Run-Time Conformance Checking of Mobile and Distributed Systems Using Executable Models

Ahmad A. Saifan
School of Computing
Queen’s University
Kingston, Ontario, Canada
saifan@cs.queensu.ca

Ernesto Posse
School of Computing
Queen’s University
Kingston, Ontario, Canada
eposse@cs.queensu.ca

Juergen Dingel
School of Computing
Queen’s University
Kingston, Ontario, Canada
dingel@cs.queensu.ca

ABSTRACT
This paper describes an approach for conformance testing of mobile and distributed systems. The approach is based on kiltera — a novel, high-level language supporting the description and execution of models of concurrent, mobile, distributed, and timed computation. In our approach, a kiltera model of the system is constructed from a high-level model which describes system behaviour using, e.g., a suitable UML profile. Check points are identified in the implementation under test (IUT) and the kiltera model and both are instrumented appropriately at these check points. During execution, relevant information flows from the IUT to the kiltera model which signals any non-conformance detected. Unique features of our approach include the support for mobility, distribution, time, dynamic creation and deletion of agents, and distributed monitoring. We describe the approach and a prototype implementation using a running example for illustration. Results of first, preliminary experiments are reported.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging, Testing tools; I.6 [Computing Methodologies]: Simulation and Modeling

General Terms
Design, Experimentation, Verification, Languages

Keywords
Software testing, conformance testing, runtime monitoring, mobile agents, process algebra, π-calculus, Java, Aglets

1. INTRODUCTION
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” [5]. 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 [24]. The use of mobility has reached a certain degree of maturity: Several different development platforms are available (e.g., Aglets [15, 2], Voyager, and Grasshopper) and comparative performance evaluations have been conducted [29]; agent-oriented software engineering (AOSE) has produced tool-supported development methodologies (e.g., Prometheus [22]) and Tropos [4], promising commercial applications exist [21], and standardization is being considered [23]. However, it seems that relatively little work has been done to support quality assurance techniques such as testing and verification of mobile systems [7, 30].

In this paper, we present an 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 kiltera [25]. 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 HLM is translated into a kiltera model (KM); 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 behaviour) makes this translation relatively straightforward; kiltera’s simulation environment allows early analysis. The HLM is then also 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 the sending or receipt of messages or agent movement; 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 IUT to the KM. During the last step of our approach, the IUT and the KM are executed, both possibly in a distributed fashion. The KM 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 analysis is possible using kiltera’s simulation environment. Finally, our approach
does not assume a central monitoring component; instead, the KM can be arbitrarily distributed, just like the IUT, which helps reduce any performance penalty. Complete automatic code generation from the KM seems possible, but is left for future work. We have implemented the approach in a prototype and used it to conduct first, preliminary experiments with promising results.

This paper is structured as follows: Section 2 provides the necessary background: it briefly describes a UML profile used to express the HLM, it gives an informal description of kiltera, and sketches how to go from the HLM to a kiltera model. Section 3 describes the approach using a running example for illustration. Section 4 discusses the results of first, preliminary uses of our prototype. Related work is reviewed in Section 5. Section 6 concludes the paper.

2. BACKGROUND

In this Section we will describe briefly our high-level modelling formalism (Subsection 2.1), our executable modelling formalism (Subsection 2.2) and an outline of the transformation of high-level models into executable models (Subsection 2.3).

2.1 Mobile Agents in UML

In order to describe distributed systems and mobile agents we use a UML profile introduced in [14] 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.

An SMD 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 new, and 2) arrows that represent agent movement between nodes, labelled move. Message arrows between life-lines in different swimlanes represent remote communication. Figure 1 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.

2.2 An executable modelling language: kiltera

In this Subsection we describe our executable modelling formalism, kiltera [25]. This language is used to describe the behaviour 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 we call πA, an extension of the π-calculus [18]. Unlike other basic process algebras kiltera provides some higher-level constructs to facilitate development. In particular, we allow the use of complex expressions and data-structures in messages, and use pattern-matching as a mechanism to extract information from data.

The language has a formal semantics and a meta-theory (see [25]) which serve as the basis for 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 a brief description of the simulator.

