People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

Mohamed Boudiaf University - M'Sila

Faculty of Mathematics and Computer Science

Computer Science Department

Master Memory

In Candidacy for the Degree of Master in Computer Science

Option

Advanced Information Systems and Software Engineering

by

Aissa ElMahdi Bourahla

TITLE

Formal Development of Mobile Applications based on Migrant Objects (MAMOs)

In front of the jury composed of:

Mrs. Amel Meliouh	University of M'Sila	President
Mr. Mustapha Bourahla	University of M'Sila	Supervisor
Mrs. Malika Boudia	University of M'Sila	Examiner

Promotion 2019 – 2020

ملخص

تعتبر مبادئ هندسة البرمجيات ضرورية جدًا لتطوير تطبيقات الأجهزة المحمولة ، والتي تعد ضرورية للعديد من تطبيقات الحياة. في هذا العمل، نقدم عملية تطوير تعتمد على تحولات لنماذج UML ، والتي تصف تطبيقات الهاتف المحمول بناءً على اشياء مهاجرة إلى لغة Mobile Maude . يمكن بناء نماذج انتقال الحالة أثناء المحاكاة لوصف سلوكيات تطبيقات الهاتف المحمول بناءً على الأشياء المهاجرة ، والتي سيتم التحقق منها باستخدام تقنية فحص النموذج. يمكن استخدام النموذج الذي تم التحقق منه لتطبيق الهاتف المحمول المستند إلى الأشياء المهاجرة لإنشاء تطبيقات Android لنشرها على الأجهزة المحمولة.

الكلمات الدالة الهندسة البرمجية القائمة على النموذج ، التحول من نموذج إلى نموذج ، تحويل نموذج إلى نص ، تطبيق الهاتف المحمول ، إعادة الكتابة المنطقية ، TGG, Xpand, Mobile . Maude, Maude LTL Model Checker

Résumé

Les principes du génie logiciel sont très nécessaires pour le développement d'applications mobiles, qui sont nécessaires pour de nombreuses applications de la vie. Dans ce travail, nous présentons un processus de développement basé sur les transformations TGG et Xpand des modèles UML, qui décrivent des applications mobiles basées sur des objets migrant vers le langage Mobile Maude qui seront exécutées pour la simulation et la vérification formelle des propriétés LTL avec la technique de vérification de modèle. Le modèle vérifié d'application mobile peut être utilisé pour générer des applications Android à déployer sur des appareils mobiles.

Mots clés **:** Architecture pilotée par modèle, transformation modèle en modèle avec triple grammaire de graphe, transformation Xpand modèle en texte, application mobile, objets migrants, logique de réécriture, Mobile Maude, Maude LTL Model Checker.

Abstract

Software engineering principles are very required for the development of mobile applications, which are necessary for many life applications. In this work, we present a development process based on TGG and Xpand transformations of UML models, which describe mobile applications based on migrant objects to Mobile Maude language. The generated rewriting theories will be executed for simulation. State transition models may be built during simulation, 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 applications to be deployed on mobile devices.

Keywords **:** Model-Driven Architecture, TGG (Triple Graph Grammars) Model-to-Model Transformation, Xpand/Xtend Model-to-Text Transformation, Mobile Application, Migrant Objects, Rewriting Logic, Mobile Maude, Maude LTL Model Checker.

Acknowledgements

I thank Allah the Almighty who gave me the health to complete this project.

My heartfelt thanks go to:

My supervisor, Mr. **Mustapha Bourahla**, professor at the Mohamed BOUDIAF University of M'Sila, to guide me, to have advised me and to encourage me throughout the realization of this project, and to my teacher **Mrs. Amel Meliouh** who did me the honor of chairing the evaluation jury for this Master memory and my teacher **Mrs. Malika Boudia** who did me the honor to examine this project.

My thanks also go to:

All the teachers who taught me during these five years in the computer science department at the University Mohamed Boudiaf of M'Sila.

Contents

List of figures

General introduction

Context and motivation

THE mobile application development has exponential growth, which are now necessary for many life applications. There are many manuals to guide developers of mobile applications to use good practices for development, 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 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.

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 [38]. There is need for customized tools to support cross-platform development and testing. The characteristics of mobile applications and their operating environments present new issues for software engineering research.

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 development of techniques for testing. Development of the mobile application is test driven [30] and it will typically be done within the context of the overall software development effort.

The mobile applications can be classified to four classes. The native mobile applications, which are designed to be run on independent mobile devices, the second class of mobile applications are web-based mobile applications, which are applications to rend services for users (clients) from web servers. The third class is the hybrid mobile applications. The fourth class of mobile applications is an extension of the web-based mobile applications, where the mobile applications representing the clients and the server can allow migration of small code (called objects) to be executed on the platform of the destiation device to realise some tasks.

It is essential to apply software engineering techniques to assure the development of high-quality complex mobile applications. Techniques for developing mobile applications are similar to software engineering for other embedded applications with additional requirements. The motivation of this master memory is to use software engineering principles, which are very required for the development of mobile applications that are necessary for many life applications in particular web-based mobile applications with migrant objects.

Problematic

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 [38, 9].

Mobile devices have more complex functionality than those of desktop or laptop computers. The smaller display and different styles of user interaction also have a major impact on interaction design for mobile applications, which in turn has a strong influence on application development. The mobile user interface is developed around widgets, touch, physical motion, physical and virtual keyboards.

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 [10, 11]. They should also design applications differently depending on the speed of the network on which they are being used.

Contribution

In this dissertation, 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 its own mobile device to other mobile client application owned by its patients.

Hence, the mobile client application now running on the patient 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 the Unified Modelling Language (UML) [6] for modelling the mobile application based on migrant objects. For each mobile application based on migrant objects, we develop the class, object and state chart diagrams. These diagrams are used to generate specifications as rewriting theories implemented by the Mobile Maude language [13], which extends the system Maude [7] 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 [17]. 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.

An implementation of automatic generation of Mobile Maude specifications from UML diagrams based on Triple Graph Grammars (TGG) [33, 4] is realized using the Eclipse Modelling Framework (EMF) for model-to-model transformation and Xpand for model-to-text transformation.

Related works

Development of mobile applications is the process of modelling, implementing, testing and finally deploying an application that is expected to satisfy the need of many users and making sure the standard of quality is met. As the demand for mobile applications continues to grow, proposing solutions that help save developers time to produce higher quality apps will continue to increase as well. 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 [36, 37] 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 in [25] have written a survey as result of interviews with mobile application developers. With this survey, we can understand the main challenges that face the 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 [2], which has replaced Eclipse Android Development Tools [14, 15] as the primary IDE for native Android application development. It provides code editing, debugging, and

testing tools within the development environment using emulators. Android Studio is used in conjunction with Gradle [21] to add external libraries, which is based on Apache Maven and Apache Ant. Gradle is one of the most popular development tools for creating large-scale applications involving Java.

Android Studio includes the Android Debug Bridge (ADB) [1], which is a bridge of communication between Android devices and computers used during development. Another useful feature of Android Studio is the AVD (Android Virtual Device) Manager [3], which is an emulator used to run Android applications on a computer. The AVD is used to work with all types of Android devices to test responsiveness and performance on different versions, screen sizes, and resolutions.

Twitter's mobile application can be developed on a development platform called Fabric[18], which includes everything from beta-testing to marketing and advertising tools. For Android game developers, GameMaker: Studio [19] is the most popular development tool, which provides everything you need to create 2D games. The Android emulator Genymotion [**Genymotion**] is a cross-platform development tool that helps developers test and preview an application. IntelliJ IDEA [23] is designed at JetBrains for ultimate programmer productivity. It?s extremely fast and features a full suite of development tools.

Dissertation organization

In addition to the general introduction and conclusion, this dissertation consists of four chapters. The first chapter presents state-of-the-art of the mobile applications, while the remaining three chapters are as much contributions for the proposition of this technique of using software engineering principles for the development of mobile applications. The dissertation is therefore organized as follows:

We have presented in chapter 1 the mobile applications, where their definition, types and characteristics are presented in detail in addition to their use in the market as their management and distribution. This chapter focuses on the development process of the mobile applications and it clarifies the need of their formal development.

The second chapter 2, which is a contributing chapter, describes in detail the proposed technique of using the principles of software engineering for the development of

mobile applications based on migrant object as extension of web-based mobile appilcations.

In chapter 3, an implemention of the technique using the model drive architecture for model to model and model to text transformations with the technique of Triple Graph Grammars (TGGs) is explained in detail. Along all this dissertation, a case study is used in the domain of health care, which is detailed in the fourth chapter 4. A set of appendices containing necessary code of implementation and case study results are also presented.

Finally, this dissertation ends with a general conclusion followed by some perspectives and a view on future work. Where the strongness and the weakness of this technique are presented.

Chapter 1

Mobile applications

1.1 Introduction

MOBILE application, which is called mobile app or simply app, is a software application programmed to run on a mobile devices such as phones or tablets. Today, apps are used in many areas like email, calendar, contact databases, mobile games, factory automation, GPS and location-based services, order-tracking, and ticket purchases. This rapid expansion makes that there are now millions of apps available.

The owner of the mobile operating system, such as the App Store (iOS) and Google Play Store have application distribution platforms from which, we can download apps, where many of them are free. Mobile applications often stand directly on the mobile device in contrast to desktop applications which are designed to run on desktop computers, and web applications which run in mobile web browsers [24].

When buying a mobile device, we can get several pre-installed apps, such as a web browser, email client, calendar, media, or more apps. Apps that are not preinstalled are usually available through distribution platforms called app stores, which are typically operated by the owner of the mobile operating system, such as the Apple App Store, Google Play, Windows Phone Store, and BlackBerry App World. Usually, they are downloaded from the platform to a target device, but sometimes they can be downloaded to laptops or desktop computers. It is possible to install manually apps, for example by running an Android application package on Android devices.

The main service tasks of mobile apps were the productivity and information retrieval, including email, calendar, contacts, the stock market and weather information. However, public demand and the availability of developer tools drove rapid expansion into other categories, such as those handled by desktop application software packages. With a growing number of mobile applications available at app stores and the improved capabilities of smartphones, people are downloading more applications to their devices [32].

1.2 Classes of mobile applications

We can classify mobile applications to four classes : native, web-based, hybrid and migrant objects-based apps.

- Native app: Native apps are developped for a particular mobile platform. Therefore, an app developped for Apple device does not run in Android devices. Developing native apps makes incorporating best-in-class user interface modules, which helps for better performance, consistency and good user experience. The main purpose for creating such apps is to ensure best performance for a specific mobile operating system.
- Web-based app: A web-based app is coded in HTML5, CSS or JavaScript. Internet access is required for proper behaviour and user-experience of this group of apps. These apps may capture minimum memory space in user devices compared to native and hybrid apps. Since all the personal databases are saved on the Internet servers, users can fetch their desired data from any device through the Internet.
- Hybrid app: The concept of the hybrid app is a mix of native and web-based apps. Apps developed using Apache Cordova, Xamarin, React Native, Sencha Touch and other similar technology fall into this category. These are made to support web and native technologies across multiple platforms. Moreover, these apps are easier and faster to develop. It involves use of single code base which works in multiple mobile operating systems. Despite such advantages, hybrid apps exhibit lower performance. Often, apps fail to bear the same look-and-feel in different mobile operating systems.

- Migrant objects-based app: This kind of mobile applications is extension of the class of web-based applications, where small piece of code representing an object (agent) program can migrate from a device to another, where it can be executed to realise a particular task. In this manuscript, we are interested by this class of mobile applications.

1.3 Mobile application development

Mobile app development is the process by which a mobile app is developed for mobile devices to be pre-installed on phones during manufacturing platforms, or delivered as web applications using server-side or client-side processing (e.g., JavaScript) to provide an "application-like" experience within a Web browser.

Developing apps for mobile devices requires considering the constraints and features of these devices, such as screen sizes, hardware specifications, and configurations. Mobile devices run on battery and have less powerful processors than personal computers and also have more features such as location detection and cameras.

Mobile application development requires the use of specialized integrated development environments. Mobile apps are first tested within the development environment using emulators, which are hardware or software that enables one computer system (called the host) to behave like mobile device (called the guest).

An emulator typically enables the host system to run software or use peripheral devices designed for the guest system and later subjected to field testing. Emulators provide an inexpensive way to test applications on mobile phones to which developers may not have physical access [20].

As part of the development process, mobile user interface (UI) design is also essential in the creation of mobile apps. Mobile UI considers constraints, contexts, screen, input, and mobility as outlines for design. Mobile UI design constraints include limited attention and form factors, such as a mobile device's screen size for a user's hand(s). Mobile UI contexts signal cues from user activity, such as location and scheduling that can be shown from user interactions within a mobile app.

Conversational interfaces display the computer interface and present interactions

through text instead of graphic elements. They emulate conversations with real humans. There are two main types of conversational interfaces: voice assistants (like the Amazon Echo) and chatbots.

1.3.1 Platform for developping mobile applications

The platform organizations needed to develop, deploy and manage mobile apps are made from many components and tools which allow a developer to write, test and deploy applications into the target platform environment.

1.3.1.1 Front-end development tools

Front-end development tools are focused on the user interface and user experience (UI-UX) and provide the following abilities:

- UI design tools
- SDKs to access device features
- Cross-platform accommodations/support

There are many notable tools. We are interested by Android, which has the following characteristics. The programming language is Java (but portions of code can be in C, C++, Kotlin). It has a debugger or debugging tool, which is a computer program used to test and debug programs.

It can be integrated in Eclipse and standalone debugging monitor, which has an emulator. It has An integrated development environment (IDE) (as Eclipse, IntelliJ IDEA, Android Studio, Project Kenai Android plugin for NetBeans), which is a software application that provides comprehensive facilities to computer programmers for software development.

An IDE normally consists of at least a source code editor, build automation tools and a debugger. Some IDEs, such as NetBeans and Eclipse, contain the necessary compiler, interpreter, or both; others, such as SharpDevelop and Lazarus, do not. It

has cross-platform deployement Android only and it has an installation tool called apk. In addition is free and the IntelliJ IDEA Community Edition is free.

1.3.1.2 Back-end servers

Back-end tools pick up where the front-end tools leave off, and provide a set of reusable services that are centrally managed and controlled and provide the following abilities:

- Integration with back-end systems
- User authentication-authorization
- Data services
- Reusable business logic

1.3.1.3 Security add-on layers

With bring your own device (BYOD) becoming the norm within more enterprises, IT departments often need stop-gap, tactical solutions that layer atop existing apps, phones, and platform component. Features include

- App wrapping for security
- Data encryption
- Client actions
- Reporting and statistics

1.3.1.4 System software

Many system-level components are needed to have a functioning platform for developing mobile apps.

- Ubuntu Touch Web-based: HTML5, CSS, JavaScript Native: QML, C, C++ Yes Yes Ubuntu SDK HTML5 app to be available web browser. Ubuntu Touch through App store, Web URL Development requires Ubuntu Desktop 12.04 or higher, Free
- Windows Mobile C, C++ Yes Free emulator (source code available), also bundled with IDE Visual Studio 2010, 2008, 2005, eMbedded VC++ (free), Satellite Forms Windows Mobile, Windows CE OTA deployment, CAB files, ActiveSync Free command-line tools or eMbedded VC++, or Visual Studio (Standard edition or better)

Criteria for selecting a development platform usually contains the target mobile platforms, existing infrastructure and development skills. When targeting more than one platform with cross-platform development it is also important to consider the impact of the tool on the user experience.

Performance is another important criteria, as research on mobile apps indicates a strong correlation between application performance and user satisfaction. Along with performance and other criteria, the availability of the technology and the project's requirement may drive the development between native and cross-platform environments.

To aid the choice between native and cross-platform environments, some guidelines and benchmarks have been published. Typically, cross-platform environments are reusable across multiple platforms, leveraging a native container while using HTML, CSS, and JavaScript for the user interface.

In contrast, native environments are targeted at one platform for each of those environments. For example, Android development occurs in the Eclipse IDE using Android Developer Tools (ADT) plugins, Apple iOS development occurs using Xcode IDE with Objective-C and/or Swift, Windows and BlackBerry each have their own development environments.

1.3.1.5 Mobile app testing

Mobile applications are first tested within the development environment using emulators and later subjected to field testing. Emulators provide an inexpensive way to test

applications on mobile phones to which developers may not have physical access. The following are examples of tools used for testing application across the most popular mobile operating systems.

- Google Android Emulator: An Android emulator that is patched to run on a Windows PC as a standalone app, without having to download and install the complete and complex Android SDK. It can be installed and Android compatible apps can be tested on it.
- The official Android SDK Emulator: A mobile device emulator which mimics all of the hardware and software features of a typical mobile device (without the calls).
- TestiPhone: A web browser-based simulator for quickly testing iPhone web applications. This tool has been tested and works using Internet Explorer 7, Firefox 2 and Safari 3.
- iPhoney: It gives a pixel-accurate web browsing environment and it is powered by Safari. It can be used while developing web sites for the iPhone. It is not an iPhone simulator but instead is designed for web developers.
- BlackBerry Simulator: There are a variety of official BlackBerry simulators available to emulate the functionality of actual BlackBerry products and test how the device software, screen, keyboard and trackwheel will work with application.
- Windows UI Automation: To test applications that use the Microsoft UI Automation technology, it requires Windows Automation API. It is pre-installed on Windows 7, Windows Server 2008 R2 and later versions of Windows. On other operating systems, you can install using Windows Update or download it from the Microsoft Web site.
- MobiOne Developer: A mobile Web integrated development environment (IDE) for Windows that helps developers to code, test, debug, package and deploy mobile Web applications to devices such as iPhone, BlackBerry, Android, and the Palm Pre.

These tools are supported by a GUI-based automated test tool for mobile app across all operating systems and devices, test automation tools for mobile, web and desktop apps and real mobile devices and test automation tools for testing mobile and web apps.

1.4 Distribution of mobile applications

The three biggest app stores are Google Play for Android, App Store for iOS, and Microsoft Store for Windows 10, Windows 10 Mobile, and Xbox One [34].

- Google Play : Google Play (formerly known as the Android Market) is an international online software store developed by Google for Android devices. Most of apps in the Google Play Store are free to download.
- App Store (iOS) : Apple's App Store for iOS was not the first app distribution service, but it ignited the mobile revolution.
- Microsoft Store (digital) : Microsoft Store (formerly known as the Windows Store) was introduced by Microsoft in 2012 for its Windows 8 and Windows RT platforms. While it can also carry listings for traditional desktop programs certified for compatibility with Windows 8, it is primarily used to distribute "Windows Store apps" which are primarily built for use on tablets and other touch-based devices (but can still be used with a keyboard and mouse, and on desktop computers and laptops).
- Amazon Appstore is an alternative application store for the Android operating system. The Amazon Appstore's Android Apps can also be installed and run on BlackBerry 10 devices.
- BlackBerry World is the application store for BlackBerry 10 and BlackBerry OS devices.
- Windows Phone Store was introduced by Microsoft for its Windows Phone platform.
- Samsung Apps offers apps for Windows Mobile, Android and Bada platforms.
- The Electronic AppWrapper was the first electronic distribution service to collectively provide encryption and purchasing electronically
- F-Droid Free and open Source Android app repository.
- Opera Mobile Store is a platform independent app store for iOS, Java, BlackBerry OS, Symbian, iOS, and Windows Mobile, and Android based mobile phones.
- There are numerous other independent app stores for Android devices.

