Approach for the development of mobile applications based on migrant objects

Aissa ElMahdi Bourahla, Mustapha Bourahla

Abstract

Software engineering principles are very required for the development of mobile applications, which are necessary for many life applications. In this paper, we present a development process based on transformation of UML models, which describe mobile applications based on migrant objects to Mobile Maude language that extends the Maude language and implements the Rewriting Logic (RL). The generated rewriting theories will be executed for simulation using located configurations, which are produced from UML object diagrams. State transition models may be built during simulation to describe behaviours of mobile applications based on migrant objects, which will be verified against LTL properties with the technique of model-checking. The verified model of mobile application based on migrant objects can be used to generate Android application to be deployed on mobile devices.

Keywords: Mobile Application, Migrant Objects, Rewriting Logic, Mobile Maude, Maude LTL Model Checker.

1 Introduction

The mobile application development has exponential growth since the iPhone AppStore opened in 2008. There are many documents for sets of best practices to guide developers of mobile applications, like those proposed to Android and Apple iPhone developers, but rarely use formal development techniques. Despite the development of large number of mobile applications, there is still not much formal research

©2022 by CSJM; A. Bourahla, M. Bourahla

around their engineering processes. There are powerful development tools and frameworks (for example, Android Studio, Eclipse and Windows Phone) offering programming environments for the major mobile platforms to simplify the task of implementing mobile applications. These tools are focusing on the individual developer who is trying to create a mobile application as quickly as possible.

A formal technique will help to develop mobile apps independently of the mobile platform as Android, iOS, etc. when the devices with different software versions or screen sizes might have issues that aren't found elsewhere, which can also help to handle testing. The formal specification and verification process is still a crucial part of development and it gives an added insight into the inner workings of mobile app, where alternative ways can be potentially found to achieve our goals. These alternatives can save work time and cost. Thus, it is not sufficient to test the app once on mobile phone and assume that it's working correctly. A quality assurance process can find problems with our app before it goes to market. It's much better to identify these using formal development; otherwise, users will find them in real life.

Techniques for developing mobile applications are similar to software engineering for other embedded applications with additional requirements. The mobile applications have potential interaction with other applications, sensor handling, data of physical location, proximity to other mobile devices, the activation of various device features, complexity of testing, power consumption, and issues associated with transmission through gateways and the telephone network [1], [2].

Software engineering techniques should assure data integrity and synchronization using for example, client-server computing. They should consider the risk of program and/or data integrity when events of potential loss of connectivity or battery power occur during a transaction or system update [3]. They should also design applications differently depending on the speed of the network on which they are being used.

The development of prototypes of the user interfaces is very required, particularly when multiple devices will be supported. An important area for mobile software engineering research is the develop-

ment of techniques for testing. Development of the mobile application is test driven [4] and it will typically be done within the context of the overall software development effort. It is insufficient to merely test mobile application on an emulator; it must be tested across many different mobile devices running different versions of the operating system on various telecom networks. Integrated test tools would help the development process [1]. There is need for customized tools to support cross-platform development and testing.

In this paper, we propose a framework for developing mobile applications based on migrant objects. A mobile application based on migrant objects is defined as a mobile application composed of a set of modules (or procedures) with small code. These modules can migrate from a mobile device to another to communicate with a located module. The module migration is the move of the procedure code and its state (context) represented by a set of attributes.

A module is represented by an object of a class with attributes and methods. This module object will be associated with a mobile object, which can move from device to another using TCP/IP sockets. The located mobile module contains a root object to play the role of server (server root object) or the role of client (client root object). These root objects are responsible for communication to move module objects between the different mobile devices.

There are many life applications for this kind of mobile applications based on migrant objects: health care, electronic commerce, electronic learning, etc. In the health care domain, for example, a doctor can use a mobile server application to migrate from his/her own mobile device to another mobile client application owned by its patients. Hence, the mobile client application now running on the patient's device can interact with its mobile server application. All required health information can be asked by this mobile module representing the doctor behaviour within the patient mobile device. When the doctor module (mobile object) returns back to the doctor mobile device, it can inform him with the patient's health information.

This development framework uses a UML like language for modelling server/client processes composing distributed mobile applications

based on migrant objects, where a mobile object resident in a process can migrate (move) to another process for communication with its resident mobile objects. For each mobile application based on migrant objects, we develop the class, object, and statechart diagrams. These diagrams are used to generate specifications as rewriting theories implemented by the Mobile Maude language [5], which extends the system Maude [6] implementing the Rewriting Logic. These specifications are executable, which help to do simulation and testing. In addition to simulation, it is possible to verify formally specified properties using Linear Temporal Logic (LTL) and the technique of model-checking with Maude LTL Model-Checker [7]. When the UML model of mobile application based on migrant objects is tested by simulation of its specification and its LTL properties are verified, an implementation code can be generated.

1.1 Related works

There are studies on software engineering issues for mobile application development, which help to be aware of some challenges during the application development life cycle and try to resolve problems to improve the performance of mobile applications. The authors in [8] provide an overview of important software engineering research issues related to the development of applications that run on mobile devices. They highlighted some software engineering issues for development processes, tools, user interface design, application portability, quality, and security. New set of research issues is asked for the different characteristics of mobile applications and their operating environments.

