Implementing and Evaluating a Runtime Conformance Checker for Mobile Agent Systems

Ahmad A. Saifan[a], Juergen Dingel[b], Jeremy S. Bradbury[c], Ernesto Posse[b]
[a]Yarmouk University, Irbid, Jordan
[b]Queen's University, Kingston, ON, Canada
[c]University of Ontario Institute of Technology, Oshawa, ON, Canada
ahmads@yu.edu.jo, dingel@cs.queensu.ca, jeremy.bradbury@uoit.ca, eposse@cs.queensu.ca

Abstract

A Mobile Agent System (MAS) is a special kind of distributed system in which the agent software can move from one physical host to another. This paper describes a new approach, together with its implementation and evaluation, for checking the conformance of a MAS with respect to an executable model. In order to check the effectiveness of our conformance check, we have built a mutation-based evaluation framework. Part of the framework is a set of 29 new mutation operators for mobile agent systems. Our conformance checking approach is used to compare the mutated agents with the executable model and determine nonconformance. Our experimental results suggest that our approach holds promise for the generation and detection of non-equivalent mutants.

1. Introduction

In Mobile Agent Systems (MAS), the code of the agent can 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"* [6]. Mobility occurs naturally in many distributed system applications such as telecommunications and electronic commerce. Moreover, mobility may reduce bandwidth consumption and coupling as well as increase flexibility [23]. The use of mobility has reached a certain degree of maturity: several different development platforms are available (e.g., Aglets [14], Voyager [1], and Grasshopper [4]); agent-oriented software engineering (AOSE) has produced tool-supported development methodologies (e.g., Prometheus [21] and Tropos [5]); promising commercial applications exist [19], and standardization is being considered [22]. However, it seems that relatively little work has been done to support quality assurance techniques such as testing and verification of mobile systems [9, 28].

In this paper, we present our work on evaluating 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` [24]. 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 behaviour 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 (KM). The high-level model is then used to identify suitable "check points" at which conformance between the implementation under test (IUT) and the KM is to be checked. Check points typically occur right before or after agent movement or the sending or receipt of messages. After these check points have been located in the IUT and the KM, both are instrumented at these check points to allow relevant information to flow from the implementation under test to the KM. Finally, the IUT and the KM are both executed, possibly in a distributed fashion. The KM will report any non-conformance at the check points that arises during execution.

In addition to describing our conformance testing approach we also present our evaluation of the technique. Our evaluation is based on mutation testing [20]. *Mutation operators* specify a single syntactic change to a program and are used to generate a set of *mutants* from the implementation under test. We generate mutants for the IUT and check if our conformance test observes a difference between the behaviour of the mutant and that of the KM. In that case, the mutant is said to be *killed*. An initial version of the approach was first presented by Saifan et al. [26]. However, no evaluation (using mutation or any other technique) was

presented.

The paper is structured as follows. The next section discusses related work with an emphasis on approaches to runtime monitoring and to testing mobile code and mobile agent systems. Section 3 introduces the relevant parts of `kiltera`, and Section 4 describes our conformance checking approach. Some of our mutation operators are sketched in Section 5. Section 6 describes the mutation-based evaluation of the conformance checker before presenting evaluation results in Section 7. Limitations and future work are discussed in Section 8. Section 9 concludes.

2. Related Work

Runtime monitoring: Runtime Monitoring is the activity of checking the conformance between the target program (the implementation) and the requirements specification of that program during execution. Approaches and tools for runtime monitoring include Java-MaC [12, 11], Java PathExplorer [10], Java Runtime Timing-constraint Monitor [17], decentralized monitoring [27], and Java Monitoring-Oriented Programming [7]. All of these approaches are based on the same fundamental idea: The monitored code is instrumented (possibly automatically) such that it produces sequences of events during execution; event sequences are analyzed by a central component for specification violations. Our approach to conformance testing differs from the previous work, in that it supports runtime monitoring and possibly distributed analysis of mobile agents. Moreover, specifications are expressed using an executable modeling language, instead of more declarative specifications (e.g., using temporal logic). Given that agent interactions may be arbitrarily complex, this kind of operational specification appears more suitable for the description of agent interactions without sacrificing mathematical rigour.

Testing Mobile Agents: A more limited amount of existing work is devoted to the testing of mobile agents. Delamaro et al. [9], present a framework that is used to support testing of mobile Java agents with respect to the standard code coverage criteria. A formal framework for conformance testing of mobile agents using labeled transition systems is presented by Marche and Quemener [15]. Unlike the Marche and Quemener work, our approach is based on an extension of the π-calculus, which appears much more suitable due to the explicit support for mobility.

Agent-oriented software engineering (AOSE) is concerned with supporting the effective construction of reliable agent systems. Most AOSE methodologies (such as Prometheus [21]) 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 [29, 18]. The work by Alberti et al. [2] 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.

