needs to know the target site. There are three ways in which an agent can know a site. The first is if the site was given in its sites declaration. The second is by using the here operator to obtain the name of the local site. The third is by receiving a site name sent by another agent. This is possible since site names are considered first-class values and therefore they can be transmitted in messages through channels.

Modeling the creation and movement of agents can be done in several ways, and it depends on how we assign the responsibility of moving agents, that is, who initiates movement: should an agent tell another agent to move, or should an agent move itself.
Also, there are different approaches to transferring the state of an agent. Furthermore, we can create an agent first locally and then move it, or we can create it at the remote site. The latter is directly captured by the semantics of the move construct: \texttt{move } A[\tilde{x}](\tilde{y}) \texttt{to } s \texttt{creates a new instance of } A \texttt{in a (possibly remote) site } s, \texttt{linked through the channels } \tilde{x} \texttt{and with initial state values given by } \tilde{y}. Creating a copy of an agent (module) locally can be achieved by \texttt{here } s \texttt{in } \texttt{move } A[\tilde{x}](\tilde{y}) \texttt{to } s. \texttt{This shows how new arrows of SMDs can be directly modeled with the move operator. So in this case, the process/module creating the agent is also responsible for moving the agent to the target site.}

How can we emulate an agent migrating on its own? The simplest way is by capturing all the necessary state, test if the agent is already in the destination, and if not, move a copy of itself there with the required initialization state and stop. For example, the HLM presented in Figure 3.3 on page 35 could be encoded in \texttt{kiltera} as in Figure 3.5.

```
module P2[\textit{x,ch}](\textit{state}):  
sites A, B  
  // Use modify \textit{state yielding newstate}
  here s in
    match s with
      A ->
      move P2[\textit{x,ch}](\textit{newstate}) to B
      | B ->
      // ... do what needs to be done in B
```

Figure 3.5: The \texttt{kiltera} model of the corresponding SMD presented in Fig. 3.3
3.1.3 Instrumentation of IUT and KM using HLM

After we have constructed the HLMs and translated them into KMs, we start the instrumentation step. In the instrumentation step, we instrument both the IUT (implemented in Aglets) and the KM to allow relevant information to flow from the IUT to the KM. We instrument the IUT first because it is going to send information to the KM. We use the HLM to start the instrumentation step. First, we identify a collection of “check points” at which conformance between the (IUT) and the KM is to be checked. Typically, check points involve the sending or receipt of messages or agent movement. Figure 3.6 shows the HLM with a check point numbered by 1 inserted before the agent P2 moves from node A to node B.

![Image of SMD with a check point](image)

Figure 3.6: SMD with a check point

Then we manually locate these check points in the IUT and the KM. Figure 3.7 shows the location of the check point 1 in the IUT (move the agent from node A to node B) and Figure 3.8 shows the corresponding location of check point 1 in KM.

Next, at each check point in the IUT, we insert appropriate instrumentation code which transmits relevant state information to the KM via sockets through a process.
we call “Python connector”. The Python connector is a Python script [1] that serves as a “bridge” between the IUT with the KM. In addition, at each check point in the KM, we insert instrumentation code that receives state information from the IUT and compares it with the expected internal information; if they are different, then non-conformance is signalled; otherwise, the KM continues. Figure 3.9 and Figure 3.10 show examples of instrumentation code inserted in the IUT and in the KM respectively.

```
public void onArrival(MobilityEvent me){
    try{
        /∗ Check point 1 */
        /∗ Inserted code begin */
        Socket s = new Socket("localhost",60002);
        PrintStream str = new PrintStream(s.getOutputStream());
        str.println("Agent P2 traveled to node B");
        /∗ inserted code end */
        /∗ Agent arrived at a new destination */
    } ...
}
```

Figure 3.9: Code inserted into the IUT of agent P2 at check point 1

The instrumentation in Figure 3.9 causes the name of the process and the destination to be sent in a string via socket s just when the process arrived at the new destination.
CHAPTER 3. DESCRIPTION OF APPROACH

//Check point 1
//Inserted code begin
HandleJavaMessages[external_ch]
when external_ch with data->
if data = ‘Agent P2 traveled to node B’ then
move P2[x_ch](newstate) to B.
else
//Error, stop execution with non-conformance
//inserted code end

Figure 3.10: Code inserted into KM of agent P2 at check point 1

The kiltera code in Figure 3.10 receives a string from the Python connector via channel external_ch. If the string has the expected content, the process moves.

3.1.4 Execution of IUT and KM

During the last step of our approach we execute the instrumented IUT and the instrumented KM, both possibly in a distributed fashion, and check the conformance between them. While the IUT is running, it sends messages possibly including state information to the KM through the Python connector. When the KM receives a message from IUT through the Python connector, it compares it with its expected internal state. In the case of non-conformance, the KM stops its execution and outputs an error message together with the trace of the execution. Figure 3.11 shows the architecture of the monitoring infrastructure of our approach.

3.2 Chapter Summary

Relatively little work has been done to support quality assurance techniques such as testing and verification of mobile agent systems [26, 89]. In this chapter we have presented a new approach for runtime conformance checking of mobile agent systems
CHAPTER 3. DESCRIPTION OF APPROACH

Figure 3.11: The architecture of the monitoring infrastructure

using executable model. Our approach combines support for mobility, distributed execution and time and uses models which can be directly executed in a distributed simulation environment.

The kiltera modeling language is an extension of the well known π-calculus and rests on a firm theoretical foundation which includes a formal semantics and different notions of equivalence [71]. The model specifies the agent system behaviour in high-level operational, rather than declarative, terms (e.g., through the use of a specification logic such as Linear Temporal Logic as in Java MaC [45, 44]) which we feel is more suitable for the description of possibly complex agent interactions.

Our approach only assumes the availability of the socket communication in the agent platform. On the downside, the kiltera model currently needs to be created manually. Moreover, our notion of conformance is not currently formally defined, but it is inspired by simulation relations defined for most process algebras such as CCS.
and the $\pi$-calculus.
In this chapter we demonstrate the effectiveness of our proposed approach described in Chapter 3 by using four case studies. Two of them are mobile agent systems, the two others are implementations of distributed algorithms. The first example is an online shopping agent application implemented by the author using Aglets. In this application, an agent is searching for a specific item (e.g., a camera) by traveling to different online shopping malls in order to find the lowest price for this item and return the result of the search to the original site. The second example presents a scenario where different partners, who are spatially separated from each other, want to sign a contract for a specific project. To this end, an agent representing the contract migrates to all the partners involved. The two other examples are implementations of a token ring mutex algorithm and Lamport’s distributed mutex algorithm (implementations of these two algorithms are taken from [35]).

We checked several properties of these examples including the correct movement of mobile agents as well as the proper realization of behavioral and timing constraints. In Section 4.1, we apply the four steps of our approach to the online shopping example.
Section 4.2 discusses the application of our approach to the second example. Section 4.3 discusses its use for the analysis of two distributed algorithms. Finally, Section 4.4 summarizes the chapter.

4.1 Case Study: Online Shopping Example

In this experiment we have applied the four steps of our conformance checking approach to the online shopping example. This section shows some examples of checking sample properties of the shopping application.

4.1.1 General Description

In this application, an agent is searching for a specific item (e.g., a camera) by traveling to different online shopping malls in order to find the lowest price for this item and return the result of the search to the original site. To facilitate the presentation, we consider a system with two malls which have two shops and one shop respectively. Note that the evaluation in Chapter 6 will be carried out on a larger system.

4.1.2 Application of our Approach

Step 1: Construction of the HLM

Our approach starts with describing the desired behavior of the system using SMDs described in Chapter 3. Figure 4.1 on page 46 shows the SMD of a scenario in the shopping application. We begin by setting up the scenario, creating two Malls and a client. The first one is Mall1 created at Mall1site with two shops (Shop1 and Shop2) and the second mall is Mall2 created at Mall2site that has one shop (Shop3). The
client creates an agent that is responsible for finding the lowest price. In this scenario, the agent is sent to Mall1 and then it goes to Mall2. In each mall, the agent queries the mall’s information kiosk for a list of shops in the mall. Then, it asks each shop for the price of the camera and updates the current best price, if necessary. After the agent has finished visiting all malls, it sends the best price and the corresponding shop back to the client.

![Diagram](image)

**Figure 4.1:** The SMD of overall online shopping system

In order to clarify the model further, we divide the HLM presented in Figure 4.1 into HLMs for the client, agent, malls and the “main” (i.e., a component that sets up
CHAPTER 4. CASE STUDIES

the entire scenario). We will only show the HLMs for the main, the client, and the agent. The HLM for the main is shown in Figure 4.2 on page 47. The initialization consists of three steps: creating the client, creating the malls by specifying their sites (in this example we have two malls: Mall1 and Mall2 created in sites Mall1site and Mall2site respectively) with different shops and by providing the addresses of these malls to the client to start the search process.

![Figure 4.2: The SMD of the initialization process](image)

Once the client gets a mall list from the main, it creates an agent at Mall1site, provides it with the addresses of the other malls, and waits for the result to be returned from the agent. The agent waits for a specific amount of time to get a response from a shop. Figure 4.3 on page 48 shows the HLM of the client.

After the agent has been created at Mall1site, it sends a message to the Mall1 asking it for the shop list (i.e., the list of shops in this mall). After that, it starts sending messages to the shops one by one and in a specific order asking and waiting for their prices of the camera. The agent has a specific amount of time to get a result from the shop whether it has the product camera or not. If that amount of time is elapsed the agent moves to the next shop. Then, the agent moves to the second mall
located at Mall2site and does the same thing. Once the agent has queried all shops and visited all malls, it sends a message to the client telling him the minimum price and in which shop and mall the cheapest camera can be found. Figure 4.4 shows the HLM of the agent. Appendix A shows all the HLMs of the online shopping example.

**Step 2: Construction of KM from HLM**

In the second step of our approach, the HLMs are translated into kiltera models (KMs); kiltera’s direct support for many relevant features (e.g., support for concurrency with (a)synchronous message passing, movement of processes, time, and site-dependent behavior) makes this translation relatively straight-forward. Figure 4.5 on page 50 shows the Main KM of the corresponding HLM presented in Figure 4.2 on page 47. The three sites are declared in line 2 and the channels to the client and the malls’s information kiosk are introduced in line 3. The statements in lines 5-8 are executed in parallel: create an instance of Client with one channel (to_cust) at site Home (line 5), create an instance of Mall1 with channel mall1_info at site Mall1site (line 6), create an instance of Mall2 with channel mall2_info at site Mall2site (line 7), and send the
mall sites and their channels to the client through the channel to_cust (line 8).

Figure 4.6 on page 50 shows the Client KM of the corresponding HLM presented in
Figure 4.3 on page 48. In this figure, after the Client has received all mall addresses
(line 2), it starts its process by creating the Agent at site mallsite (line 5). The
Agent is connected to channels from_agent (to send back the result) and mall_info
(the link to the information desk where the Agent is located). Variable mallsite
is bound to the first mall in the list of malls received from Main. Furthermore, the
client sends the agent the remaining mall addresses rest (line 2). Then, the Client
waits for the result from the Agent (line 6). The result is the price, and a link to the
module Main:
sites Home, Mall1site, Mall2site in
dchannel to_cust, mall1_info, mall2_info in
par
move Client [to_cust] to Home
move Mall1 [mall1_info] to Mall1site
move Mall2 [mall2_info] to Mall2site
trigger to_cust with [(Mall1site, mall1_info), (Mall2site, mall2_info)].

Figure 4.5: The KM of the initialization process of the shopping example

module Client [from_main]:
when from_main with [(mallsite, mall_info); rest] ->
dchannel from_agent in
par
move Agent [from_agent, mall_info] (rest, null, null) to mallsite
when from_agent with (price, shop_ch) ->
// print where the Agent found the min price

Figure 4.6: The KM of the client

Figure 4.7 on page 51 shows the agent KM of the corresponding HLM presented in Figure 4.4 on page 49. The Agent consists of four subprocesses: GetShopList, QueryShops, EnterShop and GoToNextMall. When an agent arrives at a mall it executes the process GetShopList (line 38). This process sends a message to the mall information booth asking for the shop list. When it receives the shop list (line 6), it starts querying the shops (invoking process QueryShops in line 7) one by one searching for the camera and updating the current lowest price if necessary. This is done by invoking EnterShop (lines 19-29) which asks the shop if it has the camera or not; if this is the case and the shop sends a price less than the bestprice that we have so far, then the new best price is updated (line 25); otherwise, the bestprice is left unchanged (line 27). If the Agent did not get a response from the shop within some time t (the time the agent is allowed to stay in each shop), then the Agent continues with the next shop. After the Agent exits a shop it invokes recursively QueryShops.
module Agent [to_customer , to_mall] ( malllist , bestprice , best_shop):

process GetShopList [ mall_info ]:

channel response_ch in
par
trigger mall_info with ("get shop list ", response_ch)
when response_ch with shop_list ->
QueryShops [ shop_list ] ( bestprice , best_shop )

process QueryShops [ shop_list ] ( bestpricehere , best_shop_here ) :

match shop_list with
[ ] ->
| GoToNextMall [] ( bestpricehere , best_shop_here )
| [ shop_ch ; rest ] ->
| event shop_visited in
| EnterShop [ shop_ch , shop_visited ] ( bestpricehere , best_shop_here )
| when shop_visited with ( newestprice , new_best_shop ) ->
| QueryShops [ rest ] ( newestprice , best_shop )

process EnterShop [ shop_ch , shop_visited ] ( bestprice , best_shop_here ) :

channel response_ch in
par
trigger shop_ch with ("camera" , response_ch)
when response_ch with price ->
if price < bestprice then
trigger shop_visited with ( price , shop_ch )
else
trigger shop_visited with ( bestprice , best_shop_here )
timeout t ->
trigger shop_visited with ( bestprice , best_shop_here )

process GoToNextMall [] ( best_price , best_shop ) :

match mall_list with
[ ] ->
trigger to_customer with ( bestprice , best_shop )
| [ ( mallsite , mall_info ) ; rest ] ->
move Agent [ to_customer , mallinfo ] ( rest , bestprice , best_shop ) to mallsite
in
GetShopList [ to_mall ]

Figure 4.7: The Agent KM
(line 17) with the remainder of the shop list. Once the Agent has finished the shop list (which means the shop list is empty in line 11), then it executes the GoToNextMall process which checks whether there are more malls to visit or not. If there are more malls (line 35), the Agent travels to the next mall by creating an instance of Agent at that mall with an updated state (line 36) and behaves in the same way. If the Agent has visited all malls (malllist is empty, line 33), it sends the result back to the Client telling him the lowest price and where the cheapest camera was found (line 34).

As we can see from example in Figure 4.7 on page 51, the move statement sends an instance of the process Agent in the remote site mallsite and any state information is provided as parameters to the Agent. Furthermore, while the Agent is trying to find the best price of the product camera, it uses a timeout in the shop because if the shop does not respond (e.g., because it is down), then the Agent should not wait forever. Appendix A shows all the KMs of the online shopping example.