The authors of [9] have proposed a MDD approach for mobile application development, which includes modeling and code generation strategies for Android and Windows Phone, where UML class and sequence diagrams are employed for modeling mobile applications and code is generated from these models. The paper [10] has presented processes and procedures for developing mobile cloud applications by effectively applying UML, where the Android mobile platform and Amazon Web Service are used for cloud computing in order to demonstrate

the applicability of the proposed approach to systematically apply the UML profiles and diagrams for cloud-based mobile applications.

The paper [11] has presented model-based analysis to study energy consumption issues at early stages of mobile development. It uses Real-Time Maude for analyzing energy consumption of a whole system behavior consisting of hardware components and application programs as well as the framework. Example scenarios on detecting energy bugs demonstrate that the time-bounded analysis method using Real-Time Maude is effective. The authors in [12] have written a survey as a result of interviews with mobile application developers. With this survey, we can understand the main challenges that face mobile application developers, which are developing applications across multiple platforms: lack of robust analysis, testing tools, and the problems of emulators that are slow or miss many features of mobile devices. The most useful tool for mobile application development is Android Studio [13], which has become the primary IDE for native Android application development. It provides code editing, debugging, and testing tools within the development environment using emulators.

This paper is organized as follows. In Section 2, we present how to model a mobile application based on migrant objects with UML. The model transformation to Mobile Maude specifications is presented in Section 3. Section 4 explains in detail the simulation of the specifications and their formal verification against LTL properties. An implementation of automatic generation of Mobile Maude specifications from UML diagrams based on Transformation Graph Grammars (TGGs) [14] is presented in Section 5. At the end, conclusions and perspectives are given.

2 Modelling mobile applications based on migrant objects

The key elements for modelling Mobile Applications based on Migrant Objects (we call them MAMOs) are processes and mobile objects. A process is a computational environment, which is located in a mobile

device, where mobile objects can reside. Each mobile object, which is characterized by its own code and state, has the ability to move from a process to another in different locations and it uses messages to communicate asynchronously with the other mobile objects. The mobile objects can execute their code in the located computational environment as response to incoming events.

An overall configuration of MAMO application is a set of processes, where a mobile object resident in a process can migrate to another process for communication with its resident mobile objects [6]. The code of each mobile object is represented by a small object-oriented program (module), and its state is represented by data of a set of objects and messages, which can be changed by executing its own code at the process level.

To model a MAMO, we need to describe a model composed of three diagrams (class, statechart and object diagrams). A class diagram is composed of three classes (Figure 1). The first two classes are called "ServerRootObject" and "ClientRootObject", which are specializations (subclasses) of the class "RootObject" and they have different behaviours. The third class is called "MobileObject", which is used to create mobile objects.

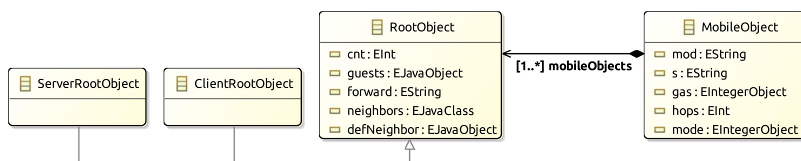

Figure 1. Class diagram of the main classes: RootObject, ServerRootObject, ClientRootObject and MobileObject

To facilitate the transformation process, the definitions of these classes are based on Mobile Maude implementation [5]. The first two classes are defined to create sockets for communication between mobile objects composing the MAMO. There will be only one object (instance) of the class "ServerRootObject" to create TCP/IP sockets for

listening on for connections from other root objects, which are of the class “ClientRootObject”. The mobile objects are instances of the class “MobileObject” and they can reside with a server root object or a client root object to be able to communicate.

The root object (“RootObject”) class has a set of attributes. The attribute “cnt” is a counter to produce names for new mobile objects that are created at the process level. The root object of each process keeps information, which is a set of names (identifiers) of the mobile objects currently in it in the “guests” attribute. Each root object has another attribute called “forward” to save the forwarding information concerning the whereabouts of its residents. The communication actions of root objects are represented by its attribute “state”, which is only “idle” during their creation to indicate their activation as a client socket (if it is a client root object) or as a server socket (if it is a server root object). Then, the root objects change their state to the state “waiting-connection” and they stay on it until getting acceptation of connection from the server or client socket, which will pass them to the state of “active” mode to begin the normal activity of communication. Each root object has a routing table maintained in its attribute “neighbors”. It is used to associate socket object identifiers with location identifiers. If the neighbours table doesn’t contain a communication socket associated with a particular location, then an additional default socket will be specified in the attribute “defNeighbor” to be associated with this particular location.

The “MobileObject” class has the attribute “mod” to store the meta-representation of the object-based module using the operator “upModule”. The mobile object’s state (context) must be stored in the attribute “s”, which is meta-representation (using the operator “upTerm”) of a pair of configurations meaningful for the module in “mod”. The attribute “hops” of the class “MobileObject” is defined to be incremented every time a mobile object has moved from one process to another. This number of hops performed by a mobile object will be used by the forwarding process. The attribute “mode” indicates if the mobile object is idle or active. Its initial value is set to be active to make the mobile object on activity from the beginning and it cannot

be in active mode if it is in motion. The other attribute in this set, is the “gas” attribute, which is used to limit the number of resources to be consumed by the mobile objects. It is possible to add attributes to this set during the modelling if necessary.