1.5 Mobile application management

Mobile application management (MAM) describes software and services responsible for provisioning and controlling access to internally developed and commercially available mobile apps used in business settings. The strategy is meant to off-set the security risk of a Bring Your Own Device (BYOD) work strategy.

When an employee brings a personal device into an enterprise setting, mobile application management enables the corporate IT staff to transfer required applications, control access to business data, and remove locally cached business data from the device if it is lost, or when its owner no longer works with the company.

Containerization is an alternate BYOD security solution. Rather than controlling an employees entire device, containerization apps create isolated and secure pockets separate from all personal data. Company control of the device only extends to that separate container.

1.5.1 App wrapping vs. native app management

Especially when employees bring your own device, mobile apps can be a significant security risk for businesses, because they transfer unprotected sensitive data to the Internet without knowledge and consent of the users. Reports of stolen corporate data show how quickly corporate and personal data can fall into the wrong hands. Data theft is not just the loss of confidential information, but makes companies vulnerable to attack and blackmail.

Professional mobile application management helps companies protect their data. One option for securing corporate data is app wrapping. But there also are some disadvantages like copyright infringement or the loss of warranty rights. Functionality, productivity and user experience are particularly limited under app wrapping. The policies of a wrapped app can't be changed. If required, it must be recreated from scratch, adding cost.

An app wrapper is a mobile app made wholly from an existing website or platform [31], with few or no changes made to the underlying application. The "wrapper" is essentially a new management layer that allows developers to set up usage policies ap-

propriate for app use. Examples of these policies include whether or not authentication is required, allowing data to be stored on the device, and enabling/disabling file sharing between users.

Because most app wrappers are often websites first, they often do not align with iOS or Android Developer guidelines. Alternatively, it is possible to offer native apps securely through enterprise mobility management without limiting the native user experience. This enables more flexible IT management as apps can be easily implemented and policies adjusted at any time.

1.6 Conclusion

This first chapter presented the utility of the mobile applications in our life. This makes their development having exponential growth. 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 around their engineering processes.

It is essential to apply software engineering techniques to assure the development of high-quality complex mobile applications. Our contribution focuses on using formal techniques for the development of mobile applications, which will be explained in detail in Chapter two (2), where its implementation is presented in Chapter three (3). The implementation is applied on a case sudy presented in Chapter four (4) with some results.

Chapter 2

Software Engineering for the Development of Mobile Applications

2.1 Introduction

Software engineering principles are very required for the development of mobile applications, which are necessary for many life applications. In this chapter, 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 Java application to be deployed on mobile devices.

We will present how to model a mobile application based on migrant objects with UML to be transformed to Mobile Maude specifications. The simulation of the specifications and their formal verification against LTL properties will be explained in detail.

2.2 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 [7]. 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, state chart and object diagrams). A class diagram is composed of three classes (Figure 2.1). The first two classes are called "ServerRootObject" and "ClientRootObject", which are generalizations of the class "RootObject" and they have different behaviours. The third class is called "MobileObject", which is used to create mobile objects.

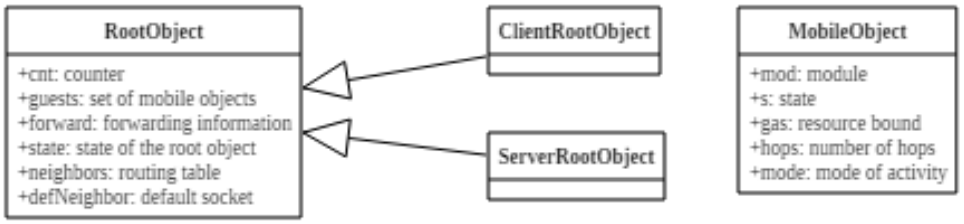

Figure 2.1 – Main classes: RootObject, ServerRootObject, ClientRootObject and MobileObject

To facilitate the transformation process, the definitions of these classes are based on

Mobile Maude implementation [13]. 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 "'Client-RootObject". 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" that will be used 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.

2.3 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 state chart 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 [13] is a mobile object language, which extends Maude [7] for specifying mobile computation. The formal semantics of Mobile Maude is defined as a rewrite theory within the rewriting logic. 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 is 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 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.

The communication can also 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 by the specified location L as first tentative, if A doesn't find the object B, it will go on looking for B in 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 operation of sending the go message to inter mobile objects,

where the sender mobile object is indicated as 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 objet of this message.

2.3.1 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.

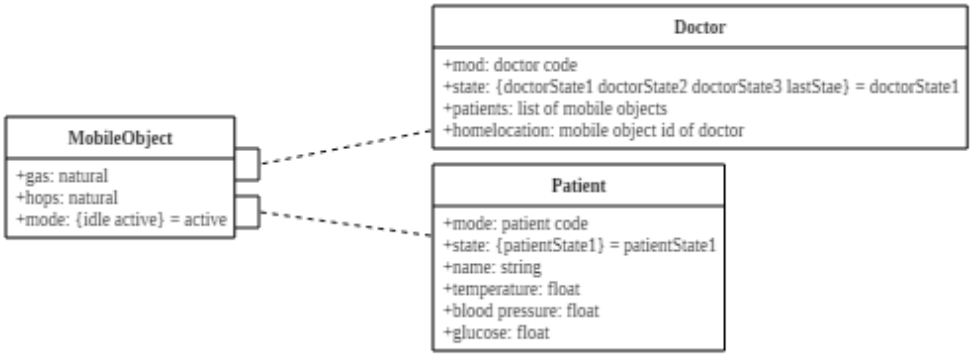

Figure 2.2 – Extension of the class diagram to model the doctor and patient classes

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 (Figure 2.2), which extends the class diagram of Figure 2.1 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 state chart 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 state chart diagram (Figure 2.3) summarizes their behaviours.

The transformation to Mobile Maude code is based on formal semantics given to the state chart 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 state chart diagram is represented by

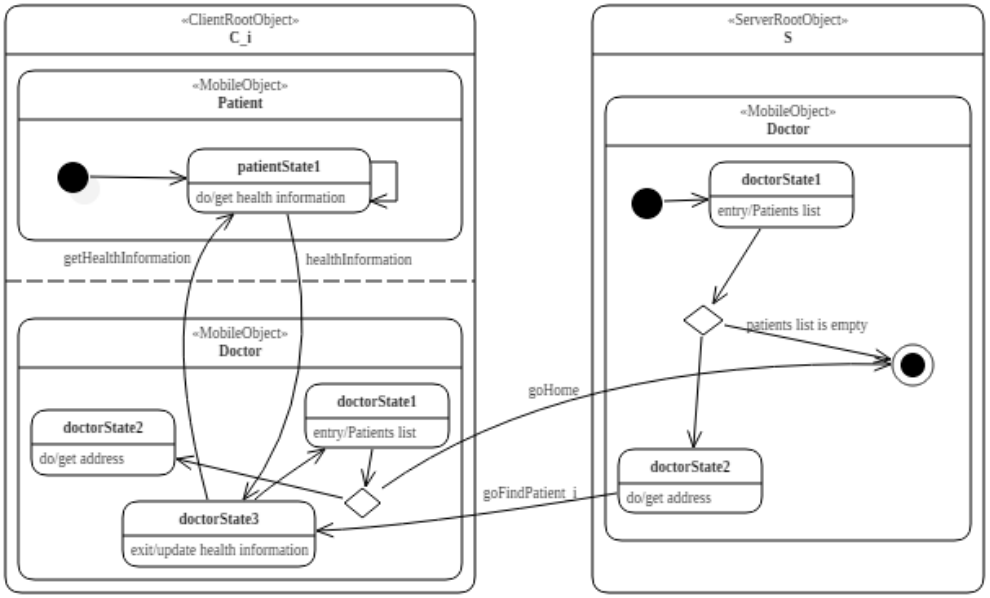

Figure 2.3 – State chart diagram of the mobile objects doctor and patient

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 state chart 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*),

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

else it goes to the state $State2$, from which it can go to find the patient location.

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

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 ($get-health-information$) 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 are in its attribute named $health-information$, which can be consulted by mobile device owner.

Patients receive from the doctor, messages of the form *get–health–information*(D), with D 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 state chart 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 state chart 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
  ex MOBILE-OBJECT-ADDITIONAL-DEFS .
  ex FLOAT .
  sort Patient .
  subsort Patient < Cid .
  op Patient : -> Patient .
  var V@Patient : Patient .
  op name :_ : String -> Attribute [ctor] .
  op temperature :_ : Float -> Attribute [ctor] .
  op blood-pressure :_ : Float -> Attribute [ctor] .
  op glucose :_ : Float -> Attribute [ctor] .
  op get-health-information : Mid -> Contents [ctor] .
  op health-information : String Float Float Float ->
                          Contents [ctor] .
  vars P D : Mid .
  vars  T BP G : Float .
  var Name : String .
  var  AtS : AttributeSet .
  var  Conf : Configuration .
  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
  ex MOBILE-OBJECT-ADDITIONAL-DEFS .
  pr LIST{Mid} * (op __ to _.._, op nil to no-id) .
  pr DEFAULT{Nat} .
  pr DEFAULT{Float} .
  pr DEFAULT{String} .
  pr DEFAULT{Oid} .
  sort DoctorState .
  ops State1 State2 State3 last : -> DoctorState [ctor] .
  sort Doctor .
  subsort Doctor < Cid .
  op Doctor : -> Doctor .
  var V@Doctor : Doctor .
  op patients :_ : List{Mid} -> Attribute [ctor gather(&)] .
  op state :_ : DoctorState -> Attribute [ctor] .
  op health-information :
     _(temperature:_, blood-pressure:_, glucose:_) :
     Default{String} Default{Float} Default{Float}
     Default{Float} -> Attribute [ctor] .
  op home-location :_ : Default{Oid} -> Attribute [ctor] .
  op get-health-information : Mid -> Contents [ctor] .
  op health-information : String Float Float Float ->
     Contents [ctor] .
  vars  P D : Oid .
  var  OP : List{Mid} .
  var  AtS : AttributeSet .
  var N : Nat .
  vars T BP G : Float .
  var Name : String .
  var  L : Loc .
  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
```

2.4 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, represented by the object diagram model in Figure 2.4, 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. The dotted lines in the object diagram represent the possibility of communication by TCP/IP sockets.

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

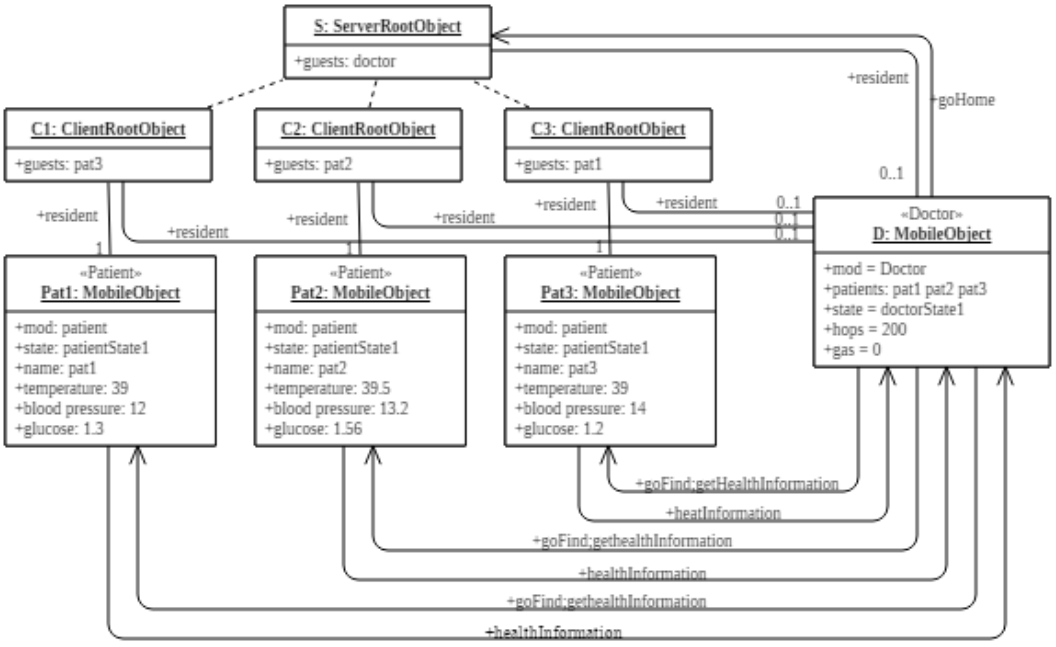

Figure 2.4 – Object diagram to generate located configurations

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 .
  eq IP = "localhost" .
  eq port = 8000 .
  op initial : -> Configuration .
  eq initial =<>
   < l(IP, 0) : ServerRootObject |
      cnt : 1, guests : o(l(IP, 0), 0),
      forward : 0 |-> (l(IP, 0), 0),
      neighbors : empty,
      state : idle, defNeighbor : null >
     < 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 and different health information. These four configurations represent an overall located configuration (processes) of the mobile application MAMO.

The mobile object of the first patient, which has the name "Pat1" is refered by the name "o(l(IP,1),0)" and it is located in a process with the client root object "l(IP,1)". This patient has the following health information: the temperature is 39.0, the blood-pressure is 12.0 and the glucose is 1.3.

```
mod PROCESS-PAT1 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, 1) : ClientRootObject |
        cnt : 1, guests : o(l(IP, 1), 0),
        forward : 0 |-> (l(IP, 1), 0),
        neighbors : empty,
        state : idle, defNeighbor : null >
     < o(l(IP, 1), 0) : MobileObject |
        mod : upModule('PATIENT, false),
        s : upTerm(
             < o(l(IP, 1), 0) : Patient |
                name : "Pat1",
                temperature : 39.0,
                blood-pressure : 12.0,
                glucose : 1.3 > & none),
        gas : 200, mode : active, hops : 0 > .
endm
```

The mobile object of the second patient, which has the name "Pat2" is called "o(l(IP,2),0)" and it is located in a process with the client root object "l(IP,2)". This patient has the following health information: the temperature is 39.5, the blood-pressure is 13.2 and the glucose is 1.56.

```
mod PROCESS-PAT2 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, 2) : ClientRootObject |
        cnt : 1, guests : o(l(IP, 2), 0),
        forward : 0 |-> (l(IP, 2), 0),
        neighbors : empty, state : idle,
        defNeighbor : null >
     < o(l(IP, 2), 0) : MobileObject |
        mod : upModule('PATIENT, false),
        s : upTerm(
            < o(l(IP, 2), 0) : Patient |
               name : "Pat2",
               temperature : 39.5,
               blood-pressure : 13.2,
               glucose : 1.56 > & none),
        gas : 200, mode : active, hops : 0 > .
endm
```

The mobile object of the third patient, which has the name "Pat3" is refered by the name "o(l(IP,3),0)" and it is located in a process with the client root object "l(IP,3)". This patient has the following health information: the temperature is 39.0, the blood-pressure is 14.0 and the glucose is 1.2.

```
mod PROCESS-PAT3 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, 3) : ClientRootObject |
        cnt : 1, guests : o(l(IP, 3), 0),
        forward : 0 |-> (l(IP, 3), 0),
        neighbors : empty,
        state : idle, defNeighbor : null >
     < o(l(IP, 3), 0) : MobileObject |
        mod : upModule('PATIENT, false),
        s : upTerm(
```

```
                < o(l(IP, 3), 0) : Patient |
                    name : "Pat3",
                    temperature : 39.0,
                    blood-pressure : 14.0,
                    glucose : 1.2 > & none),
            gas : 200, mode : active, hops : 0 > .
endm
```

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 shown in Figure 5. 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 $o(l(IP, 2), 0)$.

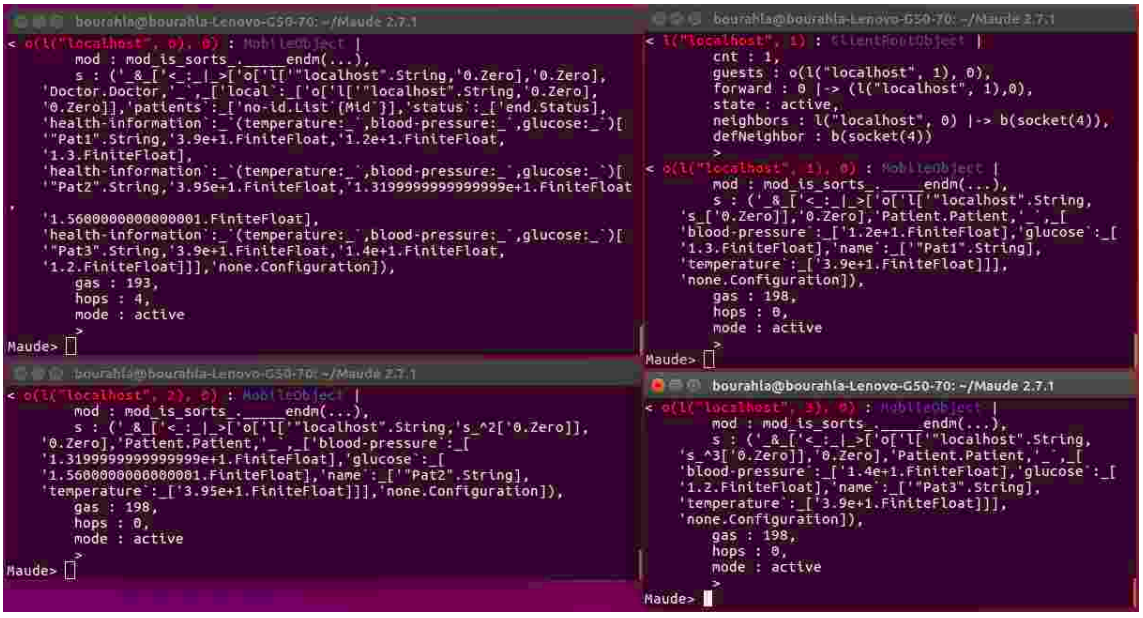

Figure 2.5 – Simulation results

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 location $l(IP, 3)$ 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. We can observe how the doctor has visited the processes of all the patients in the top left window in Figure 2.5, 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 in three other three windows show results for the mobile objects ("Pat1" in the right top window, "Pat2" in left bottom window and "Pat3" in right bottom window). All these results show that the three mobile objects didn't move (the values of their attribute "hops" are zero) and they have consumed two resources ("gas" equals $200 - 198$).

2.4.1 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 [17] 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 fulfils a mobile application MAMO property described by Linear Temporal Logic (LTL) [28], such as safety property (something bad never happens) and liveness property (something good eventually happens) [8].

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 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 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 modified doctor configuration in the object diagram.