**Step 3: Instrumentation of IUT and KM using HLM**

After we have constructed the HLMs and translated them into KMs of the shopping example, we start the instrumentation step. Before we talk about the instrumentation we have to mention here that we use the Aglets platform [49, 3] in order to implement the online shopping application illustrated in Figure 4.1 on page 46. In the instrumentation step, we instrument both the IUT (implemented in Aglets) and the KM to allow relevant information to flow from the IUT to the KM. We instrument the IUT first because it is going to send information to the KM. We use the HLM presented in Section 4.1.2 to start the instrumentation step. First we identify
a collection of “check points” at which conformance between the (IUT) and the KM is to be checked. Typically, check points involve the sending or receipt of messages or agent movement. The number and locations of checkpoints are determined by the constraints to be enforced. More precisely, the users must ensure that the information required for enforcing the constraints is communicated to the KM at the appropriate time. Figure 4.8 shows the HLM with 5 check points numbered from 1 to 5 presented as a circle.

Figure 4.8: Check points locations in camera searcher scenario SMD

Once we have decided on the checkpoints in the HLM we locate them in the IUT and the KM. Figure 4.9 on page 54 shows the location of check point 5 in the IUT
(the agent moved to a mall site) and Figure 4.10 on page 54 shows the corresponding location of check point 5 in the KM.

```java
public void onArrival(MobilityEvent me){
    /* Check point 5 */
    ...}
```

Figure 4.9: Location of check point 5 in the IUT

```java
// Check point 5
move Agent[to_customer,mall_info](rest,bestprice,best_shop) to mallsite
```

Figure 4.10: Location of check point 5 in the KM

Next, at each check point in the IUT, we insert appropriate instrumentation code which transmits relevant state information to the KM via sockets through the Python connector. In addition, at each check point in KM, we insert instrumentation code that receives state information from the IUT and compares it with the expected internal information; if they are different, then non-conformance is signalled; otherwise, the KM continues. The comparison may require the addition of a specific routine that implements the constraint that is to be enforced at that checkpoint. Constraints may restrict individual messages or entire sequences of messages. For instance, suppose each shop in a mall is to be queried exactly once. The instrumentation needs to ensure that the IUT sends the name of the shop about to be queried to the KM while the KM keeps track of the shops queried so far. Figure 4.11 and Figure 4.12 show the instrumentation code inserted in IUT and in KM respectively.
public void onArrival(MobilityEvent me) {
    try {
        // Check point 5
        // Inserted code begin
        Socket s = new Socket("localhost", 60002);
        PrintStream P = new PrintStream(s.getOutputStream());
        P.println("Agent traveled to next mall");
        // inserted code end
    }...

    //Check point 5
    // Inserted code begin
    HandleJavaMessages[external_ch]
    when external_ch with data->
    if data = "Agent traveled to next mall" then
    move Agent[to_customer,mall_info](rest,bestprice,best_shop) to mallsite.
    else
    //Error, stop execution with non-conformance
    // inserted code end

Step 4: Execution of IUT and KM

In the last step of our approach we execute the instrumented shopping example and the instrumented KM and check the conformance between them. The following Sub-section 4.1.3 lists examples of some relevant properties and some observations are presented in Section 4.1.4. A full evaluation of the approach to determine its ability to detect faults in the implementation of the online shopping example will be given in Chapter 6.

4.1.3 Checking sample properties

To implement a particular conformance check in the KM, it is important that all necessary information is sent from the IUT. For instance, to check that an agent in
the IUT has moved properly to a new node it suffices for the IUT to send a string
to the KM describing the target of the movement. However, to check more complex
behavioral constraints (e.g., that a sequence of actions has been performed in the
right order), the KM may have to maintain and query additional state information.
Finally, to check that timing constraints have been met, timing information needs
to be communicated and timeouts may have to be used. Examples for each of these
kinds of conformance checks are given below.

**Property 1: Agent visits next mall at expected time**

At the checkpoint shown in Figure 4.12 on page 55, the KM expects a message from
the IUT containing the string "Agent traveled to next mall" (lines 3-5). If such
a string is received, the KM continues its execution by creating an instance of Agent
and moves it to the next mall site mall_site (lines 6-7). If any other kind of message
is received (because, for instance, the number of malls or the number of shops in a
mall in the IUT and the KM do not match, or the IUT did not query all shops in
a mall) the behavior of the IUT is non-conformant (line 9). In that case, the KM
outputs an error message and terminates.

**Property 2: Agent queries each shop in a mall exactly once**

Suppose each shop in a mall is to be queried exactly once. The instrumentation needs
to ensure that the IUT sends the name of the shop about to be queried to the KM,
while the KM keeps track of the shops queried so far. In Figure 4.13 on page 57, after
receiving a message in line 14, the KM checks whether the IUT has already queried this
shop or not by checking the list Jshoplist (using the process Search in line 17). If
the shop name received from the IUT (Jshopname) is already contained in Jshoplist (Search sends a message ‘‘Found’’ to the listener in line 18), non-conformance is signalled (line 20); otherwise, the shop is added to Jshoplist (line 22) and execution continues.

```plaintext
1 process Search(answer_ch)(list,item):
2   match list with
3       [] ->
4       [hd;rest] ->
5           if hd = item then
6             trigger answer_ch with ‘‘Found’’.
7           else
8             Search(answer_ch)(rest,item)
9       end
10 end
```

Figure 4.13: Checking Property 2 in KM

Property 3: Timing constraints

Figure 4.14 on page 58 shows the KM that is used to check time conformance. More precisely, we check that shops respond to queries in a timely fashion (within $t$ seconds) and that the agent reacts appropriately to delayed shop responses. The instrumentation needs to ensure that the agent in the IUT informs the KM how long it had to wait in response to a query. The agent in the KM will receive this time in responsetime in line 5. If responsetime is greater than $t$ (line 6), non-conformance is reported.
Also, if the KM agent does not receive a response from the shop in the KM within \( t \) seconds (line 13), but a proper response has been received from the shop in the IUT (line 20), we have non-conformance.

```java
trigger shop_ch with ("camera", response_ch).
when response_ch with (shopprice, ...) ->
par
HandleJavaMessages[external_ch]
when external_ch with response_ch, response_ch ->
if (float(response_ch) <= t) then
  if (shopprice < minprice) then
    // New min price, visit next shop with this price
  else
    // check next shop with the current min price
  end
else
  // Time non-conformance, stop executing
  timeout t ->
par
HandleJavaMessages[external_ch]
when external_ch with data ->
if (data = "Not Found") or (data = "No Respond") then
  trigger next_ch with ("Get Next Shop", ...).
else
  // non-conformance, IUT find the camera but KM did not
```

Figure 4.14: Checking Property 3 in KM

### 4.1.4 Observations

Table 4.1 compares the size of the IUT and the KM before and after the instrumentation.

Table 4.1: Size in lines of code of the IUT and KM before and after the instrumentation

<table>
<thead>
<tr>
<th>File Name</th>
<th>IUT</th>
<th>KM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>Client</td>
<td>57</td>
<td>82</td>
</tr>
<tr>
<td>Agent</td>
<td>212</td>
<td>262</td>
</tr>
<tr>
<td>Mall1 with 3 shops</td>
<td>186</td>
<td>186</td>
</tr>
<tr>
<td>Mall2 with 2 shops</td>
<td>131</td>
<td>131</td>
</tr>
</tbody>
</table>
We make the following observations:

1. The instrumentation increases the size of the agent the most. This is because the agent carries out most of the behavior and contains the most check points. Instrumentation to the other files is much smaller. Neither the Main nor any of the malls require any instrumentation.

2. Both, before and after the instrumentation, the size of the kiltera code is considerably smaller than that of the corresponding Aglets code. We view this as an indication that kiltera does indeed allow a succinct expression of mobile and distributed computation.

The approach allowed us to find seeded faults in the implementation. A more evaluation of the ability of the approach to detect faults will be given in Chapter 6.

4.2 Case Study: Contract Signing

4.2.1 General Description

The description of this example has been taken from [13]. In this example, several distributed partners want to sign a contract for a specific project. A coordinator called notary guides the process of signing the contract by all partners. The notary first creates the partners and the contract. After that, the notary is responsible for sending the addresses of the partners to the contract. Moreover, it is responsible for assigning the partners to the contract for signing it. There is only one contract to avoid inconsistency. The partners can view and sign the contract. Moreover, each partner has a specific amount of time to sign the contract. If the time has elapsed,
the notary moves the contract to the next partner. The contract is always moved by
the notary via a remote reference from one partner to the other in order to be signed.
Finally, the notary should check whether all the partners have signed the contract
or not. A partner can only sign the contract when they are at same place. In this
example, we will consider a system with two partners just to make it simple.

4.2.2 Application of our Approach

Step 1: Construction of the HLM

We start our approach by describing the SMD of the contract signing example.
Figure 4.15 shows the SMD scenario of signing a contract. The scenario starts
when the Notary creates the Contract and the two partners. The contract is cre-
ated at Home site (the same place of Notary) and Partner1, Partner2 are cre-
at ed at Partner1Site, Partner2Site respectively. In the scenario, the Notary
asks Partner1 to sign the Contract. Since Partner1 does not have the contract,
Partner1 replied “I do not have it”. So the Notary asks the Contract to move to
Partner1Site. After that, Partner1 signs the contract. Then, the Contract up-
dates his list that contains the partners who signed the contract. The Notary then
does the same thing with Partner2. Once the Contract visits all the partners, it
goes back to the Notary. The Notary asks the Contract to check whether all the
partners have signed the contract or not. If the result is yes, then Notary will print
“All partners have signed the contract”. Otherwise, “not Ok” will be printed.

We have divided the HLM presented in Figure 4.15 on page 61 into HLMs for
the notary, the contract and the partners in order to clarify the model. We will
show the HLMs for the notary, contract, and one partner. The HLM of the notary is
Figure 4.15: The SMD of the overall contract signing example

shown in Figure 4.16. The notary is responsible for creating the contract and the two partners. Moreover, it is responsible for providing the addresses of these partners to the contract. After that, the notary sends a message to Partner1 to sign the contract. Since the contract is still at site Home and Partner1 at site Partner1Site, Partner1 cannot sign the contract and so he sends a reply message to the notary telling him that he does not have the contract to sign it. Once the Notary receives this message, he asks the Contract to move to the site of Partner1. After that, the notary sends a message to Partner2 asking him to sign the contract. And because the contract
Figure 4.16: The SMD of the notary

is still at site Partner1Site he asks the contract to move to the site of Partner2. Once the Contract visits all the partners, the Notary asks the Contract to go back to the Home site. Once the Contract arrives at the Home site, the Notary sends a message to the Contract in order to check whether all partners have the signed the contract or not. If yes, the Notary prints “All partners signed”. Otherwise, it prints “not Ok”.

The Contract moves from one site to another in order to be signed. Moreover, it updates the list of partners who signed the contract and checks whether all partners have signed the contract or not when it received a check request from Notary.
Figure 4.17 shows the SMD of the contract.

![SMD of the contract](image)

Figure 4.17: The SMD of the contract

The partner is responsible for signing the contract once they are located at the same site. Figure 4.18 shows the SMD of Partner1.

**Step 2: Construction of KM from HLM**

The HLMs are translated into kiltera models (KMs). Figure 4.19 on page 65 shows the Notary KM of the corresponding HLM presented in Figure 4.16 on page 62. The Notary module identified that has one port notary\_ch used to communicates with the contract and the partners (line 1). Moreover, other channels are introduced in line 4. The three sites are declared in line 2. Line 6 used to create an instance of Partner1 at Partner1Site with three channels: to\_partner1 used to send messages from the Notary to Partner1, from\_P1\_to\_Cont used for the communication between
Partner1 and the Contract and the third channel to_notary used to send messages from the Partner1 to the Notary. Line 7 used to create an instance of Partner2 with three channels: to_partner2, from_P2_to_Cont and to_notary. These channels used to communicate with the Partner2, Contract and Notary with the same functionality as used in Partner1. Line 8 creates an instance of Contract with one channel (notary_ch) used to send messages from the Notary to the Contract. After creating all these instances, the Notary sends the addresses of all partners to the contract using the channel notary_ch (lines 10-12). The Notary then sends a message to Partner1 to sign the contract (line 14). At the same time the Notary starts waiting for a reply from Partner1 (line 15) or waiting for the t seconds to elapse in order to ask Partner2 to sign the contract (line 19). If Partner1 does not have the contract to sign it, the Notary will receive a message from Partner1 that he does not signed
CHAPTER 4. CASE STUDIES

module Notary [notary_ch] : 
sites Home, Partner1Site, Partner2Site

process Initial [] :
  dchannel to_partner1, to_partner2, from_P1_to_Cont, from_P2_to_Cont in
  par
  move Partner1 [to_partner1, from_P1_to_Cont, notary_ch] to Partner1Site
  move Partner2 [to_partner2, from_P2_to_Cont, notary_ch] to Partner2Site
  move Contract [notary_ch] to Home
  seq
    trigger notary_ch with ('partners addresses', []).
    [Partner1Site, from_P1_to_Cont, 'Partner1'],
    [Partner2Site, from_P2_to_Cont, 'Partner2']).

  par
    trigger to_partner1 with ('Sign Contract', 'Home').
    when notary_ch with ('Partner1 not signed', response_ch) ->
      par
        trigger response_ch with ('Sign Contract', 'Partner1Site').
        move Contract [notary_ch] to Partner1Site
        wait t ->
      par
        trigger to_partner2 with ('Sign Contract', 'Partner1Site').
        when notary_ch with ('Partner2 not signed', response_ch) ->
          par
            trigger response_ch with ('Sign Contract', 'Partner2Site').
            move Contract [notary_ch] to Partner2Site
            wait 2* t ->
          par
            trigger notary_ch with ('check list', ['Partner1', 'Partner2']).
            move Contract [notary_ch] to Home
          in
  Initial []

Figure 4.19: The KM of the notary

the contract (line 15). The message also contains a channel response_ch used to send a reply from Notary to Partner1 asking him again to sign the contract (line 17) after the Notary moves the Contract to the Partner1Site (line 18). Partner1 has t seconds to sign the contract (line 19). If the time has elapsed, the Notary asks Partner2 to sign the Contract. Then the Notary does the same thing (as he did with Partner1) with Partner2 (lines 20-25). Again, Partner2 has also the same time as Partner1 to sign the contract. So the Notary waits another t seconds (t+t= 2*t) to let Partner2 to sign the contract (line 26). If the time (2*t) has elapsed, the Notary asks the Contract to go back to site Home (line 29) and asks the Contract to check whether all partners have signed the contract or not (line 28).
Figure 4.20 on page 66 shows the Contract KM of the corresponding HLM presented in Figure 4.17 on page 63. The Contract KM consists of two processes:

```plaintext
1 module Contract[from_No];
2 function Enqueue(queue, item):
3  match queue with
4    [] ->
5    [item] | [hd;tl] ->
6     [hd;Enqueue(tl, item)]
7
8 process HaveAllSign[](partnersigned):
9  when from_No with ("check list ", list)->
10    if list = partnersigned then
11      print "All partners are signed the contract'.
12    else
13      print "not Ok'.
14
15 process Sign[](partnersigned, partnerlist):
16  match partnerlist with
17     [] ->
18     HaveAllSign[](partnersigned)
19    | [(psite, plink, pname); remainder] ->
20     when plink with ("Sign Contract ", recpname)->
21       if pname = recpname then
22         let partnersigned = Enqueue(partnersigned, recpname)
23         par
24         print pname "'; i signed the contract'.
25     in
26     in
27     when from_No with ("partners addresses ", partnersigned, partnerlist) ->
28     Sign[](partnersigned, partnerlist)
```

Figure 4.20: The KM of the contract

Sign and HaveAllSign. Moreover, it has one function Enqueue (lines 2-7) used to add the partners who have signed the contract to the list partnersigned. Process HaveAllSign (lines 8-13) used to check whether all partners are signed the contract or not by comparing the list which contains all partners with partnersigned list which contains all signed partners (line 10). If these two lists are equal, then statement at line 11 will be executed. Otherwise, statement at line 13 will be executed. The execution of the Contract starts when it receives the partners addresses from the Notary through the channel from_No (line 27). Once the Contract received these addresses, it starts listening to the partners for signing the contract (line 28). The
Sign process has two parameters, the list partnersigned, and the list partnerlist. The partnerlist is a list that contains the partners who have signed the Contract. More precisely, this list contains the partner site (psite), the channel used to connect a partner with the contract (plink) and the partner name for each partner (pname).