Overview

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

\[
\text{module } A[\bar{x}](\bar{y}) : P \quad \text{or} \quad \text{module } A[\bar{x}](\bar{y}) : \text{sites } \bar{s} \in P
\]

Here \(P\) ranges over process terms, defined below. We use \(x, x_1, \ldots \) for port/channel/event names, and \(A, B, \ldots \) for process/module names, \(n, s, s_i, \ldots \) for site names, and \(y, y_1, \ldots \) for any other variable name. The notation \(\bar{x}\) denotes a list of names or values \(x_1, \ldots, x_n\). In the definition of a module, the names \(\bar{x}\) represent the interface of the module, this is, its ports, or equivalently, the events which it can use to communicate with other modules. The names \(\bar{y}\) represent local state variables and the (optional) \(\bar{s}\) 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 2\(^1\). Here \(E\) ranges over expressions, \(F\) ranges over patterns, \(op \in \{+,-,\ast,/, \text{mod}, \text{and}, \text{or}, \text{not}, <, >, =, \geq, \leq, != \}\), \(n\) 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: \(\text{function } f(\bar{x}) : E\).

The process \(E\) 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

\(^1\)In the presentation of the syntax we use braces \{ and \} to denote syntactic nesting for the \(\text{par}\) and \(\text{seq}\) operators, but in the actual implementation and the examples we use indentation-based nesting.
event are synonymous). The expression E is optional. This process performs communication by uncasting: if there are multiple listeners, only one of them accepts the message, and the choice is non-deterministic. 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 \( \text{\textit{when } \beta_1 \rightarrow P_1 | \cdots | \beta_n \rightarrow P_n} \) 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 \( x \in \text{dchannel} \rightarrow P \), where \( x \) is an event/channel name, \( P \) is a pattern, and \( y \) is a variable (the suffixes \( \text{\textit{with } E} \) and \( \text{\textit{after } y} \) 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 \( P_i \), 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. A listener process represents, thus, a process in a state with multiple listeners, only one of them accepts the message, and the choice is non-deterministic. The process \( \text{\textit{when } \beta_1 \rightarrow P_1 | \cdots | \beta_n \rightarrow P_n \text{ \textit{timeout } E \rightarrow P} \) 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 \( \text{\textit{match } E \text{ \textit{with } P_1 | \cdots | P_m \rightarrow P_r}} \) evaluates the expression \( E \) and attempts to match it with each pattern \( P_i \). If a pattern \( P_i \) matches then the corresponding process \( P_i \) is executed. If more than one pattern matches the choice is non-deterministic. The process \( \text{\textit{if } E \text{ \textit{then } P_1 \text{ \textit{else } P_2}} \) is shorthand for \( \text{\textit{match } E \text{ \textit{with } true \rightarrow P_1; false \rightarrow P_2}} \).

Simulation

Our implementation of kiltera supports both uniprocessor and truly distributed simulation. Both are based on event scheduling. The basic idea is that each term in the language is treated as a simulation event to be executed by the event-scheduler and not to be confused with a communication event in the language itself. The result of a simulation is a detailed event trace.

The event-scheduler contains a queue of simulation events (terms) to be executed, but rather than store them all in a single linear queue, we divide them into 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, and taking the first term in the first time-slot and perform 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 par construct for example, simply adds the subterms to the current time-slot. The action for event creates a new communication event object in the heap. Interaction is done by means of the observer design-pattern. The action for when creates a listener for the appropriate events and registers them with the corresponding event objects. The 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 wait simply adds the term in the appropriate time-slot of the global queue.

Distributed simulation is achieved using the TimeWarp algorithm [11]. Briefly, this is an optimistic simulation algo-

\[ P ::= \text{\textit{done}} \]
\[ \text{\textit{trigger } x \text{ \textit{with } E}} \]
\[ \text{\textit{when } \beta_i \rightarrow P_i | \cdots | \beta_n \rightarrow P_n} \]
\[ \text{\textit{event } \tilde{x} \text{ \textit{in } P}} \]
\[ \text{\textit{wait } E \rightarrow P} \]
\[ \text{\textit{par } \{P_1 | \cdots | P_n\}} \]
\[ \text{\textit{process } A[\tilde{x}](y) : P_1 \text{ \textit{in } P_2}} \]
\[ \text{\textit{move } A[\tilde{x}](y) \text{ \textit{to } s}} \]
\[ \text{\textit{here } s \text{ \textit{in } P}} \]
\[ \text{\textit{dchannel } \tilde{x} \text{ \textit{in } P}} \]

\[ \beta ::= x \text{ \textit{with } F \text{ \textit{after } y}} \]
\[ E ::= n | \text{\textit{true}} | \text{\textit{false}} | "s" | x | op E \]
\[ F ::= n | \text{\textit{true}} | \text{\textit{false}} | "s" | x \]

\[ \text{\textit{Figure 2: kiltera syntax.}} \]
2.3 From high-level to executable models