```
mod PROCESS-DOCTOR is
  ex DISTRIBUTED-MOBILE-MAUDE .
  ex DOCTOR .
  ex MC .
  op IP : -> String .
  eq IP = "localhost" .
  eq port = 8000 .
  op init : -> Configuration .
  eq init  = <>
     < l(IP, 0) : ServerRootObject |
        cnt : 2, guests : o(l(IP, 0), 0),
        forward : 0 |-> (l(IP, 0), 0),
                  1 |-> (l(IP, 0), 1),
        neighbors : empty,
        state : idle, defNeighbor : null >
     < 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),
              model-check : o(l(IP, 0), 1),
              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

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

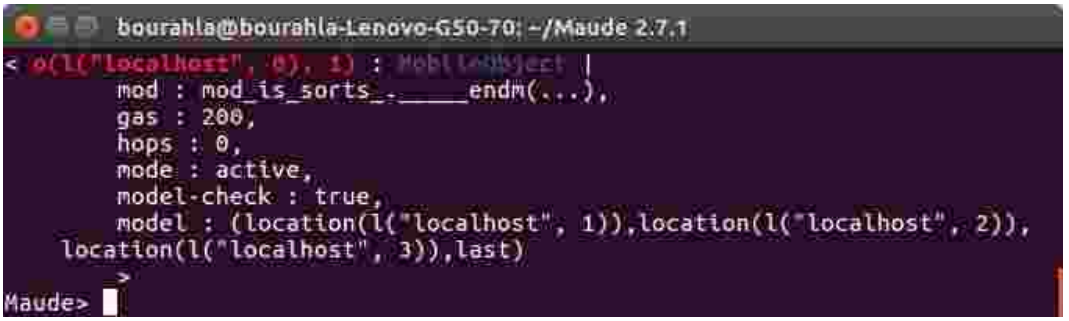

Figure 2.6 – Model construction and model checking results

The result of model checking is shown in Figure 2.6 (the value of the attribute model-check is true, which means the property is satisfied).

2.5 Conclusion

In this chapter, a technique to use software engineering principles for developing mobile applications based on migrant objects is proposed. This technique is based on modelling the mobile application (MAMO) by UML models. The class and state chart diagrams 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 desired behaviours. We can also formally verify these located configurations using the model-checking technique.

The next chapter presents the implementation of this software engineering technique as a prototype, which is extensively tested with an extended case study of the example of healthcare domain where, 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.

Chapter 3

Implementation

3.1 Introduction

This chapter presents an implementation of automatic generation of Mobile Maude specifications from UML diagrams based on Triple Graph Grammars (TGG) [33, 4]. This automatic transformation generates Mobile Maude Processes for simulation and formal verification. From the same UML diagrams, this implementation generates the Android Java mobile code to be deployed on mobile devices.

3.2 Implementation approach

The objective 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 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.

This technique of transformation is illustrated in Figure 3.1. A source model will be written with respect to a source meta-model to model the UML diagrams of class diagrams, state chart diagrams and object diagrams. These source models will be inputs to the transformation engine to generate target models based on target meta-models

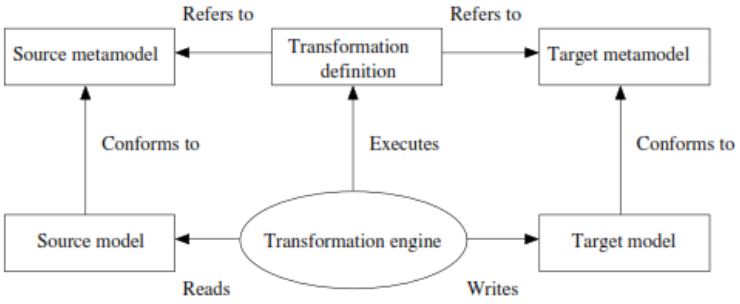

Figure 3.1 – Concepts of model-to-model transformation

of the mobile Maude.

Figure 3.2 details this automatic generation, which has three steps. In the first step, we describe the behaviours of the mobile objects as state chart diagrams based on defined attributes and methods declared in class diagrams. The Maude formal specification of modules representing the migrant objects is generated from class (declarations part) and state chart (behaviour part) diagrams.

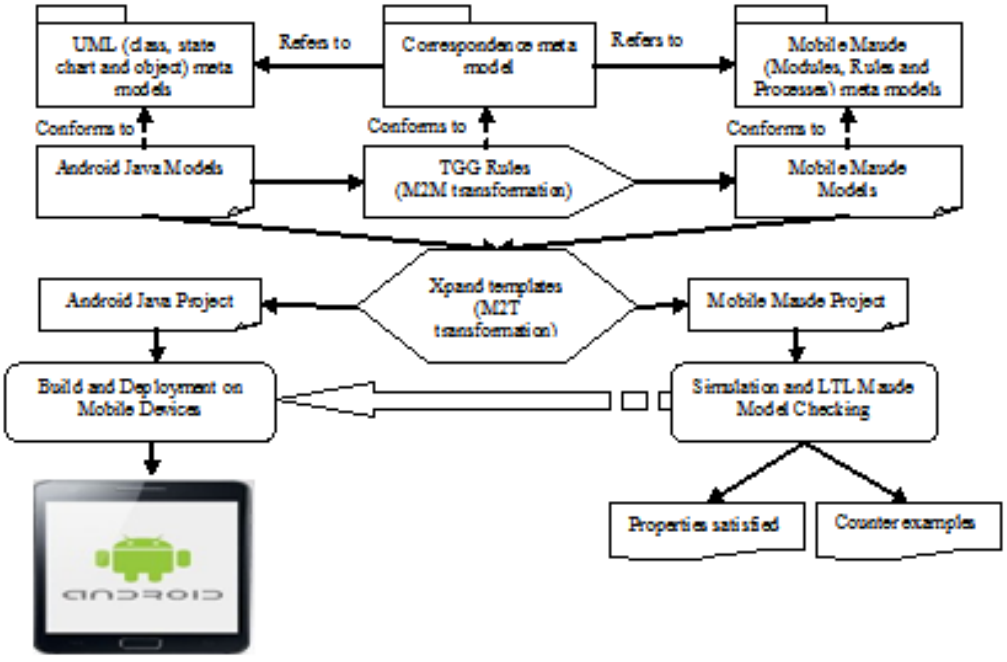

Figure 3.2 – Generation of mobile applications

The transformation of the object diagrams representing the located configurations to mobile Maude models is executed in the second step using the model transformation technique of TGG to generate models of Maude specifications for simulation and verification of these migrant objects in located configurations of processes. 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.

The generation of mobile Maude code (text) as formal specifications from mobile Maude models and the generation of Android Java code (text) for mobile applications are realized in the third step using the model to text transformation (M2T) [29], which is guided by expansion directives written with Xpand template language. The programmer will be asked to add additional Android 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.

For this automatic generation, we use generation and transformation of models based on Model Driven Architecture (MDA)[26, 39, 12]. Triple Graph Grammars (TGGs) are used to specify bidirectional model transformation [33, 4]. Consistency relations can be specified with TGG as rules [5] to control the correspondence between the source and target models. In TGG specification, bidirectional (forward/backward) transformation is supported, which means transforming a source model to a target model and vice-versa.

3.3 Model-to-model transformation with TGG

The EMF (Eclipse Modelling Framework) ECore [16] of the Eclipse platform is used for the implementation of this automatic transformation. The meta-model (Ecore) of EMF Eclipse is used to define models and it provides run-time support for handling the models of the core EMF framework, as to generate interfaces to edit objects, which separate the application development from classes implementation.

Thus, the meta-modelling layer of the EMF tool is used to graphically model the meta-models of the class and state chart 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) [22] 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.

3.3.1 Meta models of UML diagrams

The class meta-model is composed of structural part to declare attributes and methods, and the state chart is composed of a structural part to define states and transitions (in particular for mobility) and a behavioural part to define events, guards and actions. These meta-models enable us to edit attributes and state changes of migrant (mobile) object, in response to interactions with other migrant objects.

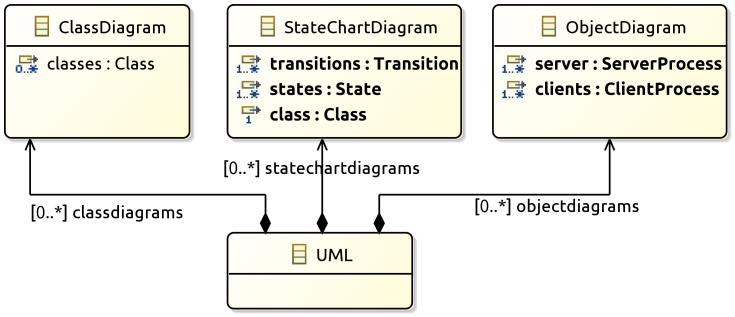

Figure 3.3 – Meta model of UML diagrams

The migrant objects are modelled as class diagrams to specify their declarative parts and state chart diagrams to specify their behaviours in terms of states and their transitions. The object diagram is used to model instances of Mobile Maude processes for simulation and formal verification.

The first meta model in Figure 3.3 is an extension of the meta models of class diagrams, state chart diagrams and object diagrams to integrate the three meta models in one. Thus, it is composed of these meta models to facilate the modelling of the mobile applications by linking the three meta models within the same global edition session.

The figure 3.3 shows that the meta model of UML diagrams is composed of instances of the Class Diagrm, State Chart Diagram and Object Diagram. The Class Diagram can contain a list of classes, the State Chart Diagram can contain a list of states, transitions and it should refer to a previously produced class. The object Diagram can model servers and clients.

The meta model of class diagrams (3.4) is the main component of the meta model of UML diagrams. This meta model contains all the necessary features for modelling Maude Mobile modules as UML class digram. It is composed of imports for importing user defined classes or predefined classe as INT, NAT, FLOAT, DISTRIBUTED-

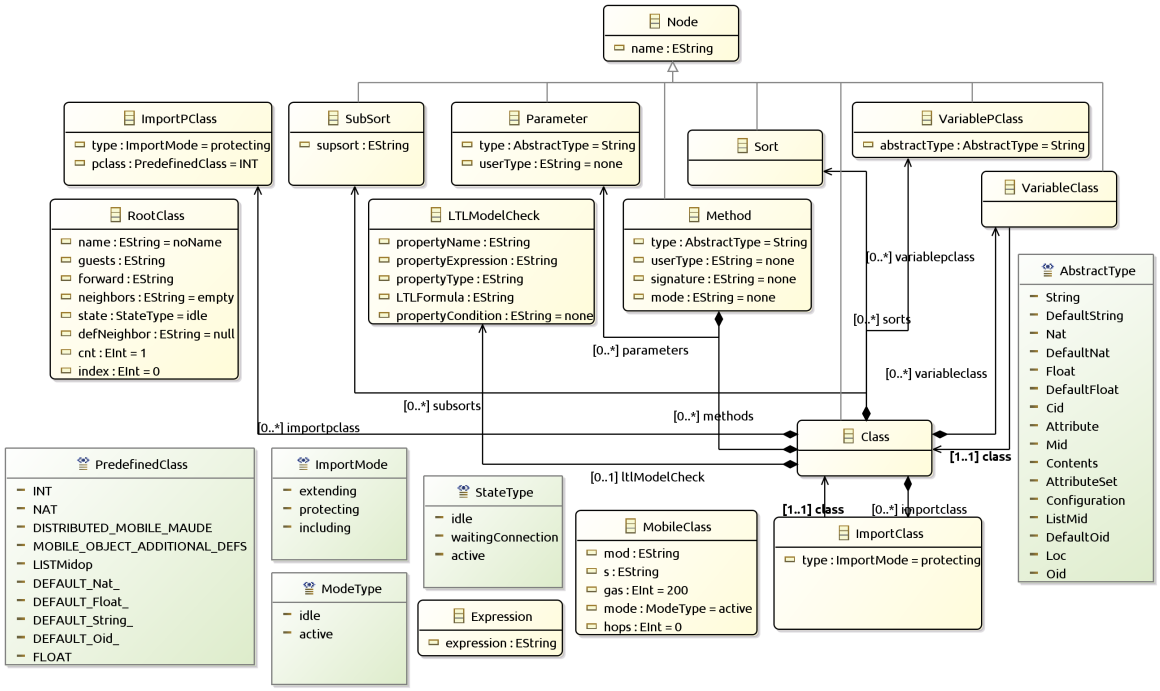

Figure 3.4 – Meta model of class diagram

MOBILE-MAUDE, etc. The importation can be an extension, protection or inclusion.

The second component is the declaration of sorts (types) which can be also an extension of an abstract type, as String, Nat, Int, Cid, etc. The third component is the declaration of methods. A method has a name and a type with a characteristic mode as constructor. It can have parameters, each parameter has a name and a type.

A class of Maude module can declare variables with their types (abstract type or a user defined class) and it can contain an LTL Model Check class for specifying information about model checking as the property name, type, expression, conditions for marking states, and the LTL formula to check the model against it.

These class diagrams define the attributes of the ROOT and MOBILE classes to be used for modeling Maude processes with the meta model of the object diagram.

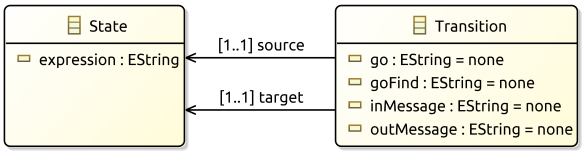

Figure 3.5 – Meta model of state chart diagram

The meta model of state chart diagrams (3.5) is the second component of the meta model of UML diagrams. This meta model contains all the necessary features for modelling behaviour of modules modelled as classes, which means implementation of methods. A transition specifies the source state, the target state, the incoming messages, the outgoing messages, the messages (go and go-find) to migrate to other locations (mobile devices).

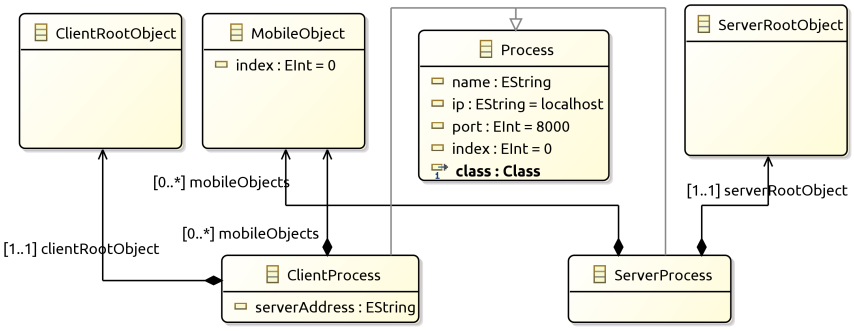

Figure 3.6 – Meta model of object diagram

The object diagrams (3.6) will be used to describe client and server processes as located configurations. A client and server process are extensions of the process class, which refers to mobile Maude class.

The server and client processes are composed of only one server root object and client root object, respectively. These objects are extensions of the root object class. They can also be composed of many mobile objects, which are extensions of the mobile class defined in the class diagram, representing classes of the mobile module as mobile objects.

3.3.2 Meta models of the mobile maude

The meta model of mobile Maude called MAMO in Figure 3.7 is composed of the meta models MAMO modules for modelling mobile Maude modules, MAMO rules for modelling the mobile Maude rules as behaviour of the mobile objects and the MAMO processes for creating located configurations.

The meta model of MAMO modules (3.8) is used to model mobile Maude modules (mobile objects), which are equivalent to UML classes. Thus, it contains all the necessary features for modelling Mobile Maude modules as UML class disgram.

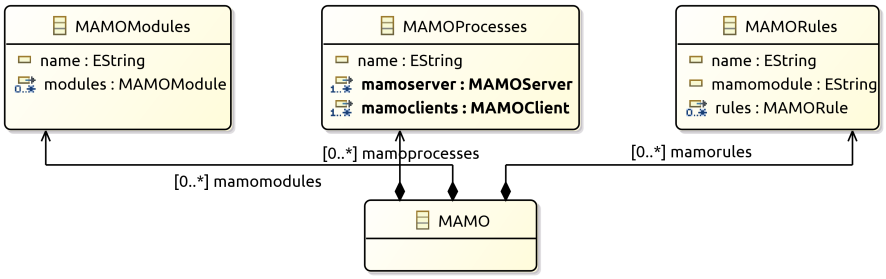

Figure 3.7 – Meta model of MAMO Components

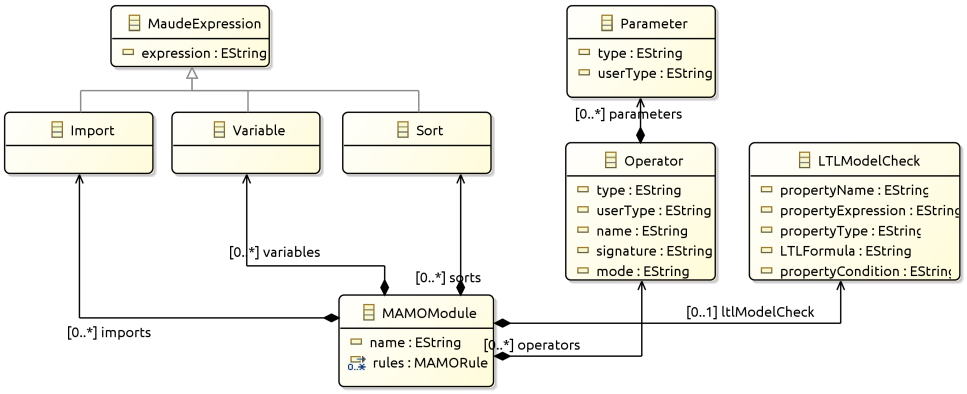

Figure 3.8 – Meta model of MAMO Modules

The meta model of MAMO rules (3.9) is equivalent to transitions in the state chart diagram to model the behaviour rules for each mobile object. Each rule has a label, incoming message, left and right hand side expressions, outgoing message and go/go-find messages for object mobility.

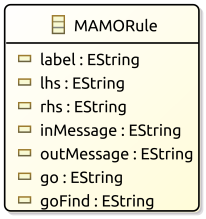

Figure 3.9 – Meta model of MAMO Rules

The meta model of MAMO processes (3.10) is used for modelling mobile Maude processes to create located configurations for simulation and formal verification, which is based on Maude LTL Model checker.

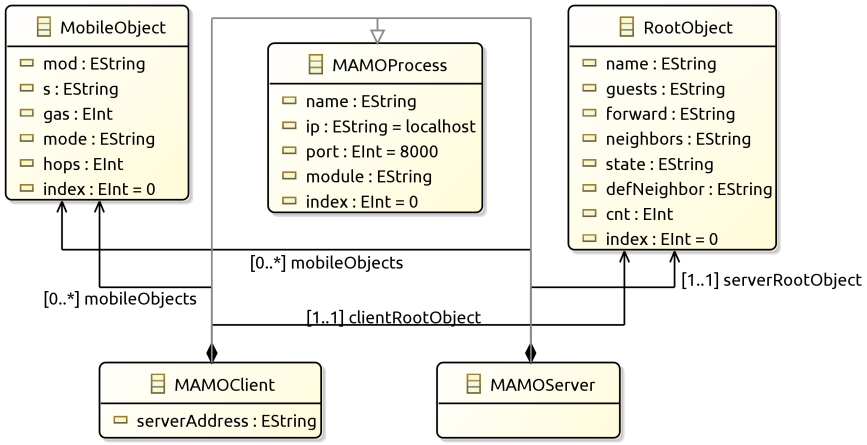

Figure 3.10 – Meta model of MAMO Processes

3.3.3 Correspondence meta model

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 3.11. 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 [27].