3. `kiltera`: An Executable Modelling Language

`kiltera` is a novel high-level language [24]. While not specifically designed for conformance checking of MAS, we find that `kiltera` is ideally suited for this purpose, due to the following features: (1) It allows a straight-forward, high-level expression of timed, concurrent, mobile, interacting processes which may be distributed over several *sites* (also called hosts or locations); more precisely, 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; (2) `kiltera` has a formal semantics and a meta-theory [24] which is based on an extension of Milner's π-calculus [16]; (3) unlike other process algebras, `kiltera` provides some higher-level constructs to facilitate development (e.g., complex expressions and data-structures can be used in messages, and pattern-matching can be used to extract information from data); (4) finally, the `kiltera` simulator[1] supports both uniprocessor and truly distributed simulation of a model. In the following, we informally introduce the subset of `kiltera` that is most significant for our work.

A model (sometimes also called specification or program) in `kiltera` consists of one or more *modules* — the smallest movable processing unit. Each module has the syntax: `module` $A[\tilde{x}](\tilde{y}) : P$ or `module` $A[\tilde{x}](\tilde{y}) :$ `sites` $\tilde{s}$ `in` P. Here P ranges over *process terms*, defined below. We use $x, x_i, ...$ for *port/channel/event names*, and $A, B, ...$ for *process/module names*, s, s_i for *site names*, and $y, y_i, ...$ for any other variable name. The notation $\tilde{x}$ denotes a list of names or values $x_1, ..., x_n$. In the definition of a module, the names $\tilde{x}$ represent the interface of the module, that is, its ports (channels), 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 behaviour of the module.

The syntax for process terms P is shown in Figure 1. Here E ranges over *expressions*, F ranges over *patterns*,

[1] available at `www.kiltera.org`.

$$
\begin{array}{lcl}
P & ::= & \texttt{done} \\
 & | & \texttt{trigger}\ x\ \texttt{with}\ E \\
 & | & \texttt{when}\ \beta_1 \rightarrow P_1 \mid \cdots \mid \beta_n \rightarrow P_n \\
 & | & \texttt{event}\ \tilde{x}\ \texttt{in}\ P \\
 & | & \texttt{wait}\ E \rightarrow P \\
 & | & \texttt{par}\ \{P_1, ..., P_n\} \\
 & | & \texttt{process}\ A[\tilde{x}](\tilde{y}) : P_1\ \texttt{in}\ P_2 \\
 & | & A[\tilde{x}](\tilde{E}) \\
 & | & \texttt{move}\ A[\tilde{x}](\tilde{y})\ \texttt{to}\ s \\
 & | & \texttt{here}\ s\ \texttt{in}\ P \\
 & | & \texttt{dchannel}\ \tilde{x}\ \texttt{in}\ P \\
\beta & ::= & x\ \texttt{with}\ F\ \texttt{after}\ y \\
E & ::= & n \mid \texttt{true} \mid \texttt{false} \mid \text{“}s\text{”} \mid \\
 & | & x\ op\ E \mid E_1\ op\ E_2 \mid f(E_1, ..., E_m) \\
 & | & (E_1, ..., E_m) \\
F & ::= & n \mid \texttt{true} \mid \texttt{false} \mid \text{“}s\text{”} \mid x \\
 & | & (F_1, ..., F_m)
\end{array}
$$

Figure 1: `kiltera` syntax

$op \in \{+, -, *, /, \mathsf{mod}, \mathsf{and}, \mathsf{or}, \mathsf{not}, <, >, =, <=, >=, != \}$, 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 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* β_i is of the form "x_i `with` F_i `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. The process "`event` $\tilde{x}$ `in` P", also written "`channel` $\tilde{x}$ `in` P", creates new events (or channels) $\tilde{x}$ within the scope of P. The process "`wait` $E \rightarrow P$" delays the execution of process P by an amount of time equal to the value of the expression E. The process "`par` $\{P_1, ..., P_n\}$" is the parallel composition of $P_1, ..., P_n$. The process "`process` $A[\tilde{x}](\tilde{y}) : P_1\ \texttt{in}\ P_2$" 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 "`move` $A[\tilde{x}](\tilde{y})$ `to` s" creates an instance of the process defined by module A in site s. The process "`here` s `in` P" binds the name of the local site to s in P. Finally, the process "`dchannel` $\tilde{x}$ `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.

