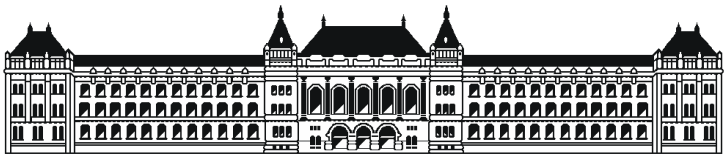


Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Using UML Sequence Diagrams for the Requirement Analysis of Mobile Distributed Systems

Diploma Thesis

Author
Áron Gergely Hamvas

Advisor
Zoltán Micskei

May 12, 2010

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

DIPLOMATERV FELADAT

Hamvas Áron Gergely
szigorló informatikus hallgató
(nappali tagozat műszaki informatikai szak)

UML Szekvencia diagramok használata mobil elosztott rendszerek követelményvizsgálatához
(a feladat szövege a mellékletben)

A tervfeladatot összeállította és a tervfeladat tanszéki konzulense:

Micskei Zoltán
egyetemi tanársegéd

A záróvizsga tárgyai: Informatikai rendszerek szolgáltatásbiztonsága
Valószínűségszámítás
Információs rendszerek fejlesztése

A tervfeladat kiadásának napja:
A tervfeladat beadásának határideje:

dr. Majzik István	*dr. Horváth Gábor*
docens, diplomaterv felelős	docens, tanszékvezető

A tervet bevette:
A terv beadásának dátuma:
A terv bírálója:

MELLÉKLET

UML Szekvencia diagramok használata mobil elosztott rendszerek követelményvizsgálatához

Napjainkban a mobil számítástechnikai rendszerek szinte mindenütt megtalálhatóak. A mobil alkalmazások verifikációja azonban új kihívásokkal szolgál a „hagyományos" elosztott rendszerekéhez képest. A mobilitás velejárója ugyanis a rendszer struktúrájának folyamatos változása. A csomópontok állandó mozgásban vannak; egymástól eltávolodhatnak, ami a kommunikációs kapcsolat megszakadásához vezethet. Továbbá, a résztvevő eszközöket (PDA, GPS, laptop, stb.) olykor ki-, bekapcsolják üzemeltetőik, ami a csomópontok dinamikus létrejöttét és megszűnését jelenti. Mobil ad hoc hálózatokra jellemző az üzenetszórás alapú kommunikáció, és a rendszer topológiájának pontos ismerete hiányában, a küldőnek nincs előzetes tudomása arról, hogy hány másik csomópont fogadja az elküldött üzenetet, illetve reagál arra.

Az ismertetett tulajdonságokkal rendelkező rendszerek működé-sének leírásához segítséget nyújt a grafikus forgatókönyv-leíró nyelvek használata. Ezeknek a nyelveknek a segítségével ugyanis megfogalmazhatók a rendszer viselkedésével kapcsolatos követelmények. Később a tesztelés során pedig a tesztcélok megfeleltethetők ezeknek a követelményeknek, a tesztcélok eléréséhez pedig tesztesetek készíthetők. A tesztesetek lefutási szálait összevetve a követelményspecifikációkkal eldönthető, hogy a rendszer a helyes viselkedést valósítja-e meg. A tanszéken korábban definiáltak egy grafikus forgatókönyv-leíró nyelvet az UML szekvencia diagramok kiterjesztésével, amely segítségével mobil elosztott rendszerekhez is készíthetők követelményspecifikációk.

A diplomatervező hallgató feladata megismerkedni az UML szekvencia diagramokkal és a mobil környezetekhez javasolt kiegészítésével, valamint elkészíteni egy eszközt, ami az ilyen követelmények megadására és ellenőrzésére képes.

1. Végezzen irodalomkutatást a mobil elosztott rendszerek tesztelésével kapcsolatban, és mutassa be annak legelterjedtebb formáit.

2. Vizsgálja meg az elterjedt grafikus forgatókönyv-leíró nyelvek alkalmazhatóságát mobil környezetben, és mutassa be a tanszéken kidolgozott nyelvet.

3. Tervezzen meg, majd készítsen el egy eszközt, amivel forgatókönyv követelményeket lehet megadni, majd ezen követelményeket ellenőrizni konkrét lefutásokon.

4. Mutassa be a program használatát példa forgatókönyvek használatával, és értékelje az elkészült megoldást.

Micskei Zoltán
egyetemi tanársegéd

HALLGATÓI NYILATKOZAT

Alulírott *Hamvas Áron Gergely*, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segédeszközök nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Tudomásul veszem, hogy az elkészült diplomatervben található eredményeket a Budapesti Műszaki és Gazdaságtudományi Egyetem, a feladatot kiíró egyetemi intézmény saját céljaira felhasználhatja.

Budapest, May 12, 2010

Hamvas Áron Gergely
hallgató

Contents

Kivonat

Napjainkban a mobil számítástechnikai rendszerek szinte mindenütt megtalálhatóak. Gyakran szükség van arra, hogy a mobil eszközök egymással összekapcsolódva, elosztott rendszerként oldjanak meg egy-egy feladatot. Ilyen elosztott rendszerek fejlesztése és tesztelése azonban felvet néhány problémát, ami a rendszer csomópontjainak mobilitásából fakad. Ilyen probléma például a hálózati topológia változékonysága, amellyel korábban, hagyományos elosztott rendszerek esetén kevésbé, vagy egyáltalán nem kellett számolni.

A diplomaterv elkészítése során megismerkedtem a hagyományos elosztott rendszerek tesztelése során alkalmazott módszerekkel, illetve tanulmányoztam ezek használhatóságát mobil rendszerek esetén. Megismertem továbbá néhány olyan tesztelési módszertant, amelyek szembenéznek a mobil elosztott rendszerek által állított új kihívásokkal, ezáltal jól használhatóak ezek tesztelésére.

A feladatom részét képezte egy olyan eszköz elkészítése, amely hozzájárul a mobil rendszerek teszteléséhez. Az elkészített eszköz a tesztelés alatt álló rendszer viselkedésével szemben támasztott, egy bizonyos forgatókönyv-leíró nyelven megfogalmazott követelmények teljesülését ellenőrizze, a tesztesetek futtatása során rögzített kommunikációs események (*lefutási szál*) vizsgálatával, illetve a követelményspecifikációkkal való összevetésével.

Az eszköz elkészítésére való felkészülés során megismerkedtem a leggyakrabban használt grafikus forgatókönyv-leíró nyelvekkel (*UML 2.0 Szekvencia Diagram*, *Message Sequence Chart* (MSC) és ennek változatai), továbbá megismertem egy kifejezetten mobil rendszerek viselkedésével szemben támasztott követelmények megfogalmazására létrehozott új nyelvvel is, amely a TERMOS névre hallgat.

A diplomaterv keretén belül elkészített eszköz képes a TERMOS nyelven megfogalmazott követelményleírások szintaktikai és szemantikai ellenőrzésére, valamint a követelményleírások alapján elkészített véges automaták használatával, a tesztelés során rögzített lefutási szálak helyességellenőrzésére. Ezáltal az eszköz hozzájárul egy - a diplomatervben ismertett - tesztelési keretrendszer működéséhez.

A diplomaterv a megismert technológiákat, tesztelési paradigmákat és a grafikus forgatókönyvleíró nyelveket tárgyalja, majd bemutatja, hogy az elkészített eszköz hogyan járul hozzá ezek gyakorlatban történő alkalmazásához, illetve miként építkezik belőlük.

Abstract

As mobile computing systems are proliferating, mobile devices are more and more often required to solve specific problems by co-operating with one another, operating as a distributed system. The development and the testing of such distributed systems, however, raises some previously unseen issues, thanks to the mobility of the nodes participating in the communication. A good example of such issues is the dynamically changing network topology, which is a new problem in the area of distributed systems.

Working on my diploma thesis project, I have become familiar with the testing methodologies applied when testing traditional distributed systems, and I have also examined their usability in case of testing mobile computing systems. I have also reviewed the state-of-the-art testing of such systems, which is also discussed in this diploma thesis.

As the main contribution of my diploma thesis project, I have developed a tool that contributes to the testing of mobile distributed systems by validating execution traces recorded during the testing of such systems against requirement scenario descriptions created using a graphical scenario language.

As I was getting prepared for the development of this tool, I got acquainted with the most widely used graphical scenario languages (such as *UML 2.0 Sequence Diagrams*, *Message Sequence Charts*, and some of their newer versions and extensions, too), as well as with a new graphical scenario language (called TERMOS), which had been created for defining requirement scenarios for mobile computing systems, specifically.

The tool designed and developed as part of my diploma thesis project is able to (i) check the syntax and the semantical soundness of requirement scenario descriptions defined in TERMOS and (ii) to validate execution traces against these scenario descriptions by using finite state automata which the tool builds from the TERMOS diagrams.

This diploma thesis, first, discusses the technologies, testing paradigms, and the graphical scenario languages I got familiar with when working on this project. Then, it presents the tool developed as a part of this project, and how it fits into one of the introduced testing methodologies.

Introduction

As mobile communication is becoming more and more common, a huge paradigm shift is taking place in the world of computing systems. It has been natural for people to communicate with others while they are sitting on a train and traveling, for a while now. Since handheld devices for different purposes became fashionable, the need to be able to make them co-operate with one another has been rising. People want to communicate with each other and also with their environment. They not only want to know where they are, but want to access any information about their current location they need. Sometimes, accessing information is not enough; people want to interact with their environment.

An example would be a smart phone that collects information about its surroundings from other smart phones within communication distance. Or we can think of a black-box in a car that collects information about the vehicle (e.g. speed, position, fuel consumption, etc.) continuously, then sends the collected data to a central database when a communication access point to this central system is within communication distance.

Distributed systems have been created for decades for 'similar' purposes, i.e. to make separate hosts co-operate with each other in order to solve a problem or to be able to provide value-added services. The development of such systems has raised several challenges for system designers and software developers. Communication, resource sharing, and reliability have been some of the most important issues to be handled, which, e.g. in case of 'traditional' standalone systems, had not meant a problem before.

Creating distributed systems that operate in a mobile setting raise, however, just as many new challenges as their earlier versions did first. Traditional distributed systems have a convenient feature, namely the fix geographical position of the participating nodes. In a mobile setting, nodes are moving all the time, joining and leaving the system in a virtually arbitrary manner.

Creating an application that is expected to operate using a network of nodes with an ever-changing topology like this is definitely challenging. Both the design and the validation parts of the development process raise previously unseen challenges. This diploma thesis addresses the problems and issues related to the validation of mobile distributed systems.

Chapter 1, first, introduces the most common testing approaches of traditional distributed systems. Then, after discussing the specificities of a mobile ad-hoc environment and the challenges it raises for system designers, the previously presented approaches are evaluated based on their usability in the validation process of mobile distributed systems. Finally, a state-of-the-art approach is introduced that addresses the problems coming with

mobile ad-hoc networking.

As the concept of *graphical scenario languages* is used in common validation approaches in case of both traditional and mobile distributed systems, the most common languages are introduced and their expressiveness and usability for modeling the behaviour of mobile computing systems are evaluated in *Chapter 2*, along with a recently proposed scenario language.

In *Chapter 3*, the design and implementation process of a tool that supports the validation of mobile distributed systems is presented. The diploma thesis also addresses the issues that needed to be handled during the realization of this tool.

The implemented tool is evaluated in *Chapter 4* by showing an example system that tries to rely on ad-hoc communication as much as possible and by presenting how the tool contributes to the testing of this system.

Chapter 1

Testing Mobile Distributed Systems

According to Larry D. Wittie's definition, distributed systems are made up by separate hosts, interconnected through a network, co-operating in order to complete a task or to facilitate the execution of different programs having some kind of logical connection with one another [30]. This means that, in case of distributed systems, communication over network is always involved, i.e. an underlying network is essential.

While communication between processes running on the very same host is facilitated as a service provided by the operating system, processes that need to co-operate with one another over a network do not have the comfort of such reliable services. *Latency* and questionable *availability* of the hosts become a real problem. Latency comes from the relatively slow communication due to the geographical distance of the communicating hosts. If the distributed system is assigned a task and, in the process of completing it, one of the hosts drop out, the completion of the task might become harder or it might even fail to be completed successfully.

Another problem is that nodes of the distributed system do not have a real-time consistent view of the global state. If a host is disconnected from the network for some reason, the system needs time to adapt to this new situation. If the disconnected host was essential to solve a problem the system was working on, the change in the topology may result in a system level failure. A host dropping out is not even the worst case scenario if we think about the possibility of a *Byzantine fault* [21] occurring in the system, which can result in providing false output, which is, in most cases, even worse than providing no output at all. Faults like these, therefore, have to be detected and isolated or removed so that they do not result in a system level failure.

To deal with these new challenges that come with distributed systems, several new algorithms and protocols have been defined. For example, to deal with the delays, system designers need to set reasonable time limits. To ensure that the nodes have a consistent view of the group they are currently the members of, a *group membership protocol* can be used, like the one defined in [4]. To provide a reliable means of communication within the group, a *group communication system* is used. It is the duty of the group communication system to ensure that messages sent within the group arrive safely to the recipient(s) [3]. Safely, here, means that the messages arrive exactly once (or if the same message arrives

to the recipient multiple times, it does not affect the result) and in the logical order they are expected to be processed. For this latter purpose a message sequencing algorithm can be used.

Designing distributed systems keeps bringing new challenges, but as we will see, there are several solutions that resolve most of the problems that come from the 'novelty' of independent nodes co-operating over a network.

1.1 Testing Traditional Distributed Systems

Testing distributed systems has brought new challenges too. These new issues are mostly related to the previously mentioned latency, and the non-determinism and concurrency that such systems exhibit. Also, participating nodes have no consistent *a priori* knowledge of the global state of the system or a common time reference. These result in observability and controllability issues that make the validation of such systems difficult [23].

The authors of [23] describe the so-called *passive testing* as one of the most applicable approaches in case of distributed systems. Passive testing means that the SUT is running with either a real or a synthetic workload and the input and output events and values are monitored, while the execution trace is recorded and analyzed to check for violation of properties. Passive testing aims not to interfere with the normal behaviour that would be inevitable if conventional *active testing* would be used. In case of active testing, validation is based on the output of the system for a given input. However, as mentioned previously, controllability is a major issue when testing distributed systems. The main advantage of passive testing over active testing is, therefore, that the previous one does not rely on the controllability of the system under test. In [1], a passive testing method is introduced that can be used for testing distributed systems.

Once the overall testing approach has been selected, the testing artifacts, e.g., test architecture, test setup or test cases, have to be described. There are several graphical scenario languages that can be used to support various test related activities, such as capturing functional behavioural requirements, designing test cases, or specifying test purposes [23]. Typical examples of such scenario languages, including *Message Sequence Charts* and *UML Sequence Diagrams*, are mentioned in [23] for these purposes, both of which will be examined more extensively in Chapter 2. It is worth noting here that the usability of Sequence Diagrams for test related purposes has increased with the introduction of the *UML 2.0 Testing Profile* (U2TP) [5].