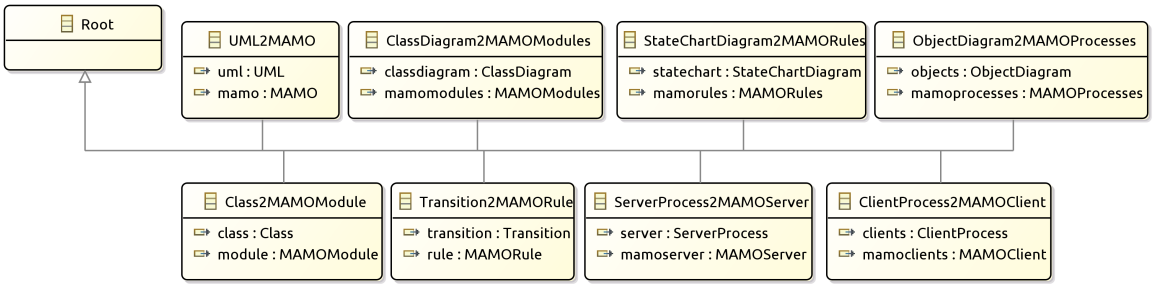

Figure 3.11 – The correspondence meta-model

The automatic transformation works properly if we connect each element from the source meta-model with element from the target meta-model. These relationships between source and target meta-models are represented by a third meta-model called the correspondence meta-model, which is composed of a class to create declarations part of mobile Maude modules and a class to create a rewriting rule for each transition in the state chart diagram.

The TGG Interpreter [35] 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. These transformations are realized between graphs modelled with Eclipse Modelling Framework (EMF).

Thus, to generate the mobile Maude models with TGG interpreter, we begin by writing transformation (TGG) rules using a graphical editor (GMF), which is sensitive to the defined meta-models. Then, writing statements with the Object Constraint Language (OCL) to assign values for specific constraints and at the end, the TGG rules will be boosted by relations of generalization.

3.3.4 Rules of Triple Graph Grammar

A TGG transformation begins by an axiom 3.12 by which we create a consistency correspondence between the UML diagrams and the Maude meta model components.

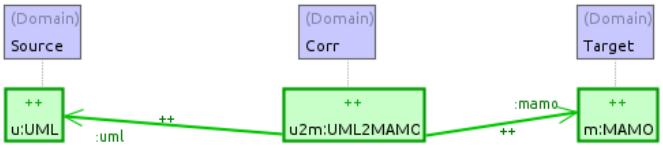

Figure 3.12 – Consistency rule (axiom) of UML and MAMO

In the context of UML diagrams and the MAMO compoenents, we can generate or check the consensitency of class diagrams vis-a-vis the MAMO modules 3.13, the state chart diagrams vis-a-vis the MAMO rules 3.14 and/or the object diagrams vis-a-vis the MAMO processes 3.15.

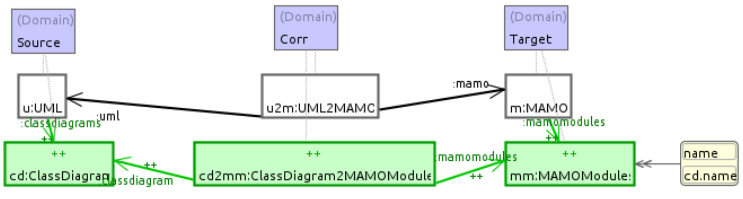

Figure 3.13 – Consistency rule of class diagram and MAMO modules

In the context of class diagram, we can generate (consistency check) the correspondence between a UML class with Maude module by the TGG rule in Figure 3.16. Their

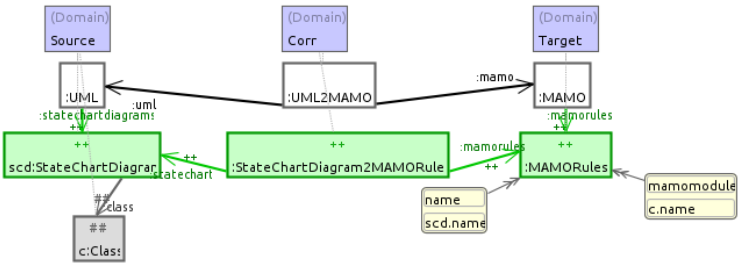

Figure 3.14 – Consistency rule of state chart diagram and MAMO Rules

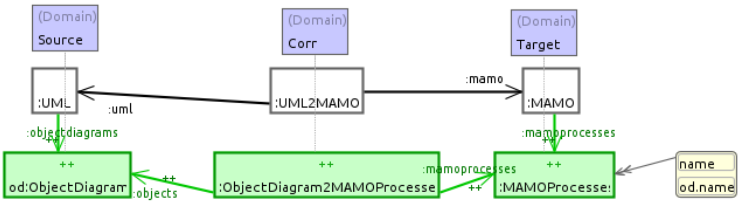

Figure 3.15 – Consistency rule of object diagram and MAMO processes

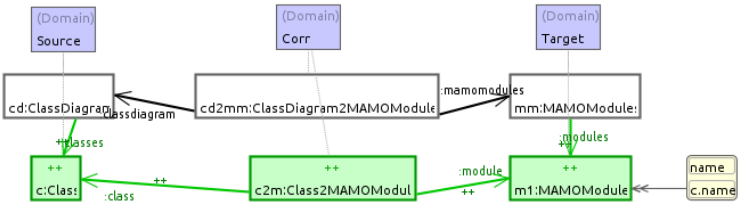

Figure 3.16 – Consistency rule of UML class and MAMO module

features are important for describing the Java classes and the mobile Maude modules.

These features are importing classes/modules, where a class/module can be user defined or predefined mobile Maude class. The second feature is the sort/type to define new types or to extend predefined type by new one. The third feature is the declaraion of methods/operators with their type and parameters. The last feature is the variables, each variable has a name and a type (user-defined type or predefined abstract type).

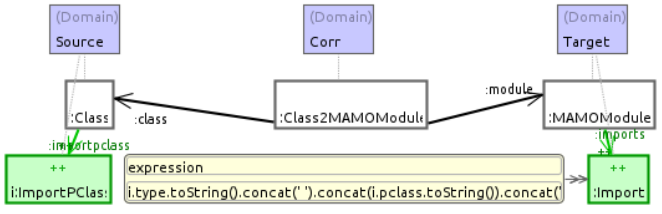

Figure 3.17 – Rule for importing predefined abstract classes

Figure 3.17 shows the TGG rule for importing predefined mobile Maude class-

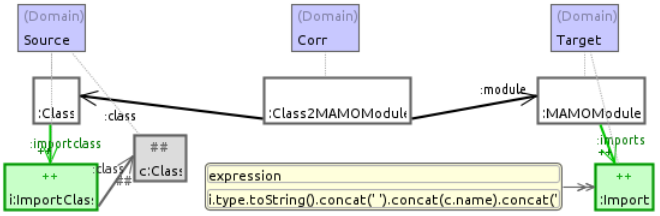

Figure 3.18 – Rule for importing user-defined classes

es/modules. Any importation has a type or mode, which can be either extending, protecting or including. In the rule, we specify the name of the imported class. Figure 3.18 shows the same TGG rule for importing user-defined classes.

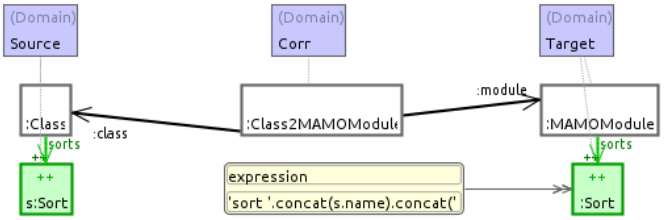

Figure 3.19 – Rule for sorts/types

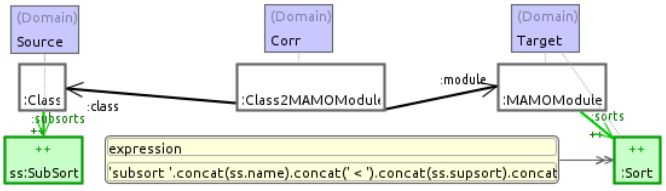

Figure 3.20 – Rule for subsorts/subtypes

It is possible in a Maude module to define new type or to extend an existing type to use it for typing variables, methods or parameters. Figure 3.19 shows the TGG rule for checking the consistency between types and sorts. In addition, Figure 3.20 shows the TGG rule for checking the consistency between subtypes and subsorts.

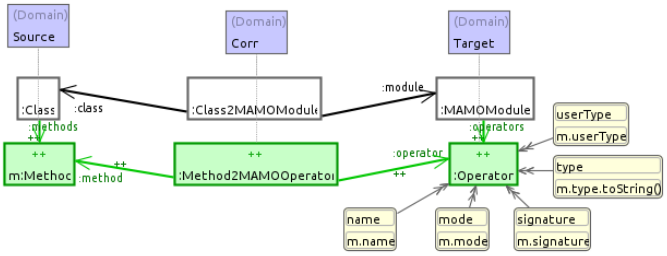

Figure 3.21 – Rule for methods/operators

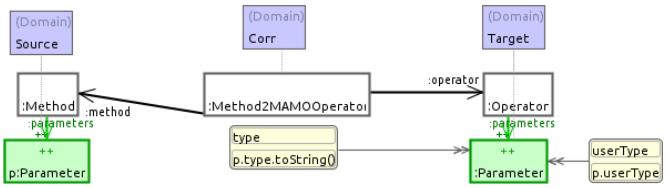

Figure 3.22 – Rule for method parameters

The methods in the module class are used to specify the behaviour by which an instance (object) can manipulate data members and communicate with other instances. A method has a resulting type and it may have parameters with different types. A method is equivalent to mobile Maude operator. Figure 3.21 illustrates the TGG rule to define the consistency between them. The TGG rule 3.22 defines the correspondence between the parameters of the methods and the Maude operators.

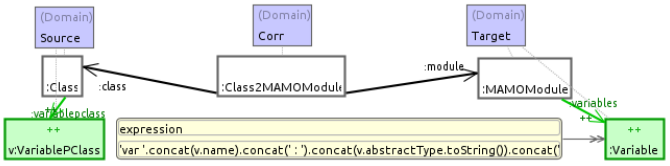

Figure 3.23 – Rule for variables with predefined abstract types

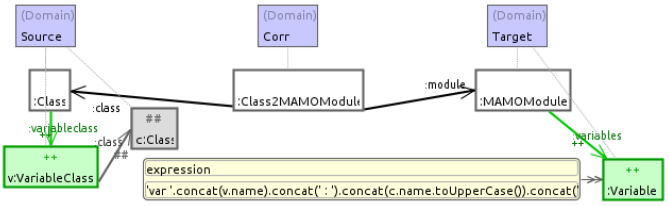

Figure 3.24 – Rule for variables with user-defined types

A variable name will take values of its type. They are considered as data members of a class and Maude variables. The variables data set represents the state of the object/module. Figure 3.23 details the correpondence between the two concepts of the variables of classes and Maude modules. The TGG rule in Figure 3.23 is defined in the case, where the variable type is a defined abstract type, howevere the rule in 3.24 is defined for user-defined types.

In the context of state chart diagram, which refers to a class in the context of class diagram, we can specify transitions that describe its behaviour. A transition is from a source state to a target state. Each state has a Maude expression representing the values of the class attributes defined during the modelling.

A transiotion may have an incoming message as a transition guard (condition for waiting a message). The incoming message from other mobile Maude objects should have information for updating the target state attributes. By the same way, it can have outgoing message to be sent to other mobile Maude objects.

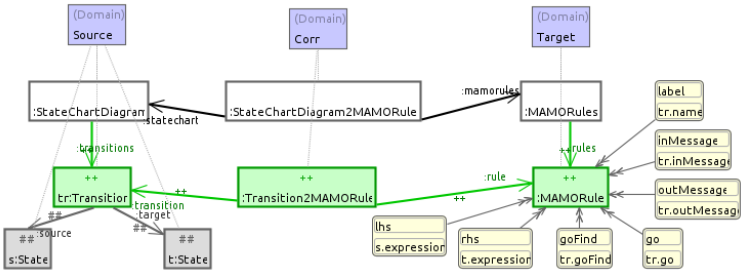

Figure 3.25 – TGG Rule for transition-to-rule of Mobile Maude

The transition can specify message to migrate the current mobile Maude object to other locations. The TGG rule in Figure 3.25 specifies the correspondence beteween state chart transitions and Mobile Maude rules.

Our objective is to simulate these specifications using the Maude system. Thus, in the context of object diagram, we can specify server and client processes. For one simulation, we specify only one server process and multiple client processes. Each process will be hosted by a machine (mobile device).

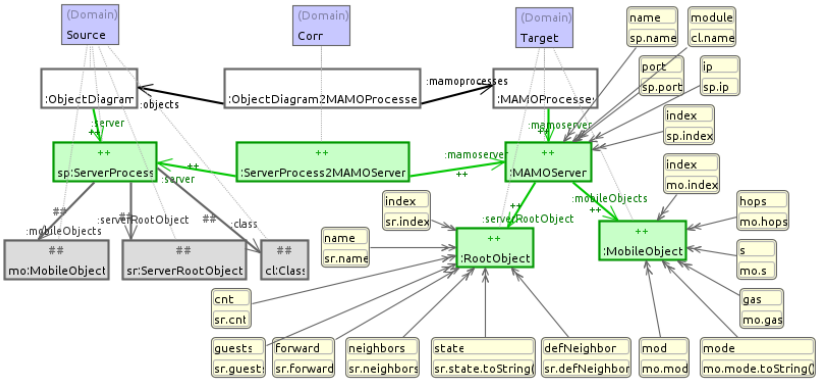

Figure 3.26 – TGG rule for creating servers

A MAMO apllication is then composed of a server, clients, where each one is deployed on a device and mobile objects that can migrate from device (located configuration) to other. The communication between the mobile objects are managed by the server root object or the client root object, which are extensions of the root object class.

Thus, if a mobile object should send a message to other mobile object, it has first to go to its location by sendind a move message to its root object (server or client). Then, the root object will send it to the specified location. The root object of the concerned located configuration will send its message to the target mobile object.

Figure 3.26 shows the TGG rule to transform a server specification in the context of an object diagram to mobile Maude server. The server process from the object diagram has reference to produced class, produced server root object and mobile object. Their features contain information for producing mobile Maude server, in addition to information given by the features specified in the server process.

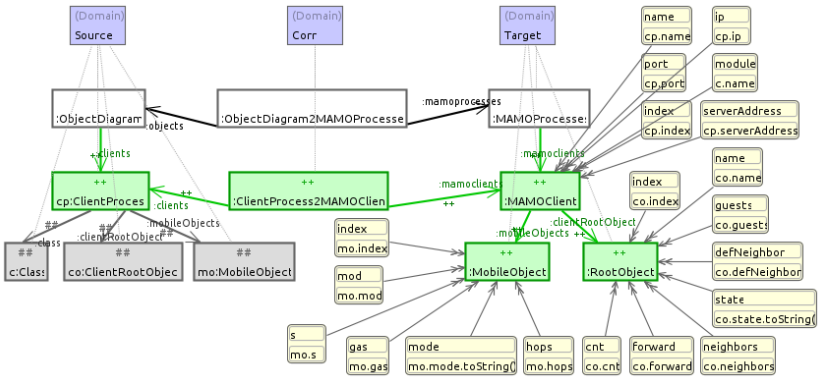

Figure 3.27 – Generation of migrant object Maude modules

By the same way, Figure 3.27 shows the TGG rule to transform a client specification in the context of an object diagram to mobile Maude client. The client process from the object diagram has reference to produced class, produced client root object and mobile object. Their features contain information for producing mobile Maude client, in addition to information given by the features specified in the client process.

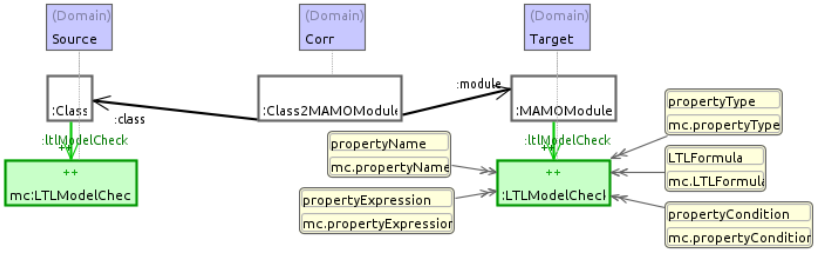

Figure 3.28 – TGG rule for Maude LTL Model Checker

During the execution of the mobile Maude module, we can mark its states by values of propositional properties if they are true or false. These propositional properties will be used for the specification of LTL formulas to be model checked with LTL Maude

Model Checker. Figure 3.28 shows the transformation of the required features information defined for the class LTL Model checker in the context of a class to its equivalent in the context of MAMO mdule. The mobile object contains then an attribute to keep the trace of these states to represent the model to be checked against the LTL property.

3.3.5 Performing M2M transformation with TGG

The M2M transformation requires edition of a source model with respect to the profile of its meta-model. The result of the transformation will be a traget model, which is also with respect to the profile of its meta-model. These two models (edited and generated) should be recognized and validated. The operations of edition and validation will be performed within projects (sets of Java files) that can be generated from the meta-models using the Eclipse Modelling Framework (EMF).

Before generation of required EMF (Eclipse Modelling Framework) projects, we have to validate all the meta-models (correspondence meta-model, UML meta-models and MAMO meta-models) in addition to the TGG model with its rules. If the validation process displays an error, it should be corrected first.

Then, we click right on the meta-model of each project (UMLDiagrams and MAMO) and select EMF Model Generator from the EMF wizard to generate an EMF model generator file. By right clicking on this generator file and select generate all, we will create four EMF projects for each meta model with their codes. These projects are located in a meta-level Eclipse application.

To edit models with respect to the meta models of UML, Class, State Chart, Object Diagrams or the MAMO meta-models, MAMO Modules, MAMO Rules or MAMO processes or to perform TGG transformation, we should go down in the level by running an Eclipse Application and importing the required projects for model edition with EMF Ecore or model transformation with TGG engine.

Before running TGG transformation, the edited source model should be validated. The selection of this source model with the TGG model and selecting create configuration from the TGG interpreter and then selecting the correspondence meta-model, a TGG interpretation configuration will be generated and saved in a file. By clicking right on this file and then selecting perform transformation from the TGG interpreter,

the M2M transfrmation engine will be executed to show the number of applied TGG rules and the produced target model.

3.4 Model-to-text transformation with Xpand

Xpand is a template engine for generating code from models. The process of writing our code generator called mamo.generator.project with Xpand is as follows (3.29). We create three folders in the src folder of the Xpand project. The first is the metamodel folder, which contains the meta-models, UML diagrams, class diagrams, state chart diagrams, object diagrams, MAMO Components, MAMO Modules, MAMO Rules and MAMO Processes. It contains also two files, the first is called "Extensions.ext" for defining meta-models extensions and the second is "Checks.chk" for checking if the edited model respects certain conditions.