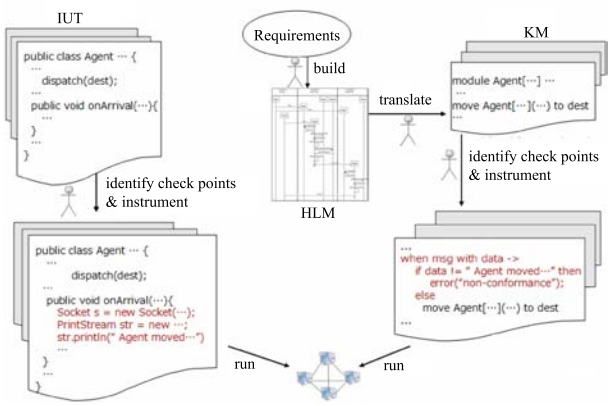


Figure 2: Our conformance testing approach

There are several derived process terms, such as sequential composition, timeouts, conditionals, etc. Here we only mention a few of them. The process "`seq` $\{P_1, ..., P_n\}$" is the sequential composition of $P_1, ..., P_n$. A timeout clause can be added to listeners: `when` $\cdots$`timeout` $E \rightarrow P$. The process "`match` E `with` $F_1 \rightarrow P_1 \mid \cdots \mid F_n \rightarrow P_n$" matches the value of E with a pattern F_i, and if successful, executes P_i. Finally, we also have terms of the form "`let` $y = E$ `in` P" which define local names in a process P. Simple examples of `kiltera` processes are given in the next section.

4. A Runtime Conformance Checker for MAS

Our conformance checking approach consists of the following four steps:

1. Construction of the high-level model (HLM) of the agent system.
2. Translation of the HLM into a `kiltera` model (KM).
3. Instrumentation of the IUT and the KM with the help of the HLM.
4. Simultaneous execution of the instrumented IUT and the instrumented KM.

Figure 2 shows the steps of our approach assuming the IUT is implemented using Aglets [14][2]. We will now describe each of these steps in detail.

[2]The Aglets Software Development Kit (ASDK) is a Java-based framework and environment for developing and running mobile agents. It was

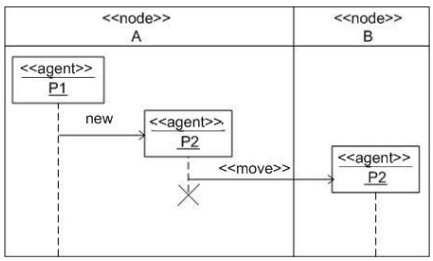


Figure 3: A simple Swimlaned Mobility Diagram.

In this diagram agent P1, located on node A, creates a new agent P2, also located on node A. Later agent P2 moves from node A to node B.

4.1. Construction of HLM

We first create a high-level model (HLM) of the agent system. The HLM constructed is assumed to capture the most relevant aspects of the system behaviour such as 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 by Kusek and Jezic [13] as our high-level modelling language. This profile extends the UML with a few new types of Sequence Diagrams. Here we use only one of these types, called *Swimlaned Mobility Diagrams* (SMDs). These diagrams are intended to represent agent location, agent creation and agent movement.

An SMD consists of one or more *swimlanes* representing *nodes* (a.k.a. *sites*, *hosts* or *locations*) (see Figure 3). 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 new, and 2) arrows that represent agent movement between nodes, labelled move. Message arrows between life-lines in different swimlanes represent remote communication.

4.2. Translation of HLM to KM

In the second step, the HLM is translated into a `kiltera` model (KM) (see Figure 4). Since `kiltera` has direct support for many relevant features including concurrency with synchronous and asynchronous message passing, movement of processes, time, and site-dependent behaviour, this translation is relatively straight-forward.

originally developed at the IBM Tokyo Research Laboratory. An Aglet is a Java agent able to autonomously and spontaneously move from one host to another.

```
module P2[x](state):
sites A, B
// use and/or modify state yielding new_state
here s in
  match s with
      A -> move P2[x](new_state) to B
    | B -> //...do what needs to be done in B
```

Figure 4: The `kiltera` model of agent P2 in Figure 3

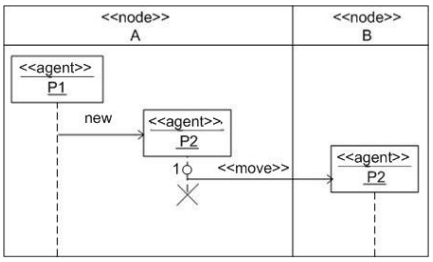


Figure 5: Check points locations in SMD

```
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 at new destination
  }
  ...
```

Figure 6: Code inserted into the IUT of agent P2

4.3. Instrumentation of IUT and KM

Next, 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 sends information to the KM. We use information in the HLM to identify a collection of "check points" at which conformance between the IUT and the KM is to be assessed (see Figure 5).

We manually locate these check points in the IUT and the KM. At each check point in the IUT, we insert the appropriate instrumentation code which will transmit relevant state information to the KM via Java sockets through a connector. 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 (see Figures 6 and 7).

```
// 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, report non-conformance
// inserted code end
```

Figure 7: Code inserted into KM of agent P2

4.4. Execution of IUT and KM

Finally, we execute the instrumented IUT and the instrumented KM, both possibly in a distributed fashion, and check conformance. While the IUT is running, it sends messages, possibly including state information, to the KM through the connector. When the KM receives a message from the IUT, 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 a trace of the execution.