Based on the requirements, defined using a graphical scenario language, the test cases can be designed as well. If the requirements are defined using UML 2.0 Sequence diagrams, the test architecture and the test behaviour can easily be designed with the help of U2TP. For the implementation of test cases, *Testing and Test Control Notation Version 3* (TTCN-3) has been one of the most powerful languages, being able to "*accommodate complex test architectures and procedures.*" [23] TTCN-3 is not only a powerful language, but includes most of the concepts that U2TP does, which makes it capable to be used for implementing the test architecture and the test behaviour defined using U2TP [26].

1.2 Mobile Computing Systems

Distributed systems that contain devices with physical mobility are often referred to as *mobile computing systems* or *mobile distributed systems*. With the emergence of mobile technologies, this type of distributed systems started to proliferate. The design, development, and validation of such systems require new methodologies, as the ones used in accordance with traditional distributed systems are not prepared to handle the characteristics specific to mobile systems. These latter ones differ from the traditional systems in the following aspects [17]:

Dynamicity of system structure. The number of mobile nodes participating in the system varies over time. Nodes can be created or shut down dynamically. Moreover, a node can move out of reach and the connection may be lost. As the devices move arbitrarily, the connection topology changes in an unpredictable manner, i.e. the topology is unstable. Since, in traditional distributed systems, the co-operating hosts reside in a fixed geographical position, system designers did not need to take this phenomenon into consideration when working out methodologies.

Communication with unknown number of partners in a local vicinity. In the ad-hoc domain, as the topology changes all the time and the nodes cannot have a priori knowledge of the global state of the system, broadcasting messages is an acceptable form of communication. The number of recipients is unknown for the sender; whoever is in communication range may listen to the message and react.

Context awareness. At all times, a node should have a consistent view of its context, and needs to adapt and react to the continuous contextual changes that arise from the node's physical mobility. The context includes any relevant attribute of the device and its interaction with other devices and the environment.

These features of mobile computing systems require an even more complex test architecture to be used for validation, and, as it will be demonstrated in the next chapter, the graphical scenario languages used in accordance with traditional distributed systems are not prepared to handle these features. Thus, new formalisms are needed for creating system models and for defining behavioural requirements for a mobile system.

1.3 State-of-the-art Testing of Mobile Systems

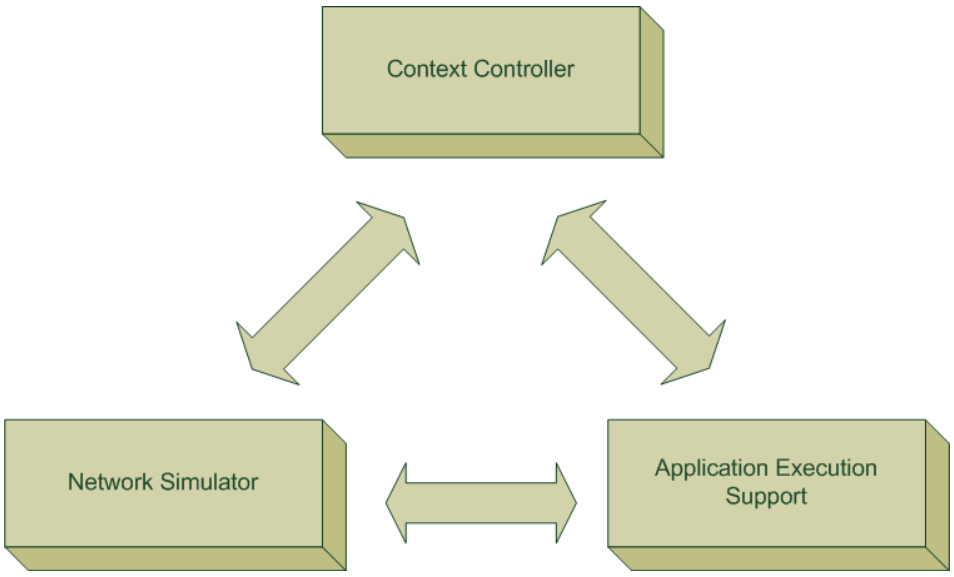
One of the many challenging things about testing mobile computing systems is the difficulty of simulating communication over a network with an ever-changing topology. One way could be people walking with handheld devices in an area and recording the interactions between the devices, and after a certain amount of time the data would be retrieved from the devices and could be checked for violation of properties. The problem with this approach is that it is time consuming and an acceptable coverage of test purposes is hard to guarantee. Also, in case of car-to-car communication applications instead of handheld device applications, it would be very expensive.

A usual approach for simulating wireless communication is to integrate a network simulator into the test platform [29]. Such network simulators have been implemented for testing specific mobile distributed systems. A good example is Ubiwise [2], which is built on 3D game engines and simulates a first-person view of the physical environment, or the topology emulator developed within the European project HIDENETS [8]. Some of them have been used to test applications that involved the previously mentioned car-to-car communication, others for evaluating a health monitoring application. In all of these cases, the network simulator was only one component of the complex test platform.

Another valuable component of such platforms is the *context controller*. As it has previously been mentioned, a feature specific for mobile computing systems is the need for the nodes to be aware of their environment, the context in which they are operating. The context, in a system like this, changes all the time, and the nodes need to adapt to these changes and act accordingly. A typical change in the context occurs when a node senses the signal of another node within communication distance. The way our node adapts to this new situation may depend on its velocity too; e.g. if it is probable that the other node will be out of reach before any reasonable interaction between the two of them could occur, there is no point in establishing a connection. During the test phase, and within the test platform, the context controller is responsible for providing the nodes with such data. By using this component, automated testing of the system in numerous different scenarios can be executed automatically.

With the help of these two components, the context controller and the network simulator, a wireless network can easily be simulated, and the nodes participating in the scenarios can be provided with contextual data at all times. The high level view of a typical test platform [17] that satisfies the needs of testing mobile distributed applications can be seen on Figure 1.1.

Figure 1.1: *High-level View of a Test Platform for Testing Mobile Distributed Systems*



The context controller provides the nodes with their relative position and information about their environment. These are usually environmental parameters that influence the operation of the nodes. Such parameters can be, for instance, the velocity and direction of the node, the geographical position of the node, and signal strength of the nodes within communication distance.

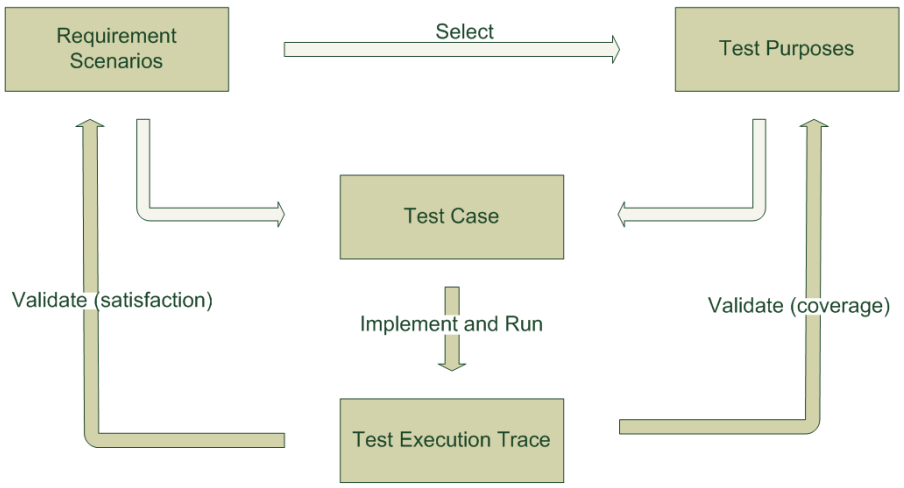
The network simulator creates an environment which is close enough to a real wireless network. It also uses data provided by the context controller; e.g. position of the nodes. With an elaborate network simulator system, we can simulate the continuous change of the geographical positions of the nodes and its effects on the communication over the simulated network.

The application execution support component facilitates the execution of the application running on the individual nodes. It can be used to simulate the underlying platform and the other applications that run on the nodes.

1.3.1 Modeling Used for Testing Mobile Distributed Applications

The test platform described above can easily serve as the underlying platform of a testing framework similar to the one defined in [17]. The framework contributes to the model-based testing of mobile distributed applications. It is also worth noting that this framework satisfies the definition of *passive testing* from Section 1.1. The framework can be seen on Figure 1.2.

Figure 1.2: *Overview of the HIDENETS Testing Framework*



Models can be used in case of mobile distributed applications to capture system level behaviour and spatial topology. Combining these two, scenarios can be described in the mobile setting the application is meant to operate in. These scenario descriptions can serve as behavioural requirement specifications, or representation of test cases or execution

traces. There are several visual modeling languages that can be used to for these purposes; their capabilities will be examined in the next chapter.

The functional testing can, then, be imagined as follows:

Test purposes specify the interaction patterns to be covered during testing are defined.

Requirement scenarios are defined to specify the expected behaviour of the SUT in different communication scenarios. The scenarios are later examined (in the form of executing and analyzing the results of test cases) in order to cover a test purpose.

Test cases can be (automatically) generated based on the test purposes and the requirement scenario descriptions.

Execution traces are recorded and produced as a result of the execution of the test cases. These traces are, then, subject to analysis. One form of the analysis is checking whether or not the requirements described in the requirement specifications or requirement scenario descriptions are satisfied by the system. Ideally, this analysis is automatic.

In [23], a graphical scenario language is proposed for the description of requirement scenarios and spatial configurations, as the available formalisms have been found inadequate for this purpose [29]. These results and conclusions will be presented in the next chapter.

Chapter 2

Graphical Scenario Languages

In this chapter, some of the most well-known scenario languages are introduced and examined from various aspects. The main idea is to find out to what extent are these languages useful in the testing process of mobile distributed systems. As it has been previously mentioned, graphical scenario languages have been found valuable assets when the system under test is a traditional distributed system.

The question is whether the existing languages provide us with the necessary formalisms or we need to extend them somehow, maybe create a completely new one. To be able to answer this question, we need to see how modeling and graphical scenario languages can help us in the testing process when we are dealing with traditional systems.

2.1 Modeling in Software Development

Several types of models are created during the lifecycle of software development. Static models are used to describe the structure, while dynamic models describe the behaviour of the application to be implemented. Sometimes, these two are integrated within an *analysis* and a *design* model. In case of *Model Driven Architecture* (MDA), as Bast, Kelppe, and Warmer compare it to other approaches [19], a so-called *Platform Independent Model* (PIM) is created first, which lacks any detail that is specific to a platform (such as types specific to Java only) enabling designers and developers to choose a development platform as late as possible. Then, after a decision is made about the platform, the *Platform Specific Model* (PSM) can be derived from the PIM using model transformations. If the model relied on a specific platform from the beginning, a new information or requirement could require changes to be made on the whole model. Whereas, having a PIM enables changes to be applied to the models easily. Then, if the previously selected platform turned out to be a bad choice, a new PSM, suitable for the new platform, can be generated from the PIM. That is, using MDA relieves us from the portability problems that can occur in other software development paradigms. Finally, and ideally, the code is generated from the PSM using transformations.

An advantage of most of these software designing approaches is that they do not specify how the models are created. They all rely on the somewhat similar definitions of models

that are well summarized in the following definitions [19]:

A model is a description of (part of) a system written in a well-defined language.

A well-defined language is a language with well-defined form (*syntax*), and meaning (*semantics*), which is suitable for automated interpretation by a computer.

There are several modeling languages that satisfy these definitions, but this chapter is restricted to a more specific subset of these languages, visual modeling languages; i.e. languages that use diagrams for the description of a system. Visual modeling languages help the documentation of the development process as well as developers to understand the structure and the requirements of the system more easily.

There are many languages that allow us to define not only the structure but also the *behaviour* of the system. One of the many ways to describe behaviour is to create *interaction diagrams*, i.e. diagrams that show the interactions between structural elements of the system. In a complex system, several types of interactions occur, and these are often grouped according to *scenarios*. In case of an ATM, a typical scenario can be, for instance, when a user enters a wrong PIN. An interaction diagram can define the behaviour of the ATM in this scenario. Those languages that allow us to visualize such scenarios are often referred to as *graphical scenario languages*.

As it is an accepted and wide-spread method to create plans, models, before creating a system, the testing process itself needs to involve some planning too. It is to be decided what kind of tests should be executed and what the executed tests will tell about the validity of our system.

There are several paradigms existing for the testing of an application, and *Model-based testing* is a paradigm worth mentioning for this approach has become popular recently [28]. Model-based testing often uses the MDA, but this is not necessarily true in all cases. When MDA is used, the approach is usually referred to as *Model-driven testing* (MDT) [18]. MDT is based on creating and transforming models using model transformations, and trying to generate as much of the actual code implementation of the test cases (or the test platform) as possible. This way, the testing process becomes well-documented, easy to understand, and definitely faster.

2.2 Graphical Scenario Languages for Requirement Analysis of Mobile Distributed Systems

In Chapter 1, the testing framework designed within the HIDENETS project was described shortly. As already mentioned, *requirement scenarios* play a significant role in it. According to the authors of the project deliverable [17], *"Scenario descriptions are useful to support test-related activities, such as representation of requirements, of test purposes (i.e. interaction patterns to be covered by testing), of test cases, or of execution traces."* Therefore, using a scenario language, and for the sake of making our own lives more comfortable, preferably a graphical scenario language, scenario descriptions can be created that represent behavioural requirements of distributed systems.

With the help of these requirement scenarios, test cases could be generated automatically, and the execution traces recorded during the execution of the test cases can be (ideally) automatically compared to these scenario descriptions. Thus, the process of determining the result of a test can be supported.

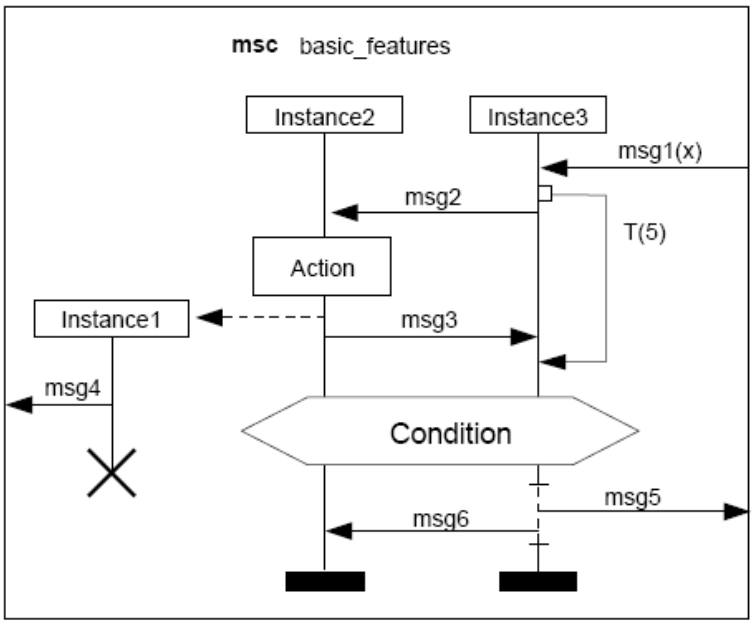
As, in Chapter 1, mobile distributed systems were shown to have some specificities that differ from those of traditional distributed systems, naturally, our expectations from graphical scenario languages are that they handle these characteristical differences. In this chapter, therefore, the existing and widely used scenario languages are analyzed, and a new scenario language is introduced which has been proposed for the specific purpose of supporting the testing of mobile computing systems.