The class MobileObjects’ attributes named “mod” for procedure code specification and “s” for procedure state context will be defined by a class association, which is associated with the class MobileObject. This class association extends the class diagram model by classes for mobile applications (modules) to define the behaviour of the mobile objects (instances of the class MobileObject). This mobile objects modelling is viewed as a distribution of located configurations; each one is executed in a different process. The created mobile objects can reside in a process to execute their code specified by the attribute “mod”. Its execution can change its state identified by the attribute “s” and it can also communicate with other mobile objects by sending and receiving messages to and from the processes locating them.

The server and client root objects are identified by names of the form $l(IP, n)$, where l stands for location, IP is an Internet Protocol address of the machine in which the process is being executed and n is a natural number. The uniqueness of names for root objects composing located configurations of distributed processes should be guaranteed for proper functionality. Only one root object of the class $RootObject$, will be in a located configuration to be responsible to maintain information concerning the process location, its mobile objects, and the situation of the mobile objects that are created in it and moved to other processes.

Mobile objects with their context state and their code written with an object-oriented language, can migrate between processes and communicate by sending and receiving asynchronous messages. The mobile objects are identified by names of the form $o(l(IP, n), k)$, where o stands for object, $l(IP, n)$ is the identifier of the root object of the process in which it was created, and k – a natural number. A mobile application based on migrant objects, which runs on a mobile device, is a configuration of an object of the class “ServerObjectManager” or “ClientObjectManager” and a set of objects of the class “MobileObject”. A mobile object can move (moving its code and its current state)

from a device configuration to another using the TCP/IP sockets created by the server and/or client objects. Two mobile objects in the same configuration can communicate using messages defined during modelling to change their states. So, if a mobile object wants to communicate with another mobile object in a different configuration, the former should first move to this configuration containing the mobile object to be able to communicate with it.

3 Generation of mobile Maude specifications

To simulate mobile application based on migrant objects (MAMO), a transformation must be applied on its models of class diagram and statechart diagram to generate Mobile Maude specifications with respect to the rewriting logic syntax. These specifications are rewriting theories, which can be executed via Mobile Maude configurations that will be generated from the object diagram modelling objects of the MAMO. Mobile Maude [5] is a mobile object language, which extends Maude [6] for specifying mobile computation. The Mobile Maude specifications are executable, which helps to use them as prototypes of the language and then applications implementing mobile object can be simulated.

The mobile object's state identified by the attribute s must be the meta-representation of a pair of configurations meaningful for the module in the attribute *mod*. These two attributes, which are specified in a class association will be added to the attributes set of the class "MobileObject". The attribute s has the form of a conjunction $Conf$ & $Msgs$, where the first part of the conjunction $Conf$, is a configuration of objects and incoming messages to be processed by the mobile Maude system. The second part of the conjunction $Msgs$, represents a multi-set of messages to be sent as a result of handling the configuration $Conf$.

The root and mobile objects are modelled by a small set of mobile Maude rules, which code their mobility and message sending. These rules are independent from the application. It is possible also to write the MAMO code as Maude object-oriented modules. The mobility of an object can be realized by sending two different messages to the

root object. The first message is $go(L)$, which means the sender mobile object request moving from its current location to the specified location L. This means the mobile object only specifies the location to go to. The second message is $go - find(O, L)$, which is sent when requesting the move to a location, where the mobile object O resides (it can be L). In this case, the mobile object only knows the identifier of the object (O) to catch up with, not the location it is at. But, it can give a tentative location (L), which can be the home location of O.

A mobile object can send messages to other mobile objects. These messages between mobile objects have the form *to* $O\ :\ msg$, where O is the receiver of the message and msg is the message content, which can be of any kind and it can indicate the name of the sender, so the receiver will know the identifier of the sender. The MAMO applications are object-oriented, where the operator "&" is used as constructor by the mobile objects to send messages and to move to other processes using the socket connections. The communication can be between objects inside the same mobile object, where the messages can be of any format and this communication may be synchronous or asynchronous. Communication can also be between objects in different mobile objects. In this case, these mobile objects can be in the same process or in different processes. This communication should be asynchronous and it is transparent to the mobile objects and the messages must be of the form *to* $O\ :\ msg$, which is explained before.

When a rule of a mobile Maude module has a configuration identifying a mobile object A is executed, and the second component of its result state has the message go or $go - find$, then the mobility of the identified mobile object A in the configuration is initiated. The $go(L)$ message will make the root object (manager) moving the mobile object A to the location L. However, the $go - find(B, L)$ message will try to move the mobile object A to reach the objet B that can be itself on move. It begins with the specified location L as the first tentative, if A doesn't find the object B, it will go on looking for B in a different location.

Each mobile object has a tray of outgoing messages; if it decides to migrate to a different process in a target location L, it will add the

message $go(L)$ to this tray of messages. This will invoke the operation of sending the *go* message to inter mobile objects, where the sender mobile object is indicated as an argument of this message. At the end of this operation, the *go* message will be then removed from the outgoing messages tray. If the mobile object is on move, it will be made inactive by the *go* message. If the location of the message's sender is different from the location of its receiver, then the root object of the sender's location will send this message to the desired location through the appropriate socket. The root object of the destination location will update its forwarding information if it receives this message. When the mobile object reaches its home location, it will be informed by the corresponding root object of this message.