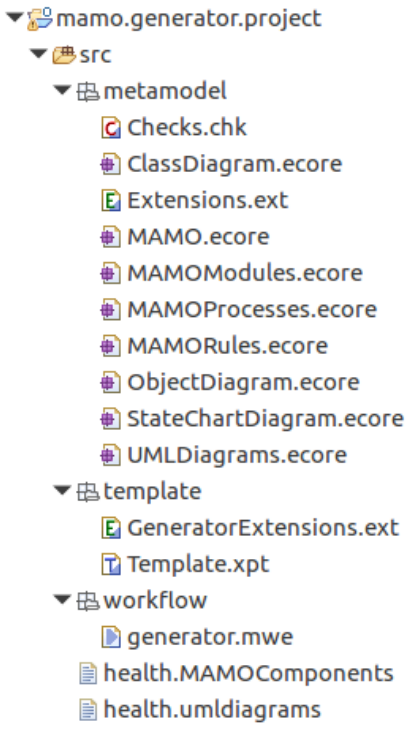

Figure 3.29 – Structure of Xpand Project

The second folder is called template to define the content of the Template.xpt file that guides the code generator how to translate our models into code (text). The second file is called GeneratorExtensions.ext to define some extensions to be used as extensions for the template file. It is worth mentioning that you can use Xpand to generate code

for almost any known programming language. Everything you can express in text can be generated using Xpand.

The third folder contains the workflow file used by the Modeling Workflow Engine. The modeling workflow engine (MWE) supports orchestration of different Eclipse modeling components to be executed within Eclipse as well as standalone. Based on a dependency injection framework, one can simply configure and wire up 'workflows' using a declarative XML-based language. The project provides the runtime used to execute workflows.

The contents of all these files are shown in the appendices. To use the code generator, we create a model and then start the code generator by opening the context menu on src/workflow/generator.mwe and select Run As -> MWE Workflow. This will start the code generator, where some log messages are dislayed in the console view.

The result of the code generation can be found in two external projects. The text files generated in the model-to-text transformation are classified to two sets of files. The first set is the mobile Maude files saved in a Eclipse Maude project in a package called Maude. The second set of files is the set of Android (Java) files saved an Android Java project in the package Android.

These texts (codes) are generated with respect to defined expansions in the Xpand template file. 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 and Android Java from EMF models of mobile Maude and UML.

3.5 Conclusion

This chapter presented the implementation of the tool based on the technology of Eclipse Modelling Framework (EMF), the Triple Graph Grammars (TGG) for M2M transformation and the Xpand/Xtend technology for M2T transformation, to generate the mobile Maude project for simulation and formal verification and Android Java project for mobile applications based on migrant objects. The next chapter will present the implementation results on a case study.

Chapter 4

Case Study: Implementation results

4.1 Introduction

This case study is an extension of the example of health care, which is explained in chapter 2. First, we describe with the editor Ecore the example models based on the defined meta models of UML diagrams in Chapter 3. We model classes for the doctor, mcdotor (doctor with model checking information) and patient mobile objects with respect to the UML class diagram and state chart diagram. We model also object diagram models for servers and clients to do simulation and formal verificatio. Then, we apply the TGG transformation to generate their equivalent models with respect to the meta-model of Mobile Maude (MAMO components).

The UML models and the Mobile Maude models will be used for production of Android Java code and Mobile Maude code using a developped Xpand project for code generation. The Mobile maude code is used for simulation and verification of LTL properties. The Android Java code is the template for mobile application to be completed for deployement on mobile devices.

4.2 Model-to-model transformation

The first step is to model with the EMF Ecore editor the model of the health care example with respect to the UML meta model. Figure 4.1 shows the result of the edition process. This model should be validated to detect and correct eventual edition errors.

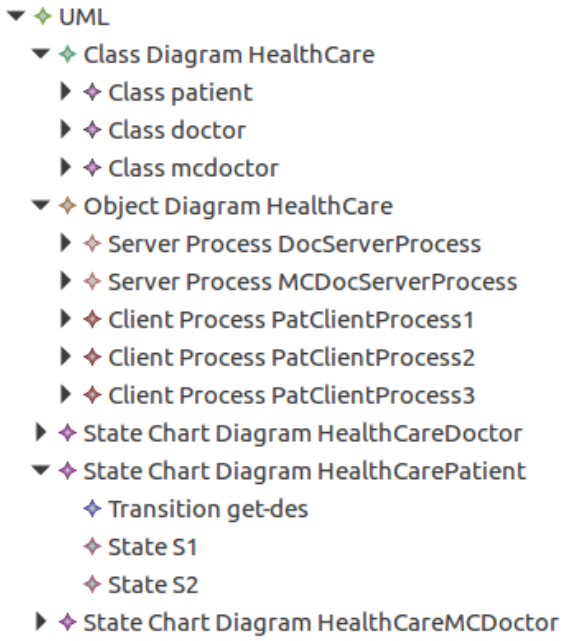

Figure 4.1 – UML model of the example

In the second step, we perform TGG transformation to produce the MAMO models. This step is composed of two operations. In the first operation, we create TGG interpreter configuration for forward transformation by selecting at the same time the file name of UML example model and the TGG model name, then right clicking on the file name of example model, to open the context menu on TGG interpreter to select the operation of creating an interpreter configuration. At this point, we have to select the corresponding model to create new file containing the configuration.

In the second operation, we right click on the file name of generated configuratin, to open the context menu on TGG interpreter to select performin the operation of transformation. The TGG engine will show information on the transformation as the number of applied rules. Figure 4.2 shows the produced MAMO model.

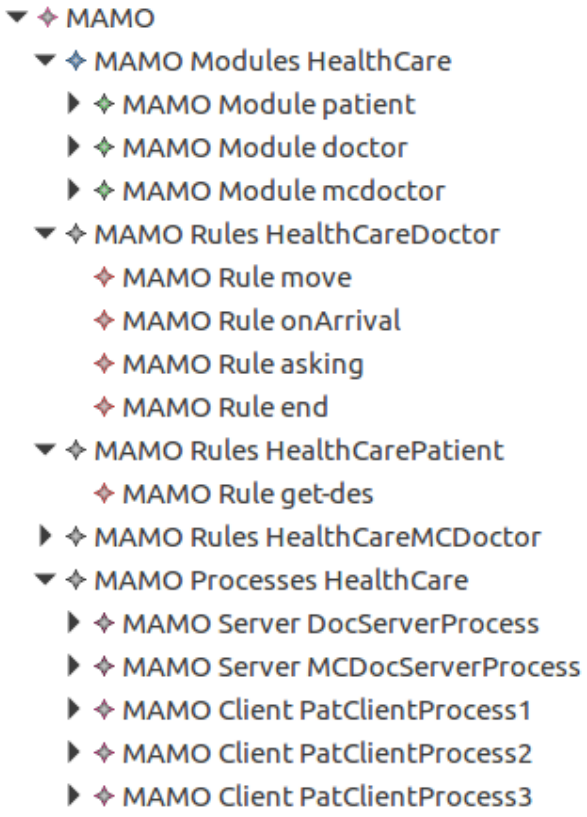

Figure 4.2 – Generated MAMO model

4.3 Model-to-text transformation

With the Xpand project, we transform these two example models to text (code). The edited UML model will be used to generate Android Java files for the Android Java project and the produced MAMO model will be used to generate Mobile Maude code for Maude project.

The transformation happens once for the two models by right clicking on the workflow file generator.mwe to open the context menu on Run As and selecting the MWE Workflow engine from the menu. We will get two sets of files, the first set will be saved in Android Java project and the second set of files will be saved in a general project for simulation and formal verification with Maude and Maude LTL Model checker.

Figure 4.3 shows the structure of the Android Java project. The contents of te Java files are shown in the appendices. The appendices show also the contents of the Maude files saved in the Maude project, which has the structure in Figure 4.4.

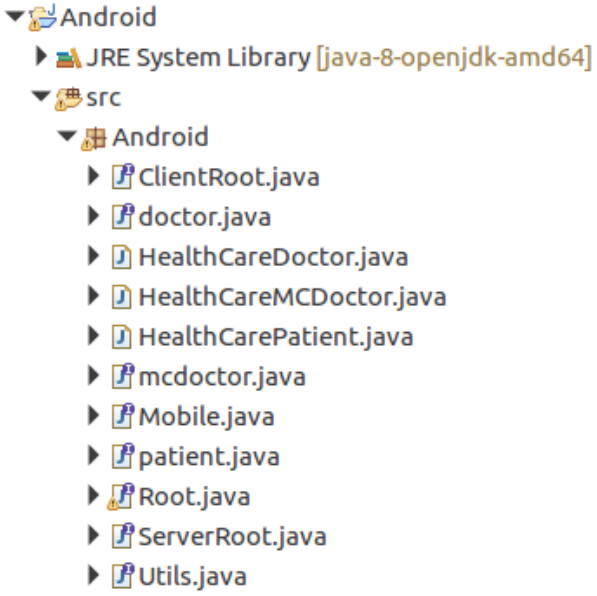

Figure 4.3 – Android Java project

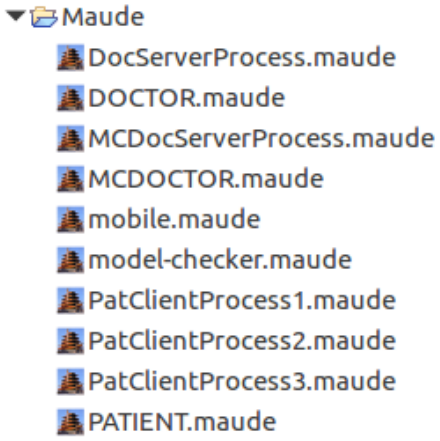

Figure 4.4 – Android Java project

4.4 Simulation

Figure 4.5 shows simulation results. Four Eclipse processes are excuted one for a server and the others are for three clients. The doctor mobile object at initial configuration is located in the server process, then it will move to the processes where the mobile objects of the patients (Pat1, Pat2 and Pat3) are located respectively to get health information of each patient (temperature, blood presure and glucose levels) at the end it will return to its home location, which means it performs four hops (jumps). When it returns to its home location it has all this health information as it is shown in its context state 's'.

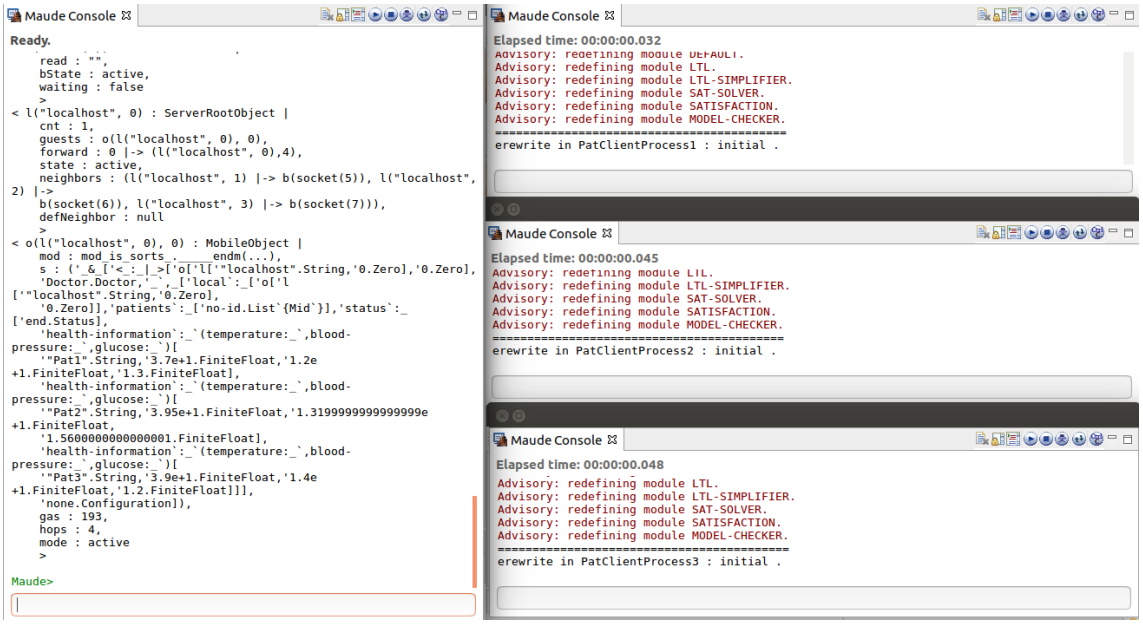

Figure 4.5 – Simulation results

From this simulation results, we can guarantee that the doctor mobile object has moved as it is expected to do for getting health information of the different patients.

4.5 Formal verification

In the case that is difficult to do simulation or it is hard to analyse its results, we use formal methods in particular model checking. We define a propostition "pr" in the code of doctor mobile object and it will be true if $T < 37.5 \wedge BP < 15.0 \wedge G < 1.4$ else it is false and all the mobile object states will be marked by this proposition value (true or false). At the end, we perform a model checking of an LTL formula which is $\Box\Diamond pr$ that means eventually we reach a state where pr is true.

Figure 4.6 shows that the LTL formula is satistied. This means that the doctor has reached a state where a patient' helath information is satisfied. It is possible to check if a patient has bad health information by changing the proposition condition.

Figure 4.7 shows a counter example for the model checking of the formula $\Box pr$ that means all the states are marked by $pr = true$. After realising simulation and formal verification, we will be in two situation, the design of the mobile application contains errors, which should be corrected by redoing all the previous tasks of modelling,

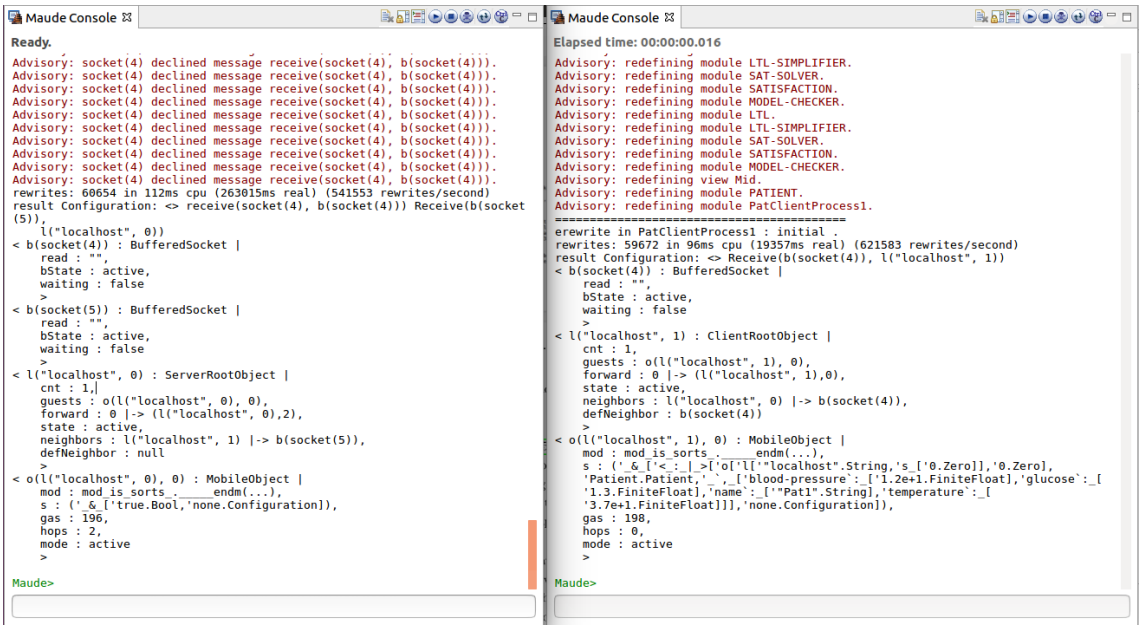

Figure 4.6 – Formal verification of $\Box\Diamond pr$

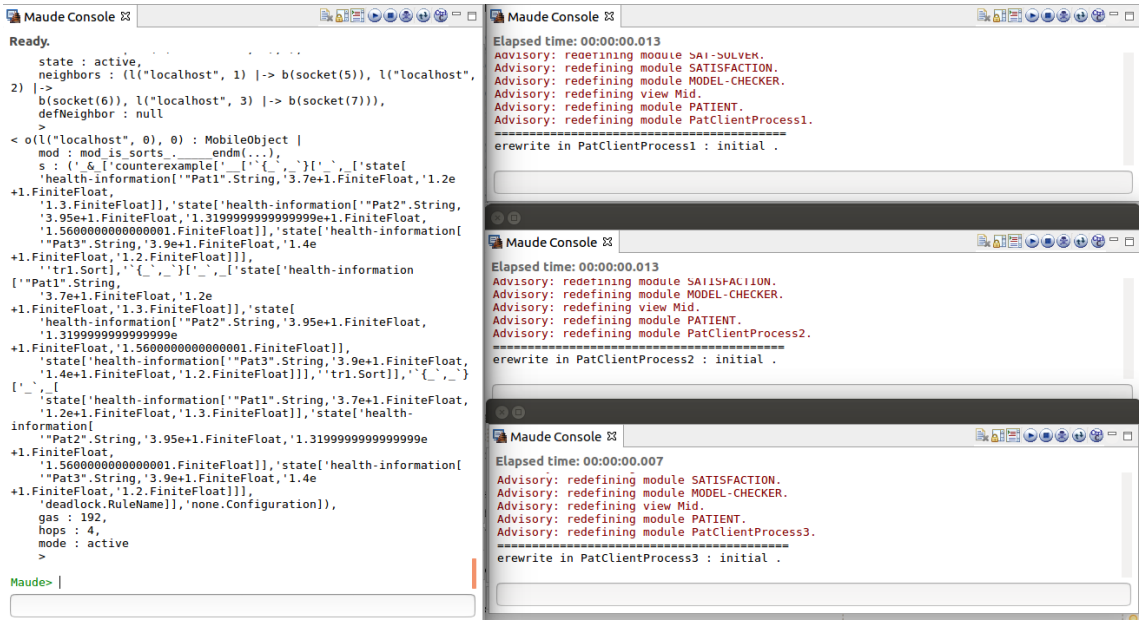

Figure 4.7 – Formal verification of $\Box pr$

transformation and code generation. If the design is satisfied, which is the second situation, we go to the developpment of the mobile application by completing the Android Java projet.

4.6 Conclusion

This chapter presented the implementation results on case study about domain of health care. The results shwo interesting and encouraging work using this implemented tool for developing mobile applications based on migrant objects.

General conclusion

THIS dissertation presented a technique to use software engineering principles in the devlopment of mobile applications, which can use migrant objects from one mobile device to another. The technique uses class diagram to model the declarative part of mobile objects, which migrate between mobile devices.

The behaviour of these mobile objects can be modelled by state chart diagrams, where behaviour rules will be specified to go from a state to another state, to send messages to the other mobile objects resident in the same mobile device or to go to distant mobile devices. These mobile objects are specified by the mobile Maude developed based on the rewriting logic. The Maude engine can help to do simulation.