2.2.1 Message Sequence Charts

"Message Sequence Charts (MSC) is a language to describe the interaction between a number of independent message-passing instances." [31] MSC has both graphical and textual representations. The graphical representation is a two-dimensional diagram that is used to overview the behaviour of communicating instances.

The language is widely used because it supports structured design, i.e. simple scenarios can be combined to form more complete specifications. These complete specifications can be defined by means of *High-level Message Sequence Charts* (HMSC).

Figure 2.1: *Basic MSC features [20]*



According to the recommendation, the behaviour defined in an MSC should at least be exhibited by the actual implementation. This is, therefore, an *existential* interpretation.

The graphical representation of an MSC, an MSC diagram, has a heading that defines the name of the MSC and a body that contains the scenario description as the simple example shows in Figure 2.1 [20]. The body is divided into different parts. An *instance area* defines an instance involved in the scenario. It has a head (a rectangular symbol) and

a body that consists of the *axis symbol* and the *stop symbol*. A message is represented by a line connecting the axis symbols of the involved instances and is navigated at the recipient end.

A widely used element is the *condition* element. There are two types of conditions: *setting* conditions that describe the current global or (less frequently) nonglobal state, and *guarding* conditions that restrict the execution of events. In the latter case, the guard is placed at the beginning of its scope. The restriction is either based on the global/nonglobal state or a Boolean expression.

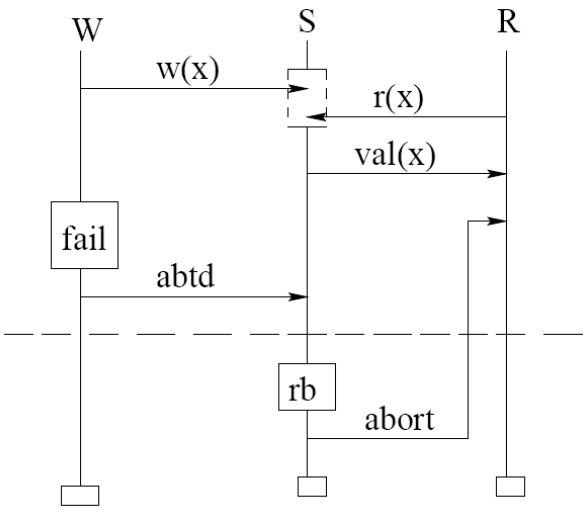
Another group of elements that can be used in MSC scenario descriptions is the group of *inline operator expressions. "The operators refer to alternative, parallel composition, iteration, exception and optional regions."*

Triggered MSCs

As MSCs are rather intended to capture the entire behavior that a system should exhibit, scenario-based notations are also heavily used in earlier stages of the development process. Requirement specification, essentially, takes place in an early phase where some design related issues are not resolved yet. [27]

In this early phase, scenario descriptions are often used to constrain the behaviour of the system rather than to prescribe it. The system is not always expected to exhibit the behaviour defined here. Imagine an error occurring in the system, which is definitely a scenario that is to be avoided. Sometimes, errors occur, and when they do, we expect the system to handle it in a specified way. These *conditional* scenario descriptions are not supported in traditional MSCs [27].

Figure 2.2: *TMSC Example [27]*



Triggered Message Sequence Charts (TMSC) are, therefore, designed to specify such scenarios. A TMSC is constructed of a *trigger* part and an *action* part. The two parts are separated in the diagram by a dashed line. A trigger describes a conditional scenario which may or may not take place during the execution of the implemented system. The

action part defines the expected behaviour, if the events specified in the trigger part occur. This means that the behaviour exhibited by the system is valid if and only if the scenario described in the trigger part never takes place or, if it does, then the steps defined in the action part of the TMSC are executed.

An example of Triggered Message Sequence Charts is shown in Figure 2.2. A writing process (W) and a reading process (R) are accessing the shared storage server S. After the failure in the writing process, a rollback (here: *abort*) needs to be executed, and reading processes must be notified about the failure, too, in order to let them know that the value they previously retrieved from the server is no longer valid.

Live Sequence Charts

According to Klose [20], some of the main problems with MSC are the following:

1. An MSC shows only one possible run of the system. Often the purpose of using MSCs is not to give a possible scenario, but to define mandatory behaviour.

2. An MSC does not express when the behvaiour should be observed, i.e. when the MSC is activated. In MSC-2000, guards and conditions can be used for this purpose, but a sequence of messages cannot be specified as activators (triggers in TMSC semantics).

3. Conditions in MSCs have no formal semantics.

Live Sequence Charts (LSCs) [7] are meant to solve these and some other issues the creators of the language found problematic, too.

Some of the interesting differences between MSCs and LSCs regarding the graphical representation is as follows:

1. Messages can be of type *asynchronous* or *instantaneous*. The latter one replaces *synchronous* messages appearing in MSCs. In fact, two *instantaneous* messages represent a synchronous call and the return message.

2. In terms of progress, LSCs contain *hot* and *cold* locations (places on the instance axis where an event is attached) and messages. Defining a location *hot* means that the following location needs to be reached (i.e. if the running is terminated there, then it is an invalid behaviour). A *hot* message means, that the message has to be sent and received for the system to exhibit the expected behaviour. In contrast, *cold* locations of an LSC do not force the system to reach the following location, and not sending the messages that were defined *cold* is an acceptable system behaviour. At a *cold* location, the instance axis segment is dashed. Similarly, a *cold* message is represented by a dashed line.

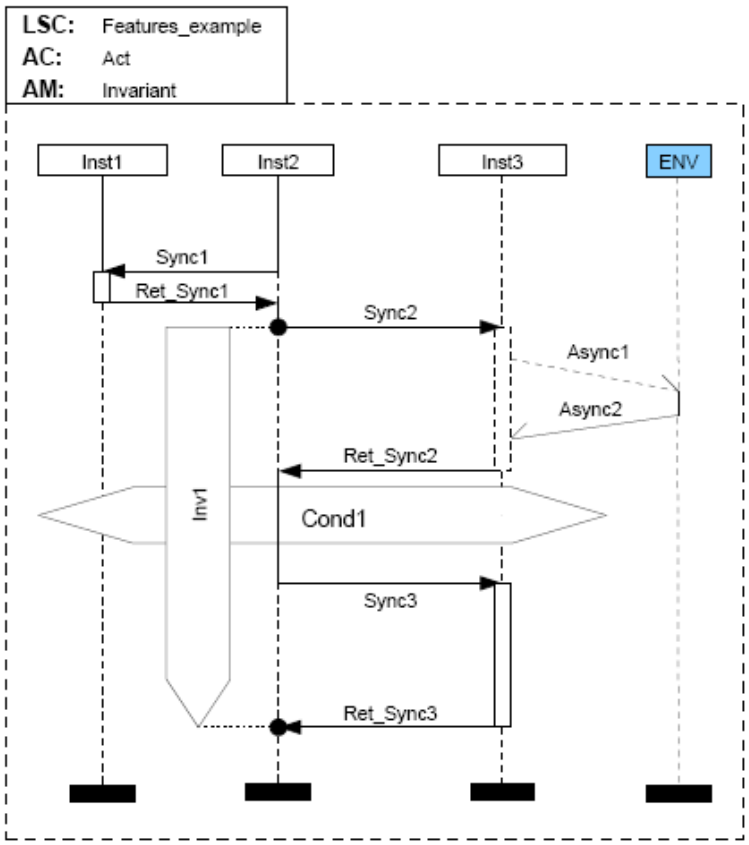
3. Conditions can be either *mandatory* or *possible*. This is a resolution of the common validation problem: 'What if a condition is not satisfied?' In an LSC, if a mandatory condition is not satisfied, then it is to be considered as a system malfunction. In contrast, if a possible condition is not satisfied, it means that the behaviour is not related to the scenario defined by the LSC.

These new features allow LSC to describe requirement scenarios that represent the expected behaviour of the system in certain situations.

An example of Live Sequence Charts can be seen in Figure 2.3. Hot locations and messages can be seen where the axes and the arrows are represented by solid lines and cold locations and messages are represented by dashed lines.

Figure 2.4 shows the difference between using synchronous messages and using instantaneous messages to designate synchronous calls and returns. [20]

Figure 2.3: *Simple LSC [20]*

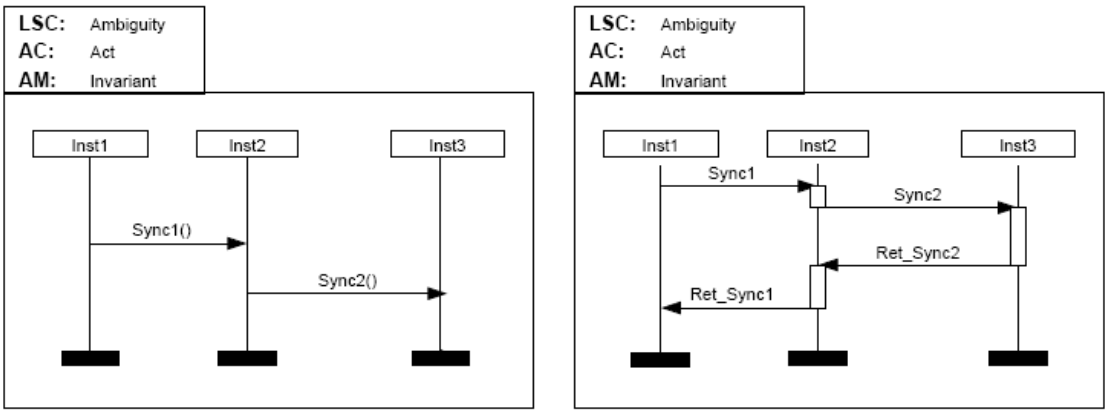


Conclusion

The main problem with Message Sequence Charts according to those, who defined Triggered Message Sequence Charts and Live Sequence Charts was that MSCs do not give any information *when* the specified behaviour should be exhibited by the system.

TMSCs allow users to define triggers, the execution of which implies that the action at the end of the diagram should be executed too. LSCs resolve the triggering by extending MSCs with a feature that makes the diagram capable of expressing that, if a certain sequence of events take place, then a situation is created which requires the system to execute some other action. LSCs are able to express these requirements on an event level and place them anywhere within the diagram, while whatever takes place within the action part of a TMSC, must be preceded by any other event that belongs to the trigger part.

Figure 2.4: *Synchronous Versus Instantaneous Messages [20]*

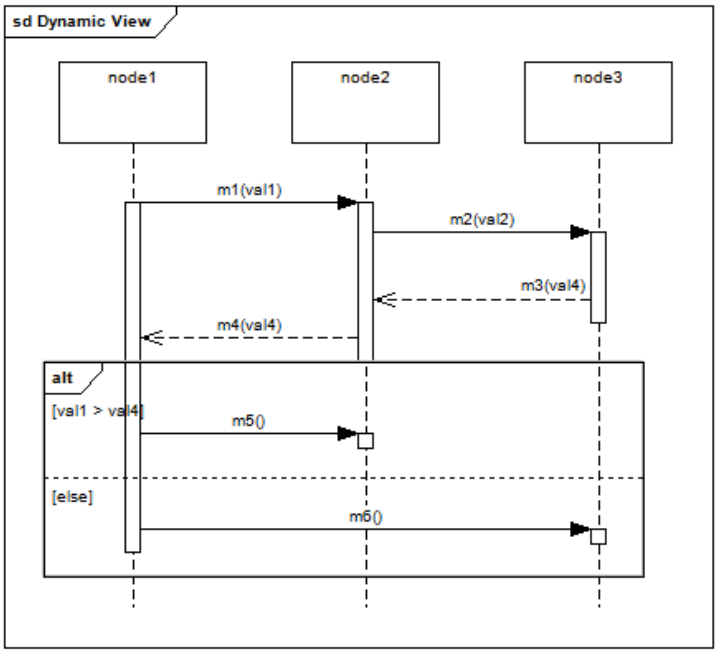


(a) ambiguous LSC (b) possible resolution

None of these languages say anything about spatial configurations of the instances and network topology. However, with the help of *conditions*, more specifically, *possible conditions* of LSCs, the topology changes could be expressed, too. The problem is that none of the languages provide a formalism to describe spatial configurations of the elements.

2.2.2 UML 2.0 Sequence Diagrams

Figure 2.5: *A Sample Sequence Diagram*



The *Unified Modeling Language 2.0* (UML) [24] is the standard of the Object Management Group (OMG). The language supports behavioural description of applications with *UML Sequence Diagrams*. A diagram represents a single scenario, or as it is called in UML,

Table 2.1: *Interaction operator types*

Type	Description
alt	represents a choice of behaviour.
opt	a choice of behaviour where the execution of the contents of the CombinedFragment is optional.
break	the contents of the CombinedFragment are executed instead of the remainder of the Sequence Diagram.
loop	the contents will be repeated the pre-defined number of times.
par	a parallel merge between the contents of the operands of the Combined Fragment.
seq	this operator represents weak sequencing between the behaviours of the operands.
strict	this operator represents a strict sequencing between the behaviours of the operands.
critical	a CombinedFragment designated with this operator represents a critical region, i.e. the traces of the region cannot be interleaved by other OccurrenceSpecifications.
neg	the CombinedFragment with this operator represents invalid traces.
assert	the CombinedFragment with an assert operator defines the only valid traces; i.e all other continuations are invalid.
consider	designates which messages should be considered within this CombinedFragment.
ignore	tells us that there can be other message types that are not present in the CombinedFragment.

Interaction. An Interaction consists of synchronous/asynchronous messages and method calls between the structural elements of the modeled system. These structural elements can be defined in forms of *UML classes*, and the elements that participate in an interaction are the representatives of instances of the defined classes (called *ConnectableElements).* The type of an instance is called *InstanceSpecification.* Every participant of an Interacion is represented in a UML Sequence Diagram by a *Lifeline* the graphical representation of which is a scattered line with a box on top, labeled with the symbolic name of the participant. Events in an *Interaction* are represented by elements called *InteractionFragments.* These fragments can have various types. Message send and receive events are defined as so-called *MessageOccurrenceSpecifications.* These belong to *Messages* that are represented by lines with arrows on one end, connecting the lifelines of the sender/caller and the recipient. The arrow is on the recipient end of the line. If a message results in a specific behaviour executed by the recipient, an *ExecutionSpecification* element is placed on its lifeline. A basic interaction usually consists of these elements.

UML Sequence diagrams can contain many other elements defined by the UML Superstructure [24]. The most significant examples are *CombinedFragments* and *StateInvariants.* CombinedFragments are used to represent special cases, often induced by the satisfaction of a condition. The type of a CombinedFragment is defined by its *InteractionOperator.* Depending on the operator type, the CombinedFragment consists of one or more operands that contain a series of InteractionFragments. The operator types are defined in Table 2.1.

based on the UML Superstructure [24].

Figure 2.5 shows a sample UML Sequence Diagram with messages and CombinedFragments. The behaviour represented by this Interaction is just a fictional one; the aim of the diagram is to show some of the most widely used elements of Sequence Diagrams.

UML 2.0 Testing Profile - An Extension for Testers

As Zhen Ru Dai writes in her publication [6], "*UML itself [...] provides no means to describe a test model.*" Therefore, since March 2004, *UML 2.0 Testing Profile* (U2TP) has been defined and made an OMG standard to support model-driven testing.