3.1 Example of healthcare domain

We explain on a simple example how a mobile application based on migrant objects (MAMO) can be developed with this formalism. In this example we have patients and a doctor in a MAMO for healthcare domain; a doctor visits several patients, who provide him information on their health. The doctor looks for the patients need care, and once he has visited all the patients, he goes back to his home location where human doctor who is the owner of the mobile device can consult his patients' health information. This description allows us to identify the actors to be represented as mobile objects, which may migrate (move) between the different processes composing the MAMO application. In this approach the specification of the MAMO applications consists of objects embedded inside mobile objects, which communicate with each other via messages. According to the semantics of the language Mobile Maude, we represent this as a dependency between the dependent class "MobileObject" and its dependency class "Patient" or "Doctor", where their codes should be executed (see the code of the processes in Section 4).

To model the doctor and patient classes, we represent patients and doctor as objects of respective classes *Patient* and *Doctor*. Such objects in the MAMO application code will then be embedded inside their corresponding mobile objects. In the class diagram, which extends the

class diagram of Figure 1 of this particular mobile application based on migrant objects, the class "Patient" has attributes description with the patient name, temperature, blood pressure and glucose levels.

$$class\ Patient \mid name\ :\ String, temperature\ :\ Float,\\ blood{-}pressure\ :\ Float, glucose\ :\ Float$$

A doctor class has an attribute called *patients* with a list of identifiers for the known patients mobile objects. It has also an attribute called *state* with its current state, which can be $State1$ (initial state), $State2$, $State3$ or $lastState$ (last state). These states (generated from the statechart diagram) are used to synchronise the rules execution, which represent its behaviour. Finally, the doctor class keeps information about the patients' health information.

$$class\ Doctor \mid patients\ :\ patients, state\ :\ DoctorState,\\ health{-}informations\ :\ patient(temperature,\\ blood{-}pressure, glucose)$$

Each mobile object will carry the representation term of its state (context) and the code managing the behaviour of the objects and messages of the configuration representing this state. In the sample mobile application MAMO, we have two different classes of mobile objects: patients and doctor. Although the objects representing the patients don't move, they should be modelled as mobile objects to be able to send and receive messages from other mobile objects through the mobile Maude system. The following statechart diagram (Figure 2) summarizes their behaviours. We remark that the mobile object doctor continues its execution from the state "doctorState3" after it moves to the client process of the mobile object patient when it was at the state "doctorState2".

The transformation to Mobile Maude code is based on formal semantics given to the statechart diagram. Every mobile object has an initial (located configuration) state, which will be defined in the object diagram. The doctor mobile object has a last state (*lastState*), however the mobile object patient has no last state, which means its execution doesn't terminate. A transition in the statechart diagram is

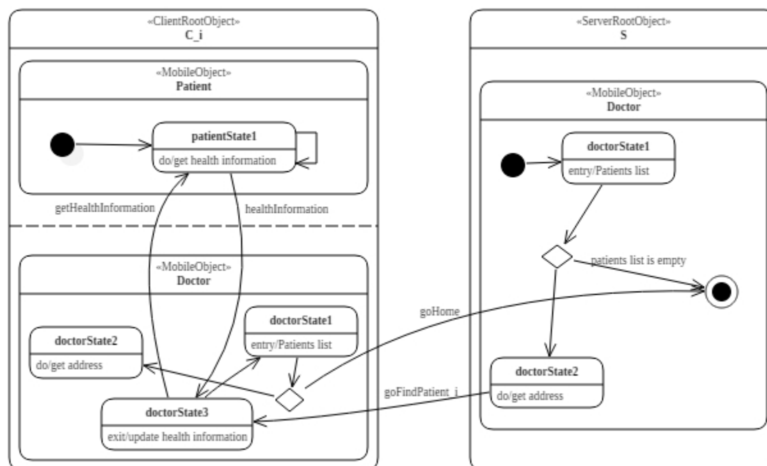

Figure 2. Statechart diagram of the mobile objects doctor and patient

represented by a rewriting rule of the form.

$$rl[State] :(State \wedge Condition) \ \& \ incoming{-}message => \\ State' \ \& \ outgoing{-}message$$

where *State* is a state value of the attribute *state*, *Condition* is any value of any attribute from the attributes set. The incoming message $incoming-message$ is a received message that labels the incoming edge of the state *State* (or *none*, if there is no label), which constitutes a transition guard with the condition *Condition*. The execution of this rule will change the mobile object configuration by changing the attribute value of *state* to be $State'$ and sending the outgoing message $outgoing-message$ that labels the outgoing edge of the state $State'$ (or *none*, if there is no label). A doctor visits several patients to ask each one he visits for the description of his health, represented here by its temperature, blood pressure and glucose levels. In the statechart diagram, when the doctor is in the server root object at the state $State1$ (the initial state) looks for the address of the first patient in the patients list. If there is no patient location to visit, it goes to its last state (named *lastState*), else it goes to the state $State2$, from which it

can go to find the patient location.

$$rl[State1] : (State1 \wedge list\ of\ patients\ is\ empty)\ \&\ none => lastState\ \&\ goHome$$

$$rl[State1] : (State1 \wedge list\ of\ patients\ is\ not\ empty)\ \&\ none => State2\ \&\ go\ find\ first\ patient\ in\ the\ list$$

A message $go - find$ is sent to the root object. As response to this message, the doctor mobile object moves to the process location of the first patient mobile object in the patients list. At this process locating the patient mobile object, the doctor mobile object continues its execution from the state $State3$ by sending the message *get–health–*$information$ to the patient mobile object.