5. Agent-based Mutation Operators

To evaluate how well our approach can detect non-conforming mobile agent programs we use mutation testing. In this section we propose six categories of mutation operators for Aglets MAS (see Table 1). These categories contain mutation operators that impact the movement, communication, run method, creation, event listeners, or the agent proxy of an agent. The purpose of these operators is to insert bugs into the IUT that represent mistakes that programmers may make when they implement mobile agent systems using Aglets. Due to space limitations, only some of these operators are presented in this section. For more details on all 29 of the mutation operators see [25].

Brief introduction to Aglets. To understand the operators, some information about Aglets is necessary. The execution environment for an Aglet is called *context*. An Aglet is created, exists, works, sleeps, and dies in such a context. The invocation of a `run` method starts the agent. Agents can migrate from one context to another (using the `dispatch` method), be destroyed (`dispose`), be cloned (`clone`), be deactivated for a certain amount of time (`deactivate`), or be moved back to the context that created it (`retract`).

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.

Event listeners can be registered for each of the major events in an agent's life. Three types of events exist:

Table 1: Mobile agent mutation operators

Operator Category	Mutation Operator
Mobility	CDD: Change Dispatch Destination
	IDS: Insert Deactivate Statement
	RAPD: Remove Aglet Proxy from Dispatch statement
	RDS: Remove Dispatch Statement
	RDD: Replace Dispatch with Dispose
	RDR: Replace Dispatch to Retract
	SICD: Shrink ifelse Containing Dispatch
	SDS: Switch Dispatch Statement
Communication	CMP: Change Message Parameter
	CMOW: Change Message to One Way message
	RSMM: Remove Send Message Method
	MSKP: Modify SameKind Parameter
	MSR: Modify SendReply Parameter
	RPSM: Remove a Parameter from a Set of Methods
	RSRM: Remove SendReply Method
	MCMC: Move the Communication Method Calls in ifElse
	NAN: Notify All message to Notify message
Agent's Run Method	RMRM: ReMove Run Method
	RPRM: RePlace Run Method
Agent Creation	MICA: Modify Create Aglet Parameter
	MFCA: Modify the File name in Create Aglet
	ROCM: Replace onCreation with other Method
	ACON: Add clone method in onCreation
Event Listeners	RCBMN: Replace CallBack Method Name
	RARL: Replace Add listener with Remove Listener
Agent Proxy	RAID: Remove AgletProxy from getAgletID
	CPCN: Change Proxy name in getAgletClassName
	CSAP: Change the State in getAgletProxies
	CNP: Change Number of Proxies

(1) mobility events such as *onDispatching* (raised right before an agent is dispatched to a new context), and *onArrival* (raised right after an agent has arrived at a new context); (2) clone events such *onCloning* (raised right before an agent is cloned) and *onCloned* (raised right after an agent has been cloned); and (3) persistence events including *onCreation* (raised right before agent is created; raised only once); *onDispose* (raised right before agent is destroyed), *onActivation* (raised right before agent is activated), and *onDeactivation* (raised right before agent is deactivated). To receive one of these events, an event listener of the appropriate type needs to be registered. Event listeners contain callback methods which are executed when an event is received and allow the agent to prepare for the event. The names of these callback methods coincide with the names of the events they are bound to. For instance, the callback method `onCreation` and `onDispose` are used for agent initialization and finalization, respectively. These callback methods can be modified to customize the behaviour of an agent.

An `AgletProxy` object is used to control and limit direct access to an agent. For example, if an agent A wants to create, migrate, communicate with, or destroy another agent B, then A needs to obtain B's proxy object first.

5.1. Mobility Mutation Operators

The `dispatch` method is used to migrate an agent. The dispatch destination is provided as an argument in the form of the URL of the context that the agent is supposed to move to. The mutation operators in this category affect the dispatch process. Examples of modifications include: change dispatch destination, remove the dispatch statement, remove the agent proxy from dispatch statement, delay the dispatch by temporarily deactivating the agent, replace the dispatch by a retraction, swap the dispatch statement with the statement immediately preceding it, or pull the dispatch out of the scope of a conditional (if applicable). Below is an example of one of these mutation operators.