The new UML profile introduces several new notations and elements that extend the UML language in form of four groups of concepts. These are:

Test Architecture Concepts A group of objects participating in the testing process can be identified as the *System Under Test* (SUT). Others, communicating with the SUT, can participate in the testing as *test components*. *Test context* allows users to specify a corresponding *test configuration*, i.e. the way components communicate with the SUT. Also, test context can be used to group test cases and to define a *test control* to specify the order in which test cases are executed. An *Arbiter* is a scheme that describes how the overall verdict for a test context is evaluated. A *scheduler* controls the test execution, creates test components when needed, and detects when the execution of a test case is finished.

Test Behaviour Concepts To define *test cases*, the interaction diagrams of the system design model can be used. At the end of a test case, a *verdict* is assigned to show the result. A *validation action* is used to communicate the test verdicts to the arbiter. For unexpected behaviours, *defaults* can be defined. The aim of a test can be set in form of a *test objective*.

Test Data Concepts In U2TP, *datapools* can be defined from where test data can be retrieved for the execution of tests using *data selector* operations. *Wildcards* are useful to define 'any value'.

Time Concepts *Timers* are helpful to control test behaviour and also to ensure that test cases finish running within a certain amount of time. To handle a distributed environment, different *time zones* can be defined so that time events within the same zone can be compared with each other.

The Expressiveness of UML Sequence Diagrams

This profile extended the language with various useful concepts to support model-driven testing using UML; however, U2TP adds very little to Sequence Diagrams, our scenario language under the microscope, and it does not resolve some of the problems that arise from the semantics of the UML language [22]. It is clear, though, that by introducing

some new restrictions on the syntax and using a well-defined semantics, UML Sequence Diagrams can be useful for defining requirement scenarios.

A specificity of the default UML semantics is that it uses *weak sequencing* between messages and fragments. This results in the following issues, hindering the checking of requirements when an execution trace is compared to requirement scenarios [17]:

- Operators have a different meaning than in structured programming languages. This means, for instance, that there is no strict sequencing of events between the iterations of a *loop*. (One lifeline may still be in the first iteration of a *loop*, while another lifeline is already in a later iteration.)
- The OMG specification [24] says that guards should be placed on the lifeline that has the first event in that operand. This means that there is no common point of time to evaluate guards, as there is no causal relation between the guards of the different operands (if not all guards are placed on the same lifeline).
- The spatial and temporal scope of the *conformance operators* (assert, ignore, consider, negate) is unclear. Instances may enter the CombinedFragments separately. Letting lifelines enter a consider fragment separately complicates checking the requirements a lot more than it contributes to the expressiveness of the language.

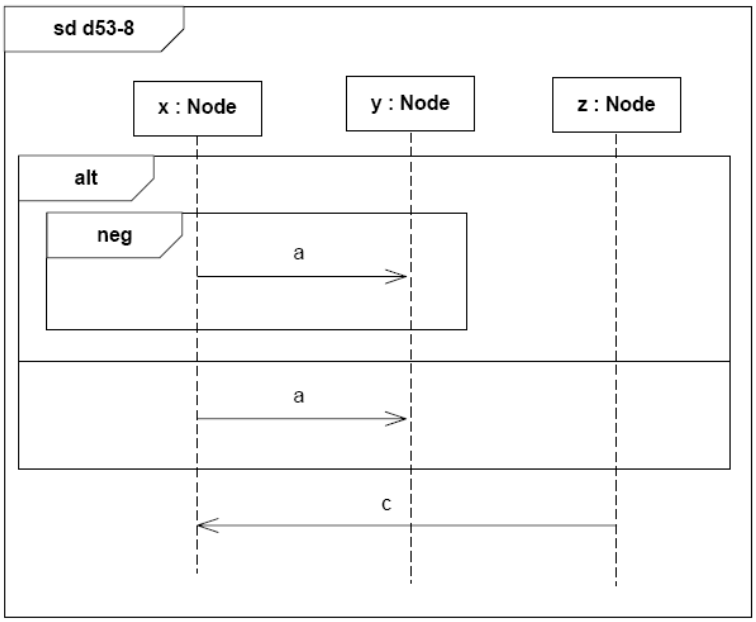
Another problem with UML Sequence Diagrams is that there is no restriction on which parameters can participate in predicates of guards and state invariants. Hence, there is no guarantee that by the time a predicate is to be evaluated, the variables already have been assigned a value. In case of requirement scenario specifications, there is no point in allowing users to define requirements that simply cannot be fulfilled by the SUT.

By default, UML allows us to create non-deterministic diagrams. The best example is when a negative conformance operator is nested in an *alt* fragment in the following way (Figure 2.6). This diagram tells us that the same message can be both valid an invalid, no difference between the two cases whatsoever. Such non-determinism can be handled in high-level specifications which will be later refined, however non-determinisms cannot be allowed in requirement scenario descriptions. When checking a trace, it would be impossible to make a correct decision whether or not it is valid.

UML also provides the user with the opportunity to create diagrams where operators are nested in a way that puts the trace validator in an impossible position. Imagine an *assert* fragment nested in a *neg* fragment, or a *par* fragment with two operands containing exactly the same messages except the contents of one of the operands are also nested in a *neg* fragment.

One of the biggest problems with UML Sequence Diagrams, however, is that there is no means provided to indicate spatial configuration changes, which can definitely take place in a mobile setting. Without this capability, the language is, unfortunately, unable to describe requirement scenarios.

Figure 2.6: *Non-determinism in a UML Sequence Diagram [17]*



2.2.3 TERMOS - A New Graphical Scenario Language

The issues presented in 2.2.2 can be resolved by introducing new restrictions to the syntax of UML 2.0 Sequence diagrams. Therefore, a new scenario language has been proposed, which is partly based on UML Sequence Diagrams but is also provided with the capability to describe spatial relation between the structural elements of the system [17]. This new language, called the *Test Requirement Language for Mobile Setting* (TERMOS), provides two different views enabling the user to define spatial relations as well as system behaviour.

Spatial View consists of different spatial configurations that define the relations between the nodes participating in the Event View.

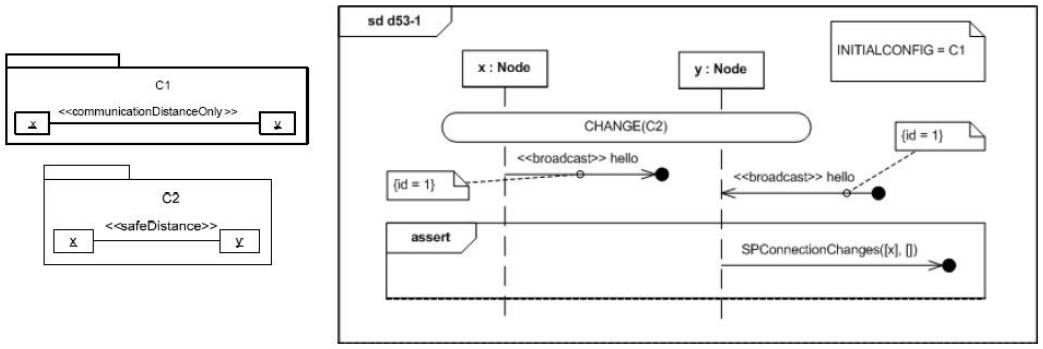
Event View is the scenario description itself, which has the capabilities of UML 2.0 Sequence diagrams with some restrictions in order to resolve the previously mentioned issues. It also provides the user with the opportunity to mark spatial configuration changes happening throughout the scenario.

Figure 2.7 shows an example for an event view and the corresponding spatial view. The spatial view defines two configurations. In the first the nodes are in communication distance, but a message sent in this configuration cannot be guaranteed to be received by the recipient. In the second configuration, the nodes are in a safe distance, where a message sent will most likely arrive at its destination. The initial configuration of the spatial view is C1. After the configuration changes to C2, communication begins, as shown in the event view.

In the following sections, the design decisions, i.e. the proposed modifications to UML Sequence Diagrams based on the issues discussed previously, are presented. Afterwards, the syntax of the TERMOS spatial view is introduced and how the spatial elements are

built in the event view of a scenario description along with other UML elements. Finally, the syntax of TERMOS is summarized.

Figure 2.7: *TERMOS Views [17]*



(a) Spatial view **(b) Event view**

Design decisions for TERMOS

As it has already been mentioned in 2.2.2, the composition of UML Sequence diagrams is based on *weak sequencing* of messages and fragments. This raises several previously mentioned issues; therefore, in TERMOS, another composing operator is used. It is still weak sequencing, only it is subject to a restriction, namely the *synchronization of lifelines on entering and exiting CombinedFragments.*

To resolve the issue with evaluating predicates, a restriction on predicates appearing in *guards* and *StateInvariants* is imposed. In TERMOS scenario descriptions, predicates can only refer to parameters of messages sent or received by any of the lifelines covered by the element containing the predicate before the evaluation of the predicate should take place, or to so-called *node label variables* for any of the involved lifelines in the current or previous spatial configurations. (As it will be shown later, spatial configurations may accommodate node label variables representing node attributes specific to a configuration.)

The problem of non-deterministic constructs in UML Sequence Diagrams (e.g. *alt* fragment with more than one guard true) is resolved by transforming the *alt* operator into an *if-then-else* construct and by introducing the previously mentioned synchronization on entering and exiting *CombinedFragments.*

To avoid ambiguous cases induced by the unlimited nesting of operators, rules exist to define which operators can be nested into other operators and how. Table 2.2. defines these nesting rules [17].

Another rule, not included in Table 2.2., is that *assert* and *consider* operators can only be nested if the containing operator is at the main level of the diagram. (E.g., an *assert* cannot be nested into a *consider* nested into an *assert.*)

There is some new interpretation introduced in TERMOS concerning the default interpretation of the diagrams, too. The reason for this is that, with TERMOS, the diagrams always define requirement scenarios. The requirements are only related to the involved

Table 2.2: *Can the operator in the row be nested in the one in the column?*

	alt	opt	par	assert	consider
alt	Y	Y	Y	Y	Y
opt	Y	Y	Y	Y	Y
par	Y	Y	Y	Y	Y
assert	N	N	N	N	Y
consider	N	N	N	Y	N

lifelines and the presented messages. Execution traces may, however, cover many different scenarios. That is, messages not included in a scenario but interleaving with those that are included are probably found in the execution trace. To be able to compare these traces to requirement scenario descriptions, a so-called *weak interpretation* needs to be used when looking at the diagrams. This means that the scenario description only specifies the communication between the nodes represented; there may be interleaving messages not relevant to the requirement defined by the diagram.

The interpretation of the operator *consider* needs to be reconsidered, too. In UML, the granularity of this operator is the message, not the event. Does a *consider* with a message m in it mean that only the lifelines involved in the considered communication are permitted to send and receive a message of type m and all the other lifelines are disallowed, then? Or do we allow other lifelines to receive a message (e.g. from other nodes not involved in the scenario) unless its sending was forbidden? To give an answer to this issue, the interpretation of *consider[m]* (where m is the message to be considered) in TERMOS is that (i) the *consider* should cover all lifelines involved in the scenario, and (ii) sending of message with type m is forbidden for all lifelines, but to receive a message with type m is allowed if its sending was not forbidden (that is, for example, sent by a node not involved in the scenario).

The Spatial View

The spatial view describes spatial configurations used in requirement scenario descriptions. A configuration can be defined by specifying its name and the nodes involved. Thus, if a configuration contains nodes x, y, and z, the scenario that refers to this configuration must contain lifelines representing these nodes. The previously mentioned *node label variables* also can be defined in the spatial view. These node labels can have three types of values:

- A constant integer value.
- A variable name, i.e. the value is let unspecified, but if more than one node has a node label having a value defined by the same variable name, then the two node labels are considered equal. Also, the value is a symbolic global constant, it has to remain stable in the configuration.
- A wildcard, which means *don't care*. These values do not need to remain stable throughout the whole scenario.

In a configuration, undirected edges represent connections between the nodes involved. That is, when two nodes are connected in the spatial configuration, then there is an edge connecting them in the spatial view. Edges can be labeled by constant values or wildcards to specify the parameters or the quality of a connection (like '*stable*' or '*unstable*').

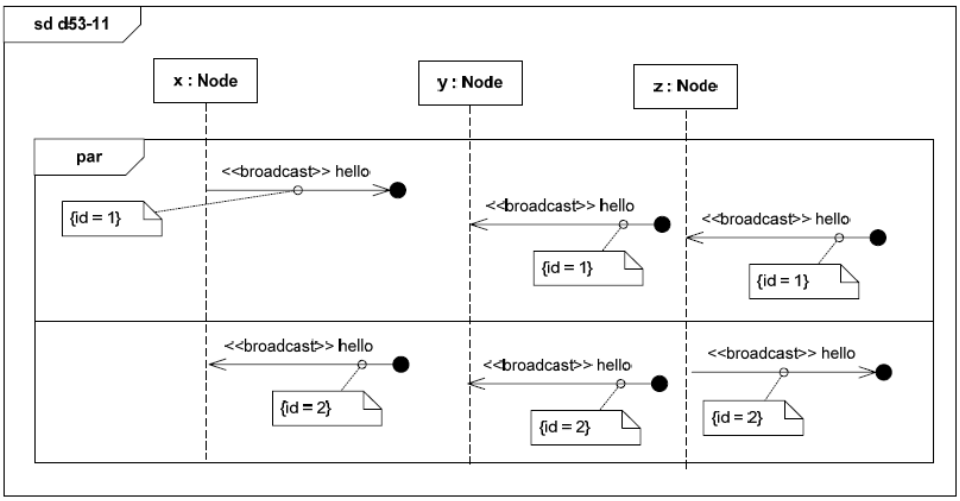
TERMOS spatial views are recommended to be described using UML object diagrams, where packages represent configurations, objects represent nodes, and association instances represent the edges between the nodes [17]. Examples will be shown in *Chapter 5*.

The Event View

The event view of a scenario is described using UML Sequence Diagrams. The scenario description, then, is a UML Interaction. Each scenario must have an *initial configuration* specified. The initial configuration determines the starting spatial configuration of the nodes involved in the scenario. As it is a spatial configuration, it must be defined in the spatial view. Later, the configuration can change, and this change can be denoted using a *configuration change* element. As there is no such element in UML, an existing UML element must be chosen for this purpose.

The structure of scenario descriptions is similar to that of Triggered MSCs, meaning, there is a prefix, which is the trigger part, and it is followed by an action that represents the expected behaviour of the system for the specified prefix. The action part of a TERMOS scenario, however, is denoted with an *assert* fragment. Everything within that fragment belongs to the action part, and every message or fragment preceding it belongs to the prefix of the scenario. Therefore, it is obvious that the last fragment of the Interaction defining the requirement scenario must be an *assert* fragment.

Figure 2.8: *Example of Broadcast Messages [17]*



TERMOS event view is also able to handle *broadcast messages*, as it was proposed, specifically, to support the validation of mobile computing systems, among the specificities of which communication with broadcast messages has been identified. The sending of broadcast messages are denoted by UML *lost message* elements (i.e. messages with type

lost), whereas for defining receive events, UML messages with type *found* are used. To be able to distinguish broadcast messages from traditional lost and found messages that are forbidden to use in TERMOS, a «broadcast» stereotype should be used. As in UML these still mean separate messages, a unique ID needs to be assigned to corresponding lost and found messages using tagged values. An example of broadcast messages appearing in the event view can be seen in Figure 2.8.