If partnerlist is empty (line 16) it means that all partners had the chance to sign the contract, but it does not mean that all partners have signed the contract. But if partnerlist still have some partners to sign the contract, then the Contract waits for a request from the current partner in the list to sign it through the connection channel between them which is plink (line 19). Line 20 is used to make sure that the partner who makes the request to sign the contract is the same partner who is on the top of the list. The partner who signed the contract will be added to partnersigned (line 21) and the Contract prints a statement showing that the current partner has signed the Contract (line 24). Line 25 makes the Contract listen to the requests of other partners if they exist.

Figure 4.21 on page 67 shows the Partner1 KM of the corresponding HLM presented in Figure 4.18 on page 64. Partner1 starts its process when he receives a

```plaintext
module Partner1 [from_notary, from_P1_to_Con, to_notary];
process SignContract [] (contractaddress);
  let pladdress = 'Partner1Site'
  in
  channel response_ch in
  if pladdress = contractaddress then
    trigger from_P1_to_Con with ('Sign Contract', 'Partner1').
  else
    par
      trigger to_notary with ('Partner1 not signed', response_ch).
      print 'Partner1: do not have the contract to sign'.
      when response_ch with ('Sign Contract', contractaddress) ->
        SignContract [] (contractaddress)
  when from_notary with ('Sign Contract', contractaddress) ->
    SignContract [] (contractaddress)
```

Figure 4.21: The KM of the Partner1
request from the Notary in order to sign the contract (line 16). Partner1 receives the request using from_notary channel with the current location of the contract (contractaddress). The partner signs the contract if they are in the same location (lines 6-7). If not, the partner sends a reply message to the Notary with a response_ch channel telling him that he does not signed the contract since it is not available (line 10). Moreover, the partner keeps waiting until he receives a response from the notary (through the channel response_ch) that the contract now is in his location. Partner1 again tries to make sure that the new location of the Contract is the same location of Partner1 by calling SignContract process again.

**Step 3: Instrumentation of IUT and KM using HLM**

Step three of our approach is to instrument both the IUT (implemented in Aglets) and KM of the contract example. As we explained earlier, first we identify a collection of “check points” at which conformance between the (IUT) and the KM is to be checked. Figure 4.22 on page 69 shows the HLM with 11 check points numbered from 1 to 11 presented as a circle.

Once we have decided on the checkpoints in the HLM we locate them in the IUT and the KM. Figure 4.23 on page 70 shows the location of check point 3 in the IUT. This check point is used to check whether the partner has the contract or not in order to sign it. Figure 4.24 on page 70 shows the corresponding location of check point 3 in the KM.

Next, we instrument the IUT by inserting code to transmit the relevant state information to the KM. Moreover, the KM is also instrumented by inserting instrumentation code allowing it to receive the state information from IUT and compare it
CHAPTER 4. CASE STUDIES

69

Figure 4.22: Check points locations in SMD of signing a contract scenario

with the expected internal information; if they are different, then non-conformance is signalled; otherwise, the KM continues. Figure 4.25 on page 71 and Figure 4.26 on page 71 show the instrumentation code inserted in IUT and in KM respectively.

Step 4: Execution of IUT and KM

In the last step of our approach we execute the instrumented signing contract example and the instrumented KM and check the conformance between them. The following Subsection 4.2.3 represents sample properties and some observations are presented in Subsection 4.2.4. A full evaluation of the approach to determine its ability to
CHAPTER 4. CASE STUDIES

4.2.3 Checking Sample Properties

We applied the four steps of our approach into the contract example. The following will check the conformance of sample properties of the signing contract application:

Property 1: Time to sign the contract

Each partner in the partner list has \( t \) seconds to sign the contract. When the notary moves the contract to the partner he asks the partner to sign the contract. If the partner does not sign the contract within the \( t \) seconds, the KM of the contract assumes that the partner is not available and so did not sign the contract. Figure 4.27 on page 72 shows the KM that is used to check time conformance. The instrumentation needs to ensure that the contract in the IUT informs the KM how long it had to wait in order to be signed by the partner (line 4 using \texttt{signtime} variable). The contract
CHAPTER 4. CASE STUDIES

```java
if (Partner1Reply.equals("Partner1 not signed")){
  /* Check point 3 */
  /* Inserted code begin */
  Socket s = new Socket("localhost", 60003);
  PrintStream P = new PrintStream(s.getOutputStream());
  P.println("Partner1 do not have the contract");
  /* inserted code end */
  Partner1Proxy.sendMessage(new Message("Sign Contract", "atp://localhost:2222"));
  ContractProxy.dispatch(new URL("atp://localhost:2222"));
}
```

Figure 4.25: Code inserted into IUT at check point 3

```java
when notary_ch with ("Partner1 not signed", response_ch) ->
  /* Check point 3 */
  /* Inserted code begin */
  par
  HandleJavaMessages[external_ch]
  when external_ch with data ->
  if data = "Partner1 do not have the contract" then
    par
    trigger response_ch with ("Sign Contract", "Partner1Site").
  else
    move Contract [notary_ch] to Partner1Site
  /* inserted code end */

Figure 4.26: Code inserted into KM at check point 3
```

in the KM will receive this time. If it is greater than \( t \) (line 5), non-conformance is reported (line 6). Otherwise, the the partner is added to partnersigned.

**Property 2: Contract movement**

After creating partners and the contract, the notary sends the addresses of all partners to the contract. So the process starts when the notary sends a message to partner to sign the contract. When the partner receives the message, it sends a reply message telling the notary that he does not have the contract. In this case the notary should moves the contract to that partner. Now suppose the notary in the IUT moved the contract into a wrong address. This case is very dangerous because the contact could be viewed or worst be updated or deleted by someone who should not even take a
look to the contract. To check this property the instrumentation needs to ensure that the IUT sends the current address of the contract to the KM, while the KM keeps track of the location of the contract. Figure 4.28 on page 72 shows the KM used to check contract movement. When the contract KM receives the current address of the contract in the IUT (line 4), the contract KM compares it with current location of the contract in the KM (line 5). If they do not match, then non-conformance is signalled and the KM outputs an error message and terminates (line 8).

Property 3: All Partners have signed the contract

Once the contract has visited all the partners, it returns to the notary. The notary then asks the contract to check whether all the partners have signed the contract or not. Once the KM of the contract has visited all the partners, it is expects to receive
a message from the implementation of the notary to check whether all the partner have signed the contract or not. So if the KM of the contract receives something else (e.g., move to another partner) then a non-conformance will signalled and stop the execution of the KM. Figure 4.29 on page 73 shows the KM used to check whether the partners in the IUT signed all the contract or not.

```java
if list = partnersigned then
    par
    HandleJavaMessages[external_ch]
    when external_ch with data->
    if data = "All Signed" then
        print "all partners are signed the contract".
    else
        // Error, non-conformance, stop executing
    else
        print "not ok".
```

Figure 4.29: Checking Property 3 in KM

### 4.2.4 Observations

Table 4.2 compares the size of the IUT and the KM before and after the instrumentation of the signing contract example.

Table 4.2: Size in lines of code of the IUT and KM before and after the instrumentation of the contract example

<table>
<thead>
<tr>
<th>File Name</th>
<th>IUT</th>
<th>KM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notary</td>
<td>87</td>
<td>180</td>
</tr>
<tr>
<td>Contract</td>
<td>132</td>
<td>201</td>
</tr>
<tr>
<td>Partner1</td>
<td>71</td>
<td>71</td>
</tr>
<tr>
<td>Partner2</td>
<td>71</td>
<td>71</td>
</tr>
</tbody>
</table>

Based on the Table 4.2, we make the following observations:
1. The instrumentation increases the size of the notary the most. This is because 
the notary carries out most of the behavior and contains the most check points. 
Instrumentation to the other files is much smaller. Neither the Partner1 nor 
Partner2 require any instrumentation.

2. Both, before and after the instrumentation, the size of the kiltera code is con-
siderably smaller than that of the corresponding Aglets code. We view this as 
an indication that kiltera does indeed allow a succinct expression of mobile and 
distributed computation.

The approach allowed us to find seeded faults in the implementation. A more eval-
uation of the ability of the approach to detect faults will be given in Chapter 6.

4.3 Other Examples

We have also applied our approach to two other examples: a token ring distributed 
mutex algorithm, and Lamport’s distributed mutex algorithm. The Java implemen-
tations of these two algorithms was taken from [35].

4.3.1 Token Ring Algorithm

To achieve the mutual exclusion in a distributed system, there are different algorithms. 
The token ring algorithm is one of them. In this algorithm, a set of processes are 
connected together in a ring. Each process is assigned a position in this ring. The ring 
positions may be allocated in numerical order of network addresses or according to 
some other criteria. Process 0 has the token when the ring is initialized. The token 
circulates around the ring. It is passed from process n to process (n +1) mod N,
where $N$ is the number of processes in the ring. When a process gets the token from its neighbor, it checks whether it needs to enter the critical section (CS) or not. If yes, the process does its work and then leaves the CS. After that, the process releases the token to the next process in the ring. If the process is not interested in entering the CS, it just passes the token to the next process. Figure 4.30 shows the KM of token ring algorithm. In Figure 4.30 we have three processes A, B, and C who are connected in a ring and waiting for the token in order to enter the CS using the token ring algorithm. In this example, the token is initially given to process A (line 17). Once process A has received the token (line 3), it enters the CS and then it stays in the CS for a specific amount of time (line 5) to do its tasks. After that, process A releases the token by sending it to the next process in the ring (line 7).

We have applied the four steps of our approach to check the conformance of the IUT with respect to the KM at runtime. Our approach was able to detect some faults that we have seeded in the implementation. These faults included:

1. a process is still in the CS although it has released the token.
2. a process sends a message to the next process that it released the token even though it does not have the token.

3. a process entered a CS although it does not have the token.

Figure 4.31 shows the KM used to check if the process has the token before it enters the CS.

```java
HandleJavaMessages[external_ch]
when external_ch with "AEnteredCS" -> //msg received from out that process A in CS
if holdingtoken = "A" then // check whether A has the token or not
  // Enter CS
else
  print "Error, A Entered the CS even it does not have the token".
```

Figure 4.31: Checking entering the CS in token ring algorithm using KM

The time it takes to find the faults depends on the location of the fault. However, most of the conformance checks described above completed in less than a minute.

### 4.3.2 Lamport Mutex Algorithm

Lamport’s algorithm is another way to achieve the mutual exclusion in distributed systems. In Lamports algorithm each process has a queue that is used to sort the critical section requests. Moreover, each process has a logical clock that is used for timestamps. The idea of the algorithm is to make sure that processes enter the critical section based on the order of timestamps of their request. When a process wants to enter a critical section, it sends a timestamp request message to all other processes. Moreover, it adds its timestamp request to the queue. On the other side, once a process receives a request message from another process, it stores this request with the timestamp in the queue. In addition, a timestamp acknowledgment is sent back. A process can enter a critical section only if all of the three conditions are met:
1. It has a request in the queue with timestamp t,
2. t is less than all other requests in the queue, and
3. It has received an acknowledgment message from every other process with timestamp greater than t.

A full *kiltera* model of the Lamport’s mutex algorithm with instrumentation is presented in Appendix B.

Our approach was able to detect seeded faults in the implementations. Again, the time it take to find the faults depends on the location of the fault. However, most of the conformance checks described above completed in less than a minute. For instance, we modified Lamport’s algorithm allowing a process to send a “release” message although it did not hold a token and was not in the critical section. Figure 4.32 shows the instrumentation in KM used to check this property. Interestingly,

```plaintext
HandleJavaMessages[external_ch]
when external_ch with "ReleaseC"->
  par
  trigger holding_token with "C". // check if process C holding the token
  when answer with true -> // if yes
    par
      ReceiveRelease[rel_c]("A")
      ReceiveRelease[rel_c]("B")
      trigger all lasting rel_c. // send release msg to all other processes A, B, C
        print "'C is Not in CS'".
      Getfromqueue[] // get the process who is on the top of Queue to enter CS
      | answer with false-> // if C not holding the token
        print "'Error, C is not in CS to Release the token'". // no-conformance
```

Figure 4.32: Checking releasing the CS in Lamport algorithm using KM

using our conformance testing approach we found that this modification caused the Java implementation to examine “intermittent” (or temporary) inconsistency which would disappear after continued execution. The example demonstrates the potential
intricacies of distributed systems and shows that a system may recover from failure “on its own” and that, therefore, partial examination of traces may not always be sufficient.

4.4 Chapter Summary

In this chapter, we have described the application of our conformance testing approach to four different sample systems. The approach allowed us to find seeded faults in the implementation. A more comprehensive evaluation of the ability of the approach to detect faults will be given in Chapter 6 and will be based on mutation testing.
Chapter 5

Mobile Agent Mutation Operators

We have applied our approach to several examples and found that it is able to detect seeded non conformance in all implementations under test. However, to check the effectiveness and the efficiency of our approach, we have developed an automatic mutation-based evaluation framework (Chapter 6). We are interested in mobile agent systems and since no mutation operators are available in the literature for mobile agent system we need to develop them. So first, a set of new mutation operators for mobile agent systems are defined based on mobility, sending and receiving messages, time, etc. Our operators are geared towards the Aglets framework. We categorize these mutation operators into six different categories: mobility, communication, the agent’s run method, agent creation, event listeners, and finally the agent proxy.

The chapter is organized as follows. Following a general motivation in Section 5.1, a new set of mutation operators for distributed and mobile agent systems are provided in Section 5.2. Finally, Section 5.3 summarizes the chapter.
5.1 Motivation

Mobility occurs naturally in many distributed system applications such as telecommunications and electronic commerce. Moreover, mobility may reduce bandwidth consumption and coupling and increase flexibility [70]. However, it seems that relatively little work has been done to support quality assurance techniques such as testing and verification of mobile systems [26, 89]. Different bugs could happen in mobile and distributed systems such as loss or death of an agent, missing communication with some host, errors in handling messages, loss of data when an agent moves to another host, incorrect deactivation of agents (i.e., delay the movement of an agent for some specific time), incorrect use of the event listener methods, etc.

In Chapter 3, we described our approach for checking conformance of mobile agent at runtime. However, to check the efficiency and the effectiveness of the approach we used mutation testing. Mutation is used to generate a set of mutants from the system under test using mutation operators. Several mutation operators have been proposed in the literature to generate a set of mutants for several types of programs. For example, in [14] a new experimental mutation analysis framework has been proposed called ExMAn. The ExMAn framework uses 25 mutation operators to test concurrent Java programs. The MuJava tool [54] uses two types of mutation operators for Java programs: one for method level operators [65] (29 mutation operators) and the second for class level operators [53] (29 mutation operators). Mothra [27, 46] is another mutation tool that uses 22 mutation operators for Fortran-77 programs.