Change Dispatch Destination (CDD). An agent can be moved to some remote context by invoking `dispatch` on the agent's proxy object with the URL of the remote context as the parameter. The URL describing the dispatch destination should specify the host and domain names of the destination context, and request the use of the Agent Transfer Protocol (ATP)[3]. The CDD operator replaces the destination of the `dispatch` method call by some another destination. For example:

Original Code:

```
dispatch(new URL("atp://balboa"));
```

CDD Mutant:

```
// mutation: modified destination
dispatch(new URL("atp://polka"));
```

5.2. Communication Mutation Operators

In Aglets, agents communicate by exchanging message objects. These message objects are sent using the `AgletProxy` class methods. There are three types of messages that can be sent: now-type (a synchronous messages which blocks execution of the sender until the receiver has handled and acknowledged the message), future-type (an asynchronous message which does not block the sender; the sender has a future handler for obtaining an acknowledgement or result), and a oneway-type (an asynchronous and unacknowledged message). An agent handles incoming message through the `handleMessage` method. Below we provide an example for a mutation operator affecting agent communication.

Modify SameKind Parameter (MSKP). The method `sameKind` allows a receiving agent to distinguish different kinds of messages. It takes a message key as parameter that is compared with the key of the incoming message. The MSKP operator is used to modify the parameter of the `sameKind` method call. For example:

Original Code:

```
boolean handleMessage(Message msg){
   if (msg.sameKind("Get_Mall_Address")) {...}
   ...
}
```

MSKP Mutant:

```
boolean handleMessage(Message msg){
   // mutation: modified message
   if (msg.sameKind("Get_Item_Price")) {...}
   ...
}
```

5.3. Mutation Operators for Agent's Run Method

The `run` method is the entry point for the agent's own thread of execution. The method is invoked upon successful creation, dispatch, cloning, retraction or activation of the agent. Below is an example for a mutation operator affecting the `run` method.

Replace Run Method (RPRM). In Aglets there are several callback methods (e.g., `onDispatching`, `onCreation`, `onArrival`). The RPRM operator swaps the name of the `run` method with the name of a callback method. For example:

Original Code:

```
public void run() {
  // code for run
}
public void onArrival(MobilityEvent e) {
  // code for onArrival
}
```

RPRM Mutant:

```
// mutation: replaced run with onArrival
public void onArrival(MobilityEvent e) {
  // code for run
}
// mutation: replaced run with onArrival
public void run() {
  // code for onArrival
}
```

When an agent is dispatched, the callback methods `run`, `onDispatching`, and `onArrival` are executed in this order: 1) `run` (to start agent), 2) `onDispatching`, 3) `onArrival`, and 4) `run` (to restart agent in the new context). In the mutated agent, however, the *onArrival* event is now handled by the code for `run` and the code for the `onArrival` method is now used to start and restart the agent. In other words, the order in which the code handling each event is executed, changes to 1) `onArrival`, 2) `onDispatching`, 3) `run`, and 4) `onArrival`.

[3]The Agent Transfer Protocol is an application-level protocol for distributed agent-based systems. It offers a simple protocol for transferring agents between networked computers.

5.4. Agent Creation Mutation Operators

The `createAglet` method 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. The method has three parameters: `codeBase`, `code`, and `init`. The `codeBase` parameter specifies the (possibly remote) directory of the agent class. If it is `null`, the context will search for the code in the local system's Aglet search path. The `code` parameter represents 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 mutation operators in this category primarily modify the `createAglet` method by, e.g., modifying the code parameter, replacing the `init` value (if it exists) with `null`, replacing the `onCreation` callback method with other callback methods or adding an invocation of the `clone` method in the `onCreation` callback method. The following is one of the mutation operators in this category.

Modify the File Name in Create Aglet (MFCA). The MFCA operator is used to change the `code` parameter in the `createAglet` method call. For example:

Original Code:

```
Prox= c.createAglet(null,"BuyingAgent",getProxy());
```

MFCA Mutant:

```
// mutation: modified agent type
Prox= c.createAglet(null,"SellingAgent",getProxy());
```

Applying the MFCA operator changes the agent that we are going to create. If no agent class of that name exists, an exception is raised at runtime. Otherwise, the agent system may start communicating with this incorrectly created agent.

5.5. Mutation Operators for Event Listeners

To receive a specific event for an agent, the appropriate event listener must be added to that agent. The callback method in the event listener corresponding to the event then allows the agent to take action when one of these events has occurred. Mutations in this category modify the name of a callback method or cause a listener to be removed rather than added.

Replace CallBack Method Name (RCBMN). The RCBMN operator is identical to the RPRM operator except it replaces the name of a callback method with that of another callback method.

Original Code:

```
public void onCloning() {code for onCloning callback}
```

RCBMN Mutant:

```
// mutation: replaced name of callback method
public void onCloned() {code for onCloning callback}
```

After using the RCBMN operator, the response to an event may be incorrect.

5.6. Mutation Operators for Agent Proxy

Agent proxies are necessary to, e.g., dispatch an agent, to obtain the id of an agent (every agent has a unique id) using `getAgletID`, or the name of the class the agent belongs to using `getAgletClassName`. The proxy of an agent in some context can be obtained by invoking the method `getAgletProxies` on the context. The operators of this category make changes that are related to agent proxies and their associated methods.