Syntax Summarized

Based on the considerations described previously and in [17], the syntax of a TERMOS scenario description can be defined as seen in table 2.3.

Table 2.3: *Changes to the original UML Sequence diagram syntax*

Type of Change	Description of Change
Remove	*"Removed elements: Events, Gate, PartDecomposition, GeneralOrdering, Continuation, ExecutionSpecification."*
Remove	*"The following operators were removed: seq, strict, loop, ignore, neg, break, critical."*
Change	*"Changed the multiplicity for the association going from StateInvariant to Lifeline from 1 to 1..* to allow global predicates. The concrete syntax remains the same, just now StateInvariants can span to multiple Lifelines."*
Constraint	*"Only the following operators can have guards: alt, opt."*
Constraint	*"The following operators have only one operand: opt, assert, consider."*
Constraint	*"The assert and consider operators should cover all Lifelines."*
Constraint	*"There should be an assert fragment at the bottom of the diagram."*
Constraint	*"If a FALSE global predicate is used, it is the only element in the assert, and covers all lifelines."*
Constraint	*"The nesting of conformance operators is only allowed as in Table 2."*
Constraint	*"The configuration change can only be in the main fragment of the diagram or nested in a consider, provided that the consider is at the main fragment of the diagram."*
Constraint	*"The diagram should contain a note with the initial configuration in it."*

Creating TERMOS Scenario Descriptions Using UML Profiles

As there is no TERMOS modeling tool available yet, the best choice for creating TERMOS scenario descriptions is to use a UML modeling tool, and to apply the necessary stereotypes on the UML elements used manually. To be able to do that, a UML profile needs to be defined. For creating scenario descriptions, a *TERMOS Profile* has been defined as part of this diploma thesis project, which contains all the stereotypes and tagged values needed to define TERMOS configurations and TERMOS scenarios. The profile will be presented

in the next chapter.

Also, as previously mentioned, UML has no elements for expressing *configuration changes*, therefore, as *Continuations* are forbidden in TERMOS anyway, I decided to make use of them by employing them to represent configuration changes in TERMOS scenario descriptions. The configuration change is distinguished from traditional UML *Continuations* based on their names. If a *Continuation* has a name like: *CHANGE(C)*, where C is a valid configuration defined in the spatial view, then it denotes a configuration change; otherwise, the element is a *Continuation* and it is to be considered a syntax violation.

2.3 Summary

By examining how existing graphical scenario languages can support the validation process of traditional distributed systems, and by defining the expectations such languages must meet to be able to support the validation of mobile distributed systems, a new language has been defined, specifically to satisfy these expectations [17]. Moreover, this new language has not been built from scratch but rather using as many existing concepts from other languages as possible.

TERMOS is able to handle those issues arising from the specificities of a mobile computing system that other, already existing and widely used, scenario languages cannot. By implementing a tool that is able to process requirement scenario descriptions and check execution traces against them, a significant part of the validation framework described in Section 1.3 can be realized.

Chapter 3

Development of a TERMOS Tool

The state-of-the art testing of mobile distributed systems, described in Section 1.3, can be invaluably supported by using graphical scenario languages. In Section 1.3.1, a testing framework (Figure 1.2) has been introduced that relies largely on models and model-transformations. As the figure shows there are numerous activities to be performed in a testing process based on this framework:

- define requirement scenarios,
- define test purposes,
- create test cases based on the requirement scenarios and the test purposes to be covered,
- execute test cases and record execution traces,
- validate traces by checking it for violations, using the requirement scenario descriptions.

To define requirement scenarios, a scenario language (preferably a graphical one) has to be chosen. For mobile distributed systems, based on the short analysis of languages performed in Chapter 3, TERMOS seems like a good choice.

The execution traces can easily be recorded in a text file. The granularity of the traces should be chosen so that it matches the granularity of the requirement scenario descriptions. For the sake of automatic analysis of traces, they should be *machine readable*, i.e. easily processable by a software, and also *human readable*, so that they can be double-checked by testers. A format for producing execution traces is introduced later in this chapter.

The definition of test cases can take place based on MDA, which supports the automation of their implementation using test models. MDA does not prescribe a specific modeling language, so it is the test designers' decision which one they are comfortable about or familiar with.

Producing execution traces can be facilitated using a logging framework (e.g. *log4j* [9]). A pre-requisite of being capable of recording communication events and building a single trace file is to have *synchronized clocks* in the system, so that the events appear in the

exact order of their occurrence. If a network simulator is used in the testing platform, the *global clock*, i.e. the clock of the simulator can be used for synchronization.

The validation of execution traces can take place either manually or automatically. Naturally, manual trace validation takes time and it does not eliminate the human factor. In this chapter, the development of a tool is presented, which performs the automatic trace validation task using the algorithms collected and presented in [17]. These algorithms will be discussed in the following sections.

3.1 The TERMOS Tool

The TERMOS Tool, designed and implemented as part of this diploma thesis project, fits in the previously presented testing framework by performing the validation of execution traces against requirement scenario descriptions defined using the TERMOS graphical scenario language.

Since there is no TERMOS editor available yet, a UML editor can be used to define requirement scenarios. As it is recommended in [17], UML Sequence Diagrams can be used to create the event view, whereas the spatial view can be created using UML Object Diagrams. For the tool to be able to identify TERMOS scenarios and configurations, a UML Profile needs to be applied when creating them. This Profile will be introduced later on in this chapter.

The TERMOS Tool contributes to the work of the testing framework by performing the following tasks:

- checking scenario descriptions for syntax violations,
- checking the semantical soundness of scenario descriptions,
- validating execution traces using the syntactically and semantically correct requirement scenario descriptions.

The first task is fairly easy. The requirement scenario descriptions defined in TERMOS are analyzed and validated against the rules presented in Section 2.2.3. The second task can be preformed simultaneously; the messages defined in the scenario description are checked to see whether or not they can be sent and received in the specified configuration.

The execution trace validation is a little more complicated. The task, here, is to compare some sort of program output and a diagram, checking if the requirements are satisfied by the recorded behaviour of the system. First of all, the scenario descriptions are rather general. They do not necessarily contain every node that participates in the system. Scenarios might only define requirements for the communication of two nodes while, in reality, there may be tens or hundreds of them. During validation, these scenarios have to be identified when processing the trace, and the communicating nodes have to be matched to those symbolic ones present in the scenario description. This is a graph matching problem and is outside the scope of this thesis. A solution to this problem is offered in [17]. To simplify the trace validation task, from now on, we assume that traces analyzed by the TERMOS Tool have

already undergone a pre-processing phase where the nodes participating in the examined scenario were identified. The only thing left for the tool is the validation itself.

The realization for this task relies on a solution presented in [17], too. The basic idea is looking at the scenario description as a *language* and the valid execution traces as the elements of the language, made up of trace records, i.e. the alphabet of the language. A decision whether an execution trace is an element of the language can, then, be made using a *finite state automaton* (FSA) built based on the scenario description. Execution trace validation, therefore, includes two phases: creating an FSA and checking if the input (i.e. the trace) brings it to an accepting state.

The method of building the automaton is based on the one presented in [20] by Klose. The diagram is first subject to a pre-processing phase, then the output of this phase is used to generate the automaton using the so-called *unwinding algorithm* [17, 20].

When the automaton is built, the job of the tool is to read the execution trace record by record. Records enable the transitions of the FSA, taking the automaton from one state to the next. When all the trace records have been processed, the tool checks the state the automaton stopped in and concludes the result of the validation process accordingly.

In the following sections, the phases of the automaton creation, pre-processing of the diagram and the unwinding algorithm, are discussed in details.

3.1.1 Pre-processing TERMOS Diagrams

The first step of the preprocessing, is to process all the *atomic* events on each lifeline in the event view. Such atomic events can be:

- Message sending and receiving events
- State invariants
- Guards
- Configuration changes
- Entering or exiting operators
- Lifeline heads
- Lifeline ends

These atomic events are packaged into *atoms*. Every atom is assigned a location that is unique on the lifeline it belongs to. (Therefore, configuration change elements, for example, have as many different atomic locations assigned as many lifelines they cover, since such elements belong to all the lifelines in the event view.) The location *0* is always assigned to the lifeline head. The locations can be determined as the example shows in Figure 3.1 [17].

After determining the atoms and their locations, *clusters* are formed. Clusters are groups of atoms that belong together on a lifeline. These are typically made up of a guard and the event that is it precedes. The location of a cluster is the one assigned to the atom with the

Figure 3.1: *Assigning Locations to Atoms of a Requirement Scenario*

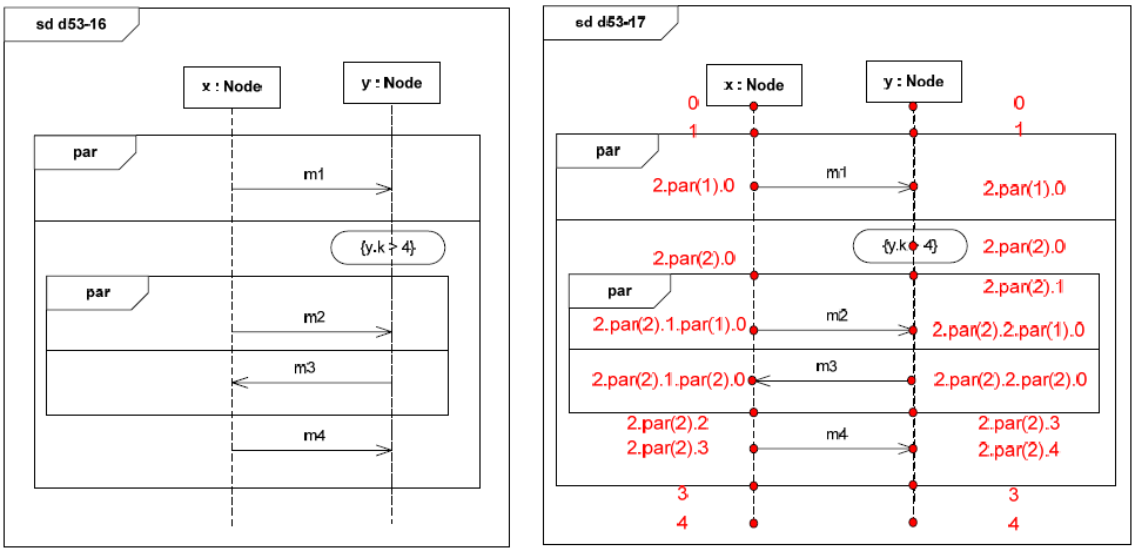
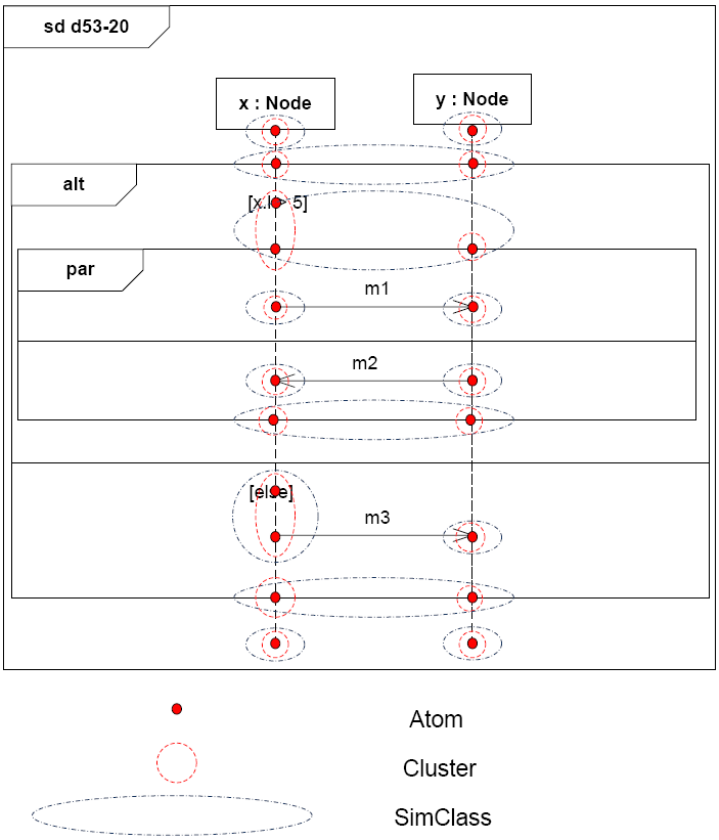


Figure 3.2: *A Pre-processed TERMOS Diagram*



smallest location within the cluster. So, for example, if two atoms, with locations i and $(i+1)$ respectively, form a cluster, the location of the cluster will be $min(i, (i+1))=i$.

Clusters, then, are grouped to form *SimClasses*, that is, simultaneous classes. SimClasses contain clusters that represent events that need to take place simultaneously; e.g. configuration changes take place at the same time on each lifeline, and there is also a synchronization on entering and exiting operators and, therefore, the clusters on the different lifelines rep-

resenting these events belong in the same SimClasses. An example of fully pre-processed TERMOS diagrams can be seen on Figure 3.2 [17].

To be able to build an automaton using the building blocks of a pre-processed diagram, a few relations need to be defined between them [17]. Two important relations between clusters are *local causality* and *local conflict*:

Local causality: If two clusters on the same Lifeline, *c1* and *c2*, are in the form: *location(c1) = p1.i.p2* and *location(c2) = p1.j.p3*, where *p1*, *p2*, *p3* can be empty strings, then

$$c1 \prec c2 \iff i < j \text{ (i.e., } c1 \text{ precedes } c2)$$

Local conflict: If two clusters on the same Lifeline, *c1* and *c2*, are in the form: *location(c1) = p1.i.p2* and *location(c2) = p1.j.p3*, then

$$c1 \# c2 \iff i \neq j \text{ (i.e., } c1 \text{ and } c2 \text{ are conflicting events)}$$

Predecessors of a cluster: With the help of these relations a *predecessor* function can be defined which can be used to identify the immediate predecessor(s) of a cluster *cl* on Lifeline *l*:

$$predecessors(cl) := \{cl' \in Clusters(l) \mid cl' \prec cl \wedge \neg \exists cl'' \in Clusters(l) : cl' \prec cl'' \prec cl\}$$

On the level of SimClasses, another auxiliary function is used, since on the global level, when ordering message related events, we need to make sure that a causality relation is only identified between events belonging to a common message. We cannot say anything, directly, about message events of different Lifelines that belong to different messages too:

messageID: *MessageSends(sd)* $\cup$ *MessageReceive(sd)* $\rightarrow$ *ID* function returns the message ID for each pair consisting of a message send and a message receive event, where *MessageSends(sd)* and *MessageReceives(sd)* contain all the message send and receive events of the diagram *sd*. [17]

SimClass Prerequisites: Using the *messageID* function, the *predecessors* function can be extended to SimClasses. Let *SimClasses(sd)* contain all the SimClasses of diagram *sd*. The immediate predecessors of SimClass *scl* in diagram *sd* are given by the *prerequisite* function:

$$Prerequisite(scl) := \{scl' \in SimClasses(sd) \mid \exists cl \in scl, \exists cl' \in scl' : cl' \in predecessors(cl) \vee (\exists a \in cl \cap MessageReceive(sd), \exists a' \in cl' \cap MessageSends(sd) : messageID(a) = messageID(a'))\}$$

Conflicting SimClasses: Similarly to the *predecessors* function, the conflict relation can be extended to SimClasses, too [17]. We say that SimClass $scl \in conflict(scl')$, where $scl, scl' \in SimClasses(sd)$, if and only if these SimClasses contain a *cl* and a *cl'* respectively so that:

- $location(cl) = prefix.k.alt(i).suffix$ on Lifeline *l*
- $location(cl') = prefix'.k'.alt(j).suffix'$ on Lifeline *l'*
- the *alt*s participating in the locations of *cl* and *cl'* coincide.