The simulation is the execution of processes containing the mobile Maude code of the different mobile objects. These processes are generated from an object diagram. During the simulation, it is possible to generate an automaton model. This model will be used with specific process that is generated from the object diagram to realise a LTL model-checking to verify an LTL formula.

An implementation of this technique based on EMF (Eclipse Modelling Framework) for editing required meta-models, Triple Graph Grammars (TGG) for model-to-model transformation and Xpand/Xtend for model-to-text transformation is realised to automatically generate two Eclipse projects. The first projet contains Maude specifications to do simulation and LTL model chacking. The second projetc is an Android Jav project to contain the generated Java code. A case sudy for the health domain is proposed to show the implementation results.

Futur works

Our major perspective is to continue on the developpment of this technique by looking for its improvments. The other perspective is to integrate this approach with the existing tools like the Android Studio.

Appendices

Extensions file of the meta-models

```
import UMLDiagrams;
import ClassDiagramModel;
import StateChartDiagramModel;
import MAMOComponents;
import MAMOModulesModel;
import MAMOProcessesModel;
import MAMORulesModel;

UML uml(ClassDiagram this) : eContainer;
UML uml(StateChartDiagram this) : eContainer;
MAMO mamo(MAMOModules this) : eContainer;
MAMO mamo(MAMORules this) : eContainer;
MAMO mamo(MAMOProcesses this) : eContainer;

List[ClassDiagram] classdiagrams(UML this) :
  classdiagrams.typeSelect(ClassDiagram);
List[StateChartDiagram] statechartdiagrams(UML this) :
  statechartdiagrams.typeSelect(StateChartDiagram);
List[MAMOModules] modules(MAMO this) :
  mamomodules.typeSelect(MAMOModules);
List[MAMORules] rules(MAMO this) :
  mamorules.typeSelect(MAMORules);
List[MAMOProcesses] processes(MAMO this) :
  mamoprocesses.typeSelect(MAMOProcesses);

ClassDiagram classes(Class this) : eContainer;
StateChartDiagram transitions(Transition this) : eContainer;
MAMOModules modules(MAMOModule this) : eContainer;
MAMORules rules(MAMORule this) : eContainer;
MAMOProcesses servers(MAMOServer this) : eContainer;
MAMOProcesses clients(MAMOClient this) : eContainer;
```

```
List[Class] class(ClassDiagram this) :
  classes.typeSelect(Class);
List[Method] methods(Class this) :
  methods.typeSelect(Method);
List[Transition] transition(StateChartDiagram this) :
  transitions.typeSelect(Transition);
List[MAMOModule] module(MAMOModules this) :
  modules.typeSelect(MAMOModule);
List[MAMORule] rule(MAMORules this) :
  rules.typeSelect(MAMORule);
List[MAMOServer] server(MAMOProcesses this) :
  mamoserver;
List[MAMOClient] client(MAMOProcesses this) :
  mamoclients.typeSelect(MAMOClient);
```

Checks file of the meta models

```
import UMLDiagrams;
extension metamodel::Extensions;
context UML ERROR "No class diagram defined" :
  classdiagrams.exists(e|ClassDiagram.isInstance(e));
```

Extensions file of the template

```
import MAMOModulesModel;
extension metamodel::Extensions reexport;
getter(MAMOModule this) : "get"+name.toFirstUpper();
setter(MAMOModule this) : "set"+name.toFirstUpper();
```

The template file

```
«IMPORT UMLDiagrams»
«IMPORT ClassDiagramModel»
«IMPORT StateChartDiagramModel»
«IMPORT MAMOComponents»
«IMPORT MAMOModulesModel»
```

```
«IMPORT MAMOProcessesModel»
«IMPORT MAMORulesModel»

«EXTENSION template::GeneratorExtensions»

«DEFINE main FOR UML»
«EXPAND utils»
«EXPAND root»
«EXPAND serverRoot»
«EXPAND clientRoot»
«EXPAND mobile»
«EXPAND classdiagrams FOREACH classdiagrams()»
«EXPAND statechartdiagrams FOREACH statechartdiagrams()»
«ENDDEFINE»

«DEFINE main FOR MAMO»
«EXPAND mamomodules FOREACH modules()»
«EXPAND mamorules FOREACH rules()»
«EXPAND mamoprocesses FOREACH processes()»
«ENDDEFINE»

«DEFINE utils FOR UML»
«FILE "Utils.java"»
package Android;
public interface Utils {
    public class MaudeType<Type> {
       private Type t;
       public void set(Type t) { this.t = t; }
       public Type get() { return t; }
    };
    public class MaudeString<Type> extends MaudeType<Type> {};
    public class MaudeState<Type> extends MaudeType<Type> {};
    public class MaudeNat<Type> extends MaudeType<Type> {};
    public class MaudeInt<Type> extends MaudeType<Type> {};
    public class MaudeFloat<Type> extends MaudeType<Type> {};
    public class MaudeAttribute<Type> extends MaudeType<Type> {};
    public class MaudeAttributeSet<Type> extends MaudeType<Type> {};
    public class MaudeVoid<Type> extends MaudeType<Type> {};
    public class MaudeDefaultNat<Type> extends MaudeType<Type> {};
    public class MaudeMid<Type> extends MaudeType<Type> {};
    public class Object<Type> extends MaudeType<Type> {};
    public class Forward<Type> extends MaudeType<Type> {};
    public class RoutingTable<Type> extends MaudeType<Type> {};
    public class Socket<Type> extends MaudeType<Type> {};
    public class Location<Type> extends MaudeType<Type> {};
    public class MaudeContents<Type> extends MaudeType<Type> {};
    public class MaudeOid<Type> extends MaudeType<Type> {};
    public class MaudeLoc<Type> extends MaudeType<Type> {};
```

```
    public class MaudeListMid<Type> extends MaudeType<Type> {};
    public class MaudeDefaultString<Type> extends MaudeType<Type> {};
    public class MaudeDefaultFloat<Type> extends MaudeType<Type> {};
    public class MaudeDefaultOid<Type> extends MaudeType<Type> {};
    public class Content<Type> extends MaudeType<Type> {};
}
«ENDFILE»
«ENDDEFINE»

«DEFINE root FOR UML»
«FILE "Root.java"»
package Android;
import java.util.List;
public interface Root extends Utils {
    RoutingTable<String> empty = new RoutingTable<String>();
    public enum RootState {idle}
    String IP = "localhost";
    int Port = 8000;
    int  cnt = 1;
    List<Object<String>> guests = null;
    List<Forward<String>> forward = null;
    RoutingTable<String> neighbors = empty;
    RootState state = RootState.idle;
    Socket defNeighbor = null;
    public void go(Location<String> l);
    public void goFind(Object<String> o, Location<String> l);
    public void to(Object<String> o, Content<String> c);
}
«ENDFILE»
«ENDDEFINE»

«DEFINE serverRoot FOR UML»
«FILE "ServerRoot.java"»
package Android;
public interface ServerRoot extends Root {}
«ENDFILE»
«ENDDEFINE»

«DEFINE clientRoot FOR UML»
«FILE "ClientRoot.java"»
package Android;
public interface ClientRoot extends Root {
   String serverAddress = "localhost";
}
«ENDFILE»
«ENDDEFINE»

«DEFINE mobile FOR UML»
```

```
«FILE "Mobile.java"»
package Android;
public interface Mobile extends Utils {
    public class MobileType<Type> {
       private Type t;
       public void set(Type t) { this.t = t; }
       public Type get() { return t; }
    };
    public class MobileState<Type> extends MobileType<Type> {};
    public enum MobileMode {active}
    MobileState<String> s = new MobileState<String>();
    int gas = 200;
    MobileMode mode = MobileMode.active;
    int hops = 0;
    void go(String l);
    void goFind(String l);
    void sendMessage(String c);
}
«ENDFILE»
«ENDDEFINE»

«DEFINE classdiagrams FOR ClassDiagram»
«EXPAND classes FOREACH class()»
«ENDDEFINE»

«DEFINE statechartdiagrams FOR StateChartDiagram»
«FILE name+".java"»
package Android;
import java.util.List;
import com.sun.javafx.collections.MappingChange.Map;
public class «name» implements
   «LET class AS c» «c.name» {Map<String,List<String>> Conf;«EXPAND methods(")
   { return null; };") FOR c»
   «ENDLET»
   public void transitions() {
      do {
        «EXPAND transitions(name) FOREACH transition()»
      } while (!Conf.map(«"\""+name+"\""»).isEmpty());
    }
    @Override
    public void go(String l) {
        // TODO Auto-generated method stub
    }
    @Override
    public void goFind(String l) {
        // TODO Auto-generated method stub
    }
    @Override
```

```
    public void sendMessage(String c) {
        // TODO Auto-generated method stub
    }
}
«ENDFILE»
«ENDDEFINE»

«DEFINE methods(String text) FOR Class»
«FOREACH methods AS m»«EXPAND header(m.name, m.userType, m.type, text) FOR m»
«ENDFOREACH»
«ENDDEFINE»

«DEFINE methods(String text) FOR Method»
    «EXPAND header(name, userType, type, text)»
«ENDDEFINE»

«DEFINE header(String name, String userType,
  ClassDiagramModel::AbstractType type, String text) FOR Method»
      public Maude«IF userType == "none"»«type.toString().
      replaceAll("[-{}(),_\\*\\ \\.]","")»«ELSE»«userType»«ENDIF»<String>
      «name.toString().replaceAll("[-{}(),_\\*\\ \\.]","")» (
      «EXPAND parameters("<String>")»«text»
«ENDDEFINE»

«DEFINE parameters(String subtype) FOR Method»
«FOREACH parameters AS p»
            Maude«IF p.userType == "none"»«p.type.toString().
            replaceAll("[-{}(),_\\*\\ \\.]","")»«ELSE»«p.userType»«ENDIF»
            «subtype» «p.name», «ENDFOREACH»
            MaudeVoid«subtype» p
«ENDDEFINE»

«DEFINE mamomodules FOR MAMOModules»
«EXPAND modules FOREACH module()»
«ENDDEFINE»

«DEFINE mamorules FOR MAMORules»
«FILE mamomodule.toUpperCase()+".maude" APPEND_OUT»
    «FOREACH rules AS r»rl [«r.label»] :
    Conf «IF r.inMessage != "none"»«r.inMessage»«ENDIF» «r.lhs» & none =>
    Conf «r.rhs» «IF r.outMessage != "none"» & «r.outMessage»«ENDIF»
    «IF r.go != "none"» & «r.go»«ENDIF»«IF r.goFind != "none"» & «r.goFind»
    «ENDIF»
    «IF r.outMessage == "none" && r.go == "none" && r.goFind == "none"» & none
    «ENDIF» .
    «ENDFOREACH»
endm
«ENDFILE»
```

```
«ENDDEFINE»

«DEFINE mamoprocesses FOR MAMOProcesses»
«EXPAND mamoserver FOREACH server()»
«EXPAND mamoclients FOREACH client()»
«ENDDEFINE»

«DEFINE classes FOR Class»
«FILE name+".java"»
package Android;
public interface «name» extends Mobile {
    «FOREACH importpclass AS ip»public class Maude«ip.pclass.toString().
    replaceAll("[-{}(),_\\*\\ \\.]","")» {};«ENDFOREACH»
    «FOREACH importclass AS ic»public class Maude«ic.class.name» {};
    «ENDFOREACH»
    «FOREACH sorts AS s»public class Maude«s.name»<Type> extends
    MaudeType<Type> {};
    «ENDFOREACH»
    «FOREACH subsorts AS ss»public class Maude«ss.supsort» {};
    public class Maude«ss.name» extends Maude«ss.supsort» {};«ENDFOREACH»
    «FOREACH variablepclass AS var»Maude«var.abstractType.toString().
    replaceAll("[-{}(),_\\*\\ \\.]","")»<String> «var.name» = new Maude
    «var.abstractType.toString().replaceAll("[-{}(),_\\*\\ \\.]","")»
    <String>();
    «ENDFOREACH»
    «FOREACH variableclass AS var»«var.class.name» «var.name» = null;
    «ENDFOREACH»
    «FOREACH methods AS m»«EXPAND methods(");") FOR m»«ENDFOREACH»
}
«ENDFILE»
«ENDDEFINE»

«DEFINE methodparameters FOR Method»
«FOREACH parameters AS p»
   Maude«IF p.userType == "none"»«p.type.toString().
     replaceAll("[-{}(),_\\*\\ \\.]","")»«ELSE»«p.userType»<String>,
   «ENDIF»
«ENDFOREACH») ;
«ENDDEFINE»

«DEFINE transitions(String class) FOR Transition»
«LET source AS s»if (Conf.map(«"\""+class+"\""»).
   contains(«"\""+s.expression+"\""»)
   «IF inMessage != "none"» && Conf.map(«"\""+class+"\""»).
   contains(«"\""+inMessage+"\""»)«ENDIF») {
     Conf.map(«"\""+class+"\""»).remove(«"\""+s.expression+"\""»);
«ENDLET»
   «LET target AS t»
```

```
     Conf.map(«"\""+class+"\""»).add(«"\""+t.expression+"\""»);
     «IF outMessage != "none"» sendMessage(«"\""+outMessage+"\""»);«ENDIF»
     «IF go != "none"» go(«"\""+go+"\""»);«ENDIF»
     «IF goFind != "none"» goFind(«"\""+goFind+"\""»);«ENDIF»
   «ENDLET»
   }
«ENDDEFINE»

«DEFINE modules FOR MAMOModule»
«FILE name.toUpperCase()+".maude"»
load model-checker.maude .
view Mid from TRIV to MID is
  sort Elt to Mid .
endv
mod «name.toUpperCase()» is
    «FOREACH imports AS i»«i.expression»«ENDFOREACH»
    sort «name.toLowerCase().toFirstUpper()» .
    subsort «name.toLowerCase().toFirstUpper()» < Cid .
    op «name.toLowerCase().toFirstUpper()» : ->
    «name.toLowerCase().toFirstUpper()» .
    var V@«name.toLowerCase().toFirstUpper()» :
    «name.toLowerCase().toFirstUpper()» .
    «FOREACH sorts AS s»«s.expression»«ENDFOREACH»
    «FOREACH operators AS o»
      op «IF o.name != "none"»«o.name+" :"»«ENDIF»
      «IF o.signature != "none"»«o.signature+" : "»«ENDIF»
      «EXPAND parameters FOR o»       ->
         «IF o.userType == "none"»«o.type+" "»«ELSE»«o.userType+" "»«ENDIF»
         «IF o.mode != "none"»«" ["+o.mode+"]"»«ENDIF» .
    «ENDFOREACH»
    «FOREACH variables AS v»«v.expression»«ENDFOREACH»
    var  Conf : Configuration .
    «IF ltlModelCheck != null»
       «LET ltlModelCheck AS mc»
          including MODEL-CHECKER .
          sort StateList .
          subsort StateList < State .
          op nil : -> StateList [ctor] .
          op _,_ : StateList StateList -> StateList [ctor assoc id: nil] .
          op St :_ : StateList -> Attribute [ctor] .
          op ModelCheck :_ : Bool -> Attribute [ctor] .
          subsort Status < StateList .
          subsort «mc.propertyType» < StateList .
          subsort ModelCheckResult < Msg .
          op «mc.propertyName» : -> Prop [ctor] .
          op state : StateList -> StateList [ctor] .
          vars st st' : StateList .
          var s : State .
```

```
          «IF mc.propertyExpression == "none"»ceq st , state(
              «mc.propertyExpression» , st' |= pr = true .
          «ELSE»
              ceq st , state(«mc.propertyExpression») , st' |=
              pr = true if T < 37.5 /\ BP < 15.0 /\ G < 1.4 .
          «ENDIF»
          eq state(st) |= pr = false [owise] .
          crl [tr1] : state(s, st) => state(s) , state(st)
              if s =/= nil /\ st =/= nil .
          rl [tr2] : state(nil) => state(end) .
          rl [tr3] : state(end) => state(end) .
          rl [end] : < D : V@«name.toLowerCase().toFirstUpper()» |
                       St : st, ModelCheck : true, AtS > & none =>
                     modelCheck(state(st), «mc.LTLFormula») & none .
       «ENDLET»
    «ENDIF»
«ENDFILE»
«ENDDEFINE»

«DEFINE parameters FOR Operator»
       «FOREACH parameters AS p»
         «IF p.userType == "none"»«p.type+" "»
         «ELSE»«p.userType+" "»
         «ENDIF»
       «ENDFOREACH»
«ENDDEFINE»

«DEFINE load FOR MAMOModule»
load «name.toUpperCase()+".maude ."»
«ENDDEFINE»

«DEFINE mamoserver FOR MAMOServer»
«FILE name+".maude"»
load model-checker.maude .
load mobile.maude .
load «module.toUpperCase()+".maude ."»
mod «name» is
  ex DISTRIBUTED-MOBILE-MAUDE .
  ex «module.toUpperCase()» .
  op IP : -> String .
  eq IP = "«ip»" .
  eq port = «port» .

  op initial : -> Configuration .
  eq initial = <>
  «LET serverRootObject AS srv»< l(IP, «srv.index») : ServerRootObject |
  cnt : «srv.cnt»,
  guests : «srv.guests»,
```

```
  forward : «srv.forward»,
  neighbors : «srv.neighbors»,
  state : «srv.state»,
  defNeighbor : «srv.defNeighbor» >
  «FOREACH mobileObjects AS mo»
  < o(l(IP, «srv.index»), «mo.index») : MobileObject |
  mod : upModule('«mo.mod», false),
  s : upTerm(< o(l(IP, «srv.index»), «mo.index») :
  «mo.mod.toLowerCase().toFirstUpper()» |
  «mo.s» > & none),
  gas : «mo.gas»,
  mode : «mo.mode»,
  hops : «mo.hops» > .
  «ENDFOREACH»
  «ENDLET»
endm
erew initial .
«ENDFILE»
«ENDDEFINE»

«DEFINE mamoclients FOR MAMOClient»
«FILE name+".maude"»
load mobile.maude .
load model-checker.maude .
load «module.toUpperCase()+".maude ."»
mod «name» is
  ex DISTRIBUTED-MOBILE-MAUDE .
  ex «module.toUpperCase()» .
  op IP : -> String .
  eq IP = "«ip»" .
  eq port = «port» .
  eq server-address = "«serverAddress»" .

  op initial : -> Configuration .
  eq initial = <>
  «LET clientRootObject AS clt»< l(IP,
  «clt.index») : ClientRootObject |
  cnt : «clt.cnt»,
  guests : «clt.guests»,
  forward : «clt.forward»,
  neighbors : «clt.neighbors»,
  state : «clt.state»,
  defNeighbor : «clt.defNeighbor» >
  «FOREACH mobileObjects AS mo»
  < o(l(IP, «clt.index»), «mo.index») : MobileObject |
  mod : upModule('«mo.mod», false),
  s : upTerm(< o(l(IP, «clt.index»), «mo.index») :
  «mo.mod.toLowerCase().toFirstUpper()» |
```

```
  «mo.s» > & none),
  gas : «mo.gas»,
  mode : «mo.mode»,
  hops : «mo.hops» > .
  «ENDFOREACH»
  «ENDLET»
endm
erew initial .
«ENDFILE»
«ENDDEFINE»
```