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 modelled 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. The following listing shows such an example.

```plaintext
1 process Sender[x]:
2 channel response in
3 seq
4 trigger x with ("message", response)
5 when response with result ->
6 // Do something with result
7 // Do something with data
8 trigger response with answer
9
10 process Receiver[x]:
11 channel a in
12 par
13 Sender[a]
14 Receiver[a]
```

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 know the name of the local site. The third is by receiving a site name sent by another agent elsewhere. 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 responsibilities of moving agents. This is, who initiates movement: should an agent tell another agent to move, or should an agent move itself. Also, there are different approaches to transfer the state of an agent. Furthermore, we can create an agent first locally and then move it, or we create it at the remote site. The latter is directly captured by the semantics of the move construct: move A[＠](ｙ) to s creates a new instance of A in a (possibly remote) site s, linked through the channels ｐ and with initial state values given by ｙ. Creating a copy of an agent (module) locally can be achieved by here s in move A[＠](ｙ) to s. This shows how new arrows of SMDs can be directly modelled 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 it is already in the destination, and if not, move a copy of itself there with the required initialization state, and stop.

For example, agent P2 from Figure 1 could be emulated as follows:

```plaintext
1 module P2[x](state):
2 sites A, B
3 3. Use/modify state yielding new_state
4 here s in
5 match s with
6 | A ->
7 | B ->
8 | B ->
9 3. do what needs to be done in B
```

3. DESCRIPTION OF THE APPROACH

Our approach consists of the following four steps:

1. Construction of the high-level model (HLM) of the system using SMDs,
2. Translation of the HLM into a kiltera model (KM),
3. Initialization of the implementation under test (IUT) and KM using the HLM, and
4. Execution of the instrumented IUT and the instrumented KM in parallel.

See Figure 3 for illustration. We will now describe each of these steps in more detail using a running example.
3.1 Step 1: Construction of the HLM

Our approach starts with describing the desired behaviour of the system using SMDs described in Section 2.1. Consider, for instance, an online shopping agent application. 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. Figure 4 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 Mallsite 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.

In order to clarify the model, we divide the HLM presented in Figure 4 into HLMs for the client, agent, malls and the “main” (i.e., a component that sets up 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 5. 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 Mallsite and Mall2site respectively) with different shops and by providing the addresses of these malls to the client to start the search process.

Once the client gets all mall list from the main, it creates an agent at Mallsite, provides it with the addresses of the other malls, and waits for the result to be returned from the agent. Figure 6 shows the HLM of the client.

After the agent has been created at Mallsite, 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. 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 7 shows the HLM of the agent.

3.2 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 behaviour) makes this translation relatively straight-forward. Figure 8 shows the Main KM of the corresponding HLM presented in Figure 5. The three sites are declared in line 2 and the channels to the client and the mall’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 mall1info at site Mall1site (line 6), create an instance of Mall2 with channel mall2info at site Mall2site (line 7), and send the mall sites and their channels to the client through the channel to_cust (line 8).

Figure 9 shows the Client KM of the corresponding HLM presented in Figure 6. In this figure, after the Client has received all mall addresses (line 2), it starts its process by creating the Agent at site mall_site (line 5-6). 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 mall_site 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 (lines 2). Then, the Client waits for the result from the Agent (line 7). The result is the price, and a link to the shop where the agent found the minimum price of the camera.

Figure 10 shows the agent KM of the corresponding HLM presented in Figure 7. The Agent consists of four subprocesses: GetShopList, QueryShops, EnterShop and GoToNextMall. When an agent arrives at a mall it executes the process GetShopList (line 48). This process sends a message to the mall information booth asking for the shop list. When it receives the shop list (line 8), it starts querying the shops (invoking process QueryShops in line 9) one by one searching for the camera and updating the current lowest price if necessary. This is done by invoking EnterShop (lines 24-37) 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 best_price that we have so far, then the new best price is updated; otherwise, the best_price is left unchanged (line 34). 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 (line 22) with the remainder of the shop list. Once the Agent has finished the shop list (which means the shop list is empty in line 14), then it executes the GoToNextMall process which checks whether there are more malls to visit or not. If there are more malls (line 43), the Agent travels to the next mall by creating an instance of Agent at that mall with an updated state (lines 44-45) and behaves in the same way. If the Agent has visited all malls (mall_list is empty, line 41), it sends the result back to the Client telling him the lowest price and where the cheapest camera was found (line 42).