3.1.2 The Unwinding Algorithm

After the pre-processing of the requirement scenario description is finished, the *unwinding* of SimClasses may begin. The original version of the algorithm is presented in [20], while its modified version, the one used in TERMOS Tool, is discussed in details in [17]. The unwinding algorithm is presented in this section just in order to introduce the main concepts that are used to generate an automaton based on requirement scenario descriptions in TERMOS Tool.

The automaton can be given with the tuple $(\Sigma, Q, q_0, F_t, F_s, \rightarrow, Var, Def)$, where:

- Σ is the set of transition labels with predicates using the variables of Var,
- Q is the set of states,
- q_0 is the initial state of the automaton,
- $F_t \subseteq Q$ and $F_s \subseteq Q$ are disjoint subsets of accept states. F_t contains states that express trivial satisfaction of the requirements (i.e. the trigger before the Assert did not match), while F_s contains states that express stringent satisfactions (i.e. the trigger and the contents of the Assert both matched).
- $\rightarrow \subseteq Q \times \Sigma \times Q$ is the set of transitions.
- Var is the set of variables extracted from the TERMOS scenario. It contains the variables that appear in the spatial view as well as the ones in the event view.
- $Def \subseteq Q \times Var$ gives the subset of variables defined for each state.

The algorithm uses the notion of *phase*, a tuple $(Ready, History, Cut, Variables)$, where:

- $History$ is the set of SimClasses that have been unwound,
- $Ready$ is the set of SimClasses ready to be unwound,
- Cut represents a borderline between already unwound elements and the currently enabled ones. It is a tuple of clusters $(c_1, \ldots, c_n)$ where c_i is a cluster on Lifeline i.
- $Variables$ is the set of variables that are already valuated.

Each phase corresponds to an automaton state. A $STATE(ph : Phase)$ function can be used for assigning a unique state name for a phase. If, during the execution of the unwinding algorithm, a phase recurs several times, the function is expected to assign the same state to it that once has already been assigned.

The algorithm starts with an initial phase, where the only unwound SimClass is the one containing the Lifeline heads. The SimClasses ready to be unwound are the ones that have virtually no prerequisites (i.e. the prerequisite is the very beginning of the scenario). The cut in the initial state is the set of clusters containing the atoms that represent Lifeline

heads. The set of variables only contain the ones defined in the initial spatial configuration and the variables that denote the nodes participating in the scenario.

Therefore:

- $History_0 = \{\{\{\perp_1\}\}\}, \{\{\{\perp_2\}\}\}, \ldots, \{\{\{\perp_n\}\}\}$ (where $\perp_i$ denotes a Lifeline head)
- $Ready_0 = \{scl \in Simclasses(sd) \mid Prerequisite(scl) \subseteq History_0\}$
- $Cut_0 = \{\perp_1\}, \ldots, \{\perp_2\}, \{\perp_n\}$
- $Variables_0 =$ the variables describe above.

The phases, then, are recursively calculated based on their preceding phases. For the calculation the function $STEP(ph : Phase, scl : Simclass)$ is used. That is, given a phase $ph = (History_i, Ready_i, Cut_i, Variables_i)$, the $STEP$ function returns the next phase by processing one of the ready SimClasses, scl, of the diagram sd.

$STEP(ph, scl)$ returns $ph' = (History_{i+1}, Ready_{i+1}, Cut_{i+1}, Variables_{i+1})$, where:

- $History_{i+1} = Hisotry_i \cup scl \cup conflict(scl)$ as events represented by the conflicting SimClasses cannot occur in the execution trace.
- $Ready_{i+1} = \{scl' \in Simclasses(sd) \mid \forall scl'' \in Prerequisite(scl') : scl'' \in History_{i+1}\}$
- $Cut_{i+1} = (c'_1, c'_2, \ldots, c'_n)$ where c'_i is the same cluster as c_i from Cut_i or is replaced by a cluster of the unwound SimClass, which cluster is immediately preceded by c_i on Lifeline i.
- $Variables_{i+1}$ is the union of $Variables_i$ and the set of variables valuated in the new phase. (Variables are only valuated in case of a configuration change or a communication event.)

After ph' and the corresponding state is calculated, the transition incoming to this state from the one corresponding to ph are to be created and added to the set of transitions, and the self-loop belonging to the previous state is to be refreshed. Most of the defining features of the transitions are available after the execution of the $STEP$ function, like $from$ and to states, but the labels of the transitions are yet to be determined at this point. Labels can contain one or more elements. The labels of transitions connecting two different states typically contain a condition describing a communication event or a configuration change event and may or may not contain a predicate of a guard or constraint. The format of a label element describing one of these events is one the followings:

- $(!msgName(parameterNames), senderNode, recipientNode, msgId)$, where ! denotes that it is a send event, and $msgId$ is the variable with the value of a unique ID belonging to the message sent by the sender node having a format: e.g *$1.*
- $(?msgName(parameterNames), senderNode, recipientNode, msgId)$, where ? denotes that it is a receive event.

- $CHANGE(configId)$ denotes configuration change event where $configId$ is the name of the new configuration.

If the label has multiple elements, that is one of the events described above and a guard of an operand, then the two elements are conjoined using an ' ^ ' sign, denoting a logical AND relationship between the elements. Self-loop labels typically contain the negated form of the label elements appearing on the outgoing transitions. If the pre-condition of firing a transition is not a configuration change event, the self-loop label also contains the label element *~CHANGE(-)* (where ' ~ ' represents the negation operator) that restricts the SUT from undergoing an unexpected configuration change.

The labels are determined or updated when unwinding the atoms of the clusters of a SimClass. The labels in case of atoms representing communication events and configuration changes have been introduced, but there are other events in a TERMOS diagram. The labels in this case are determined as follows:

- On entering and exiting a fragment (*opt, alt, par, consider, assert*), the label contains a single *true* element. In case of these events, no self-loop label is needed.
- Within a consider fragment a label element restricting the sending of the messages considered is added to the self-loop labels.
- In case of a state invariant, the predicate in the StateInvariant element is in the label of the transition, and its negated form is added to the self-loop label.

One more thing needs to be determined for the state created (or updated) within a cycle of the unwinding algorithm. This is the mode of the state, i.e. accepting or rejecting. States preceding the event of entering the Assert fragment, i.e. the states representing the trigger part of the scenario, are all trivial accepting states. When entering the Assert, the states become rejecting states. The stringent satisfaction of the requirements takes place on exiting the Assert fragment, so the state that can be reached through the transition representing this event is the non-trivial accepting state.

Once the unwinding algorithm is finished, the automaton can be built from the states and transitions created by the algorithm. The elements of the automaton are, then:

- Σ: the set of transition labels determined in the cycles of the algorithm,
- Q: the set of states created by the $STATE$ function after a phase is calculated by the $STEP$ function,
- q_0: the initial state that belongs to the initial phase,
- F_t: the states preceding the transition representing the event of entering the Assert,
- F_s: the state following the transition representing the event of exiting the Assert,
- $\rightarrow$: the set of transitions determined as shown previously,

- Var: the set of variables created by gradually expanding the initial set of variables (containing only the ones defined in the description of the initial configuration) by the $STEP$ function,
- Def: the $Variables_i$ sets assigned to the state belonging to $Phase_i$.

3.1.3 Validation of Execution Traces

Validating an execution trace is ready to be started once the automaton is ready. The text file containing the trace needs to be processed record by record, and, simultaneously with the processing of a record, one of the transitions starting from the current state of the automaton has to be selected and fired. Since TERMOS was designed to avoid non-determinisms, there should be only one enabled transition in each state of the automaton. Then, the next state is to be stored for the next cycle of the trace processing. Obviously, at the beginning of the validation, the automaton is in the initial state. When the validator runs out of trace records, the result of the validation is determined based on the current state of the automaton. If the automaton is in a rejecting state, the validation failed; if the state is an accepting state, the validation succeeded. As it was previously shown, there are trivial accepting states and stringent accepting states. The validation result should express which of these two kinds of successful validation results has been achieved.

3.2 The Environment of the TERMOS Tool

It would be an obvious decision to implement the tool, from now on referred to as *TERMOS Tool*, as a standalone application. However, there are several solutions that provide APIs that help the development.

At the time of writing this diploma thesis, there is no available application for creating TERMOS diagrams. In [17], using a UML editor for this purpose is recommended, since the spatial view can be described using UML Object diagrams, while the event view can be described using UML Sequence diagrams. TERMOS Tool, therefore, will work with UML Models containing UML Interactions and UML Packages. For the tool to be able to distinguish TERMOS requirement scenario descriptions from regular UML Interactions, a UML Profile needs to be applied on them which will be presented in 3.3.1.

Since the tool will have to handle UML elements, an API to a UML implementation would be beneficial for the development. In Section 3.2.1, a development environment is introduced that provides an API and a runtime environment for plug-in creation. Then, in Section 3.2.2, an existing plug-in, created for this environment specifically, is introduced, which is a Java based implementation of UML providing an API for developers to be able to work with the elements of UML. TERMOS Tool will operate in this environment as a plug-in, using the services provided by the above mentioned extension.

3.2.1 The Eclipse IDE and the Plugin Development Environment

The environment TERMOS Tool will operate in is the *Eclipse Integrated Development Environment* (Eclipse IDE). The IDE is the product of the *Eclipse Platform* project of the *Eclipse Foundation* [12]. "*Eclipse is a kind of universal tool platform - an open extensible IDE for anything and yet nothing in particular. The real value comes from tool plug-ins that "teach" Eclipse how to work with things - java files, web content, graphics, video - almost anything one can imagine.*" With the help of Eclipse, independent tools can be developed and integrated with other tools developed by others [11].

Extending Eclipse is facilitated by *Eclipse Plug-in Development Environment* (Eclipse PDE) [15], another project of the Eclipse Foundation. It enables users to develop, test, build, and deploy Eclipse plug-ins.

Developing a plug-in for Eclipse is aided in the PDE with an API, a *Plug-in project* wizard, and several project templates. Using the *Eclipse Rich Client Platform* (RCP) [16], which is a platform for building so-called *rich client applications*, Eclipse plug-ins can extend the IDE with new *Views* and *Perspectives*, graphical elements that are parts of the graphical user interface (GUI) of Eclipse.

TERMOS Tool will be created as an Eclipse plug-in written in Java, extending some of the local pop-up menus of Eclipse and adding a *Console* view to the workbench to display informational messages.

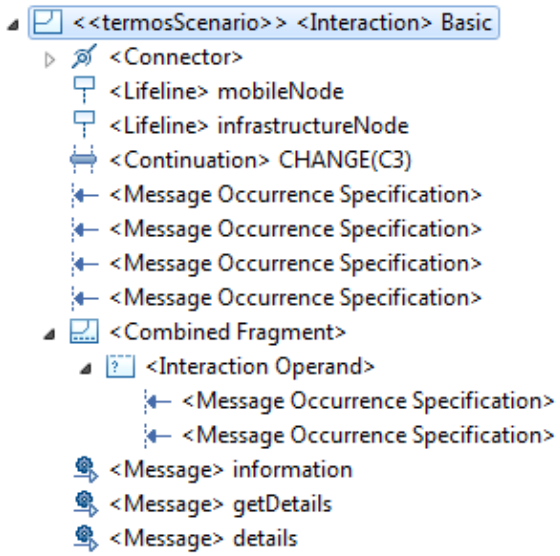
3.2.2 UML2 Tools for Eclipse

UML2 and *UML2 Tools* are parts of the *Eclipse Model Development Tools* project [14]. UML2 is an implementation of the Unified Modeling Language 2.x metamodel based on the *Eclipse Modeling Framework* [13], the modeling and code-generating framework of Eclipse. UML2 Tools is *"a set of GMF-based (i.e., using Eclipse Graphical Modeling Framework) editors for viewing and editing UML models."* [10] Unfortunately, at the time of writing this thesis, UML2 Tools has only limited abilities which are not enough to enable users to describe TERMOS requirement scenarios, but according to the project website, new features will soon be added. Hopefully, these new features will enable tools to use it for creating TERMOS diagrams too. Currently, the main disadvantage of UML2 Tools is that stereotypes cannot be applied to elements of a Sequence Diagram.

UML2 makes Eclipse capable of handling **.uml* files that contain UML models, and provides our TERMOS Tool an easy-to-use API to access the elements these models contain. Although the UML2 Tools has limited capabilities in supporting the creation of UML diagrams, it supports the editing of UML models as a tree structure. The main advantage of this tool over other UML editors is that it enables designers to fully access the set of UML 2.0 elements. Most editors do not allow elements like Continuations, Messages of type *lost* or *found* and StateInvariants, i.e. elements that are vital parts of TERMOS diagrams. This advantage of UML2 Tools and the API it provides for accessing UML model elements played a significant role in choosing Eclipse as the operational environment of the TERMOS Tool.

A part of a **.uml* file, opened in Eclipse, can be seen in Figure 3.3. The figure shows a UML Interaction created using UML Sequence Diagrams.

Figure 3.3: *A UML Interaction in Eclipse*



3.3 Developing TERMOS Tool

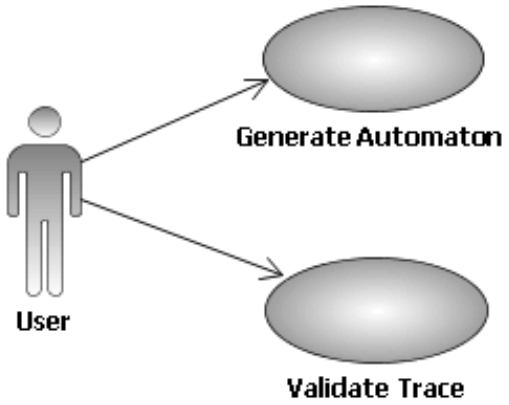
As previously mentioned, the operational environment of TERMOS Tool is the Eclipse IDE. The tool is a plug-in that enables users to validate execution traces against requirement scenarios defined in TERMOS. The scenario descriptions are defined using a UML editor, and they can be imported into the Eclipse environment, thanks to the UML and UML2 Tools plug-ins.