Change the State in getAgletProxies (CSAP). The method `getAgletProxies` has one parameter describing the state of an agents whose proxies are to be retrieved. The possible values of this parameter are ACTIVE, INACTIVE or ACTIVE/INACTIVE with ACTIVE being the default. The CSAP operator modifies the value of this parameter. For example:

Original Code:

```
Enumeration e;
e=AgletContext().getAgletProxies(ACTIVE);
```

CSAP Mutant:

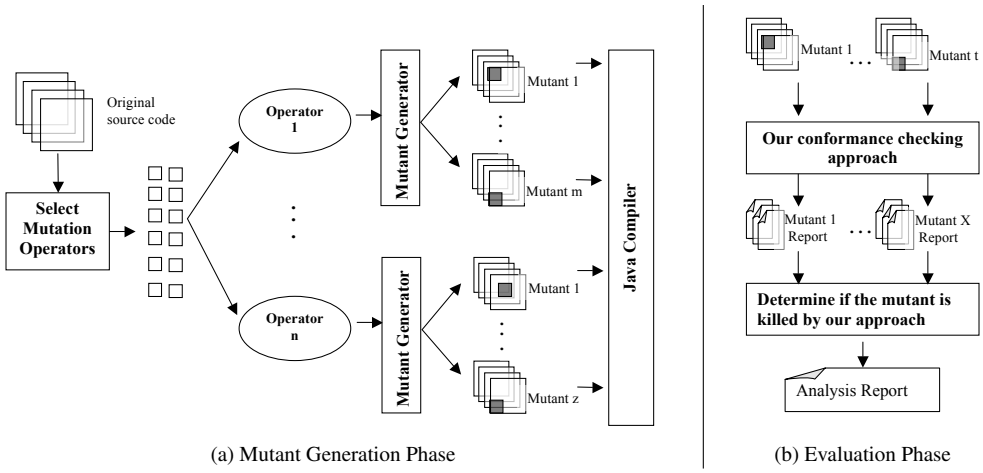
```
Enumeration e;
// mutation: modified state
e=AgletContext().getAgletProxies(INACTIVE);
```

The CSAP operator thus causes the wrong proxies to be retrieved.

6. Evaluating the Runtime Conformance Checker

To evaluate the effectiveness of our runtime conformance checking approach we have developed a mutation-based evaluation framework (see Figure 8). This framework automatically runs mutants that have been automatically generated from the IUT using the mutation operators described in Section 5. The framework consists of two phases, the *Mutant Generation Phase* and the *Evaluation Phase*.

Mutant Generation Phase: First, some mutation operators are selected by the user and then automatically applied



(a) Mutant Generation Phase (b) Evaluation Phase

Figure 8: The mutation-based evaluation framework

to the IUT. Our generator uses a TXL-based mutation process [8] to mutate code of the IUT. If the IUT contains a set of files, then a set of mutants is created 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 generate more than one mutant for a given source file, we distinguish between them by adding a number to the name of the directory. Then, the generator compiles all mutant files. If a mutant does not compile, it is discarded.

Evaluation Phase: Every mutant created in the previous phase is executed simultaneously with the `kiltera` model KM. If the runtime behaviour of the mutant does not match that of the KM and the KM detects non-conformance, analysis of the mutant is stopped and the mutant is considered killed. Otherwise, it is terminated after a user-specified amount of time (since mutants may not terminate on their own). In each case, an appropriate report summarizing the analysis result is generated. This process is repeated for each mutant. After all mutants have been analyzed, a summary report is automatically generated listing for each mutant whether it was killed or not and some additional statistics such as the number of mutants generated and killed.

7. Experiment

In our experiment we used all of the 29 mutation operators listed in Table 1. We used our framework to automatically generate the mutants for two mobile agent systems implemented in Aglets — an Online Shopping example [26] and a Contract Signing example [25]. The Online Shopping example is a MAS in which an agent is searching for a specific item to be purchased (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 Contract Signing example is a MAS in which several distributed agents representing partners have to sign a contract (represented by a mobile agent) for a specific project. A notary agent coordinates and supervises the process.

Results: A few details of the application of our approach to the Contract Signing example with two partner agents follow: Step 1 of our approach (construct HLM) results in the specification of the notary agent given in Figure 9. Due to space limitations, the result of Step 2 (translate HLM into KM), the notary's KM, is not shown (see [25] instead). While the Java Aglets code (i.e., the IUT) is 361 lines long, the KM has just 90 lines. Step 3 of our approach (instrument IUT and KM) is carried out in such a way that the conformance test checks, e.g., that partners sign in time, that the movement of the contract is correct, and that the notary's check of the contract at the end produces the correct result. After instrumentation, the IUT has 523 lines and the KM just 179 lines. In the Online Shopping example, the KM is more succinct than the IUT by a similar factor.