$$rl[State2] : State2\ \&\ none => State3\ \&\ get\ health\ information$$

The patient sends back his health information stored in its attributes, which will be received when the mobile object doctor is in its state $State3$ and then saved in its attribute $health-information$.

$$rl[State3] : (State3 \wedge health\ information)\ \&\ none => State1\ \&\ none$$

From the state $State3$, the doctor returns back to the state $State1$ if it has received the health information for looking the next patient address to move to its process location by the same way. Once the mobile object doctor has visited all the patients it knows (the patients list is empty), it goes back to its home location by sending the message $go(home-location)$. Then, all the patients' health information is in its attribute named $health-information$, which can be consulted by the mobile device owner. Patients receive from the doctor messages of the form $get-health-information(D)$ with D as the identifier of the doctor mobile object sending the message. Patients can send messages of the form *health–*$information(Name, T, BP, G)$ with $Name$ – string representing the patient's name, T, BP, and G – real numbers representing the patient's temperature, blood pressure, and glucose levels, respectively.

$$rl[State1] : (State1 \wedge get\ health\ information)\ \&\ none => State1\ \&\ health\ information$$

From the extended class diagram and the statechart diagram, we generate Mobile Maude modules for patient's behaviour and doctor's behaviour. Patient's behaviour is represented by the rewrite rule labelled with $State1$ (indicated in the statechart diagram), which corresponds to its unique state $State1$. When a patient receives a heath information request, it sends back the information to the doctor. The whole module defining the patient's behaviour in Maude is below.

```
mod PATIENT is
  ...
  rl [State1] : Conf (to P : get-health-information(D))
    < P : V@Patient | name : Name, temperature : T,
       blood-pressure : BP, glucose : G, AtS > & none =>
    Conf < P : V@Patient | name : Name, temperature : T,
       blood-pressure : BP, glucose : G, AtS > &
       (to D : health-information(Name, T, BP, G)) .
endm
```

However, the doctor module, which is presented below, is more complex. Its behaviour is composed of four states. In the state encoded by the rule labelled by $State1$, it moves to the process containing the mobile object, where the object patient identified by $o(L, N)$ is in and it changes its state to $State2$ to ask the patient object its health information by sending the message *to* P : $get{-}health{-}information(D)$, where P is the object identifier of the patient (i.e., $o(L, N)$) and D is the object identifier of the doctor requesting health information.

The patient object, which is now located with the doctor mobile object in the same process, responds with the message *to* D : *health–information*$(Name, T, BP, G)$ (the rule encoding the state $State1$ of the patient mobile object). When it is in the state $State3$ and the patients list is not empty, it will return back to the state $State1$ to move to the located configuration of the next patient mobile object, else it will return back to its original home location (the server root object) carrying the health information of all the patients.

```
mod DOCTOR is
  ...
  rl [State1] : < D : V@Doctor | patients : o(L,N) . OP,
    state : State1, AtS > & none => < D : V@Doctor |
    patients : o(L,N) . OP, state : State2, AtS > &
    go-find(o(L,N), L) .
  rl [State1] : < D : V@Doctor | patients : no-id,
```

```
      state : State1 , home-location : o(L,N) , AtS > &
      none => < D : V@Doctor | patients : no-id ,
      state : last , home-location : o(L,N),AtS> & go(L).
  rl [State2] : < D : V@Doctor | patients : P . OP,
      state : State2 , AtS > & none => < D : V@Doctor |
      patients:P . OP, state:State3 , AtS > &
      (to P : get-health-information(D)) .
  rl [State3] : (to D : health-information(Name, T, BP,
     G)) < D : V@Doctor | patients : P . OP, state :
      State3 , AtS > => < D : V@Doctor | patients : OP,
      health-information : Name(temperature: T,
      blood-pressure: BP, glucose: G), state : State1 , AtS > .
endm
```

4 Simulation and verification

We will show how we can simulate and verify this mobile application based on migrant objects by using the Maude system. Our sample doctor/patients configuration is constituted of four located configurations; each one will be executed in a Maude process. Each located configuration contains one root object (there are four configurations).

The mobile object doctor is the resident of the server root object; however, the patients are residents of the client root objects. From the object diagram, we generate four located configurations. The first located configuration (shown below) contains a *ServerRootObject*, with identifier $l(IP, 0)$, and a mobile object identified by $o(l(IP, 0), 0)$ with a doctor module in its belly. This configuration represents the central process of the star network and it must be executed first, because it has the object *ServerRootObject*, which is responsible for creating the server socket to listen and then accept connections from the other objects.

```
mod PROCESS-DOCTOR is
  ex DISTRIBUTED-MOBILE-MAUDE . ex DOCTOR . op IP : -> String .
  op initial : -> Configuration . eq IP = "localhost" .
  eq port = 8000 . eq initial =<><l(IP, 0) : ServerRootObject |
    cnt : 1,state:idle,guests : o(l(IP, 0),0),defNeighbor:null,
    neighbors : empty, forward : 0 |-> (l(IP, 0), 0) >
  < o(l(IP,0),0):MobileObject | mod:upModule('DOCTOR, false),
      s : upTerm(< o(l(IP, 0), 0) : Doctor | status : done,
      home-location : o(l(IP, 0), 0), patients : o(l(IP, 1),0).
      o(l(IP, 2),0) . o(l(IP, 3),0)> & none), gas : 200,
```

```
    mode : active , hops : 0 > .
endm
```