In the design phase of the development of the tool, the use cases were to be determined. The functions identified in Section 3.1 form two different groups of tasks. The first group of tasks include the processing and analyzing of requirement scenarios, that is (i) checking their syntax and their semantical soundness and (ii) building automatons. The input of the execution of these tasks is, therefore, a requirement scenario description, while the output is an automaton. The second group of tasks includes the processing and the validation of execution traces. Thus, the inputs in this case are an execution trace to be validated and the automaton that is used to perform the validation, while the output is the desired validation result. Based on the expected outputs of these two groups of tasks, the use cases of the tool are shown in Figure 3.4.

The user of the tool can generate an automaton and validate execution traces. These are the two separate actions a user can invoke. Thanks to the separation of these two phases, the automatons need not be generated every time the user wants to validate an execution trace, only when the requirements are changed.

As the inputs of the system, in both phases, come from outside of the tool, a well-defined

Figure 3.4: *Use Cases of the TERMOS Tool*



format needs to be specified both for the scenario descriptions and for the execution traces. It has already been mentioned that scenarios and the configurations are defined using UML Sequence Diagrams and Object Diagrams respectively but with special attention payed to the syntax and semantics of TERMOS. In the following smaller sections some further constraints and expectations toward the inputs of the tool are discussed: a UML Profile (Section 3.3.1) to be applied on scenario descriptions and the expected format of the execution traces (Section 3.3.2) are presented. Then, in Section 3.3.3, the implementation of the tool is discussed.

3.3.1 A UML Profile Recommended for Creating TERMOS Diagrams

In [17], a few stereotypes were recommended to be used when creating TERMOS diagrams. In the design phase of TERMOS Tool, when the example TERMOS diagrams were created, I decided to extend the idea to more elements of requirement scenario descriptions as stereotypes were found useful in many cases.

TERMOS configurations can, indeed, be defined using UML Object Diagrams, as previously mentioned, but the most important thing about them is that they are represented in the UML Model by *Packages* that have the *termosConfiguration* stereotype. The nodes participating in a configuration are UML *InstanceSpecifications* using the *termosNode* stereotype, while the connections are represented by instances of UML *Associations* given in the structural model of the SUT and they use the *termosConnection* stereotype which has a required property of type enumeration with the values {*communicationDistanceOnly, safeDistance*} to specify the quality of the connection between the nodes.

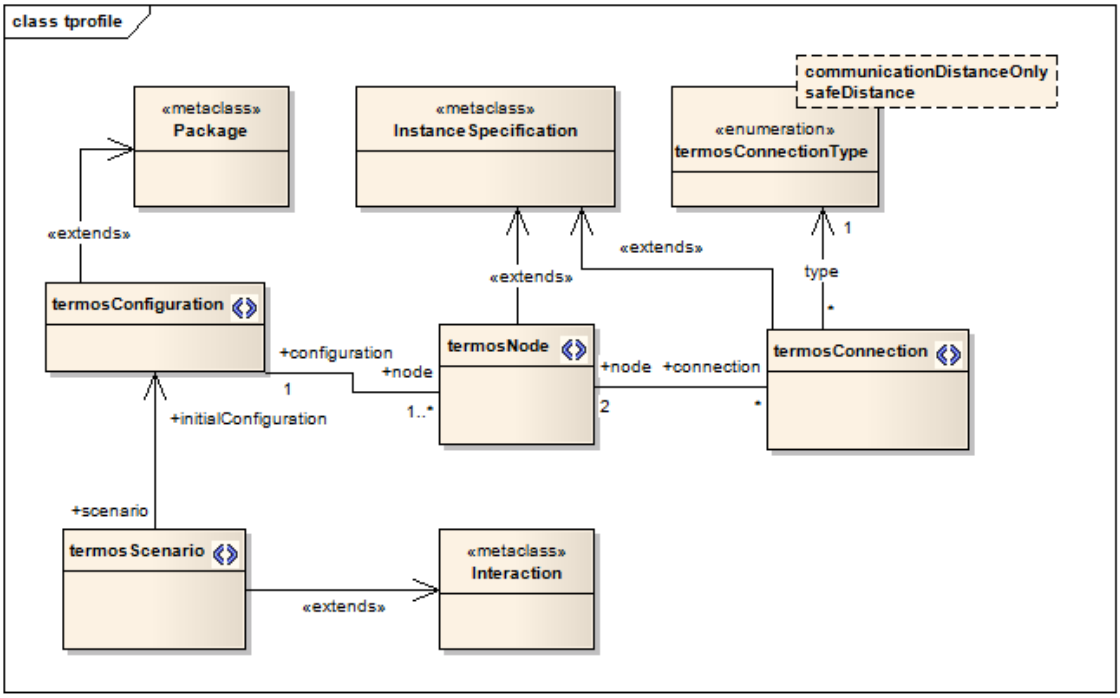
Distinguishing TERMOS configurations from other UML Packages in the UML Model enables designers to use Packages for more traditional purposes, as well; like packaging classes into a logical unit. The *termosConnection* stereotype simply provides the opportunity to explicitly express the function of the link between two nodes and to facilitate the description of connection properties on a meta level.

As recommended in [17], UML *Interactions* that describe requirement scenarios have the *termosScenario* stereotype with the *initialConfiguration* property, which is required and

specifies the initial configuration in the scenario.

The TERMOS Tool developed as part of this diploma thesis project, in order to be able to process them, requires UML Models that contain TERMOS configurations and requirement scenario descriptions to have a profile applied containing these stereotypes and stereotype properties. Figure 3.5 shows the structure of the UML Profile to be used for defining TERMOS requirement scenarios.

Figure 3.5: *The TERMOS UML Profile Elements*



3.3.2 The Recommended Format of Execution Traces

In [17] a format for execution traces has been recommended. In this format, a trace record describing the event of a node sending a message is denoted with the following expression:

(!messageName([list of parameter values]), senderNodeName, messageID)

In this expression the exclamation mark (*!*) means that this is a message send event. In case of a message receive event an expression very similar to this previous one is used:

(?messageName([list of parameter values]), recipientNodeName, messageID)

The question mark (?) is used to express that this is a message receive event. With the help of these two expression kinds, message events can be identified easily in *most* situations. Another expression can be used to designate configuration change events:

(CHANGE(newConfigId)

In the expression *newConfigId* is to be substituted with the name of the new configuration (e.g. *C2*).

Unfortunately, using this format is not always satisfactory if we want to keep the trace processing algorithm simple. Imagine that a node sends the same message to two different nodes, and the order of the message sends is not pre-defined; the requirement specifies that these two messages should be sent simultaneously, e.g. using a *par* fragment. The messages have different IDs of course, but both of the message send events have the same format. For example, the labels of the transitions that are enabled in the imaginary situation are:

(!m(val1), sender, $1)
(!m(val2), sender, $2)

While the trace record to be processed looks like this:

(!m(2), 192.168.1.2, 12)

Which one of the two transitions should be fired? *$1* and *$2* are symbolic IDs, i.e. variables valuated when validating a trace. The value *12* on its own does not specify which of the two possible and acceptable events it represents. Note that it does matter which one of the transitions the algorithm chooses to fire, because the recipient of message with the ID *$1* is not the same as of the one with the ID *$2*. If we use this representation of events in the traces, and on the labels of the transitions, we need to look (maybe far) into the future in the trace to find out who the recipient is and, therefore, which transition should be fired. In order to simplify the trace processing and trace validation algorithm, I recommend (and, in case of TERMOS Tool, decided) to use a format containing information regarding both the sender and the recipient of the message in a trace record denoting a message event. The modified format for denoting message events are, then:

(?messageName(parameter values), senderNode, recipientNode, messageID)
(!messageName(parameter values), senderNode, recipientNode, messageID)

This way, the above mentioned problem can be easily resolved, as the labels on the transitions would look something like this:

(!m(val1), sender, recipient1, $1)
(!m(val2), sender, recipient2, $2)

The example trace record, then, is shown below. As the graph matching took place before the validation phase, the validator knows exactly which recipient the address value denotes, therefore, it means no problem to choose the one and only enabled transition to be fired.

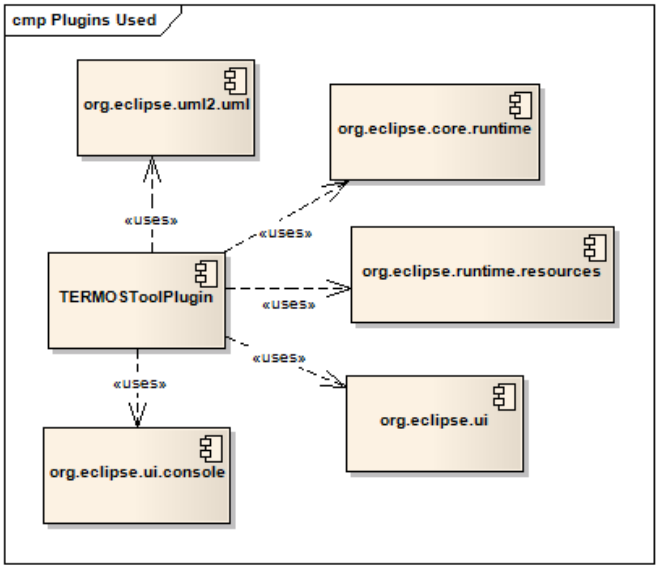
(!m(2), 192.168.1.2, 192.168.2.13, 12)

As during the development of the tool it was assumed that the graph matching is performed before the trace arrives on the input of the TERMOS Tool, the tool expects a trace to contain the meta-information for the identification of the participating nodes in the head of the trace file. The TERMOS Tool also expects the trace files to end at the end of the scenario. The result of the validation shall not be compromised by trace records denoting events not represented in the scenario, but the tool will not carry out any kind of validation regarding these events. The result of the validation shall not be compromised by any kind of invalid trace record either.

3.3.3 The TERMOS Tool as an Eclipse Plug-in

Since the tool has been developed to be an Eclipse plug-in, it shows some significant differences compared to standalone applications. The main differences, the great advantages of being able to use the services of already existing tools (Eclipse UML, Eclipse RCP) has already been mentioned. As the tool extends the workspace of the Eclipse IDE, it has necessary points of connection to its environment. These *points of extension* are defined, among other important characteristics like the plug-in dependencies, etc., in an XML file called *plugin.xml*. TERMOS Tool extends the workspace by adding a submenu containing two actions to its local pop-up menus. These are the previously identified actions, namely, *Generate Automaton* and *Validate Trace* in the *TERMOS Tool* submenu.

Figure 3.6: *Plug-in Dependencies of the TERMOS Tool*



Extending the workspace can be achieved by adding extension points to the plug-in. The actions are '*attached*' to these points and, thereby, they can be invoked through these points. The services of the TERMOS Tool can be invoked through clicking on the above mentioned elements of the pop-up menu of Eclipse, created by extending the user interface using the *org.eclipse.ui.popupMenus* extension point. When extending the user interface, the contribution of the plug-in has to be specified, too. This can be a *viewer contribution* or

an *object contribution*. The first one is used to contribute to the pop-up menu of a specific view, while the second one will cause the menu item to appear in the pop-up menu when the specified object type is selected. TERMOS Tool contributes to pop-up menus related to specific objects, namely: UML Interactions (requirement scenario descriptions) and files having the extension *.aut* (automatons generated by the tool).

As already mentioned, plugin.xml also contains very important information about what other plug-ins are used by the '*owner*' plug-in of the file. TERMOS Tool uses five plug-ins. The plug-in dependencies of the tool are shown in Figure 3.6.

org.eclipse.core.runtime supports the runtime environment, the environment the plug-in has to operate in,

org.eclipse.core.resources provides services for managing the workspace and its resources,

org.eclipse.ui is essential to extend the user interface of Eclipse,

org.eclipse.uml2.uml provides an API to the UML implementation of Eclipse, and

org.eclipse.ui.console is needed to be able to display information for the user in the *Console* view of Eclipse.

The classes of the plug-in can be divided into three different groups. These groups are shown in Figure 3.7. The classes of the package *termostoolplugin.impl* are discussed in Section 3.3.4. The first group contains only one class, which is the *Activator* class. This is the main class of the plug-in. It has a *start* method, which is called upon the activation of the plug-in, and a *stop* method, which is called when the plug-in is stopped. The second group contains the *action* classes. These are the ones that implement the behaviour associated with the action. Of course, two classes, at least in our case, are not enough to implement the whole logic of the TERMOS Tool, so we need the third group, the *implementation classes* that actually contain the business logic of the tool and are called from within the *run* methods of the action classes. The runtime environment calling the *run* method is the point from which the control is fully given to the plug-in to carry out its expected behaviour. In case of the automaton generation action, for example, the run method of the action class invokes the run method of the single instance of the implementation class called *TermosAutomatonGenerationImpl*, passing it the TERMOS diagram to be processed. Then, the syntax of the diagram is validated. If the diagram is a valid TERMOS requirement scenario, then an automaton is generated. Both of these actions (syntax validation and automaton generation) are supported by the implementation classes presented in Section 3.3.4. The high-level view of this process can be seen on Figure 3.8.

In case of our TERMOS Tool, the implementation classes form the largest group with no less than 20 members. In Section 3.3.4, these classes, along with the design and implementation of the tool, are introduced.

Figure 3.7: *Packages of the TERMOS Tool Plug-in*

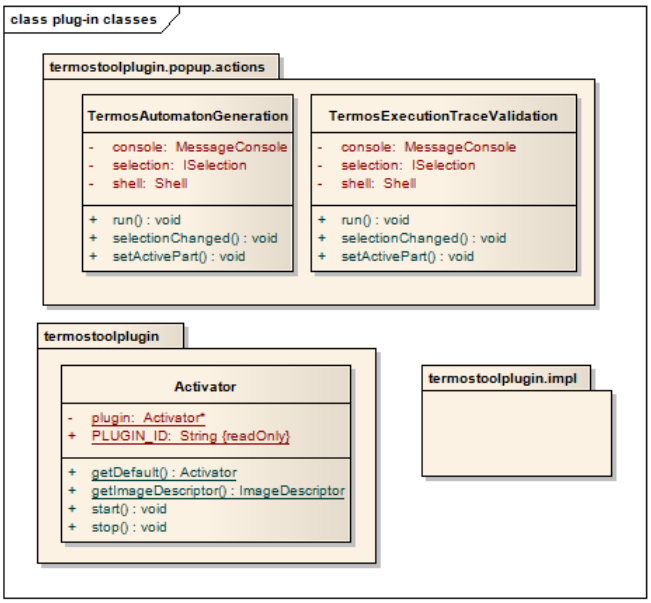
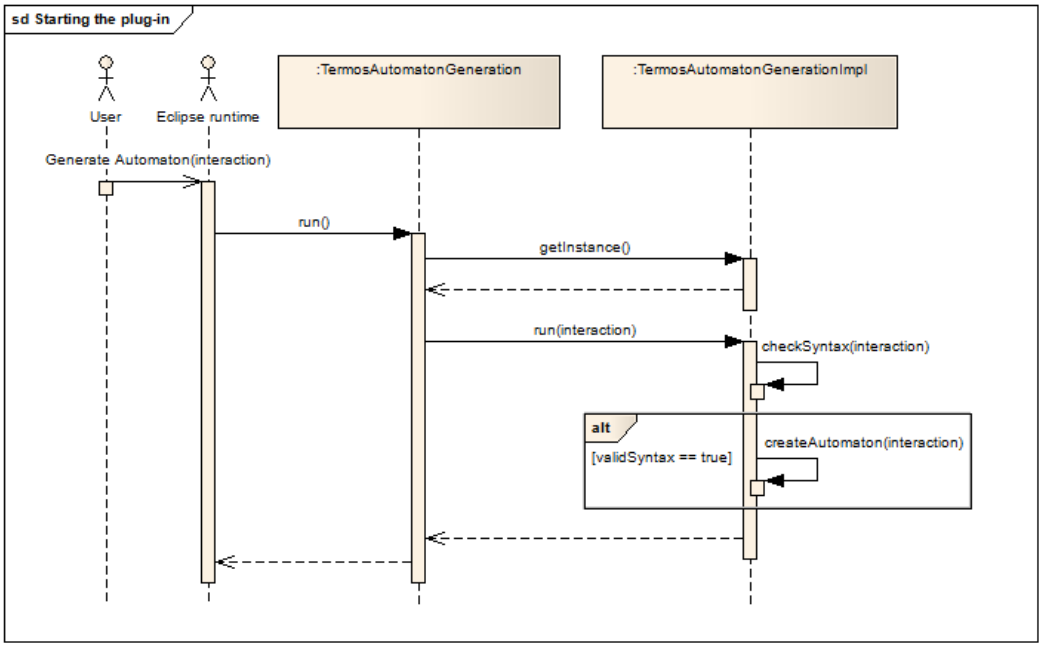


Figure 3.8: *Starting the TERMOS Tool Plug-in*



3.3.4 The TERMOS Tool Classes and Behaviour

To understand how the tool works, let us see the basic steps taken from the beginning to the end of the trace validation process.