In this chapter we present a set of mutation operators for mobile agent systems. These mutation operators are used in Chapter 6 to automatically generate the mutants, run these mutants in parallel with a kilter model, and finally to automatically
CHAPTER 5. MOBILE AGENT MUTATION OPERATORS

collect the results of the conformance tests.

5.2 Mutation Operators for Mobile Agent Systems

In this section we propose six categories of mutation operators for Aglets mobile agent systems. These categories contain mutation operators that impact the following aspects of a mobile agent system implemented using Aglets: mobility, communication, the agent’s run method, agent creation, event listeners, and the agent proxy. Table 5.1 provides a complete list of mutation operators. These operators are presented in this section. The purpose of these operators is to insert some bugs into the system under test that the programmers may make when they implement mobile agent system using Aglets.

5.2.1 Mobility

In Aglets the dispatch statement is used to migrate an agent from a host (context) to another. It has only one parameter used to specify the destination. The operators of this category are primarily interested in modifying the dispatch statement or add some method calls that affect on the dispatch statement. Examples of modifications include: change dispatch destination, remove dispatch statement, remove proxy name from dispatch statement, insert a deactivate statement before the dispatch statement, replace dispatch statement with dispose or retract statement, switch dispatch statement with its previous statement, and shrink if statement by moving dispatch statement outside the if statement. We now discuss these operators in detail.
### Table 5.1: Mobile agent mutation operators

<table>
<thead>
<tr>
<th>Operator category</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mobility</strong></td>
<td>CDD</td>
<td>Change Dispatch Destination</td>
</tr>
<tr>
<td></td>
<td>IDS</td>
<td>Insert Deactivate Statement</td>
</tr>
<tr>
<td></td>
<td>RAPD</td>
<td>Remove Aglet Proxy from Dispatch statement</td>
</tr>
<tr>
<td></td>
<td>RDS</td>
<td>Remove Dispatch statement</td>
</tr>
<tr>
<td></td>
<td>RDD</td>
<td>Replace Dispatch with Dispose</td>
</tr>
<tr>
<td></td>
<td>RDR</td>
<td>Replace Dispatch to Retract</td>
</tr>
<tr>
<td></td>
<td>SIHD</td>
<td>Shrink if/else has Dispatch</td>
</tr>
<tr>
<td></td>
<td>SDS</td>
<td>Switch Dispatch Statement</td>
</tr>
<tr>
<td><strong>Communication</strong></td>
<td>CMP</td>
<td>Change Message Parameter</td>
</tr>
<tr>
<td></td>
<td>CMOW</td>
<td>Change Message to One Way Message</td>
</tr>
<tr>
<td></td>
<td>RSMM</td>
<td>Remove Send Message Method</td>
</tr>
<tr>
<td></td>
<td>MSKP</td>
<td>Modify SameKind Parameter</td>
</tr>
<tr>
<td></td>
<td>MSR</td>
<td>Modify SendReply Parameter</td>
</tr>
<tr>
<td></td>
<td>RPSM</td>
<td>Remove a Parameter from A Set of Method</td>
</tr>
<tr>
<td></td>
<td>RSRM</td>
<td>Remove SendReply Method</td>
</tr>
<tr>
<td></td>
<td>MCMC</td>
<td>Move the Communication Method Calls in IfElse</td>
</tr>
<tr>
<td></td>
<td>NAN</td>
<td>Notify All Message to Notify Message</td>
</tr>
<tr>
<td><strong>Agent’s Run Method</strong></td>
<td>RMRM</td>
<td>Remove Run Method</td>
</tr>
<tr>
<td></td>
<td>RPRM</td>
<td>Replace Run Method</td>
</tr>
<tr>
<td><strong>Agent Creation</strong></td>
<td>MICA</td>
<td>Modify Create Aglet Parameter</td>
</tr>
<tr>
<td></td>
<td>MFCA</td>
<td>Modify the File Name in Create Aglet</td>
</tr>
<tr>
<td></td>
<td>ROCM</td>
<td>Replace onCreation with other Method</td>
</tr>
<tr>
<td></td>
<td>ACON</td>
<td>Add clone method in onCreation</td>
</tr>
<tr>
<td><strong>Event Listeners</strong></td>
<td>REN</td>
<td>Replace Event Name with another event</td>
</tr>
<tr>
<td></td>
<td>RARL</td>
<td>Replace Add listener with Remove Listener</td>
</tr>
<tr>
<td><strong>Agent Proxy</strong></td>
<td>RAID</td>
<td>Remove AgletProxy from getAgletID</td>
</tr>
<tr>
<td></td>
<td>CPCN</td>
<td>Change proxy name in getAgletClassName</td>
</tr>
<tr>
<td></td>
<td>CSAP</td>
<td>Change the State in getAgletProxies</td>
</tr>
<tr>
<td></td>
<td>CNP</td>
<td>Change number of Proxies</td>
</tr>
</tbody>
</table>

**CDD: Change Dispatch Destination**

In Aglets the agent can dispatch itself to some remote context by calling its dispatch method with the URL of the remote context as the parameter. This URL should specify the host and domain names of the destination context, and the protocol ATP\(^1\) to be used for transferring the agent over the network. Moreover, the agent

\(^1\)The Agent Transfer Protocol (ATP) is an application-level protocol for distributed agent-based systems. It offers a simple protocol for transferring agents between networked computers.
can dispatch another agent by calling the dispatch method of that agent. In this case we need the AgletProxy of that agent. The AgletProxy interface is a placeholder for an agent. The purpose of this interface is to provide a mechanism to control and limit direct access to agents. The CDD operator replaces the destination of the dispatch() method call by some another destination. For example:

Original Code
\[
\text{dispatch(new URL(\'atp://balboa\'));}
\ldots
\]

Apply CDD Operator
\[
\text{dispatch(new URL(\'atp://polka\'));}
/*replace the destinations*/
\]

The CDD operator when applied will migrate the agent to a wrong destination which may mean losing the agent.

**IDS : Insert Deactivate Statement**

By using the deactivate(t) method, the agent is temporarily stopped and be removed from its current context. It returns to the context and resumes execution after the specified period t has elapsed. So inserting a deactivate method call before a dispatch method call using the IDS operator means delaying the movement of that agent for some time. Of course this delay in movement could affect the behavior of that agent and the whole system.

Original Code
\[
\text{dispatch(new URL(\'atp://balboa\'));}
\ldots
\]

Apply IDS Operator
\[
\text{long time= 30000;}
\text{deactivate(time);}
/*insert deactivate methods*/
\text{dispatch(new URL(\'atp://balboa\'));}
\]
RAPD: Remove Agent Proxy from Dispatch statement

We have explained before that an agent can dispatch another agent by calling the dispatch method of that agent. To do that, we need the AgletProxy of that agent. Suppose that agent A wants to move agent B to some destination dest. And suppose the proxy name of A is Aproxy and the proxy name of B is Bproxy. To achieve the move, A executes Bproxy.dispatch(dest). However, if we removed the proxy name Bproxy from the previous statement (new statement is dispatch(dest)), then agent A will move itself to that destination instead. For example:

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Apply RAPD Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL dest = new URL('atp://balboa'); Contract派遣 (dest); ...</td>
<td>URL dest = new URL('atp://balboa'); dispatch(dest); /<em>remove agent proxy</em>/</td>
</tr>
</tbody>
</table>

The RAPD operator when applied will migrate the wrong agent and the agent who should migrate will die if the dispatch statement is the last statement of the agent code.

RDS: Remove Dispatch statement

Removing the dispatch statement means that the agent will stay in its context and will not be moved to the new destination. So if there is some code that will be executed after the movement of the agent, then this code will never be executed. The following shows an example of using the RDS operator.

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Apply RDS Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>try{</td>
<td>try{</td>
</tr>
<tr>
<td>Contract派遣 (dest);</td>
<td>/<em>removed dispatch</em>/</td>
</tr>
<tr>
<td>} catch ...</td>
<td>} catch ...</td>
</tr>
</tbody>
</table>
The RDS operator is another example where an agent will die in its context instead of moving to another context.

**RDD: Replace Dispatch with Dispose**

In Aglets the `dispose()` method is used to destroy and remove the agent from its current context (i.e. kill the thread). The RDD operator replaces the `dispatch` method call with the `dispose` method call. So instead of moving the agent to a new destination we are killing that agent. For example:

Original Code
```
URL dest = new URL('atp://balboa');
Contract.dispatch(dest);
...
```

Apply RDD Operator
```
URL dest = new URL('atp://balboa');
Contract.dispose();
/*replace dispatch with dispose*/
```

**RDR: Replace Dispatch with Retract**

The `retractAglet` method is used to move an agent back to the URL where it was created. So by using the RDR operator instead of dispatching the agent to a new destination we retract it to the original place where it was created. For example:

Original Code
```
URL dest = new URL('atp://balboa');
dispatch(dest);
...
```

Apply RDR Operator
```
URL dest = new URL('atp://balboa');
getAgletContext().retractAglet(OrgDest);
/*replace dispatch with retract*/
```
SIHD Shrink ifelse that has Dispatch

This operator is used to shrink an if statement that contains a dispatch method call by moving the dispatch method call outside the if statement. For example:

Original Code
if (condition){
    ...
    statement n ;
    Contract . dispatch ( dest );
}

Apply SIHD Operator
Contract . dispatch ( dest );
/*move dispatch outside if*/
if (condition){
    ...
    statement n :}

The SIHD operator may also cause an agent to move incorrectly.

SDS: Switch Dispatch Statement

The SDS operator is an example where an agent is moved early. In this operator we switch the dispatch statement with its previous statement. For example:

Original Code
statement n ;
Contract . dispatch ( dest );

Apply SDS Operator
Contract . dispatch ( dest );
statement n ;
/*switch dispatch statements*/

The SDS operator makes an agent move before the execution of the statement that we have switched it with. This situation may cause an agent to lose some data when it is moved.

5.2.2 Communication

One of the basic abilities of an agent is its ability to communicate with other agents. In Aglets the communication is performed by exchanging message objects. These message objects are sent using the Agletproxy class methods. In Aglets there are three
types of messages that can be sent: now-type (synchronous message which blocks further execution until receiver has handled and acknowledged the message); future-type (asynchronous message where the sender has a future handler used for obtaining result and where the sender does not have to wait for the receiver’s response); oneway-type (sends a oneway message to the agent and no acknowledgement will be sent back to the sender). The Agent can handle the message through the `handleMessage` method.

In this section we propose some of the mutation operators that affect the communication between agents in Aglets.

**CMP: Change Message Parameter**

The CMP operator can be applied to the three types of messages. It is applied to the `sendMessage()`, `sendFutureMessage()`, and `sendOneWayMessage()` method calls that include a parameter of type `Message` which represents the content of the message. For example, in Aglets a call of the `sendMessage` for a specific agent will send a message to that agent and will cause the thread to no longer be executed until the receiver has handled and acknowledged the message. The CMP operator replaces the content of the message by another message.

**Original Code**

```java
Message msg = new Message("Arrived?");
String ack = Contproxy.sendMessage(msg);
if(ack.equals(...)) { ...;
```

**Apply CMP Operator**

```java
Message msg = new Message("Mutation");
/* change msg parameter*/
String ack = Contproxy.sendMessage(msg);
if(ack.equals(...)) { ...;
```

The CMP operator will make the agent send an incorrect message to another agent through its proxy. This message will not be handled by the receiver since it is expecting to receive specific kinds of messages. In this case the receiver’s `handleMessage` method will return the truth value “false” to the sender telling him that he did not
CHAPTER 5. MOBILE AGENT MUTATION OPERATORS

handle the message.

CMOW: Change sendMessage to OneWayMessage

The `sendMessage` method is used to send a message to an agent and waiting for an acknowledgment. The `sendOnewayMessage` is used to send a message without waiting for an acknowledgment. The CMOW operator is used to replace the `sendMessage` method call with a `sendOnewayMessage` method call. For example:

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Apply CMOW Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Message msg = new Message('Get price');</code></td>
<td><code>Message msg = new Message('Get price');</code></td>
</tr>
<tr>
<td><code>Sprice = Agentproxy.sendMessage(msg);</code></td>
<td><code>Agentproxy.sendOnewayMessage(msg);</code></td>
</tr>
<tr>
<td></td>
<td><code>/*change msg types*/</code></td>
</tr>
</tbody>
</table>

By using the CMOW operator, the agent who sends the message will lose the acknowledgment message that it is expecting to receive from the receiver of the original message.

RSMM: Remove Send Message Method

The RSMM operator removes calls to all methods used to send messages: `sendMessage()`, `sendFutureMessage()`, and `sendOnewayMessage()`. Removing the calls of these methods means removing the communication between the agents. The example below shows an example of removing the `sendFutureMessage()` method call:

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Apply RSMM Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Message msg = new Message('Search');</code></td>
<td><code>Message msg = new Message('Search');</code></td>
</tr>
<tr>
<td><code>FutureReply reply = A.sendFutureMessage(msg);</code></td>
<td><code>FutureReply reply = null;</code></td>
</tr>
<tr>
<td></td>
<td><code>/*Remove msg statements*/</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MSKP: Modify SameKind Parameter

The `sameKind` method allows the receiving agent to distinguish among messages he wants to understand and respond to. Its parameter is a message key to be compared with keys of incoming messages. The MSKP operator is used to modify the parameter of the `sameKind` method call. For example:

Original Code
```java
boolean handleMessage(Message msg){
  if (msg.sameKind("Get Mall Address")){
  ...
}
```

Apply MSKP Operator
```java
boolean handleMessage(Message msg){
  if (msg.sameKind("Mutation")){
    /*update samekind arguments*/
    ...
  }
```

By modifying the `sameKind` method call parameter from `M` to `M'` using the MSKP operator, the receiver agent will not be able to handle the incoming message `M` that has been sent by another agent. Therefore, the sender agent will never get a reply for that message.

MSR: Modify SendReply Parameter

As we discussed before, sometimes when an agent sends a message to another agent it expects an acknowledgment or a reply from the receiving agent. The `sendReply` method is used to send a reply message. The MSR operator is applied to change the contents of the reply message. For example:

Original Code
```java
boolean handleMessage(Message msg){
  if (msg.sameKind("Search Camera")){
    msg.sendReply("Found");
    msg.sendReply("Not Found");
    ...
}
```

Apply MSR Operator
```java
boolean handleMessage(Message msg){
  if (msg.sameKind("Search Camera")){
    msg.sendReply("Found");
    /*modify reply message*/
```

By modifying the `sendReply` method call parameter the sender agent will receive an incorrect reply from the receiver agent.
CHAPTER 5. MOBILE AGENT MUTATION OPERATORS

RPSM: Remove a Parameter from A Set of Methods

The RPSM operator can be applied to the \texttt{getArg(x)}, \texttt{sendReply(x)}, \texttt{waitMessage(-t)}, and \texttt{waitForReply(t)} method calls that include an optional parameter. In Aglets, an agent can send a message with different values. To distinguish between these values there should be a key for each of them. The \texttt{setArg(x)} method is used to set the key for each value. The receiver agent uses this key to get the value of a message by using the \texttt{getArg(x)} method. The \texttt{waitMessage(t)} is used to wait until it is notified or the timeout (\texttt{t}) expires. The \texttt{waitForReply(t)} is used to wait for a reply with specific timeout value \texttt{t}. The RPSM operator removes the optional parameter for all these method calls. For example, removing the \texttt{t} parameter from the \texttt{waitForReply(t)} method call makes the agent wait for a reply until the reply is available. Removing the parameter \texttt{x} from the \texttt{sendReply(x)} method call makes the receiver agent send an acknowledgment reply without any value. Removing the parameter \texttt{t} from the \texttt{waitMessage(t)} method call makes the agent wait until it is notified. For example:

Original Code

\begin{verbatim}
Sender Agent:
Message msg = new Message('const');
msg.setArg('pi', 3.1416);
msg.setArg('myname', 'Ahmad');
receiver.sendMessage(msg);
...