Worklow file

```
<?xml version="1.0"?>
<workflow>
  <property name="metamodel" value="mamo.generator.project/src/metamodel" />
  <property name="model1" value=
    "mamo.generator.project/src/health.umldiagrams" />
  <property name="Android" value=
    "../../workspace-android/Android/src/Android" />
  <property name="model2" value=
    "mamo.generator.project/src/health.MAMOComponents" />
  <property name="Maude" value="../../workspace-maude/Maude" />

  <!-- set up EMF for standalone execution -->
  <bean class="org.eclipse.emf.mwe.utils.StandaloneSetup" >
    <platformUri value=".."/>
    <RegisterEcoreFile value=
       "platform:/resource/${metamodel}/UMLDiagrams.ecore"/>
    <RegisterEcoreFile value=
       "platform:/resource/${metamodel}/ClassDiagram.ecore"/>
    <RegisterEcoreFile value=
       "platform:/resource/${metamodel}/StateChartDiagram.ecore"/>
    <RegisterEcoreFile value=
       "platform:/resource/${metamodel}/ObjectDiagram.ecore"/>
    <RegisterEcoreFile value=
       "platform:/resource/${metamodel}/MAMO.ecore"/>
    <RegisterEcoreFile value=
       "platform:/resource/${metamodel}/MAMOModules.ecore"/>
    <RegisterEcoreFile value=
       "platform:/resource/${metamodel}/MAMORules.ecore"/>
    <RegisterEcoreFile value=
       "platform:/resource/${metamodel}/MAMOProcesses.ecore"/>
  </bean>
```

```
  <!-- load model and store it in slot 'model' -->
  <component class="org.eclipse.emf.mwe.utils.Reader">
    <uri value="platform:/resource/${model1}" />
    <modelSlot value="model1" />
  </component>

  <!-- check model -->
  <component class="org.eclipse.xtend.check.CheckComponent">
    <metaModel id="mm" class=
      "org.eclipse.xtend.typesystem.emf.EmfRegistryMetaModel"/>
    <checkFile value="metamodel::Checks" />
    <emfAllChildrenSlot value="model1" />
  </component>

  <!--  generate code -->
  <component class="org.eclipse.xpand2.Generator">
    <metaModel idRef="mm"/>
    <expand value="template::Template::main FOR model1" />
    <outlet path="${Android}">
      <postprocessor class="org.eclipse.xpand2.output.JavaBeautifier" />
</outlet>
  </component>

  <!-- load model and store it in slot 'model' -->
  <component class="org.eclipse.emf.mwe.utils.Reader">
    <uri value="platform:/resource/${model2}" />
    <modelSlot value="model2" />
  </component>

  <!-- check model -->
  <component class="org.eclipse.xtend.check.CheckComponent">
    <metaModel id="mm" class=
       "org.eclipse.xtend.typesystem.emf.EmfRegistryMetaModel"/>
    <checkFile value="metamodel::Checks" />
    <emfAllChildrenSlot value="model2" />
  </component>

  <!--  generate code -->
  <component class="org.eclipse.xpand2.Generator">
    <metaModel idRef="mm"/>
    <expand value="template::Template::main FOR model2" />
    <outlet path="${Maude}" name="APPEND_OUT" append="true"/>
    <outlet path="${Maude}">
      <postprocessor class="org.eclipse.xpand2.output.JavaBeautifier" />
    </outlet>
  </component>
</workflow>
```

Files generated for Android Java project

```
package Android;
public interface Utils {
    public class MaudeType<Type> {
        private Type t;
        public void set(Type t) {this.t = t;}
        public Type get() {return t;}
    };
    public class MaudeString<Type> extends MaudeType<Type> {};
    public class MaudeState<Type> extends MaudeType<Type> {};
    public class MaudeNat<Type> extends MaudeType<Type> {};
    public class MaudeInt<Type> extends MaudeType<Type> {};
    public class MaudeFloat<Type> extends MaudeType<Type> {};
    public class MaudeAttribute<Type> extends MaudeType<Type> {};
    public class MaudeAttributeSet<Type> extends MaudeType<Type> {};
    public class MaudeVoid<Type> extends MaudeType<Type> {};
    public class MaudeDefaultNat<Type> extends MaudeType<Type> {};
    public class MaudeMid<Type> extends MaudeType<Type> {};
    public class Object<Type> extends MaudeType<Type> {};
    public class Forward<Type> extends MaudeType<Type> {};
    public class RoutingTable<Type> extends MaudeType<Type> {};
    public class Socket<Type> extends MaudeType<Type> {};
    public class Location<Type> extends MaudeType<Type> {};
    public class MaudeContents<Type> extends MaudeType<Type> {};
    public class MaudeOid<Type> extends MaudeType<Type> {};
    public class MaudeLoc<Type> extends MaudeType<Type> {};
    public class MaudeListMid<Type> extends MaudeType<Type> {};
    public class MaudeDefaultString<Type> extends MaudeType<Type> {};
    public class MaudeDefaultFloat<Type> extends MaudeType<Type> {};
    public class MaudeDefaultOid<Type> extends MaudeType<Type> {};
    public class Content<Type> extends MaudeType<Type> {};
}

package Android;
import java.util.List;
public interface Root extends Utils {
    RoutingTable<String> empty = new RoutingTable<String>();
    public enum RootState {idle}
    String IP = "localhost";
    int Port = 8000;
    int cnt = 1;
    List<Object<String>> guests = null;
    List<Forward<String>> forward = null;
    RoutingTable<String> neighbors = empty;
    RootState state = RootState.idle;
    Socket defNeighbor = null;
```

```
    public void go(Location<String> l);
    public void goFind(Object<String> o, Location<String> l);
    public void to(Object<String> o, Content<String> c);
}

package Android;
public interface ServerRoot extends Root {}

package Android;
public interface ClientRoot extends Root {
    String serverAddress = "localhost";
}

package Android;
public interface Mobile extends Utils {
    public class MobileType<Type> {
        private Type t;
        public void set(Type t) {this.t = t;}
        public Type get() {return t;}
    };
    public class MobileState<Type> extends MobileType<Type> {};
    public enum MobileMode {active}
    MobileState<String> s = new MobileState<String>();
    int gas = 200;
    MobileMode mode = MobileMode.active;
    int hops = 0;
    void go(String l);
    void goFind(String l);
    void sendMessage(String c);
}

package Android;
public interface doctor extends Mobile {
    public class MaudeMOBILEOBJECTADDITIONALDEFS {};
    public class MaudeLISTMidoptoopniltonoid {};
    public class MaudeDEFAULTNat {};
    public class MaudeDEFAULTFloat {};
    public class MaudeDEFAULTString {};
    public class MaudeDEFAULTOid {};
    public class MaudeStatus<Type> extends MaudeType<Type> {};
    MaudeOid<String> P = new MaudeOid<String>();
    MaudeNat<String> N = new MaudeNat<String>();
    MaudeLoc<String> L = new MaudeLoc<String>();
    MaudeOid<String> D = new MaudeOid<String>();
    MaudeFloat<String> T = new MaudeFloat<String>();
    MaudeFloat<String> BP = new MaudeFloat<String>();
    MaudeFloat<String> G = new MaudeFloat<String>();
    MaudeString<String> Name = new MaudeString<String>();
```

```
    MaudeAttributeSet<String> AtS = new MaudeAttributeSet<String>();
    MaudeListMid<String> OP = new MaudeListMid<String>();

    public MaudeStatus<String> done(MaudeVoid<String> p);
    public MaudeStatus<String> onArrival(MaudeVoid<String> p);
    public MaudeStatus<String> asking(MaudeVoid<String> p);
    public MaudeStatus<String> end(MaudeVoid<String> p);
    public MaudeAttribute<String> patients(MaudeListMid<String> p1,
            MaudeVoid<String> p);
    public MaudeAttribute<String> status(MaudeStatus<String> p1,
            MaudeVoid<String> p);
    public MaudeAttribute<String> healthinformation(
     MaudeDefaultString<String> p1, MaudeDefaultFloat<String> p2,
     MaudeDefaultFloat<String> p3, MaudeDefaultFloat<String> p4,
     MaudeVoid<String> p);
    public MaudeAttribute<String> local(
     MaudeDefaultOid<String> p1, MaudeVoid<String> p);
    public MaudeContents<String> gethealthinformation(
     MaudeMid<String> p1, MaudeVoid<String> p);
    public MaudeContents<String> healthinformation(
     MaudeString<String> p1, MaudeFloat<String> p2,
     MaudeFloat<String> p3, MaudeFloat<String> p4,
     MaudeVoid<String> p);
}

package Android;
import java.util.List;
import com.sun.javafx.collections.MappingChange.Map;
public class HealthCareDoctor implements doctor {
    Map<String, List<String>> Conf;
    public MaudeStatus<String> done(MaudeVoid<String> p) {return null;};
    public MaudeStatus<String> onArrival(MaudeVoid<String> p) {return null;};
    public MaudeStatus<String> asking(MaudeVoid<String> p) {return null;};
    public MaudeStatus<String> end(MaudeVoid<String> p) {return null;};
    public MaudeAttribute<String> patients(
            MaudeListMid<String> p1, MaudeVoid<String> p) {return null;};
    public MaudeAttribute<String> status(
            MaudeStatus<String> p1, MaudeVoid<String> p) {return null;};
    public MaudeAttribute<String> healthinformation(
            MaudeDefaultString<String> p1, MaudeDefaultFloat<String> p2,
            MaudeDefaultFloat<String> p3,
            MaudeDefaultFloat<String> p4, MaudeVoid<String> p) {return null;};
    public MaudeAttribute<String> local(
            MaudeDefaultOid<String> p1, MaudeVoid<String> p) {return null;};
    public MaudeContents<String> gethealthinformation(
            MaudeMid<String> p1, MaudeVoid<String> p) {return null;};
    public MaudeContents<String> healthinformation(
            MaudeString<String> p1, MaudeFloat<String> p2,
```

```
        MaudeFloat<String> p3, MaudeFloat<String> p4,
        MaudeVoid<String> p) {return null;};
public void transitions() {
  do {
    if (Conf.map("HealthCareDoctor").contains(
      "< D : V@Doctor | patients : o(L,N) . OP, status : done, AtS >")) {
      Conf.map("HealthCareDoctor").remove(
      "< D : V@Doctor | patients : o(L,N) . OP, status : done, AtS >");
      Conf.map("HealthCareDoctor").add(
      "< D : V@Doctor | patients : o(L,N) . OP, status : onArrival,
      AtS >");
      goFind("go-find(o(L,N), L)");
    }
    if (Conf.map("HealthCareDoctor").contains(
      "< D : V@Doctor | patients : P . OP, status : onArrival, AtS >")) {
      Conf.map("HealthCareDoctor").remove(
      "< D : V@Doctor | patients : P . OP, status : onArrival, AtS >");
      Conf.map("HealthCareDoctor").add(
      "< D : V@Doctor | patients : P . OP, status : asking, AtS >");
      sendMessage("(to P : get-health-information(D))");
    }
    if (Conf.map("HealthCareDoctor").contains(
      "< D : V@Doctor | patients : P . OP, status : asking, AtS >")
      && Conf.map("HealthCareDoctor").contains(
      "(to D : health-information(Name, T, BP, G))")) {
      Conf.map("HealthCareDoctor").remove(
      "< D : V@Doctor | patients : P . OP, status : asking, AtS >");
      Conf.map("HealthCareDoctor").add(
      "< D : V@Doctor | patients : OP,
      health-information : Name(temperature: T, blood-pressure: BP,
      glucose: G), status : done, AtS >");
    }
    if (Conf.map("HealthCareDoctor").contains(
      "< D : V@Doctor | patients : no-id, status : done, local : o(L,N),
      AtS >")) {
      Conf.map("HealthCareDoctor").remove(
      "< D : V@Doctor | patients : no-id, status : done, local : o(L,N),
      AtS >");
      Conf.map("HealthCareDoctor").add(
      "< D : V@Doctor | patients : no-id, status : end, local : o(L,N),
      AtS >");
      go("go(L)");
    }
  } while (!Conf.map("HealthCareDoctor").isEmpty());
}
@Override
public void go(String l) {
    // TODO Auto-generated method stub
```

```
    }
    @Override
    public void goFind(String l) {
        // TODO Auto-generated method stub
    }
    @Override
    public void sendMessage(String c) {
        // TODO Auto-generated method stub
    }
}

package Android;
public interface mcdoctor extends Mobile {
    public class MaudeMOBILEOBJECTADDITIONALDEFS {};
    public class MaudeLISTMidoptoopniltonoid {};
    public class MaudeDEFAULTNat {};
    public class MaudeDEFAULTFloat {};
    public class MaudeDEFAULTString {};
    public class MaudeDEFAULTOid {};
    public class MaudeStatus<Type> extends MaudeType<Type> {};

    MaudeOid<String> P = new MaudeOid<String>();
    MaudeNat<String> N = new MaudeNat<String>();
    MaudeLoc<String> L = new MaudeLoc<String>();
    MaudeOid<String> D = new MaudeOid<String>();
    MaudeFloat<String> T = new MaudeFloat<String>();
    MaudeFloat<String> BP = new MaudeFloat<String>();
    MaudeFloat<String> G = new MaudeFloat<String>();
    MaudeString<String> Name = new MaudeString<String>();
    MaudeAttributeSet<String> AtS = new MaudeAttributeSet<String>();
    MaudeListMid<String> OP = new MaudeListMid<String>();

    public MaudeStatus<String> done(MaudeVoid<String> p);
    public MaudeStatus<String> onArrival(MaudeVoid<String> p);
    public MaudeStatus<String> asking(MaudeVoid<String> p);
    public MaudeStatus<String> end(MaudeVoid<String> p);

    public MaudeAttribute<String> patients(
            MaudeListMid<String> p1, MaudeVoid<String> p);
    public MaudeAttribute<String> status(
            MaudeStatus<String> p1, MaudeVoid<String> p);
    public MaudeAttribute<String> healthinformation(
            MaudeDefaultString<String> p1, MaudeDefaultFloat<String> p2,
            MaudeDefaultFloat<String> p3,
            MaudeDefaultFloat<String> p4, MaudeVoid<String> p);
    public MaudeAttribute<String> local(
            MaudeDefaultOid<String> p1, MaudeVoid<String> p);
    public MaudeContents<String> gethealthinformation(
```

```
            MaudeMid<String> p1, MaudeVoid<String> p);
    public MaudeContents<String> healthinformation(
            MaudeString<String> p1, MaudeFloat<String> p2,
            MaudeFloat<String> p3, MaudeFloat<String> p4,
            MaudeVoid<String> p);
}

package Android;
import java.util.List;
import com.sun.javafx.collections.MappingChange.Map;
public class HealthCareMCDoctor implements mcdoctor {
    Map<String, List<String>> Conf;
    public MaudeStatus<String> done(MaudeVoid<String> p) {return null;};
    public MaudeStatus<String> onArrival(MaudeVoid<String> p) {return null;};
    public MaudeStatus<String> asking(MaudeVoid<String> p) {return null;};
    public MaudeStatus<String> end(MaudeVoid<String> p) {return null;};
    public MaudeAttribute<String> patients(
            MaudeListMid<String> p1, MaudeVoid<String> p) {return null;};
    public MaudeAttribute<String> status(
            MaudeStatus<String> p1, MaudeVoid<String> p) {return null;};
    public MaudeAttribute<String> healthinformation(
            MaudeDefaultString<String> p1, MaudeDefaultFloat<String> p2,
            MaudeDefaultFloat<String> p3,
            MaudeDefaultFloat<String> p4, MaudeVoid<String> p) {return null;};
    public MaudeAttribute<String> local(
            MaudeDefaultOid<String> p1, MaudeVoid<String> p) {return null;};
    public MaudeContents<String> gethealthinformation(
            MaudeMid<String> p1, MaudeVoid<String> p) {return null;};
    public MaudeContents<String> healthinformation(
            MaudeString<String> p1, MaudeFloat<String> p2,
            MaudeFloat<String> p3, MaudeFloat<String> p4,
            MaudeVoid<String> p) {return null;};
    public void transitions() {
      do {
        if (Conf.map("HealthCareMCDoctor").contains(
          "< D : V@Mcdoctor | patients : o(L,N) . OP, status : done,
          AtS >")) {
          Conf.map("HealthCareMCDoctor").remove(
          "< D : V@Mcdoctor | patients : o(L,N) . OP, status : done,
          AtS >");
          Conf.map("HealthCareMCDoctor").add(
          "< D : V@Mcdoctor | patients : o(L,N) . OP, status : onArrival,
          AtS >");
          goFind("go-find(o(L,N), L)");
        }
        if (Conf.map("HealthCareMCDoctor").contains(
          "< D : V@Mcdoctor | patients : P . OP, status : onArrival,
          AtS >")) {
```

```
          Conf.map("HealthCareMCDoctor").remove(
          "< D : V@Mcdoctor | patients : P . OP, status : onArrival,
          AtS >");
          Conf.map("HealthCareMCDoctor").add(
          "< D : V@Mcdoctor | patients : P . OP, status : asking, AtS >");
          sendMessage("(to P : get-health-information(D))");
        }
        if (Conf.map("HealthCareMCDoctor").contains(
          "< D : V@Mcdoctor | patients : P . OP, status : asking, St : st,
          AtS >")
          && Conf.map("HealthCareMCDoctor").contains(
          "(to D : health-information(Name, T, BP, G))")) {
          Conf.map("HealthCareMCDoctor").remove(
          "< D : V@Mcdoctor | patients : P . OP, status : asking, St : st,
          AtS >");
          Conf.map("HealthCareMCDoctor").add(
          "< D : V@Mcdoctor | patients : OP,
          health-information : Name(temperature: T, blood-pressure: BP,
          glucose: G),
          status : done, St : (st, health-information(Name, T, BP, G)),
          AtS >");
        }
        if (Conf.map("HealthCareMCDoctor").contains(
          "< D : V@Mcdoctor | patients : no-id, status : done, local : o(L,N),
          AtS >")) {
          Conf.map("HealthCareMCDoctor").remove(
          "< D : V@Mcdoctor | patients : no-id, status : done, local : o(L,N),
          AtS >");
          Conf.map("HealthCareMCDoctor").add(
          "< D : V@Mcdoctor | patients : no-id, status : end,
          local : o(L,N), ModelCheck : true, AtS >");
          go("go(L)");
        }
      } while (!Conf.map("HealthCareMCDoctor").isEmpty());
    }
    @Override
    public void go(String l) {
        // TODO Auto-generated method stub
    }
    @Override
    public void goFind(String l) {
        // TODO Auto-generated method stub
    }
    @Override
    public void sendMessage(String c) {
        // TODO Auto-generated method stub
    }
}
```

```
package Android;
public interface patient extends Mobile {
    public class MaudeMOBILEOBJECTADDITIONALDEFS {};
    public class MaudeFLOAT {};
    MaudeMid<String> P = new MaudeMid<String>();
    MaudeMid<String> D = new MaudeMid<String>();
    MaudeFloat<String> T = new MaudeFloat<String>();
    MaudeFloat<String> BP = new MaudeFloat<String>();
    MaudeFloat<String> G = new MaudeFloat<String>();
    MaudeString<String> Name = new MaudeString<String>();
    MaudeAttributeSet<String> AtS = new MaudeAttributeSet<String>();
    public MaudeAttribute<String> temperature(
            MaudeFloat<String> p1, MaudeVoid<String> p);
    public MaudeAttribute<String> bloodpressure(
            MaudeFloat<String> p1, MaudeVoid<String> p);
    public MaudeAttribute<String> glucose(
            MaudeFloat<String> p1, MaudeVoid<String> p);
    public MaudeAttribute<String> name(
            MaudeString<String> p1, MaudeVoid<String> p);
    public MaudeContents<String> gethealthinformation(
            MaudeMid<String> p1, MaudeVoid<String> p);
    public MaudeContents<String> healthinformation(
            MaudeString<String> p1, MaudeFloat<String> p2,
            MaudeFloat<String> p3, MaudeFloat<String> p4,
            MaudeVoid<String> p);
}

package Android;
import java.util.List;
import com.sun.javafx.collections.MappingChange.Map;
public class HealthCarePatient implements patient {
    Map<String, List<String>> Conf;
    public MaudeAttribute<String> temperature(
        MaudeFloat<String> p1, MaudeVoid<String> p) {return null;};
    public MaudeAttribute<String> bloodpressure(
        MaudeFloat<String> p1, MaudeVoid<String> p) {return null;};
    public MaudeAttribute<String> glucose(
        MaudeFloat<String> p1, MaudeVoid<String> p) {return null;};
    public MaudeAttribute<String> name(
        MaudeString<String> p1, MaudeVoid<String> p) {return null;};
    public MaudeContents<String> gethealthinformation(
        MaudeMid<String> p1, MaudeVoid<String> p) {return null;};
    public MaudeContents<String> healthinformation(
        MaudeString<String> p1, MaudeFloat<String> p2,
        MaudeFloat<String> p3, MaudeFloat<String> p4,
        MaudeVoid<String> p) {return null;};
    public void transitions() {
```

```
      do {
        if (Conf.map("HealthCarePatient").contains(
            "< P : V@Patient | name : Name,
            temperature : T, blood-pressure : BP, glucose : G, AtS >")
            && Conf.map("HealthCarePatient").contains(
            "(to P : get-health-information(D))")) {
            Conf.map("HealthCarePatient").remove(
            "< P : V@Patient | name : Name, temperature : T,
            blood-pressure : BP, glucose : G, AtS >");
            Conf.map("HealthCarePatient").add(
            "< P : V@Patient | name : Name, temperature : T,
            blood-pressure : BP, glucose : G, AtS >");
            sendMessage("(to D : health-information(Name, T, BP, G))");
         }
       } while (!Conf.map("HealthCarePatient").isEmpty());
    }
    @Override
    public void go(String l) {
        // TODO Auto-generated method stub
    }
    @Override
    public void goFind(String l) {
        // TODO Auto-generated method stub
    }
    @Override
    public void sendMessage(String c) {
        // TODO Auto-generated method stub
    }
}
```