As we can see from example in Figure 10, 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.
module Main:
sites Home, Mall1site, Mall2site in
dchannel to_cust, mallinfo, mall2info in
par
move Client [to_cust] to Home
move Mall1 [mallinfo] to Mall1site
move Mall2 [mall2info] to Mall2site
trigger to_cust with [(Mall1site, mallinfo), (Mall2site, mall2info)]

Figure 8: The KM of the initialization process of the shopping example

3.3 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 [15, 2] in order to implement the online shopping application illustrated in Figure 4. The Aglets Software Development Kit (ASDK) is a framework and environment for developing and running mobile agents. It is originally developed at the IBM Tokyo Research Laboratory. The Aglets system is one of the most popular Java-based open source mobile agent systems [16]. An Aglet is a Java agent able to autonomously and spontaneously move from one host to another. 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 HLM presented in Section 3.1 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 numbers and locations of checkpoints are determined by the constraints to be enforced. More precisely, the users must enforce that the information required for enforcing the constraint is communicated to the KM at the appropriate time. Figure 11 shows the HLM with 5 check points numbered from 1 to 5 presented as a circle.

Once we have decided on the checkpoints in the HLM we locate them in the IUT and the KM. Figure 12 shows the location of check point 5 in the IUT (move the agent to a mall site) and Figure 13 shows the corresponding 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 a process we call “Python connector”. In addition, at each check point in KM,

module Agent [to_customer, to_mall]
  (mall_list, best_price, best_shop):
4 process GetShopList [mallinfo]:
  channel response in
  par
  trigger mallinfo with ("get shop list ", response)
  when response 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, rest) ->
      event shop_visited in
      EnterShop [shop, shop_visited]
      (best_price_here, best_shop_here)
      when shop_visited with
        (new_best_price, new_best_shop) ->
        QueryShops [rest] (new_best_price, new_best_shop)
    process EnterShop [shop, shop_visited]
      (best_price, best_shop_here):
      channel response in
      par
      trigger shop with ("camera ", response)
      when response with price ->
      if price < best_price then
        trigger shop_visited with (price, shop)
      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
        | [] ->
        | (mallsite, mallinfo), rest ->
        move Agent [to_customer, mallinfo]
          (rest, best_price, best_shop) to mallsite
      in
      GetShopList [to_mall]

Figure 10: The Agent 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. 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 14 and Figure 15 show the instrumentation code inserted in IUT and in KM respectively.

3.4 Step 4: Execution of IUT and KM

In the last step of our approach we execute the instrumented IUT and the instrumented KM and check the conformance between them. While the IUT is running, it sends messages possibly including state information to the KM through a “Python connector”. The Python connector is a Python script [1] that serves as a “bridge” between the
IUT with the KM. 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 16 shows the architecture of the monitoring infrastructure of our approach.

### 4. EXPERIMENTATION

We now describe some of the experiments we have conducted.

#### 4.1 Checking sample properties of the shopping application

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 in some format to the KM describing 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 check will now be given. At the end of the section, another experiment will be discussed briefly.

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

At the checkpoint shown in Figure 15, the KM expects a message from the IUT containing the string "Agent traveled to next mall" (lines 3-5). If such string is received, the KM continues its execution by creating an instance of `Agent` and moving it to the next mall site `mall_site` (line 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 17, 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.

**Property 3: Timing constraints.**

Figure 18 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` time units) and that...
1 // Check point 5
2 // Inserted code begin
3 HandleJavaMessages [ external ]
4 when external with data ->
5 if data = 'Agent Traveled to Next Mall' then
6 move Agent [ to _ customer , mallinfo ]
7 ( rest , best_price , best_shop ) to mallsite .
8 else
9 // Error, stop execution with non-conformance
10 // Inserted code end

Figure 15: Code inserted into KM at check point 5

Figure 16: The architecture of the monitoring infrastructure

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 rsponsetime in line 5. If rsponsetime is greater than t (line 5), non-conformance is reported. Also, if the KM agent does not receive a response from the shop in the KM within t time units (line 13), but a proper response has been received from the shop in the IUT (line 20), we have non-conformance.