Table 2 provides the results of applying our framework to both example systems. We see that a total of 587 mutants were generated. Manual inspection revealed that 46 (or about 8%) of these mutants were equivalent. 99% of the non-equivalent mutants have been killed by our approach. Only 6 non-equivalent mutants are not killed. All of them were generated by the RPSM operator. In all of these mutants, the maximum wait time for receiving a reply from another agent was removed. Removing this time parameter causes the agent to keep waiting for a reply indefinitely. However, during execution, all these six mutants receive a reply before their behaviour becomes non-conformant.

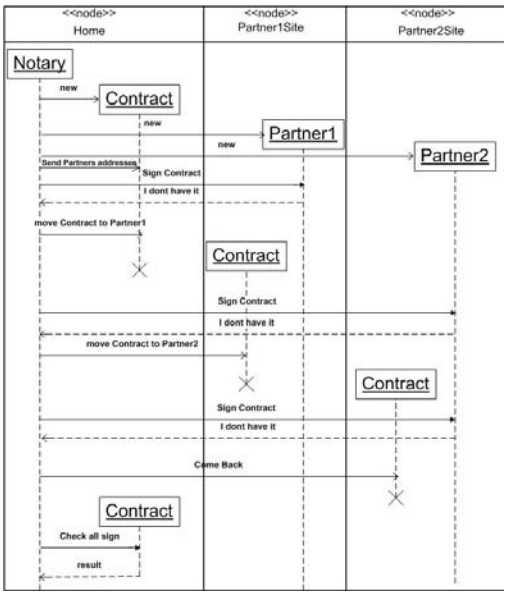


Figure 9: The HLM of the notary agent in the Contract Signing example

Since we have 587 mutants, our runtime conformance checking approach has been executed 587 times. Each execution was allowed to run for at most 90 seconds. So running the approach for all the mutants for both examples takes 587 * 90 seconds = 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 Redhat Linux.

8. Limitations and Future Work

Our experiments suggest that both our mutation operators and our analysis framework hold some promise for the testing of MAS and the detection of non-conformance.

However, it is important to note the limitations of our evaluation experiment and our analysis framework. First, we currently have no hard evidence that our mutation operators are representative of the kinds of mistakes that MAS developers using the Aglets Software Development Kit make. Note, for instance, that the results reported by Andrews et al. in [3] do not apply to our agent-based mutation operators and that their work therefore cannot be used to conclude that mutation testing of the kind suggested here is even appropriate for MAS. More specifically, we have not yet ruled out the possibility that our analysis approach performs less well on real MAS systems, because the bugs introduced by our mutation operators turn out to be much easier to find than those contained in real MAS systems. More experimentation is necessary to strengthen the evaluation of our work, ideally using agent systems developed in industry.

Table 2: Number of equivalent and killed mutants detected

Op.	Mut. Gen.	Equiv.	Non-Equiv.	Killed	Killed %
RDR	15	0	15	15	100%
CDD	15	0	15	15	100%
IDS	15	0	15	15	100%
RAND	13	0	13	13	100%
RDS	15	0	15	15	100%
RDD	7	0	7	7	100%
SICD	2	0	2	2	100%
SDS	7	0	7	7	100%
CMP	11	0	11	11	100%
CMOW	8	0	8	8	100%
RSMM	11	0	11	11	100%
RPSM	50	11	39	33	84.6%
MSKP	27	0	27	27	100%
MSR	41	13	38	38	100%
RSRM	44	13	31	31	100%
MCMC	46	0	46	46	100%
NAN	0	-	-	-	-
RMRM	2	0	2	2	100%
RPRM	18	0	18	18	100%
MICA	2	0	2	2	100%
MFCA	25	0	25	25	100%
ROCM	87	0	87	87	100%
ACON	11	6	5	5	100%
RCBMN	80	0	80	80	100%
RARL	10	0	10	10	100%
RAID	8	3	5	5	100%
CPCN	9	0	9	9	100%
CNP	5	0	5	5	100%
CSAP	3	0	3	3	100%
Total	**587**	**46**	**541**	**535**	**98.9%**

Second, the instrumentation of the IUT and the KM in Step 3 of our approach was performed by hand which is error-prone and time-consuming. At least a partial automation of the instrumentation should be possible and is another topic for future work.

Third, at the moment, our approach does not help the user find appropriate test inputs. Ideally, `kiltera` models would also be used to generate test inputs — a topic that is well-researched in the context of model-based testing (albeit not on `kiltera` models).

Finally, the translation of the high level model to a `kiltera` model was also performed manually. However, thanks to the good support in `kiltera` for the features in Swimlaned Mobility Diagrams in particular and UML Sequence Diagrams in general, the automation of this translation should be possible.