Receiver Agent:

boolean handleMessage(Message msg){
    if (msg.sameKind('const')){
        name = (String) msg.getArg('myname');
        /* do something */
    }
    ...
}
\end{verbatim}

Apply RPSM Operator

\begin{verbatim}
Sender Agent:
Message msg = new Message('const');
msg.setArg('pi', 3.1416);
msg.setArg('myname', 'Ahmad');
receiver.sendMessage(msg);
...

Receiver Agent:

boolean handleMessage(Message msg){
    if (msg.sameKind('const')){
        name = (String) msg.getArg();
        /* remove parameter from getArg*/
    }
    ...
}
\end{verbatim}

Applying RPSM operator on the \texttt{getArg(x)} method call may cause the sender message to not be handled properly by the receiver agent.
CHAPTER 5. MOBILE AGENT MUTATION OPERATORS

RSRM: Remove SendReply Method

The RSRM operator removes a `sendReply()` method call. Using this operator the sender agent will never receive a reply message from the receiver agent. And in this case the sender agent will assume that the receiver agent did not get his message.

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Apply RSRM Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>msg.sendReply(‘‘Found’’);</code></td>
<td>/<em>removed sendReply method calls</em>/</td>
</tr>
</tbody>
</table>

MCMC: Move the Communication Method Calls in IfElse

The MCMC operator moves calls to all methods used to send and reply messages from the `if` part to the `else` part and vise versa. These method calls are: `sendMessage`, `sendFutureMessage()`, `sendOneWayMessage()`, and `sendReply()`. The following are examples of using MCMC operator:

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Apply MCMC Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (condition){</code></td>
<td><code>else{</code></td>
</tr>
<tr>
<td><code>X.sendMessage(msg);</code></td>
<td><code>X.sendMessage(msg);</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>}</code></td>
</tr>
<tr>
<td><code>else{</code></td>
<td><code>else{</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Apply MCMC Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (condition){</code></td>
<td><code>else{</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>Y.sendMessage(msg);</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>}</code></td>
</tr>
<tr>
<td><code>else{</code></td>
<td><code>else{</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>Y.sendReply(msg);</code></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Apply MCMC Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (condition){</code></td>
<td><code>else{</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>Y.sendReply(msg);</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>}</code></td>
</tr>
<tr>
<td><code>else{</code></td>
<td><code>else{</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>Y.sendMessage(msg);</code></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>
CHAPTER 5. MOBILE AGENT MUTATION OPERATORS

NAN: Notify All Message to Notify Message

The NAN operator changes the notifyAllMessage() (which notifies all waiting threads) method call into the notifyMessage() (which notifies only a single waiting thread) method call. By using the NAN operator, some of the waiting threads will not be notified. For example:

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Apply NAN Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>notifyAllMessage()</td>
<td>notifyMessage()</td>
</tr>
</tbody>
</table>

5.2.3 Agent’s Run Method

The run() method is the entry point for the agent’s own thread of execution. It is called every time the agent arrives at, or is activated in, a new context. The method is invoked upon successful creation, dispatch, cloning, retraction or activation of the agent. For example, if a createAglet method is used to create an agent A, then agent A will first execute the onCreation method and then the run method. If dispatching is used, then before the agent is moved to the new destination, the methods will be executed in the following order: the run method, the dispatch method call, and then the onDispatching method. Once the agent has arrived, the order of executed methods will be the onArrival method then again the run method. The following discusses the mutation operators used to modify the run method:

RMRM: Remove Run Method

Because the run method is called whenever it occupies the context, this is a good place to define the common task of an agent. Removing the run method using the
RMRM operator will remove some tasks of the agent. Moreover, sometimes removing the `run` method causes the agent to become idle. For example:

```
Original Code
public void run(){
    try{
        // do some tasks
    } catch{...}
}
```

```
Apply RMRM Operator
  /* removed run */
```

RPRM: Replace Run Method

As we have described in Table 2.1 Chapter 2, there are several Aglets callback methods. The RPRM operator replaces the `run` method with these callback methods. For example:

```
Original Code
public void onDispatching(){
    /* onDispatching stm */
    ...
}
public void run(){
    /* run stm */
    ...
}
```

```
Apply RPRM Operator
public void onDispatching(){
    /* onDispatching stm */
    ...
}
public void onArrival(MobilityEvent e){
    /* run stm */
    ...
}
```

Before using the RPRM operator, the `run` method is executed first. Then before the agent is actually moved, the `onDispatching` method is executed. Once the agent arrives at the new destination, the `run` method is executed again. Replacing the `run` method by the `onArrival` method using the RPRM operator changes the execution order of the agent method. So after applying the RPRM operator and before the agent moves to the new destination, the `onDispatching` method is executed first. And once the agent has arrived at the new destination, the `onArrival` method will be executed. Now suppose the callback method that we want to replace the `run`
method with already exists in the agent code. The RPRM operator switches the code of these methods. For example:

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Apply RPRM Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void onDispatching(){</td>
<td>public void run(){</td>
</tr>
<tr>
<td>/* onDispatching stm */</td>
<td>/* run stm */</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>public void run(){</td>
<td>}</td>
</tr>
<tr>
<td>/* run stm */</td>
<td>}</td>
</tr>
<tr>
<td>...</td>
<td>}</td>
</tr>
</tbody>
</table>

5.2.4 Agent Creation

The third category of our mobile agent mutation operators concerns the Aglets statements that have to do with agent creation. The `createAglet` statement is used to create an instance of an agent class in a context. The agent’s class code file can be located on the local file system as well as on a remote server. It has three parameters: `codeBase`, `code`, and `init`. The `codeBase` specifies the directory (possibly remote) of the agent class. If it is `null`, the context will search for the code in the local system’s Aglet search path (AGLET-PATH). The `code` parameter gives the name of the file that contains the Aglet’s compiled class code. The `init` parameter is an object passed on to the agent’s `onCreation` method. The operators of this category are primarily interested in modifying the `createAglet` method by, e.g., modifying the `code` parameter, replacing the `init` value (if it is exist) with `null`, replacing the `onCreation` callback method with other callback methods and adding an invocation of the `clone` method in the `onCreation` callback method. The following discusses all these operators.
CHAPTER 5. MOBILE AGENT MUTATION OPERATORS

MICA: Modify init Parameter in Create Aglet

As we described earlier, the *init* parameter in the *createAglet* method is used to pass on an object to the agent when it is created. For example, this parameter can be used to pass the *agletProxy* of the parent to its child when it is created. For example:

\[
\text{ChildProxy} = \text{context.createAglet}(\text{null}, \text{Child}, \text{getProxy}());
\]

Suppose this statement is part of the *Parent* code. This statement is used to create an agent *Child* that is located in the local systems Aglets search path (since the first parameter is null). Moreover, the proxy (using the *getProxy* method call) of the *Parent* will be passed on to the child’s *onCreation* method. When the child agent is moved to another context, it can use the value of this parameter to communicate with its parent remotely. The MICA operator is used to modify the *init* parameter of the *createAglet* method call by replacing its value with *null*. For example:

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Apply MICA Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prox= c.createAglet(null,'Child',getProxy());</td>
<td>Prox= c.createAglet(null,'X',null); /* modify 3rd parameter */</td>
</tr>
</tbody>
</table>

Applying the MICA operator makes the agent who is using this parameter miss its value. If the value of this parameter is the proxy of another agent, then by using this mutation operator the agent will lose the communication with the other agent.

MFCA: Modify the File Name in Create Aglet

The MFCA operator is used to change the *code* parameter in the *createAglet* method call. In other words, this operator is used to modify the class name of the agent that we are going to create. For example:
CHAPTER 5. MOBILE AGENT MUTATION OPERATORS

Applying the MFCA operator changes the agent name that we are going to create. If that file does not exist, a runtime exception occurs. Otherwise, the agent may start to communicate incorrectly with that agent.

**ROCM: Replace onCreation with other Method**

We have already seen the replacement of an agent callback method with another agent callback method (Subsection 5.2.3). The ROCM operator is used to replace the onCreation method with other callback methods. If the method that the ROCM operator wants to exchange it with is already in the code, the ROCM operator swaps the contents of these two methods.

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Apply ROCM Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void onCreation(Object o){</td>
<td>public void onArrival(){</td>
</tr>
<tr>
<td>...</td>
<td>/<em>replace method name</em>/</td>
</tr>
<tr>
<td>}</td>
<td>...</td>
</tr>
</tbody>
</table>

**ACON: Add clone method in onCreation**

The ACON operator adds an invocation method of the clone() method in the onCreation callback method. This addition creates an instance of the created agent. In other words, instead of creating one instance of an agent by using this operator two instances of that agent will be created.

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Apply ACON Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>public void clone(){</td>
</tr>
<tr>
<td></td>
<td>/<em>add clone method</em>/</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Apply ACON Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>public void onArrival(){</td>
</tr>
<tr>
<td></td>
<td>/<em>use clone method</em>/</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
Applying the ACON operator creates an incorrect number of agents.

5.2.5 Event Listeners

As we have mentioned before Aglets is an event-based programming language. It allows the programmer to plug in event listeners into an agent. These event listeners are customizable. These listeners are used to catch particular events during the life cycle of an agent. Moreover, they allow the programmers to take an action when such event occurs, e.g., cloned, disposed, etc. There are three event listeners in Aglets:

1. **Clone listener**: used to listen for cloning events. The listener can be customized to catch specific actions when an agent is about to clone (the `onCloning` callback method), when a clone is actually created (onClone callback method), and when a clone was created (the onCloned callback method).

2. **Mobility listener**: used to listen for mobility events. The listener can be customized to catch specific actions when an agent is about to be dispatched to another context (the onDispatching callback method), when an agent about to be retracted (the onReverting callback method), and when an agent has arrived in a new context (the onArrival callback method).

3. **Persistence listener**: used to listen for the persistent event. The listener can be customized to catch specific actions when an agent is about to be activated.
(the onDeactivating callback method) and after it has been activated (the onActivation callback method).

In this section we propose some of the mutation operators that are used to replace these events (callback methods) with other events and replacing the addition of an event listener with the removal of an event listener.

**REN: Replace Event Listener Name with another event**

We have already seen the replacement of one method with another (ROCM and RPRM). The REN operator is identical except it replaces the event listener with another event listener.

Original Code

```java
public void onCloning()
{
    ...
}
```

Apply REN Operator

```java
public void onCloned()
{
    /*replace event listener name*/
    ...
}
```

By using the REN operator, the time of the execution statement will be incorrect which means changing the behavior of an agent.

**RARL: Replace Add listener with Remove Listener**

In Aglets we use the `addCloneListener(CloneListener listener)` method to add the specified clone listener to receive clone events from this agent, the `addMobilityListenerMobilityListener listener)` method to add the specified mobility listener to receive mobility events from this agent, and the `addPersistencyListener(PersistencyListener listener)` method to add the specified persistency listener to receive persistency events from this agent. On the other side, Aglets uses the `removeCloneLi-
stener(CloneListener l) to remove the specified clone listener so it no longer receives clone events, the removeMobilityListener(MobilityListener l) to remove the specified mobility listener so it no longer receives mobility events and the removePersistencyListener(PersistencyListener l) to remove the specified persistency listener so it no longer receives persistency events. The RARL operator replaces the addXListener method call with the removeXListener method call where X is Clone, Mobility, or Persistency. For example:

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Apply RARL Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void onCreation(Object o){</td>
<td>public void onCreation(Object o){</td>
</tr>
</tbody>
</table>
|   ... |   ...
|   addMobilityListener( |   removeMobilityListener( |
|     new MobilityAdapter(){ |     new MobilityAdapter(){ |
|       public void onArrival(...){ |       public void onArrival(...){ |
|         ... |         ...
|       public void onReverting(...){ |       public void onReverting(...){ |
|         ... |         ...
|     }) |     })
|   } |   }
| ... | ... |

By applying the RARL operator the agent will no longer receive clone, mobility, and persistency events.

5.2.6 Agent Proxy

The AgletProxy in Aglets is used to control and limit direct access to an agent. For example, if an agent A wants to communicate with another agent B, then we need to know the Aglets proxy of B. There are different ways to get the Aglets proxy of agents, one of them is by getting an enumeration of proxies in a context by calling the primitive AgletContext.getAgletProxies(). An agent can be distinguished from
another agent by its ID. This ID is unique over the agent’s life time. We can get the ID of an agent by using the \texttt{getAgletID()} method call. We can get the agent class name by using the \texttt{getAgletClassName} method call. The operators of this category are primarily interested in making some changes that are related to the Aglets proxy and its methods.

**RAID: Remove AgletProxy from getAgletID**

The \texttt{getAgletID} method call is used to get the Aglets ID for a specific agent. To do that we need the Aglets proxy of that agent. If we do not specify the Aglets proxy of the agent, then we will get the Aglets ID of the current agent. The RAID operator is used to remove the proxy name from the \texttt{getAgletID} method call.

**Original Code**

```java
/* code of Agent A */
int index = 0;
while (index <= proxyList.size()-1){
    AgletProxy proxy = (AgletProxy) proxyList.get(index);
    if (getAgletID() != proxy.getAgletID()){
        /* Agent A tries to visit all agent in the context except himself */
        ...
    }
    ...
    index++;
}
```

**Apply RAID Operator**

```java
/* code of Agent A */
int index = 0;
while (index <= proxyList.size()-1){
    AgletProxy proxy = (AgletProxy) proxyList.get(index);
    if (getAgletID() != getAgletID()){ /* removed proxy from second getAgletID()*/
        /* Agent A tries to visit all agent in the context except himself */
        ...
    }
    ...
    index++;
}
```
Removing the proxy name from the `getAgletID` method call makes the program retrieve the ID of the current agent instead of the ID of a specific agent. By applying the RAID operator in the previous example, agent A will not visit any of the agents in its context. Moreover, if the condition of the while loop is comparing the Aglets ID of the current agent with the Aglets ID’s of the other agent in a context, then applying the RAID operator may make the agent enter an infinite loop.

**RPCN: Replace proxy name in `getAgletClassName`**

The `X.getAgletClassName()` method is used to get the class name for an agent X. The RPCN operator is used to change the proxy name in the `getAgletClassName` method call. For example:

```
Original Code                                      Apply RPCN Operator
/*/ code of Agent A*/
ShopName=proxy.getAgletClassName();
if(ShopName.equals("Shop1"){
  ...
}
```

In the above example replacing the proxy name “proxy” with the `getAgletInfo()`\(^2\) method call changes the value of the `ShopName` from the class name of the `proxy` into the class name of agent A.