Note how the Maude meta-level function *upModule* is used to obtain the meta-representation of the module *DOCTOR*, and how the function *upTerm* is used to meta-represent the initial state of the inner object, where the list of patients is declared as value of the attribute *patients*. The other configurations below are for the three mobile objects patients, which contain *ClientRootObject* with a Patient object in the belly of a mobile object. Each patient has a different name "PatI" ("I" stands for 1, 2 or 3) and different health information: the temperature T (39.0, 39.5, and 39.0), the blood-pressure B (12.0, 13.2, and 14.0) and the glucose G (1.3, 1.56, and 1.2). The mobile object of the patient, which has the name "PatI" is called "o(l(IP,I),0)" and it is located in a process with the client root object "l(IP,I)", where "I" stands for 1, 2 or 3.

```
mod PROCESS-PATI is
  ex DISTRIBUTED-MOBILE-MAUDE . ex PATIENT . op IP : -> String .
  eq IP = "localhost" . eq port = 8000 .
  eq server-address = "localhost" . op initial :->Configuration .
  eq initial = <> < l(IP, I) : ClientRootObject | state : idle,
     cnt : 1, guests : o(l(IP, I), 0), neighbors : empty,
     forward : 0 |-> (l(IP, I), 0), defNeighbor : null >
    < o(l(IP, 1), 0):MobileObject|mod:upModule('PATIENT, false),
       s : upTerm(< o(l(IP, I), 0) : Patient | name : "PatI",
       temperature : TI, blood-pressure : BI, glucose : GI >
       & none), gas:200, mode:active, hops:0 > .
endm
```

These four configurations represent an overall located configuration (processes) of the mobile application MAMO. In this case the four processes run on the same machine, with IP address *localhost*. The execution results of these four different Maude processes are described as follows. First, the doctor travels to the location $l(IP,1)$ of the first patient $o(l(IP,1),0)$ and asks him about his health information. The patient resident in this location responds with his information. Then, the doctor travels to the next location $l(IP,2)$ to ask its resident, which is the patient $o(l(IP,2),0)$.

The last move before returning to its home location $l(IP,0)$, the doctor identified by $o(l(IP,0),0)$ travels to the third patient loca-

tion $l(IP, 3)$ to ask its resident, which is the patient identified by $o(l(IP, 3), 0)$. The doctor has finished his travel at his home location and has the names and health information of all the patients. Its attribute "hops" has the value 4, which means that it has 4 jumps, it has moved to 3 locations and it has returned back home. The value of the attribute "gas" is $200 - 193$, which means it has consumed 7 resources. The simulation results for the mobile objects "Pat1", "Pat2", and "Pat3" show that these three mobile objects didn't move (the values of their attribute "hops" are zero) and they have consumed two resources ("gas" equals $200 - 198$).

4.1 Formal verification

Formal verification is useful to check properties that cannot be verified by simulation as, for example, the property a doctor mobile object should visit only once every patient. The Maude LTL model checker implemented within the system Maude [7] can be used to verify the satisfaction of LTL properties by Maude specification if its set of states reachable from an initial state is finite. The Maude LTL model checker can be used to check whether a given initial overall configuration fulfills a mobile application MAMO property described by Linear Temporal Logic (LTL), such as safety property (something bad never happens) and liveness property (something good eventually happens) [15]. Verifying LTL properties on mobile applications MAMOs is not easy. The MAMOs applications are distributed among several hosts; therefore, the Maude LTL model checker cannot be applied directly to prove global properties. In the following, we show how this issue is addressed.

The problem has been solved by specifying additional mobile object responsible for model-checking models that can be built during the activity of the other mobile objects. A model checker mobile object is another associated class with attributes, the first attribute *model* is for capturing the states sent by the specified mobile objects, the second attribute *formula* is to specify the LTL formula to verify the model built, and the last attribute $model-check$ will show the model checking result, which is true if the property is satisfied or a counterexample (a

trace) showing why the property is falsified. This mobile object is located in the process of the server root object, and it will receive states from the other mobile objects (in the patients/doctor example, it receives location of the doctor mobile object each time it travels to it). The following Mobile Maude code represents the modified doctor configuration in the object diagram.

```
mod PROCESS-DOCTOR is
   ...
  < o(l(IP, 0), 0) : MobileObject | mod :upModule('DOCTOR, false),
      s : upTerm(< o(l(IP, 0), 0) : Doctor | status : done,
      model-check:o(l(IP,0), 1), home-location : o(l(IP, 0), 0),
      patients : o(l(IP, 1), 0) . o(l(IP, 2), 0) . o(l(IP, 3), 0)>
      & none), gas : 200, mode : active, hops : 0 >
  < o(l(IP, 0), 1) : MobileObject | model : nil,
      model-check : waiting-model,
      formula : ((<> (location(l(IP, 2)))) /\
         (location(l(IP,2)) -> [](~ location(l(IP,2))))),
      mod : upModule('MC, false),gas : 200,mode :active,hops :0> .
endm
```

The mobile object identified by $l(IP,0),1)$ is the model checker mobile object and its belly module MC is the Maude module containing rewriting theory for building the required model to be the value of the attribute *model*. The LTL property in the attribute $formula$ is used to verify the constructed model against the property that the mobile object doctor eventually reaches the location of the mobile object of the patient $pat2$ and it will never return to it, which formally is expressed by

$$\Diamond(loc(l(IP,2))) \wedge (loc(l(IP,2)) \implies \Box(\neg loc(l(IP,2))))$$

The result of model checking is shown in Figure 3 (the value of the attribute model-check is true, which means the property is satisfied).