9. Conclusion

In this paper, we have described a new approach for checking the conformance of a mobile, distributed application with respect to an executable model at runtime. To evaluate the effectiveness of our runtime conformance checking approach, we have proposed a mutation-based framework which is based on a collection of novel agent-based muta-

tion operators. We have used this framework to automatically generate and analyze the mutants obtained for two different kinds of mobile agent systems. Our results suggest that `kiltera` is a suitable language to model and simulate MAS succinctly, and that our mutation operators and the conformance checking approach are promising. However, more work needs to be done to strengthen our evaluation and increase the degree of automation of our approach.

10. Acknowledgements

This research was made possible through the financial support of NSERC and of Yarmouk University, which funded the first author during his stay at Queen's University.

References

[1] Voyager pervasive platform. Web site: `http://www.recursionsw.com/Products/voyager.html`, last visited: Jan 9, 2011.

[2] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Compliance verification of agent interaction: a logic-based tool. *Applied Artificial Intelligence*, 20(4-5), Apr. 2006.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. of ICSE*, pages 402–411, 2005.

[4] C. Bäumer and T. Magedanz. Grasshopper - a mobile agent platform for active telecommunication. In *Proc. of the 3rd Int. Work. on Intelligent Agents for Telecommunication Applications*, pages 19–32, 1999.

[5] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. An agent-oriented software development methodology. *J. of Autonomous Agents and Multi-Agent Systems*, 8:203–236, 2004.

[6] A. Carzaniga, G. Picco, and G. Vigna. Designing distributed applications with mobile code paradigms. In *Proc. of ICSE*, 1997.

[7] F. Chen, M. d'Amorim, and G. Roşu. Checking and correcting behaviors of Java programs at runtime with JavaMOP. *Electronic Notes in Theoretical Computer Science*, 144(4):3–20, 2006.

[8] J. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[9] M. Delamaro and A. R. Vincenzi. Structural testing of mobile agents. In *Proc. of the 3rd Int. Work. on Scientific Engineering of Distributed Java Applications*, LNCS 2952, Nov. 2003.

[10] K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[11] M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, checking, and steering of real-time systems. In *Proc. of the 2nd Int. Work. on Run-time Verification*, Jul. 2002.

[12] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[13] M. Kusek and G. Jezic. Extending UML sequence diagrams to model agent mobility. In *Proc. of the Int. Work. on Agent Oriented Software Engineering*, pages 51–63, 2006.

[14] D. B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[15] M. Marche and Y.-M. Quemener. A model for conformance testing of mobile agents in a MASIF framework. In *Proc. of the 2nd Int. Work. on Formal Approaches to Agent-Based Systems*, Oct. 2002.

[16] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. Technical reports ECS-LFCS-89-85 and 86, University of Edinburgh, Mar. 1989.

[17] A. Mok and L. Guangtian. Efficient run-time monitoring of timing constraints. In *Proc. of the IEEE Real-Time Technology and Applications Symp.*, pages 252–262, Jun. 1997.

[18] F. Mokhati, M. Badri, L. Badri, R. Hamidane, and S. Bouazdia. Automated testing sequences generation from AUML diagrams: a formal verification of agents' interaction protocols. *Int. J. of Agent-Oriented Software Engineering*, 2(4):422–448, 2008.

[19] S. Munroe, T. Miller, R. Belecheanu, M. Pechoucek, P. McBurney, and M. Luck. Crossing the agent technology chasm: Experiences and challenges in commercial applications of agents. *Knowledge Engineering Review*, 21(4):345–392, 2006.

[20] A. Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, 1992.

[21] L. Padgham and M. Winikoff. *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley and Sons, 2004.

[22] L. Padgham, M. Winikoff, S. DeLoach, and M. Cossentino. A unified graphical notation for AOSE. In *Proc. of the Int. Work. on Agent Oriented Software Engineering*, 2008.

[23] G. Picco. Mobile agents: An introduction. *J. of Microprocessors and Microsystems*, 25:65–74, 2001.

[24] E. Posse. *Modelling and simulation of dynamic structure discrete-event systems*. Ph.D. thesis, McGill University, Oct. 2008.

[25] A. Saifan. *Runtime Conformance Checking of Mobile Agent Systems Using Executable Models*. Ph.D. thesis, Queen's University, Apr. 2010.

[26] A. Saifan, E. Posse, and J. Dingel. Run-time conformance checking of mobile and distributed systems using executable models. In *Proc. of PADTAD*, 2009.

[27] K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *Proc. of ICSE*, pages 418–427, 2004.

[28] M. Winikoff. Future directions for agent-based software engineering. *Int. J. of Agent-Oriented Software Engineering*, 3:402–410, 2009.

[29] Z. Zhang, J. Thangarajah, and L. Padgham. Automated unit testing intelligent agents in PDT. In *Proc. of the Int. Conf. on Autonomous Agents and Multi-Agent Systems*, pages 1673–1674, May 2008.