**CSAP: Change the State in `getAgletProxies`**

As we explained before one of the ways to get an enumeration of proxies in a context is calling the primitive `AgletContext.getAgletProxies()`. The `getAgletProxies()` has one parameter describing the state of an agent you want to get. The possible

\(^2\)The `getAgletInfo()` is used to get the info object of this agent.
values of this parameter are ACTIVE, INACTIVE or ACTIVE/INACTIVE (default is ACTIVE). If we use ACTIVE then the primitive `AgletContext.getAgletProxies()` will get only all the enumerated active proxies. The CSAP operator modifies the value of this parameter. For example:

```
Original Code
public Enumeration e;
...
e=getAgletContext().getAgletProxies(ACTIVE);

Apply CSAP Operator
public Enumeration e;
...
e=getAgletContext().getAgletProxies(INACTIVE);
```

Changing the state in the `getAgletProxies` method call using the CSAP operator means changing the proxies that we are going to retrieve.

**CNP: Change number of Proxies**

The `getAgletProxies()` yields an enumeration of proxies in the current context. To use the proxies we add the enumeration elements to an `ArrayList` of type `AgletProxy`. And we use the method `size` to get the number of proxies in the array list (the proxies in the current context). The CNP operator is used to modify the retrieved number of the proxies in the current context by reducing the `size()` method by 1. For example:

```
Original Code
ArrayList Plist = new ArrayList();
while (e.hasMoreElements()){
    Plist.add((AgletProxy)e.nextElement());
}
index = 0;
while (index <= proxyList.size()){
    ...
}

Apply REN Operator
ArrayList Plist = new ArrayList();
while (e.hasMoreElements()){
    Plist.add((AgletProxy)e.nextElement());
}
index = 0;
while (index <= proxyList.size()-1){
    /* # of proxies decreased */
    ...
}
5.3 Chapter Summary

We have presented our mobile agent mutation operators that have been classified in six categories: mobility, communication, the agent’s run method, agent creation, event listeners, and the agent proxy. These mutation operators will be used in the evaluation of our approach in Chapter 6. Our new mutation operators should be viewed as a complement to the mutation operators identified for Java in e.g., [14] and [65]. For example, the mutation operators in [14] are used to cause concurrency bugs and the concurrency could be available in mobile agent system. In other words, concurrency mutant operators can cause indirect mobile agent bugs. Moreover, MuJava mutation operators are used to cause bugs in a statement that performs actions such as modification, replacement, and deletion. These modifications could also affect indirectly the behavior of the mobile agent system.
Chapter 6

Evaluation and Comparison

To check the efficiency and the effectiveness of our runtime conformance checking approach presented in Chapter 3 and Chapter 4, in this chapter, we propose a mutation-based evaluation framework. This framework automatically and efficiently generates and runs the mutants that have been automatically generated from the system under test. The purpose of this framework is to determine how effective our approach is for finding non conformance in mobile agent systems.

Moreover, we compare our runtime conformance checking approach with runtime monitoring based on aspects. This comparison provides further evidence of the effectiveness of our approach.

In this chapter, we will first provide the details of our evaluation framework in Section 6.1. In Section 6.2 we describe the experimental setup of our study. Our findings and analysis results are presented in Section 6.3. The comparison with aspect-oriented runtime monitoring is discussed in Section 6.4. Finally, Section 6.5 summarizes the chapter.
6.1 The Evaluation Framework

This section provides the description of the major components of our mutation-based framework. Figure 6.1 shows the framework. The framework has two phases, the Mutant Generation Phase, in which mutants are generated automatically from the original code, and the Evaluation Phase in which all the mutants are tested for conformance by applying our runtime conformance checking approach then to see how many mutants are killed. The following subsections describe the two phases in more detail.

6.1.1 Mutant Generation Phase

The purpose of the first phase of the framework is to automatically generate a number of mutated versions of the original code. This phase has the following steps:

- **Select Mutation operator**: After we have selected the program that we are going to use in this framework, a set of mutation operators that have been identified in Chapter 5 are selected.

- **Mutant Generation**: A mutant generator is used to automatically generate a number of mutants for each of the selected mutation operators. In more detail, the mutant generator uses a TXL-based mutation process [24] to mutate the original code of the selected program. If the program contains a set of files, then the mutant generator uses the TXL-based mutation process to generate a set of mutants for each file of the program. Each mutant is stored as a file in a directory. The name of this directory represents the mutation operator used to generate this mutant and the file name. Since the mutation operator can
(a) Mutant Generation Phase

(b) Evaluation Phase

Figure 6.1: The mutation-based evaluation framework
generate more than one mutant for a given file, we distinguish between them by adding a number to the name of the directory.

- **Compile Mutants**: The mutant generator is also used to automatically compile all the mutants file since Aglets requires compiled source code as input. If the mutant generator produces a mutant that is not syntactically correct, then it will not compile. Only mutants files that compile successfully is used in the next phase.

### 6.1.2 Evaluation Phase

The purpose of this phase is to run all the mutated programs that have been generated in the previous phase one by one in parallel with the correct kiltera model for evaluation. If the mutated program and the kiltera model are non-conforment (which should be discovered by the kiltera model), then we say the mutant is killed. This phase has the following steps:

- **Run the Runtime Conformance Checking Approach**: After we generate the mutants, another script is used to automatically replace the compiled mutated file with the compiled original file of the implementation under test. Moreover, the script automatically runs our runtime conformance checking approach where the mutant program and the kiltera model are automatically executed in parallel. At the end of the execution kiltera will produce a report. This process will be repeated for each mutant generated in the previous phase.

- **Detection Evaluation**: Another script is used to automatically check all the kiltera reports in order to see whether the execution of each of the mutation
programs conforms to that of the kiltera model or not. The script will also produce a report showing each mutant name and whether it conforms to kiltera model or not. Moreover, it shows the total number of the generated mutants and how many of them have been killed. Subsection 6.3.4 will show such kind of a report.

6.2 Experimental Setup

In this experiment we have applied the mutation-based evaluation framework using the steps specified in Section 6.1. The setup involves selecting the example program which will be used in the experiment and the mutation operators that will be used to generate the mutants.

6.2.1 Examples Used

In this experiment we used the two case studies that are presented in Chapter 4. However, the number of shops in the online shopping example and the number of distributed partners in the contract signing example are increased to obtain more information about the potential scalability of our approach. The two case studies are:

- **OnlineSearching**: A simulation program where an agent is searching for a specific item (e.g., a camera) by traveling to different online shopping malls in order to find the lowest price for this item and return the result of the search to the original site. We consider a system with four malls which have two shops, three shops, four shops, and five shops respectively.
• **SigningContract**: A simulation program in which several distributed partners have to sign a contract for a specific project. A coordinator called notary guides the process of signing the contract by all partners. Here we consider a system with four distributed partners.

Table 6.1 provides a statistical overview of these examples. The examples we used are

<table>
<thead>
<tr>
<th>Aglets Program</th>
<th>loc</th>
<th>classes</th>
<th>methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnlineSearching</td>
<td>1157</td>
<td>20</td>
<td>29</td>
</tr>
<tr>
<td>Main.java</td>
<td>30</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Client.java</td>
<td>61</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Agent.java</td>
<td>212</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Mall1 with 3 shops</td>
<td>186</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Mall2 with 2 shops</td>
<td>131</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Mall3 with 4 shops</td>
<td>241</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Mall4 with 5 shops</td>
<td>296</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>SigningContract</td>
<td>549</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>Notary.java</td>
<td>121</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Contract.java</td>
<td>144</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Partner1.java</td>
<td>71</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Partner2.java</td>
<td>71</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Partner3.java</td>
<td>71</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Partner4.java</td>
<td>71</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

created manually by the author. We tried our best to find existing Aglets programs that feature mobility, communication, and timing issues from other sources. However, all the existing Aglets program examples were too small. Moreover, we had difficulty finding Aglets program examples that used all these features together. However, the program examples we used in Table 6.1 have many of the features found in mobile agent programs such as mobility (an agent moves or is moved to a new context), remote communication, local communication, time, negotiation between agents, etc. Moreover, the program examples that we have selected are presented and implemented
as mobile agent systems in several works using other mobile agent languages [10, 55, 13].

6.2.2 Mutation Operators used

In this experiment we have used and applied all the mutation operators described in Chapter 5 in order to generate mutants for both Aglets programs.

6.3 Experimental Results

After we setup the experiment we applied the two Aglets program examples into the two phases of the mutation-based framework. The following subsections describe the results and analyze them. Our analysis is based on the analysis performed on other mutation tools reported in the literature, e.g., [52, 66, 14, 23]

6.3.1 Mutant Generation

We start our experiment by selecting the mutation operators to generate mutants from the two Aglets programs. In this experiment we have selected all the 29 mutation operators specified in Chapter 5. After that, the script generator automatically generates the mutants from the two Aglets programs. Table 6.2 provides details on the total number of mutants generated according to the mutation operators for both Aglets programs. As we can see from Table 6.2, the OnlineSearching example has more mutants than the SigningContract example. This is because the first example has more lines of code than the second one. One of the mutation operators (NAN) did not generate any mutants which indicates that the NotifyAllMessage() call method
Table 6.2: The number of mutants generated for the two Aglets programs

<table>
<thead>
<tr>
<th>Operator</th>
<th>OnlineSearching</th>
<th>SigningContract</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDR</td>
<td>6</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>CDD</td>
<td>6</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>IDS</td>
<td>6</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>RAPD</td>
<td>4</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>RDS</td>
<td>6</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>RDD</td>
<td>6</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>SHD</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>SDS</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>CMP</td>
<td>4</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>CMOW</td>
<td>3</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>RSM</td>
<td>4</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>RPSM</td>
<td>30</td>
<td>20</td>
<td>50</td>
</tr>
<tr>
<td>MSKP</td>
<td>17</td>
<td>10</td>
<td>27</td>
</tr>
<tr>
<td>MSR</td>
<td>28</td>
<td>13</td>
<td>41</td>
</tr>
<tr>
<td>RSRM</td>
<td>29</td>
<td>15</td>
<td>44</td>
</tr>
<tr>
<td>MCMC</td>
<td>30</td>
<td>16</td>
<td>46</td>
</tr>
<tr>
<td>NAN</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RMRM</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>RPRM</td>
<td>9</td>
<td>9</td>
<td>18</td>
</tr>
<tr>
<td>MICA</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MFCA</td>
<td>20</td>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>ROCM</td>
<td>47</td>
<td>40</td>
<td>87</td>
</tr>
<tr>
<td>ACON</td>
<td>6</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>REN</td>
<td>40</td>
<td>40</td>
<td>80</td>
</tr>
<tr>
<td>RARL</td>
<td>5</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>RAID</td>
<td>7</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>CPCN</td>
<td>0</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>CNP</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>CSAP</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>324</td>
<td>263</td>
<td>587</td>
</tr>
</tbody>
</table>

is not used in either examples. Moreover, the ROCM operator produced the largest number of mutants which indicates that the onCreation method has been used in most Java files of both examples. The other reason for this large number is that this operator is used to replace the onCreation method with all the event listeners in Aglets. There are 8 event listeners in Aglets. All of them were presented in Table 2.1.

Below we provide some analysis based on these generated mutants.

**Mutant Generation per Program**

Table 6.3 shows the number of mutants generated from each of the two programs. A
Table 6.3: Number of program mutants

<table>
<thead>
<tr>
<th>Program</th>
<th>Num Mutants</th>
<th>Mutants/m file</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnlineSearching</td>
<td>324</td>
<td>15.43</td>
</tr>
<tr>
<td>SigningContract</td>
<td>263</td>
<td>43.83</td>
</tr>
<tr>
<td>Total</td>
<td>587</td>
<td>21.74</td>
</tr>
</tbody>
</table>

total of 587 mutants were generated for the two applications, and the mean number of mutants per Java file was 21.74. On average the SigningContract example (43.83 mutants per Java file) had more generated mutants than the OnlineSearching (15.43 mutants per Java file) example. This is because in the OnlineSearching example, there are 21 Java files, but 14 of them (Shop1.java,..., Shop14.java) had only 9 generated mutants for each. However, in the SigningContract example, there are 6 Java files and each had at least 32 generated mutants. This explains why the SigningContract example on average had more mutants than the OnlineSearching example although it has more lines of code.

**Mutant Generation per Operator Category**

Figure 6.2 shows the number and the percentage of the generated mutants for each category\(^1\) of the mutation operators. We observe that our mutant operators succeeded in generating mutants for each category. Moreover, the figure shows that 42% (which is the highest percentage) of the generated mutants are generated by the communication operators category. It means that there are a lot of negotiations between the agents in both examples. Operators in the agent’s run method category generated only 3% of all mutants. This is because not all the Java files in both examples had a

\(^1\)As we explained in Chapter 5, the mutation operators are classified into 6 categories.
run method. The agent proxy category also only led to 4% of the mutants. This is because some of the modifications of the Aglets proxy are defined under other categories of the mutation operators such as the RAPD operator in mobility category.

**Mutant Generation per Java File**

This subsection further explores the number of mutants generated per Java file. Figure 6.3 summarizes the data by dividing the Java files of both examples into five groups.

Although the mean number of mutants per Java file was 21.76, 52% of all the Java files produced less than ten mutants and about 7% (one Java file) of all the Java files produced between ten and 20 mutants. Only two Java files (7%) produced more than 50 mutants: Notary.java had 92 mutants and Agent.java had 56 mutants. This is because the Notary and Agent carry out most of the behavior of the two examples.
and contain the most mutated statements. The other observation that we can make from this figure is that there is no Java file that did not produce any mutant. This is because all the Java files in both examples contain statements to send or receive messages. This means that at least we will have some mutants generated using the communication operators category. Figure 6.4 shows the number of mutants per Java file for the 27 Java files against the file size in lines of code. The result shows that the number of mutants per Java file tends to increase as the number of lines of code increases.

6.3.2 Compile Mutants

All the generated mutants in the previous step are automatically compiled using a script (we used javac). The compiled file for each mutants is saved in the same directory of that mutant. We compiled the generated mutants because Aglets requires
6.3.3 Run our Runtime Conformance Checking Approach

A script is used to automatically run our runtime conformance checking approach. For each compiled mutant, we run the runtime conformance checking approach by replacing the original compiled file of the system under test with its compiled mutant file. kiltera is run on the model in real time simulation mode and produces an output file of this execution to be traced later on. If kiltera finds the error in the mutant while the system is running, kiltera will print in the output file the word “Error,” followed by the type of error. Then this output file will also be saved in the same directory of the mutant file. Figure 6.5 shows an output file of kiltera after running our runtime conformance checking approach for a specific mutant. The progression of the execution from check point to check point can clearly be traced and the error

![Figure 6.4: The number of mutants per Java file against lines of code](image_url)