5 Implementation

The objective of the implementation is to develop a tool by which we can generate automatically the Maude specifications from the UML diagrams to do simulation and model checking of LTL properties. When

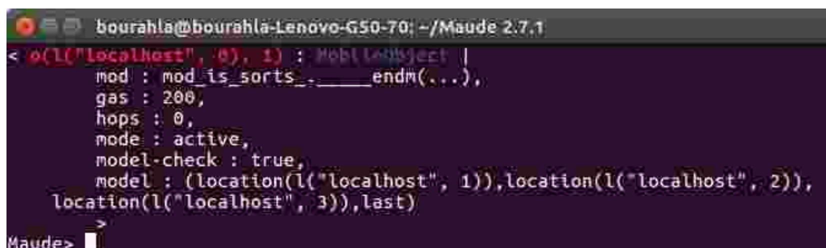

Figure 3. Model construction and model checking results

the simulation results and LTL properties are satisfied, we can use this implemented tool to generate concrete mobile application (its code), which can be deployed on mobile devices. For this automatic generation, we use generation and transformation of models based on Model Driven Architecture (MDA) [16]. Triple Graph Grammars (TGGs) are used to specify bidirectional model transformation [14]. Consistency relations can be specified with TGGs as rules [17] to control the correspondence between the source and target models. With TGGs, to guarantee that a source model is consistent with a target model, we specify a correspondence model between them, which will be used by a set of TGG rules for checking the consistency. This correspondence model should be conforming to a defined correspondence meta-model. Also, the source and target models conform to their corresponding meta-models that are connected by the correspondence meta-model. The triple of source, target and correspondence meta-models is referred to as a TGG schema [18].

The class (declarative parts) and statechart (behaviour parts) diagrams are used to generate the Maude specifications of modules representing the migrant objects. The EMF (Eclipse Modelling Framework) ECore of the Eclipse platform is used for the implementation of this automatic transformation. Thus, the meta-modelling layer of the EMF tool is used to graphically model the meta-models of the class and statechart diagrams, which represent the source meta-models and the meta-model of the mobile Maude as the target meta-model. The

correspondence meta-model is added to connect the source and the target meta-models. The Graphical Modelling Framework (GMF) [19] tool will be then used for editing the different models of the MAMO application based on the specified formalism defined by the created meta-models.

The TGG Interpreter [20] is used to generate mobile Maude models by executing graph rewriting rules based on the source, target, and correspondence meta-models. These rules can be specified to do model-to-model (M2M) or model-to-text (M2T) transformations using the TGG technique. The text generated in the model-to-text transformation is a mobile Maude code, which is generated with respect to defined templates. The template is a target text containing holes as variable parts. These holes contain meta-code, this means code creating code. The variable parts are computed during the template instantiation time. In this case, we use Xpand language to create templates for code generation of mobile Maude modules from EMF models of mobile Maude.

For the verification (simulation and formal verification) of these migrant objects in the located configurations of processes, we generate from object diagrams representing the located configurations the mobile Maude processes. By the same way, using TGG transformations as explained before, we realize this task. We use Maude LTL Model-Checker to verify the state-transition model constructed during simulation against specified LTL properties, a counter example is generated in case a property fails.

By the same process, Figure 4 shows the steps of the code generation, where the mobile Maude modules and located configurations are used as inputs to generate Android Java Code [21] to be deployed on Android devices. However, the programmer will be asked to add additional Java code for user interface, as to make secure logins, to input user data and to handle information as key data or SQL data base, etc.

6 Conclusions and perspectives

A technique to use software engineering principles for developing mobile applications is proposed. The UML class and statechart diagrams

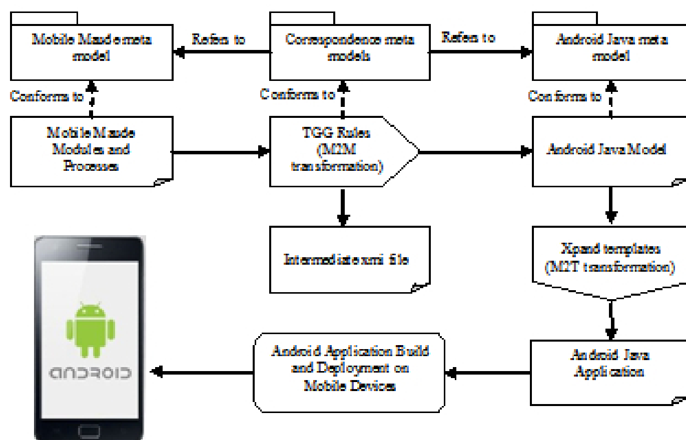

Figure 4. Generation of mobile application code

modelling the mobile app, are first transformed to mobile rewriting theories to be the belly modules of mobile objects that will reside in processes containing root objects (server root object and client root objects) to manage communication between the different mobile objects and hence creating located configurations by transformation of object diagrams. Thus, it is possible to execute each located configuration in a process to simulate these UML models and the execution results will be checked and compared with the desired behaviours. We can also formally verify these located configurations using the model-checking technique. At the end, an equivalent Android mobile application can be generated using a transformation process of mobile rewriting theories to Android Java code to be built and deployed on mobile devices. This software engineering technique is implemented as a prototype and extensively tested with small examples. The implementation results encouraged us to continue with this work. As perspectives, we would like to integrate it with the existing tools for development of mobile applications.