Table 1 compares the size of the IUT and the KM before and after the instrumentation.

<table>
<thead>
<tr>
<th>File Name</th>
<th>IUT Before Inst</th>
<th>IUT After Inst</th>
<th>KM Before Inst</th>
<th>KM After Inst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>26</td>
<td>26</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Client</td>
<td>57</td>
<td>60</td>
<td>7</td>
<td>20</td>
</tr>
<tr>
<td>Agent</td>
<td>212</td>
<td>262</td>
<td>44</td>
<td>151</td>
</tr>
<tr>
<td>Mall1 with 4 shops</td>
<td>272</td>
<td>272</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>Mall2 with 2 shops</td>
<td>152</td>
<td>152</td>
<td>21</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 1: Size in lines of code of the IUT and KM before and after the instrumentation

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.

Figure 18: Checking Property 3 in KM

2. The instrumentation of the Client KM is larger than the corresponding instrumentation on the IUT side. This is because the Aglets platform offers a primitive which allows the easy determination of how many processes reside on a node; in the KM, on the other hand, this primitive does not exist and needed to be implemented.

3. 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.

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 con-
formance checks described above completed in less than a minute.

4.2 Other examples

We have also applied our framework to two other examples: a centralized distributed mutex algorithm, and Lamport’s distributed mutex algorithm (implementations of these two algorithms was taken from [9]). Our prototype was able to detect seeded faults in both implementations. 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 region. Interestingly, this modification caused the code to examine “intermittent” (or temporary) inconsistency which would disappear after continued execution. The example shows that a system may recover from failure “on its own” and that, therefore, partial examination of traces may not always be sufficient.

5. RELATED WORK

Relevant related work seems to fall into three categories: run-time monitoring, testing of mobile code, and testing of agent systems.

1. Run-time monitoring: Many approaches and tools for run-time monitoring exist (e.g., Java MaC [13, 12], Java PathExplorer [10], Java Run-time Timing-constraint Monitor [19], decentralized monitoring [26], and Java Monitoring-Oriented Programming (MOP) [6]). With the exception of Java MOP, 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 [19, 12], distributed systems [26], and sequential and concurrent systems [13, 6]), the degree of automation of the instrumentation phase (e.g., automatic instrumentation from specifications [13, 26], automatic instrumentation using scripts [10], and manual instrumentation [19]), and the specification formalism supported (e.g., temporal logic [13], timing constraints [19], and rewrite logic [10]). Our approach 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.

2. Testing of mobile code: A more limited amount of existing work is devoted to testing of mobile code. In [7], a framework is presented to support testing mobile Java code with respect to the standard code coverage criteria (e.g., statement coverage, branch coverage, etc.). In [8], test patterns are obtained from design patterns for mobile agents. Finally, a formal framework for conformance testing of mobile code is presented in [17]. The work extends the conformance testing theory for distributed systems developed by Tretmans [28] and uses labeled transition systems. Our approach, on the other hand, is based on an extension of the π-calculus, which appears much more suitable. However, our work currently lacks a formal definition of conformance.

3. Testing of agent systems. In this context, 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 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 [22]) 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 [27, 32, 31, 20]. The work in [3] discusses conformance testing and thus is closer to ours: agents are monitored with respect to “interaction constraints” that capture properties of interactions between agents as logical formulas; constraint checking is implemented using constraint logic programming; timing constraints are supported, but support for mobility and distributed monitoring appears to be missing.

In summary, to the best of our knowledge, our approach is unique in that it simultaneously supports mobility, distributed execution and time and uses models which can be directly executed in a simulation environment.

6. CONCLUSION AND FUTURE WORK

An approach for runtime conformance checking of mobile and distributed systems has been described. The approach is based on kiltera, a novel modeling language for concurrent, distributed, mobile, and timed systems. Due to kiltera’s simulation environment, early analysis of the models is possible. Moreover, execution of the kiltera model may be distributed. Our use of the approach in two case studies has been sketched. The results are preliminary, yet encouraging.

The most important topic for future work is a more thorough evaluation of the approach. In particular, we are interested in determining how effective it really is for detecting faults in mobile and distributed systems. To this end, we are currently considering the use of mutation testing. 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 kiltera models). Finally, another promising idea to pursue is the automatic generation of, e.g., Java code from kiltera. Our approach could then be used to develop techniques for testing code generators.

7. REFERENCES