Since we have 587 mutants, our runtime conformance checking approach has been executed 587 times. Each execution is allowed to run for at most 1.5 minutes. So running the approach for all the mutants for both examples takes 14 hours and 41 minutes. The environment used for the experiment was a single user, single processor machine (Pentium 4 3.06GHZ) with 3GB of memory running the Redhat Linux operating system.

6.3.4 Detection Evaluation

At the end of the analysis and to answer the question at the beginning of this chapter, all the kiltera outputs generated by our runtime conformance checking approach will be automatically checked by a script. The purpose of this script is to detect whether the mutant is killed by our runtime conformance checking approach or not. The script
creates an analysis file that contains a list of all the mutants and whether each of
them is killed or not and how many mutants are killed and how many mutants are
not killed.

The script traces the kiltera outputs for all the mutants one by one. If it finds the
word “Error” in the output, then it means that our runtime conformance checking
approach was able to detect the error and was able to kill the mutant. Otherwise,
the mutant has not been killed. If the mutant was killed, the script will increase
the variable that has the number of killed mutants by 1. Otherwise, the variable
that has the number of non killed mutants will be increased by 1. Figure 6.5 in the
previous subsection shows that the kiltera model and the mutant program are non
conformant, since the kiltera model prints out “Error” at the end of the file. The
reason for this non conformance is that the agent was moved to an incorrect address.
The non conformance here means that the kiltera model was able to kill this mutant.
In this case, the script will increase the number of killed mutants by 1. Figure 6.6
shows part of the analysis report for the Notary.java file.

At the beginning of Figure 6.6 we can see that 9 mutants are generated from the
Notary.java file using the RDR mutation operator and all of them are killed. The
end of the figure shows that 88 out of 92 of the generated mutants for this file were
killed by our approach and only 4 mutants are not killed. All of them are generated
by the RPSM operator. The result shows for example that the RPSM_Notary.java_3
was not killed, where the RPSM is the name of the mutation operator, Notary is the
name of the file that we applied the mutation operator on, and 3 means that 3 mutants
were generated by this mutation operator.
Table 6.4 shows the number of equivalent and killed mutants found by our runtime conformance checking approach for the two Aglets programs (27 Java files). Note that we manually determine whether a non-killed mutant is equivalent or not. Table 6.4 shows that almost 99% of the non-equivalent mutants have been killed by our approach. This analysis indicates that our runtime conformance approach is effective for monitoring mobile agent systems. Moreover, since only around 8% of all the generated mutants are equivalent, we also have some evidence that these mutation operators are effective for producing non-equivalent mutants (more than 83% of the 29 mutation operators did not produce any equivalent mutants). Only 6 non-equivalent mutants that are generated by the RPSM operator are not killed. In
all these mutants, the time \( t \) for waiting a reply from another agent was removed.

Removing this time parameter causes the agent to keep waiting until it gets a reply from the other agent. When we run the experiment, all these six mutants get a reply in time so that is the reason why these six mutants are not killed.

Table 6.4: Number of equivalent and killed mutants detected

<table>
<thead>
<tr>
<th>Operator</th>
<th>Mutant Generated</th>
<th>Equivalent</th>
<th>Non-Equivalent</th>
<th>killed</th>
<th>Killed Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDR</td>
<td>15</td>
<td>0 (0%)</td>
<td>15</td>
<td>15</td>
<td>15 (100%)</td>
</tr>
<tr>
<td>CDD</td>
<td>15</td>
<td>0 (0%)</td>
<td>15</td>
<td>15</td>
<td>15 (100%)</td>
</tr>
<tr>
<td>IDS</td>
<td>15</td>
<td>0 (0%)</td>
<td>15</td>
<td>15</td>
<td>15 (100%)</td>
</tr>
<tr>
<td>RAND</td>
<td>13</td>
<td>0 (0%)</td>
<td>13</td>
<td>13</td>
<td>13 (100%)</td>
</tr>
<tr>
<td>RDS</td>
<td>15</td>
<td>0 (0%)</td>
<td>15</td>
<td>15</td>
<td>15 (100%)</td>
</tr>
<tr>
<td>RDD</td>
<td>7</td>
<td>0 (0%)</td>
<td>7</td>
<td>7</td>
<td>7 (100%)</td>
</tr>
<tr>
<td>SHD</td>
<td>2</td>
<td>0 (0%)</td>
<td>2</td>
<td>2</td>
<td>2 (100%)</td>
</tr>
<tr>
<td>SDS</td>
<td>7</td>
<td>0 (0%)</td>
<td>7</td>
<td>7</td>
<td>7 (100%)</td>
</tr>
<tr>
<td>CMP</td>
<td>11</td>
<td>0 (0%)</td>
<td>11</td>
<td>11</td>
<td>11 (100%)</td>
</tr>
<tr>
<td>CMOW</td>
<td>8</td>
<td>0 (0%)</td>
<td>8</td>
<td>8</td>
<td>8 (100%)</td>
</tr>
<tr>
<td>RSMO</td>
<td>11</td>
<td>0 (0%)</td>
<td>11</td>
<td>11</td>
<td>11 (100%)</td>
</tr>
<tr>
<td>RPSM</td>
<td>50</td>
<td>11 (22%)</td>
<td>39</td>
<td>33</td>
<td>33 (84.62%)</td>
</tr>
<tr>
<td>MSKP</td>
<td>27</td>
<td>0 (0%)</td>
<td>27</td>
<td>27</td>
<td>27 (100%)</td>
</tr>
<tr>
<td>MSR</td>
<td>41</td>
<td>13 (31.71%)</td>
<td>38</td>
<td>38</td>
<td>38 (100%)</td>
</tr>
<tr>
<td>RSRM</td>
<td>44</td>
<td>13 (29.55%)</td>
<td>31</td>
<td>31</td>
<td>31 (100%)</td>
</tr>
<tr>
<td>MCMC</td>
<td>46</td>
<td>0 (0%)</td>
<td>46</td>
<td>46</td>
<td>46 (100%)</td>
</tr>
<tr>
<td>NAN</td>
<td>0</td>
<td>0</td>
<td>( - )</td>
<td>0</td>
<td>( - )</td>
</tr>
<tr>
<td>RMMM</td>
<td>2</td>
<td>0 (0%)</td>
<td>2</td>
<td>2</td>
<td>2 (100%)</td>
</tr>
<tr>
<td>RPRM</td>
<td>18</td>
<td>0 (0%)</td>
<td>18</td>
<td>18</td>
<td>18 (100%)</td>
</tr>
<tr>
<td>MICA</td>
<td>2</td>
<td>0 (0%)</td>
<td>2</td>
<td>2</td>
<td>2 (100%)</td>
</tr>
<tr>
<td>MFCA</td>
<td>25</td>
<td>0 (0%)</td>
<td>25</td>
<td>25</td>
<td>25 (100%)</td>
</tr>
<tr>
<td>ROCM</td>
<td>87</td>
<td>0 (0%)</td>
<td>87</td>
<td>87</td>
<td>87 (100%)</td>
</tr>
<tr>
<td>ACON</td>
<td>11</td>
<td>6 (54.55%)</td>
<td>5</td>
<td>5</td>
<td>5 (100%)</td>
</tr>
<tr>
<td>REN</td>
<td>80</td>
<td>0 (0%)</td>
<td>80</td>
<td>80</td>
<td>80 (100%)</td>
</tr>
<tr>
<td>RARL</td>
<td>10</td>
<td>0 (0%)</td>
<td>10</td>
<td>10</td>
<td>10 (100%)</td>
</tr>
<tr>
<td>RAID</td>
<td>8</td>
<td>3 (37.50%)</td>
<td>5</td>
<td>5</td>
<td>5 (100%)</td>
</tr>
<tr>
<td>CPCN</td>
<td>9</td>
<td>0 (0%)</td>
<td>9</td>
<td>9</td>
<td>9 (100%)</td>
</tr>
<tr>
<td>CNP</td>
<td>5</td>
<td>0 (0%)</td>
<td>5</td>
<td>5</td>
<td>5 (100%)</td>
</tr>
<tr>
<td>CSAP</td>
<td>3</td>
<td>0 (0%)</td>
<td>3</td>
<td>3</td>
<td>3 (100%)</td>
</tr>
<tr>
<td>Total</td>
<td>587</td>
<td>46 (7.84%)</td>
<td>541</td>
<td>535</td>
<td>(98.89%)</td>
</tr>
</tbody>
</table>
6.4 Comparison with Aspect-Oriented Runtime Monitoring

In this section we compare our runtime conformance checking approach with a different monitoring approach. In this approach, the monitor code is not contained in a separate process, but rather in the unit under test itself. Aspect languages such as AspectJ [33] have been proposed to implement this approach [56, 20]. We will call it aspect-oriented monitoring. Aspect-oriented programming aims at managing crosscutting concerns. The monitoring information is formulated as a separate aspect. These aspects are woven into the program code at runtime. AspectJ is a variant of Java. The aspect Java files contains a set of pointcuts. A pointcut is a predicate that matches join points. A join point is a point during the execution of a program, such as the execution of a method or the handling of an exception. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). Different types of advice include “around” “before” and “after” advice.

We developed some pointcuts for both Aglet programs (OnlineSearching and SigningContract) in order to monitor them using AspectJ. We compared our approach with aspect-oriented approach based on the following three criteria:

1. **Lines of code:** Table 6.5 shows the size in lines of code of the Java files of the original code without any modification, after applying our approach, and after applying the aspect-oriented approach. Based on the size of lines of code we can see that our approach requires less instrumentation than the aspect-oriented approach.
Table 6.5: A comparison between our approach and aspect-oriented approach based on lines of code

<table>
<thead>
<tr>
<th>File name</th>
<th>original code</th>
<th>with our approach</th>
<th>with AspectJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent.java</td>
<td>212</td>
<td>262</td>
<td>391</td>
</tr>
<tr>
<td>Cline.java</td>
<td>57</td>
<td>82</td>
<td>113</td>
</tr>
<tr>
<td>Notary.java</td>
<td>87</td>
<td>180</td>
<td>409</td>
</tr>
<tr>
<td>Contract.java</td>
<td>132</td>
<td>201</td>
<td>423</td>
</tr>
</tbody>
</table>

2. **Mutant detection:** We developed a set of pointcuts for both examples in order to monitor the behavior of these systems. Surprisingly, we found that mutation testing does not work very well with the aspect-oriented approach. In AspectJ a piece of advice is a block of code that is executed when a pointcut matches a join point. Suppose we have an advice A used for monitoring which is executed after the join point J. Suppose J is a join point for the execution of the method that has the name M in the original program. So after the execution of method M, the aspect code in advice A will be executed. Suppose we have a mutation operator used to rename the method M into $M'$ (e.g., the RPRM operator in Chapter 5). In this case, the monitoring code in advice A will never be executed because join point J does not exist anymore since this join point is for method M and method M has been renamed into $M'$. In this case, once this method is executed, it will not be monitored. Because of that some of the mutants that have been killed by our approach are not killed by the aspect-oriented approach.

3. **Changes in implementation:** The aspect-oriented approach suffers from the severe limitation that pointcuts cannot be attached to arbitrary variable assignments inside a method.
CHAPTER 6. EVALUATION AND COMPARISON

6.5 Chapter Summary

We have presented an evaluation framework that uses mutants that have been automatically generated from two different Aglets programs. The purpose of this framework is to check the effectiveness and the efficiency of our runtime conformance checking approach that has been presented in Chapter 3 and Chapter 4. Moreover, we also compared our approach with an aspect-oriented monitoring approach.

We presented the result of the evaluation and we conclude that our approach is very effective in killing the non-equivalent mutants with respect to the programs used. Moreover, we showed that most of the mutation operators did not generate any equivalent mutants. With respect to the efficiency, we do not yet have comprehensive performance data indicating, e.g., how much of a performance penalty was introduced by the instrumentation. However, all of the conformance checks described above completed in less than a 90 seconds and the analysis of the 587 mutants took less than 15 hours. Our experiments suggest that our approach will scale to larger and perhaps even industrial size agent systems.

We have to note here that our experiments are based only on two mobile agent programs, so we cannot conclude that the results generalize to all mobile agent systems. The two examples are not that large in size. However, the experiment shows the ability of our approach to detect bugs and improve the quality of mobile agent systems.
Chapter 7

Conclusion

This chapter summarizes and concludes this thesis. Section 7.1 summarizes the contribution of this thesis. Section 7.2 outlines the limitations of our work and future research directions.

7.1 Summary

In Chapter 1 we motivated this work by arguing that testing mobile agent systems is a challenging open problem. Moreover, we also sketched the kiltera high-level programming language and our runtime monitoring technique which are used to build a prototype that helps us to validate these kinds of systems.

We reviewed the background and related work in Chapter 2. We described the kiltera modeling language in detail. Moreover, we gave a brief description of the Aglets mobile agent programming language. Furthermore, a survey of existing approaches to monitoring and testing mobile agent systems has been given in this chapter. Moreover, a brief description of program mutation was provided.
In this thesis, we have described a new approach for checking the conformance of a mobile, distributed application with respect to an executable model at runtime (Chapter 3). Our approach has four steps:

1. Construction of the high-level model of the system,

2. Translation of the high-level model into a kiltera model

3. Instrumentation of the implementation under test and kiltera model using the high-level model, and

4. Execution of the instrumented implementation under test and the instrumented kiltera model in parallel

We have demonstrated the effectiveness of our approach using four different examples: two mobile agent systems and two distributed algorithms (Chapter 4). In Chapter 5 we have identified and implemented a set of new mutation operators for mobile agent systems and classified them into six different categories. To check the efficiency and the effectiveness of our runtime conformance checking approach we proposed a mutation-based framework (Chapter 6). We have used this framework to automatically and efficiently generate and run the mutants that have been automatically generated from larger versions of the two mobile agent systems presented in Chapter 3. The results of our evaluation suggest that our approach is very effective in killing the non-equivalent mutants with respect to the programs used. Moreover, we showed that most of the mutation operators did not generate any equivalent mutants. We also compared our approach with runtime monitoring based on aspects. We found that our approach requires less instrumentation. Additionally, we observed
that the standard mutation testing techniques do not work well on aspect-oriented languages.

Our approach has the following features:

- Models are expressed in a high-level language that has been specifically designed for the modeling of timed, mobile, and distributed systems; the language is an extension of Milner’s π-calculus [59], has been formally defined and its meta-theory has been studied [71], and has been implemented in an interpreter allowing distributed simulation.

- Models allow the operational description of the expected behaviour of the implementation under test which appears more suitable to capture complex interactions than declarative descriptions.

- Only very modest assumptions are made about the language that the agent system is implemented in. More precisely, the agent language only needs to support socket communication for our approach to be applicable.

- Although we were not able to evaluate the approach on industrial-size agent code, our experiments suggest that it should scale well to large agent systems.

### 7.2 Limitations and Future Work

We have showed the effectiveness of our approach in detecting bugs in distributed and mobile agent systems. However, it is also important to point out the limitations of our work and suggest future work to address them.
• **Further case studies:** The study presented in Chapter 6 demonstrates the ability and the effectiveness of our prototype in improving the quality of mobile agent systems. However, to get a stronger and more general conclusion more experiments are necessary, ideally using agent systems developed in industry.

• **Enforcing more than safety properties:** Following Schneider’s work [76], we are bound to safety properties. In [76], the foundations of monitoring frameworks are provided and the question of which security policies can be enforced by monitors is investigated. Schneider focuses on monitors which observe the execution of a target program with no knowledge of its possible future behavior and with no ability to affect the execution except by aborting it. He found that a monitor can enforce only safety properties, which are informally characterized by prohibiting a certain bad thing from occurring in a given execution. However, recent work has identified circumstances under which the expressiveness of monitoring can be increased [38]. The applicability of this work to our approach is currently unclear and would have to be checked in future work.

• **Automatic instrumentation:** In this approach the instrumentation process was performed by hand. Using tools that automate the instrumentation process is preferable to instrumenting code by hand, since instrumenting code by hand can be time-consuming and error-prone. Automating at least parts of the instrumentation should be possible and is another topic for future work.

• **Test input generation:** At the moment, our approach does not help the user find appropriate test inputs. It would thus be more comprehensive, if the kiltera models could also be used to generate test inputs — a topic that
is well-researched in the context of model-based testing (albeit not on \textit{kiltera} models).

- \textbf{Model Translation}: The translation of the high level model to a \textit{kiltera} model was also performed manually. Thanks to the good support in \textit{kiltera} for the features in Swimlaned Mobility Diagrams in particular and UML Sequence Diagrams in general, the automation of this translation should be possible.

- \textbf{Formal conformance relation}: Although inspired by and similar to existing process algebraic simulation relations, the conformance relation used in our approach is currently not formally defined which suggest another avenue for future work.

- \textbf{Correctness of the model and the instrumentation}: Just like most other model-based testing approaches, our approach assumes the correctness of the model and the instrumentation. However, the executability of our models mitigates the issue through support of model debugging and analysis.
Bibliography


Ravi Jain, Farooq Anjum, and Amjad Umar. A comparison of mobile agent and client-server paradigms for information retrieval tasks in virtual enterprises. In


[72] Antonio Puliafito, Salvatore Riccobene, and Marco Scarpa. Which paradigm should I use? An analytical comparison of the client-server, remote evaluation


Appendix A

The HLMs and the KMs of the Online Shopping Example

The following represents the HLMs and the KMs of online shopping example with two malls which have two shops and one shop respectively presented in Chapter 4:
APPENDIX A. THE HLMS AND THE KMS OF THE SHOPPING EXAMPLE

Figure A.1: The SMD of overall online shopping system

Figure A.2: The SMD of the initialization process
Figure A.3: The SMD of the client

Figure A.4: The SMD of the Mall1 with two shops
APPENDIX A. THE HLMS AND THE KMS OF THE SHOPPING EXAMPLE 144

Figure A.5: The SMD of the agent

:module Main:
: 2sites Home, Mall1site, Mall2site in
  3dchannel to_cust, mall1_info, mall2_info in
  4 par
  5 move Client[to_cust] to Home
  6 move Mall1[mall1_info] to Mall1site
  7 move Mall2[mall2_info] to Mall2site
  8 trigger to_cust with [(Mall1site, mall1_info), (Mall2site, mall2_info)].

Figure A.6: The KM of the initialization process of the shopping example
APPENDIX A. THE HLMS AND THE KMS OF THE SHOPPING EXAMPLE

Figure A.7: The KM of the client