Maude Code

```
load model-checker.maude .
view Mid from TRIV to MID is
  sort Elt to Mid .
endv
mod DOCTOR is
extending MOBILE-OBJECT-ADDITIONAL-DEFS .
protecting LIST{Mid}* (op __ to _._, op nil to no-id) .
protecting DEFAULT{Nat} .
protecting DEFAULT{Float} .
protecting DEFAULT{String} .
protecting DEFAULT{Oid} .
sort Doctor .
    subsort Doctor < Cid .
```

```
    op Doctor : -> Doctor .
    var V@Doctor : Doctor .
sort Status .
op done : -> Status  [ctor] .
op onArrival : -> Status  [ctor] .
op asking : -> Status  [ctor] .
op end : -> Status  [ctor] .
op patients :_ : List{Mid} -> Attribute  [ctor gather(&)] .
op status :_ : Status -> Attribute  [ctor] .
op health-information :_(temperature:_,blood-pressure:_,glucose:_) :
       Default{String} Default{Float} Default{Float} Default{Float}
       -> Attribute  [ctor] .
op local :_ : Default{Oid} -> Attribute  [ctor] .
op get-health-information : Mid -> Contents  [ctor] .
op health-information : String Float Float Float -> Contents  [ctor] .

var P : Oid .
var N : Nat .
var L : Loc .
var D : Oid .
var T : Float .
var BP : Float .
var G : Float .
var Name : String .
var AtS : AttributeSet .
var OP : List{Mid} .
var  Conf : Configuration .

rl [move] :
Conf  < D : V@Doctor | patients : o(L,N) . OP, status : done, AtS > & none =>
Conf < D : V@Doctor | patients : o(L,N) . OP, status : onArrival, AtS >
& go-find(o(L,N), L) .
rl [onArrival] :
Conf  < D : V@Doctor | patients : P . OP, status : onArrival, AtS > & none =>
Conf < D : V@Doctor | patients : P . OP, status : asking, AtS >
& (to P : get-health-information(D)) .
rl [asking] :
Conf (to D : health-information(Name, T, BP, G)) < D : V@Doctor |
   patients : P . OP, status : asking, AtS > & none =>
Conf < D : V@Doctor | patients : OP, health-information : Name(temperature: T,
   blood-pressure: BP, glucose: G), status : done, AtS >  & none .
rl [end] :
Conf  < D : V@Doctor | patients : no-id, status : done, local : o(L,N), AtS >
& none =>
Conf < D : V@Doctor | patients : no-id, status : end, local : o(L,N), AtS >
& go(L) .
endm
```

```
load model-checker.maude .
view Mid from TRIV to MID is
  sort Elt to Mid .
endv
mod MCDOCTOR is
    extending MOBILE-OBJECT-ADDITIONAL-DEFS .
    protecting LIST{Mid}* (op __ to _._, op nil to no-id) .
    protecting DEFAULT{Nat} .
    protecting DEFAULT{Float} .
    protecting DEFAULT{String} .
    protecting DEFAULT{Oid} .
    sort Mcdoctor .
    subsort Mcdoctor < Cid .
    op Mcdoctor : -> Mcdoctor .
    var V@Mcdoctor : Mcdoctor .
    sort Status .
    op done : -> Status  [ctor] .
    op onArrival : -> Status  [ctor] .
    op asking : -> Status  [ctor] .
    op end : -> Status  [ctor] .
    op patients :_ : List{Mid} -> Attribute  [ctor gather(&)] .
    op status :_ : Status -> Attribute  [ctor] .
    op health-information :_(temperature:_,blood-pressure:_,glucose:_) :
       Default{String} Default{Float} Default{Float} Default{Float}
       -> Attribute  [ctor] .
    op local :_ : Default{Oid} -> Attribute  [ctor] .
    op get-health-information : Mid -> Contents  [ctor] .
    op health-information : String Float Float Float -> Contents  [ctor] .

    var P : Oid .
    var N : Nat .
    var L : Loc .
    var D : Oid .
    var T : Float .
    var BP : Float .
    var G : Float .
    var Name : String .
    var AtS : AttributeSet .
    var OP : List{Mid} .
    var  Conf : Configuration .

    including MODEL-CHECKER .
    sort StateList .
    subsort StateList < State .
    op nil : -> StateList [ctor] .
    op _,_ : StateList StateList -> StateList [ctor assoc id: nil] .
    op St :_ : StateList -> Attribute [ctor] .
    op ModelCheck :_ : Bool -> Attribute [ctor] .
```

```
    subsort Status < StateList .
    subsort Contents < StateList .
    subsort ModelCheckResult < Msg .
    op pr : -> Prop [ctor] .
    op state : StateList -> StateList [ctor] .
    vars st st' : StateList .
    var s : State .

    ceq st , state(health-information(Name, T, BP, G)) , st' |=
        pr = true if T < 37.5 /\ BP < 15.0 /\ G < 1.4 .

    eq state(st) |= pr = false [owise] .
    crl [tr1] : state(s, st) => state(s) , state(st)
        if s =/= nil /\ st =/= nil .
    rl [tr2] : state(nil) => state(end) .
    rl [tr3] : state(end) => state(end) .
    rl [end] : < D : V@Mcdoctor | St : st, ModelCheck : true, AtS > & none
               => modelCheck(state(st), [] <> pr) & none .
    rl [move] :
    Conf  < D : V@Mcdoctor | patients : o(L,N) . OP, status : done, AtS >
    & none =>
    Conf < D : V@Mcdoctor | patients : o(L,N) . OP, status : onArrival, AtS >
    & go-find(o(L,N), L) .
    rl [onArrival] :
    Conf  < D : V@Mcdoctor | patients : P . OP, status : onArrival, AtS >
    & none =>
    Conf < D : V@Mcdoctor | patients : P . OP, status : asking, AtS >
    & (to P : get-health-information(D)) .
    rl [asking] :
    Conf (to D : health-information(Name, T, BP, G))
    < D : V@Mcdoctor | patients : P . OP, status : asking, St : st, AtS >
    & none =>
    Conf < D : V@Mcdoctor | patients : OP, health-information :
    Name(temperature: T, blood-pressure: BP, glucose: G), status : done,
    St : (st, health-information(Name, T, BP, G)), AtS >  & none .
    rl [end] :
    Conf  < D : V@Mcdoctor | patients : no-id, status : done, local : o(L,N),
    AtS > & none =>
    Conf < D : V@Mcdoctor | patients : no-id, status : end, local : o(L,N),
    ModelCheck : true, AtS >  & go(L) .
endm

load model-checker.maude .
view Mid from TRIV to MID is
  sort Elt to Mid .
endv
mod PATIENT is
    extending MOBILE-OBJECT-ADDITIONAL-DEFS .
```

```
    extending FLOAT .
    sort Patient .
    subsort Patient < Cid .
    op Patient : -> Patient .
    var V@Patient : Patient .
    op temperature :_ : Float -> Attribute  [ctor] .
    op blood-pressure :_ : Float -> Attribute  [ctor] .
    op glucose :_ : Float -> Attribute  [ctor] .
    op name :_ : String -> Attribute  [ctor] .
    op get-health-information : Mid -> Contents  [ctor] .
    op health-information : String Float Float Float -> Contents  [ctor] .

    var P : Mid .
    var D : Mid .
    var T : Float .
    var BP : Float .
    var G : Float .
    var Name : String .
    var AtS : AttributeSet .
    var  Conf : Configuration .

    rl [get-des] :
    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

load model-checker.maude .
load mobile.maude .
load DOCTOR.maude .
mod DocServerProcess is
  ex DISTRIBUTED-MOBILE-MAUDE .
  ex DOCTOR .
  op IP : -> String .
  eq IP = "localhost" .
  eq port = 8000 .
  op initial : -> Configuration .
  eq initial = <>
  < l(IP, 0) : ServerRootObject |
    cnt : 1, guests : o(l(IP, 0), 0), forward : 0 |-> (l(IP, 0), 0),
    neighbors : empty, state : idle, defNeighbor : null >
  < o(l(IP, 0), 0) : MobileObject | mod : upModule('DOCTOR, false),
    s : upTerm(< o(l(IP, 0), 0) : Doctor | status : done,
                 local : 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
erew initial .

load model-checker.maude .
load mobile.maude .
load MCDOCTOR.maude .
mod MCDocServerProcess is
  ex DISTRIBUTED-MOBILE-MAUDE .
  ex MCDOCTOR .
  op IP : -> String .
  eq IP = "localhost" .
  eq port = 8000 .
  op initial : -> Configuration .
  eq initial = <>
  < l(IP, 0) : ServerRootObject | cnt : 1, guests : o(l(IP, 0), 0),
    forward : 0 |-> (l(IP, 0), 0), neighbors : empty, state : idle,
    defNeighbor : null >
  < o(l(IP, 0), 0) : MobileObject | mod : upModule('MCDOCTOR, false),
       s : upTerm(< o(l(IP, 0), 0) : Mcdoctor | St : nil, status : done,
       local : o(l(IP, 0), 0), patients : o(l(IP, 1), 0)  > & none),
    gas : 200,
    mode : active,
    hops : 0 > .
endm
erew initial .

load mobile.maude .
load model-checker.maude .
load PATIENT.maude .
mod PatClientProcess1 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, 1) : ClientRootObject | cnt : 1, guests : o(l(IP, 1), 0),
      forward : 0 |-> (l(IP, 1), 0), neighbors : empty,
      state : idle, defNeighbor : null >
  < o(l(IP, 1), 0) : MobileObject | mod : upModule('PATIENT, false),
      s : upTerm(< o(l(IP, 1), 0) : Patient | name : "Pat1",
                   temperature : 37.0, blood-pressure : 12.0, glucose : 1.3 >
                   & none),
      gas : 200,
```

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

Bibliography

[1] *Android Debug Bridge*. `https://developer.android.com/studio/command-line/adb.html`. Accessed: 2019-06-01.

[2] *Android Studio*. `https://developer.android.com/studio/index.html`. Accessed: 2019-06-01.

[3] *Android Virtual Device Manager*. `https://developer.android.com/studio/run/managing-avds.html`. Accessed: 2019-06-01.

[4] Anthony Anjorin, Erhan Leblebici, and Andy Schürr. "20 Years of Triple Graph Grammars: A Roadmap for Future Research". In: *ECEASST* 73 (2015).

[5] Messaoud Bendiaf et al. "A Model Transformation Approach for Specifying Real-Time Systems and Its Verification Using RT-Maude". In: *IJITWE* 12.4 (2017), pp. 22–41.

[6] Grady Booch, James E. Rumbaugh, and Ivar Jacobson. *The unified modeling language user guide - covers UML 2.0, Second Edition*. Addison Wesley object technology series. Addison-Wesley, 2005.

[7] Manuel Clavel et al., eds. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Vol. 4350. Lecture Notes in Computer Science. Springer, 2007.

[8] Manuel Clavel et al. "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.

[9] Luis Corral, Alberto Sillitti, and Giancarlo Succi. "Software assurance practices for mobile applications - A survey of the state of the art". In: *Computing* 97.10 (2015), pp. 1001–1022.

[10] Luis Corral et al. "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*. Ed. by Hausi A. Müller et al. ACM, 2014, pp. 31–37.

[11] Luis Corral et al. "DroidSense: A Mobile Tool to Analyze Software Development Processes by Measuring Team Proximity". In: *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings*. Ed. by Carlo A. Furia and Sebastian Nanz. Vol. 7304. Lecture Notes in Computer Science. Springer, 2012, pp. 17–33.

[12] Krzysztof Czarnecki and Simon Helsen. "Feature-based survey of model transformation approaches". In: *IBM Systems Journal* 45.3 (2006), pp. 621–646.

[13] Francisco Durán, Adrián Riesco, and Alberto Verdejo. "A Distributed Implementation of Mobile Maude". In: *Electr. Notes Theor. Comput. Sci.* 176.4 (2007), pp. 113–131.

[14] *Eclipse*. `https://www.eclipse.org/`. Accessed: 2019-06-01.

[15] *Eclipse for Android*. `https://www.altexsoft.com/blog/engineering/pros-and-cons-of-android-app-development/`. Accessed: 2019-06-01.

[16] *Eclipse Modeling Framework*. `https://www.eclipse.org/modeling/emf/`. Accessed: 2019-04-10.

[17] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. "The Maude LTL Model Checker". In: *Electr. Notes Theor. Comput. Sci.* 71 (2002), pp. 162–187.

[18] *Fabric*. `https://fabric.io/kits?utm_campaign=fabric-marketing&utm_medium=natural`. Accessed: 2019-06-01.

[19] *GameMaker: Studio*. `https://www.yoyogames.com/gamemaker`. Accessed: 2019-06-01.

[20] J. Gao et al. "Mobile Application Testing: A Tutorial". In: *Computer* 47.2 (2014), pp. 46–55.

[21] *Gradle*. `https://gradle.org/`. Accessed: 2019-06-01.

[22] *Graphical Modeling Framework*. `https://www.eclipse.org/modeling/gmp/`. Accessed: 2019-04-10.

[23] *IntelliJ IDEA*. `https://www.jetbrains.com/idea/`. Accessed: 2019-06-01.

[24] Ajay Kumar Jha, Deok Yeop Kim, and Woo Jin Lee. "A framework for testing Android apps by reusing test cases". In: *Proceedings of the 6th International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2019, Montreal, QC, Canada, May 25, 2019*. Ed. by Eli Tilevich. IEEE, 2019, pp. 20–24.

[25] Mona Erfani Joorabchi, Ali Mesbah, and Philippe 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.

[26] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA explained - the Model Driven Architecture: practice and promise*. Addison Wesley object technology series. Addison-Wesley, 2003. ISBN: 978-0-321-19442-8.

[27] Erhan Leblebici et al. "A Comparison of Incremental Triple Graph Grammar Tools". In: *ECEASST* 67 (2014).

[28] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.

[29] M.L. Murphy. *The Busy Coder's Guide to Advanced Android Development*. CommonsWare, LLC, 2009. ISBN: 9780981678016. URL: `https://books.google.dz/books?id=fQmvPwAACAAJ`.

[30] Ingrid do Nascimento Mendes and Arilo Claudio 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*. Ed. by Aaron Marcus. Vol. 9746. Lecture Notes in Computer Science. Springer, 2016, pp. 456–468.

[31] D. Nicol. *Mobile Strategy: How Your Company Can Win by Embracing Mobile Technologies*. IBM Press. Pearson Education, 2013. ISBN: 9780133094947. URL: `https://books.google.dz/books?id=8w-VlkM0oUwC`.

[32] Xuan Lam Pham, Thi Huyen Nguyen, and Gwo Dong Chen. "Research Through the App Store: Understanding Participant Behavior on a Mobile English Learning App". In: *Journal of Educational Computing Research* 56.7 (2018), pp. 1076–1098.

[33] Andy Schürr. "Specification of Graph Translators with Triple Graph Grammars". In: *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings*. Ed. by Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer. Vol. 903. Lecture Notes in Computer Science. Springer, 1994, pp. 151–163.

[34] Norman Shaw and Ksenia Sergueeva. "The non-monetary benefits of mobile commerce: Extending UTAUT2 with perceived value". In: *Int. J. Inf. Manag.* 45 (2019), pp. 44–55.

[35] *TGG Interpreter*. `http://jgreen.de/tools/tgg-interpreter/`. Accessed: 2019-04-10.

[36] Anthony 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*. Ed. by Imed Hammouda et al. Vol. 378. IFIP Advances in Information and Communication Technology. Springer, 2012, pp. 401–402.

[37] Anthony 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*. Ed. by Imed Hammouda et al. Vol. 378. IFIP Advances in Information and Communication Technology. Springer, 2012, pp. 401–402.

[38] Anthony 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*. Ed. by Gruia-Catalin Roman and Kevin J. Sullivan. ACM, 2010, pp. 397–400.

[39] Dirk Weise et al. *MDA: Principles of Model-Driven Architecture*. Addison-Wesley Professiona, 2004. ISBN: 0201788918.