References

[1] A. I. Wasserman, "Software engineering issues for mobile application development," in *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, G. Roman and K. J. Sullivan, Eds. ACM, 2010, pp. 397–400.

[2] L. Corral, A. Sillitti, and G. Succi, "Software assurance practices for mobile applications - A survey of the state of the art," *Computing*, vol. 97, no. 10, pp. 1001–1022, 2015.

[3] L. Corral, A. B. Georgiev, A. Sillitti, and G. Succi, "Can execution time describe accurately the energy consumption of mobile apps? an experiment in android," in *Proceedings of the 3rd International Workshop on Green and Sustainable Software, GREENS 2014, Hyderabad, India, June 1, 2014*, H. A. Müller, P. Lago, M. Morisio, N. Meyer, and G. Scanniello, Eds. ACM, 2014, pp. 31–37.

[4] I. do Nascimento Mendes and A. C. Dias-Neto, "A process-based approach to test usability of multi-platform mobile applications," in *Design, User Experience, and Usability: Design Thinking and Methods - 5th International Conference, DUXU 2016, Held as Part of HCI International 2016, Toronto, Canada, July 17-22, 2016, Proceedings, Part I*, ser. Lecture Notes in Computer Science, A. Marcus, Ed., vol. 9746. Springer, 2016, pp. 456–468.

[5] F. Durán, A. Riesco, and A. Verdejo, "A distributed implementation of mobile maude," *Electr. Notes Theor. Comput. Sci.*, vol. 176, no. 4, pp. 113–131, 2007.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, J. Meseguer, N. Martí-Oliet, and C. L. Talcott, Eds., *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, ser. Lecture Notes in Computer Science, vol. 4350. Springer, 2007.

[7] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The maude LTL model checker," *Electr. Notes Theor. Comput. Sci.*, vol. 71, pp. 162–187, 2002.

[8] A. I. Wasserman, "Developing mobile software with FLOSS," in *Open Source Systems: Long-Term Sustainability - 8th IFIP WG 2.13 International Conference, OSS 2012, Hammamet, Tunisia, September 10-13, 2012. Proceedings*, ser. IFIP Advances in Information and Communication Technology, I. Hammouda, B. Lundell, T. Mikkonen, and W. Scacchi, Eds., vol. 378. Springer, 2012, pp. 401–402.

[9] L. Brisolara, A. Parada, and M. Marques, "Automating mobile application development: Uml-based code generation for android and windows phone," *Revista de Informática Teórica e Aplicada: RITA*, vol. 22, pp. 31–50, 11 2015.

[10] D. Kim, "Development of mobile cloud applications using uml," *International Journal of Electrical and Computer Engineering*, vol. 8, pp. 596–604, 02 2018.

[11] S. Nakajima, "Formal analysis of android application behavior with real-time maude," in *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*, 2015, pp. 7–12.

[12] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11, 2013*. IEEE Computer Society, 2013, pp. 15–24.

[13] "https://developer.android.com/studio/index.html, Android Studio," accessed: 2019-06-01.

[14] A. Anjorin, E. Leblebici, and A. Schürr, "20 years of triple graph grammars: A roadmap for future research," *ECEASST*, vol. 73, 2015.

[15] M. Clavel, F. Durán, S. Eker, P. Lincoln, J. Meseguer, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, "LTL model checking," in *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, 2007, pp. 385–418.

[16] A. Kleppe, J. Warmer, and W. Bast, *MDA explained - the Model Driven Architecture: practice and promise*, ser. Addison Wesley

object technology series. Addison-Wesley, 2003.

[17] M. Bendiaf, M. Bourahla, M. Boudia, and S. Rehab, "A model transformation approach for specifying real-time systems and its verification using rt-maude," *IJITWE*, vol. 12, no. 4, pp. 22–41, 2017.

[18] E. Leblebici, A. Anjorin, A. Schürr, S. Hildebrandt, J. Rieke, and J. Greenyer, "A comparison of incremental triple graph grammar tools," *ECEASST*, vol. 67, 2014.

[19] "URL: https://www.eclipse.org/modeling/gmp/, Graphical Modeling Framework," accessed: 2019-04-10.

[20] "http://jgreen.de/tools/tgg-interpreter/, TGG Interpreter," accessed: 2019-04-10.

[21] M. Murphy, *The Busy Coder's Guide to Advanced Android Development*. CommonsWare, LLC, 2009. [Online]. Available: https://books.google.dz/books?id=fQmvPwAACAAJ

Aissa ElMahdi Bourahla, Mustapha Bourahla

Received July 5, 2020
Accepted October 6, 2021

Aissa ElMahdi Bourahla
Computer Science Department, University of M'Sila
BP. 166 Ichebilia, M'Sila, 28000, Algeria
Phone: +213 0540834764
E–mail: `aissa.bourahla@univ-msila.dz`

Mustapha Bourahla
Laboratory of Informatics and its Applications, Computer Science Department, University of M'Sila
BP. 166 Ichebilia, M'Sila, 28000, Algeria
Phone: +213 0778574572
E–mail: `mustapha.bourahla@univ-msila.dz`