```
module Client [from_main]:
when from_main with [(mallsite, mall_info); rest] →
  dchannel from_agent in
  par
  move Agent [from_agent, mall_info]((rest, null, null) to mallsite
when from_agent with (price, shop_ch) →
  // print where the Agent found the min price
```

Figure A.8: The KM of the Mall1 with two shops

```
module Mall1 [info_link]:
  process Shop [query, buy]((product, price, speed):
  when
    query with (item, response_ch) →
    wait speed
    if item = product then
      trigger response_ch with (price)
    | buy with (item, amount) →
      par
      print 'sold ' item ' for ' amount.
  Shop [query, buy]((product, price, speed)
  process InfoKiosk [question]((list):
  when question with ('get shop list', response_ch) →
    trigger response_ch with list
  InfoKiosk [question]((list)
  in
  channel s1, s2, buy in
  par
  Shop [s1, buy]('camera', 100.0, 2)
  Shop [s2, buy]('tv', 75.0, 3)
  InfoKiosk [info_link]((s1, s2))
```
module Agent [ to_customer , to_mall ] ( mall_list , best_price , best_shop ) :
  channel response_ch in
  par
    trigger mall_info with ( 'get shop list ' , response_ch )
    when response_ch with shop_list ->
    QueryShops [ shop_list ] ( best_price , best_shop )

  process QueryShops [ shop_list ] ( best_price_here , best_shop_here ) :
    match shop_list with
    | [] ->
    | GoToNextMall [] ( best_price_here , best_shop_here )
    | [ shop_ch ; rest ] ->
    | event shop_visited in
    EnterShop [ shop_ch , shop_visited ] ( best_price_here , best_shop_here )
    when shop_visited with ( new_best_price , new_best_shop ) ->
    QueryShops [ rest ] ( new_best_price , best_shop )

  process EnterShop [ shop_ch , shop_visited ] ( best_price , best_shop_here ) :
    channel response_ch in
    par
      trigger shop_ch with ( 'camera' , response_ch )
      when response_ch with price ->
      if price < best_price then
        trigger shop_visited with ( price , shop_ch )
      else
        trigger shop_visited with ( best_price , best_shop_here )
      timeout t ->
      trigger shop_visited with ( best_price , best_shop_here )
  
  process GoToNextMall [] ( best_price , best_shop ) :
    match mall_list with
    | [] ->
    | trigger to_customer with ( best_price , best_shop )
    | [ ( mall_site , mall_info ) ; rest ] ->
      move Agent [ to_customer , mall_info ] ( rest , best_price , best_shop ) to mallsite
    in
    GetShopList [ to_mall ]

Figure A.9: The Agent KM
Appendix B

The Kiltera Model of Lamport’s Mutex Algorithm with Instrumentation

The following represents the KM of Lamport’s mutex algorithm with three processes A, B and C:

```plaintext
module LamportMutexAlg:
  python java_listener
  let ignore = initialize()
  in
    function enqueue(queue, item):
      match queue with
        [ ] ->
        [ item ]
        | [ hd; tl ] ->
          [ hd; enqueue(tl, item) ]
    function dequeue(queue):
      match queue with
```
APPENDIX B. KM WITH INSTRUMENTATION OF LAMPORT MUTEX ALG.

function peek(queue):
    match queue with
    [] ->
    []
    [hd; tl] ->
    tl

process Queue{put, get, top, empty} (queue):
    when
    put with data ->
        Queue{put, get, top, empty} (enqueue(queue, data))
    | get with answer_link ->
        trigger answer_link with peek(queue)
        Queue{put, get, top, empty} (dequeue(queue))
    | top with top_queue ->
        trigger top_queue with peek(queue)
        Queue{put, get, top, empty} (queue)
    | empty with answer ->
        match queue with
        [] ->
            trigger answer with true
            Queue{put, get, top, empty} (queue)
        | anything ->
            trigger answer with false.

process HandleJavaMessages{external, complete}:
    let data = receive_message()
    in
    trigger lasting external with data
    when complete ->
APPENDIX B. KM WITH INSTRUMENTATION OF LAMPORT MUTEX ALG.

HandleJavaMessages [external, complete]

process ReceiveRelease [release_i](name):
  event ack_r in
  when release_i ->
    par
    trigger ack_r.
    ClientReleaseAck [ack_r](name)

process ClientReleaseAck [receive_rel_ack](name):
  match name with
  'A' ->
    when receive_rel_ack ->
    done
  | 'B' ->
    when receive_rel_ack ->
    done
  | 'C' ->
    when receive_rel_ack ->
    done

process ClientRequest [request_i,ack1,ack2,ack3](Reqname,Recname):
  when request_i ->
    match Recname with
    'A' ->
      trigger lasting ack1 with Reqname.
    | 'B' ->
      trigger lasting ack2 with Reqname.
    | 'C' ->
      trigger lasting ack3 with Reqname.

process WhetherAinCS [from_EntCS, holding_token, answer](name):
  process NotListening []:
    when
    from_EntCS with 'A' ->
      Listening []
    | holding_token with 'A' ->
trigger answer with false.

process Listening():
  when holding_token with 'A'->
    lpar
      trigger answer with true.
    in
      WhetherAinCS[from_EntCS, holding_token, answer]('A')
  NotListening[]

process WhetherBinCS[from_EntCS, holding_token, answer](name):
  process NotListening[]:
    when
      from_EntCS with 'B'->
        Listening[]
        | holding_token with 'B'->
          trigger answer with false.
    process Listening():
      when holding_token with 'B'->
        lpar
          trigger answer with true.
        in
          WhetherBinCS[from_EntCS, holding_token, answer]('B')
      NotListening[]

process WhetherCinCS[from_EntCS, holding_token, answer](name):
  process NotListening[]:
    when
      from_EntCS with 'C'->
        Listening[]
        | holding_token with 'C'->
          trigger answer with false.
    process Listening():
      when holding_token with 'C'->
        lpar
          trigger answer with true.
        in
          WhetherCinCS[from_EntCS, holding_token, answer]('C')
APPENDIX B. KM WITH INSTRUMENTATION OF LAMPORT MUTEX ALG.

125 NotListening []

127 process SendYesToA[check_requestA](name):
128   if name="A" then
129     when check_requestA with answer->
130       trigger answer with true.

132 process SendYesToB[check_requestB](name):
133   if name="B" then
134     when check_requestB with answerb->
135       trigger answerb with true.

137 process SendYesToC[check_requestC](name):
138   if name="C" then
139     when check_requestC with answerc->
140       trigger answerc with true.

142 process Initial[external, complete, req_a, req_b, req_c, ack_a, ack_b, ack_c, rel_a, rel_b, rel_c]
143   (queue):
144   event who, put, get, answer, top, empty, all_ack, check_request, check_requestA, check_requestB,
145       check_requestC, AinCS, BinCS, CinCS, holding_token in
146 process ClientRequestAck [receive_req_ack1, receive_req_ack2, receive_req_ack3, all_ack]:
147   event check_req in
148   when
149     receive_req_ack2 with "A"->
150     when receive_req_ack3 with "A"->
151       lpar
152       Get_all_ack[all_ack]("A")
153       trigger lasting all_ack with ("A").
154       when receive_req_ack3 with "A"->
155     when receive_req_ack2 with "A"->
156       lpar
157       Get_all_ack[all_ack]("A")
158       trigger lasting all_ack with ("A").
159       when receive_req_ack1 with "B"->
when receive_req_ack3 with 'B'->
  lpar
  Get_all_ack[all_ack]('B')
  trigger lasting all_ack with ('B').

| receive_req_ack3 with 'B'->
  when receive_req_ack1 with 'B'->
    lpar
    Get_all_ack[all_ack]('B')
    trigger lasting all_ack with ('B').

| receive_req_ack1 with 'C'->
  when receive_req_ack2 with 'C'->
    lpar
    Get_all_ack[all_ack]('C')
    trigger lasting all_ack with ('C').

| receive_req_ack2 with 'C'->
  when receive_req_ack1 with 'C'->
    lpar
    Get_all_ack[all_ack]('C')
    trigger lasting all_ack with ('C').

process Get_all_ack[all_ack](name):
  when
    all_ack with ('A')->
      lpar
      SendYesToA[check_requestA]('A')
      HandleNextJavaMessage[]
    | all_ack with ('B')->
      lpar
      SendYesToB[check_requestB]('B')
      HandleNextJavaMessage[]
    | all_ack with ('C')->
      lpar
      SendYesToC[check_requestC]('C')
      HandleNextJavaMessage[]
APPENDIX B. KM WITH INSTRUMENTATION OF LAMPORT MUTEX ALG.

199  process EneterCS[](Pname):
200       event first , getyes , getyesb , getyesc in
201       lpar
202           trigger top with first .
203           when first with value =>
204               if value = Pname then
205                 match Pname with
206                   'A' =>
207                     par
208                        trigger check_requestA with getyes .
209                        when
210                           getyes with true =>
211                             seq
212                               print Pname "is in Critical Section" .
213                               trigger AinCS with 'A' .
214                               HandleNextJavaMessage []
215                     | 'B' =>
216                     par
217                        trigger check_requestB with getyes .
218                        when
219                           getyes with true =>
220                             seq
221                               print Pname "is in Critical Section" .
222                               trigger BinCS with 'B' .
223                               HandleNextJavaMessage []
224                     | 'C' =>
225                     par
226                        trigger check_requestC with getyes .
227                        when
228                           getyes with true =>
229                             seq
230                               print Pname "is in Critical Section" .
231                               trigger CinCS with 'C' .
232                               HandleNextJavaMessage []
233                       else
234                         seq
print "Error: The first element in the Queue is "' value' "'.

print "'But Java allow ' 'Pname ' ' To Enter CS'.

print "'Maybe the previous process is still in CS and does not release the token'.

process Putinqueue[] (name):
  trigger put with name.

process Getfromqueue[] :
  event first in
    trigger get with first.

process HandleNextJavaMessage[] :
  lpar
    trigger complete.
    Start []

process Start[] :
  event answer a, answer b, answer c in
  when external with data->
    match data with
      'RequestA' ->
        lpar
          ClientRequestAck [ack a, ack b, ack c, all ack]
          ClientRequest [req a, ack a, ack b, ack c]('A', 'B')
          ClientRequest [req a, ack a, ack b, ack c]('B', 'A')
          trigger all lasting req a.
          Putinqueue []('A')

      | 'RequestB' ->
        lpar
          ClientRequestAck [ack a, ack b, ack c, all ack]
          ClientRequest [req b, ack a, ack b, ack c]('B', 'A')
          ClientRequest [req b, ack a, ack b, ack c]('C', 'B')
          trigger all lasting req b.
          Putinqueue []('B')
| 'RequestC' | lpar
| ClientRequestAcknowledgement [ack_a, ack_b, ack_c, all_ack]
| ClientRequest ['req_c', ack_a, ack_b, ack_c] ('C', 'A')
| ClientRequest ['req_c', ack_a, ack_b, ack_c] ('C', 'B')
| trigger all lasting req_c.
| Putinqueue [] ('C')

| 'ReleaseA' | lpar
| trigger holding token with 'A'.
| when
| answer with true-
| seq
| lpar
| ReceiveRelease [rel_a] ('B')
| ReceiveRelease [rel_a] ('C')
| trigger all lasting rel_a.
| print 'A is Not in CS'.
| Getfromqueue []
| HandleNextJavaMessage []
| answer with false-
| print 'Error, A is not in CS to Release the token'.

| 'ReleaseB' | lpar
| trigger holding token with 'B'.
| when
| answer with true-
| seq
| lpar
| ReceiveRelease [rel_b] ('A')
| ReceiveRelease [rel_b] ('C')
| trigger all lasting rel_b.
| print 'B is Not in CS'.
| Getfromqueue []
| HandleNextJavaMessage []
| answer with false ->
| print "'Error, B is not in CS to Release the token'."

| 'ReleaseC' ->
lpars
| trigger holding_token with 'C'.
when
| answer with true ->
seq
| lpar
ReceiveRelease[rel_c]['A']
ReceiveRelease[rel_c]['B']
| trigger all lasting rel_c .
| print 'C is Not in CS'.
Getfromqueue[]
| HandleNextJavaMessage[]
| answer with false ->
| print "'Error, C is not in CS to Release the token'."

| 'okayA' ->
lpars
| Get_all_ack[all_ack]['A']
| EneterCS[]('A')

| 'okayB' ->
lpars
| Get_all_ack[all_ack]['B']
| EneterCS[]('B')

| 'okayC' ->
lpars
| Get_all_ack[all_ack]['C']
| EneterCS[]('C')

| Queue[put, get, top, empty]()
| WhetherAinCS[AinCS, holding_token, answer]('A')
WhetherBinCS[BinCS, holding_token, answer](‘B’)
WhetherCinCS[CinCS, holding_token, answer](‘C’)
wait 2->
Start[

in

event check_request, external, put, get, top, empty, complete, rel_a, rel_b, rel_c, req_a, req_b, req_c, ack_a, ack_b, ack_c, answer, from_EntCS, holding_token in

par
Queue[put, get, top, empty]()
wait 1->
par
HandleJavaMessages[external, complete]

Initial[external, complete, req_a, req_b, req_c, ack_a, ack_b, ack_c, rel_a, rel_b, rel_c]()