- First, in order to be able to perform the syntax validation and the semantical soundness test of a TERMOS diagram, the configurations need to be identified. TERMOS Tool as a first step, looks for TERMOS configurations in the model containing the UML Interaction that describes the requirement scenario.

- Second, all the fragments of the diagram are checked separately and together to see if there are any syntax violations present in the scenario descriptions.
- When checking message events, the tool checks if the communication between the sender and the recipient node can take place in the current configuration.
- Next, if the diagram is valid, both syntactically and semantically, the pre-processing task is performed. Atoms are assigned locations, clusters are formed then grouped to form SimClasses, finally, the relations of the SimClasses are identified (prerequisites and conflicts).
- After the pre-processing, the SimClasses are unwound and the automaton is built and written into an XML file.
- The second phase starts with reading the XML file and building the automaton.
- Then, the trace file is read line by line, and processed by firing the transition enabled by the line last read.
- When the trace file is processed completely, i.e. there are no more lines to read, the result of the validation is decided based on the state of the automaton.

Execution of these steps start when the *run* method of one of the two action classes, i.e. either *TermosAutomatonGeneration* or *TermosExecutionTraceValidation*, is called. The *run* method then delegates the co-ordination to one of the implementation classes until the complete action is executed.

Accordingly, the first group of implementation classes consists of the ones responsible for the completion of the syntax validation, the checking of the soundness of the diagram, and automaton generation. These classes are:

TermosAutomatonGenerationImpl is responsible for initiating the syntax validation and the checking of semantical soundness of the requirement scenario description then the pre-processing of the diagram and the automaton generation.

TermosSyntaxValidator first looks for the configurations in the model of the Interaction, builds them in the system memory, then executes the syntax validation along with the checking of the semantics.

TermosAutomatonGenerator co-ordinates the generation of the automaton. It also implements the pre-processing of the diagram.

BetterTermosUnwindingStrategy implements the unwinding algorithm, that is, it builds the automaton. As its name shows, there have been previous versions of this class, this one is the result of optimizing and refactoring of the early implementations.

Syntax validation would not require the services of other classes, since it only analyzes the diagram using the API provided by the Eclipse UML plug-in. However, as semantic

validation takes place simultaneously, for which the configurations need to be known in depth, a few further services are needed that are implemented by the following classes:

TermosConfiguration represents a configuration defined in the spatial view of a requirement scenario.

TermosNode represents a node participating in a TERMOS configuration.

TermosConnection represents a pair of nodes that are connected in a TERMOS configuration.

The pre-processing of a diagram contains many concepts, such as atoms, clusters, SimClasses, their locations, and relations between them. These concepts are realized through these classes:

TermosAtom represents an *atom* in a pre-processed TERMOS diagram. It has a location on the lifeline it belongs to and a reference to the *InteractionFragment* it represents.

TermosMessageAtom extends *TermosAtom*, and it contains a *message* property to ease the access of this UML element.

TermosConfigAtom extends *TermosAtom*, and it contains a config property of type *TermosConfiguration* to ease the access of this element when needed (typically in the unwinding phase).

AtomKind is actually an enumeration used to distinguish atoms based on what kind of *InteractionFragments* they represent.

TermosCluster represents a *cluster* in a pre-processed TERMOS diagram. It has references to the atoms that form the cluster and a location derived from the locations of the atoms. TermosClusters also know their predecessors.

TermosSimClass represents a *SimClass* in a pre-processed TERMOS diagram. It contains references to its member clusters, to the conflicting SimClasses, and to the prerequisites.

SimClassKind is another helpful enumeration to ease the identification of the types of SimClasses.

Based on the information these classes contain after the pre-processing of a TERMOS diagram is finished, the tool knows enough to run the unwinding algorithm and to build the automaton. The algorithm implemented in the class *BetterUnwindingStrategy* produces the states and transitions that make up the output automaton of the action. The automaton concept is realized by the following classes:

TermosPhase represents a phase in the unwinding algorithm. It contains the *History*, the *Cut*, the *Ready*, and the *Variables* sets that are necessary for executing a cycle of the algorithm. A phase is the result of the *STEP* function described previously.

TermosAutomatonState represents a state in the output automaton. It contains the set of variables valuated in the represented state of the automaton, the ID of the state (an integer), and the state kind, which can be REJECT, ACCEPT-TRIVIAL, or ACCEPT-STRINGENT.

TermosAutomatonTransition represents a transition in the output automaton. It has references to the states it connects, and it also has a label which describes the conditions that should be satisfied in order to enable the transition.

TermosAutomaton represents the output automaton. It contains a set of states and transitions. An automaton has a name and an initial state specified. It has a public method that returns the XML representation of the automaton.

Once the XML representation of the output automaton is acquired, it is written into an XML file for later use, and the first phase is finished.

The second group of implementation classes is the one responsible for performing the validation of a trace. Along with the right to co-ordinate the execution of this action, a file containing an automaton and an execution trace file are passed to the instances of these classes by the run method of the action class. For reading the XML file and building the automaton and for the validation of the execution trace, the following implementation classes are responsible:

TermosExecutionTraceValidationImpl is responsible for the co-ordination of the entire trace validation action until it passes the control back to the invoking *run* method of the activator class to finish the running of the plug-in. It reads the XML file to build a *TermosAutomaton* object that will support the trace validation procedure.

TermosExecutionTraceValidator implements the logic to process the trace file using the *TermosAutomaton* built based on the input XML file and makes the decision concerning the final result of the trace validation process.

The classes, along with the dependency relations between them, are shown in the class diagrams in the *Appendices* part. The figure in A.2 shows the implementation classes co-operating in order to realize the automaton generation action, while the figure in A.1 shows those implementation classes that realize the execution trace validation action.

Chapter 4

Evaluating TERMOS Tool

In the previous chapter, the development of a new tool to support the automation of the HIDENETS Testing Framework was discussed. Both the structural and the behavioural features of this tool were presented as well as its operational environment and the format of the input data.

In this chapter, using a concrete example, the operation of TERMOS Tool is shown.

4.1 Blackboard Application: A Case Study

In [25], a so-called *blackboard application* was described as a typical example of mobile distributed systems. The description of the application is the following:

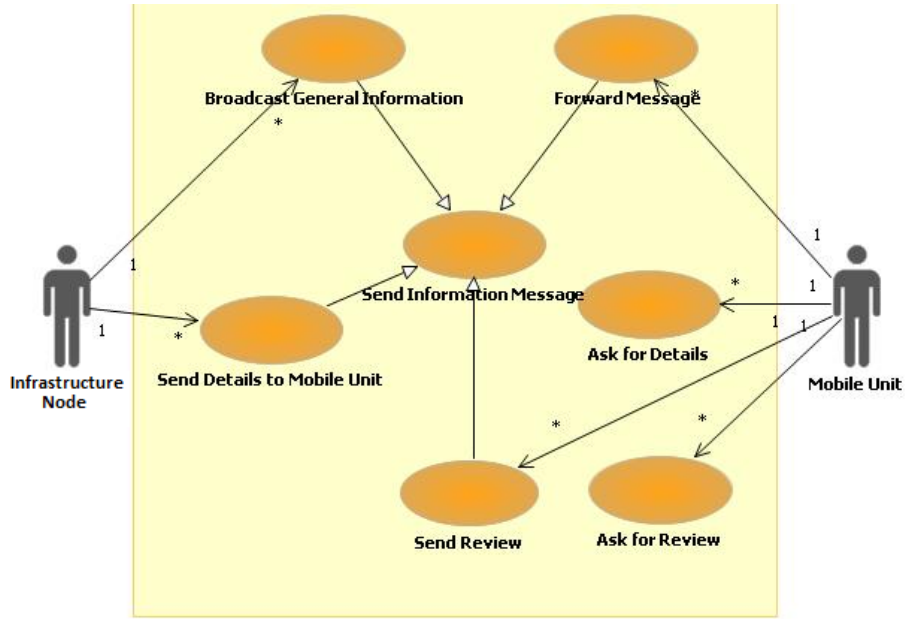
"The blackboard application shall serve to distribute information which is relevant for a certain geographic area. The notion behind is that a lot of information can be broadcast into the network but the user only wants to see the information which is relevant for him. A major share of information is relevant for a certain geographic area only, such as speed traps, fuel prices, restaurant offers, warning about a slippery road, etc. This special type of application assumes that the information is not necessarily permanently repeated by its source but sent only once (or with larger intervals). This means that the distribution has to happen by cars on the road which means also that the cars are the ones to store the data and to make sure the data does not disappear. A car may have received the message outside the relevant geographic area already but it is displayed only upon entering the area. This may be relevant for cases where no car is in the considered region but the message shall still be preserved in the network."

From a description like this, the relevant information to be inferred is that it is a mobile distributed application, therefore, its testing meets the challenges HIDENETS Testing Framework and TERMOS Tool are meant to address. Imagine a few communication scenarios that take place in a blackboard system like this. There may be one or more nodes with geographically fixed positions that send information to mobile nodes (like smart phones, GPS devices, car-computer systems) that approach them. Then, the mobile nodes that receive this information may forward it to other nodes, maybe even outside the geographic area the information relates to. In order to avoid network congestion, possibly only the

header of the message is sent first. Then, if the recipient node wants to learn more, it asks for details.

Figure 4.1 shows the possible use cases an application like this might have. Requirement scenarios can be defined to test each of these functions. In this section, the behaviour of this system is validated, using TERMOS Tool, in a scenario when a mobile node communicates directly with a node having a geographically fixed position. To abbreviate the name of a node with a fixed position, from now on, it is referred to as an *infrastructure node* as it can easily be assumed to be connected to a wired infrastructure; that is, mobility is not a factor outside the mentioned communication scenarios.

Figure 4.1: *Use Cases of a Blackboard Application*



The examined requirement scenario contains a mobile node and an infrastructure node communicating as soon as the topology enables them to. The infrastructure node communicates with the mobile node directly. The mobile node receives an informational message, after which it requests and, therefore, receives more details. The requirement scenario is shown in Figure 4.2, and the used configurations can be seen in Figure 4.3.

The TERMOS diagram expresses the requirement that if a node asks for details after receiving an informational message, then, if no configuration change occurs, the details should be delivered to the node.

4.2 Using TERMOS Tool

Once the model with the Interaction and the configurations is created, it can be imported into the Eclipse environment. Figure 4.4 shows an example of a model opened in this environment. By selecting the Interaction that represents the TERMOS Scenario with a right click, the *Generate Automaton* action can be selected from the *TERMOS Tool* submenu of the pop-up menu that appears. As a result, if the TERMOS diagram is valid, an XML file is generated into the *automaton* sub-directory of the workspace. If the directory

Figure 4.2: *Requirement Scenario for the Blackboard Application*

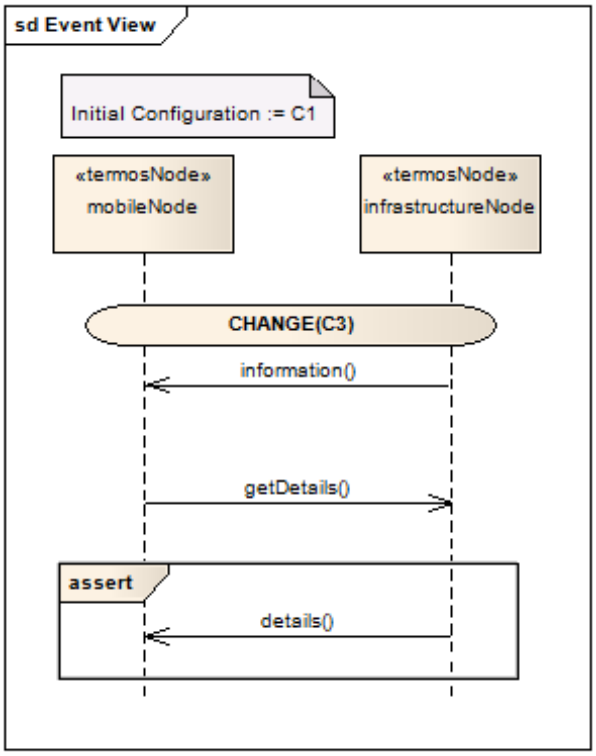
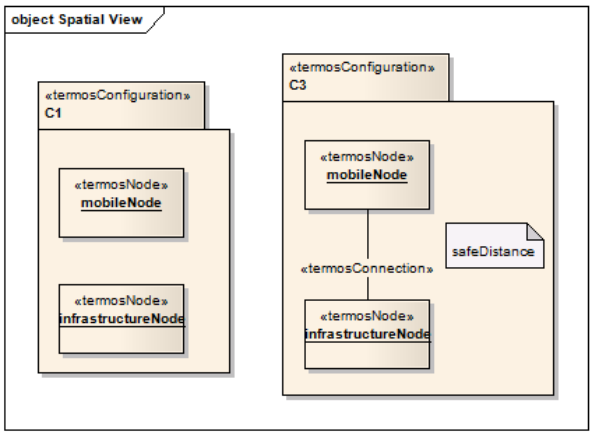


Figure 4.3: *Configurations Used in the Requirement Scenario*



does not exists, the tool creates it. The name of the file will be the same as the name of the TERMOS scenario, and its extension will be *.aut*.

If we call the *Generate Automaton* action on the requirement scenario described above, the automaton presented in Figure 4.5 is generated in an XML format. The XML content generated for this TERMOS diagram can be found in Section A.3. Since there are no *par, alt, opt* fragments in the scenario, the automaton is quite simple, but as we can see, even in case of such primitive scenarios, the automaton has relatively many states. From the scenario description shown in Figure 4.6, an automaton with as many as 30 states is generated.

Figure 4.4: *A UML File in the Eclipse Environment*

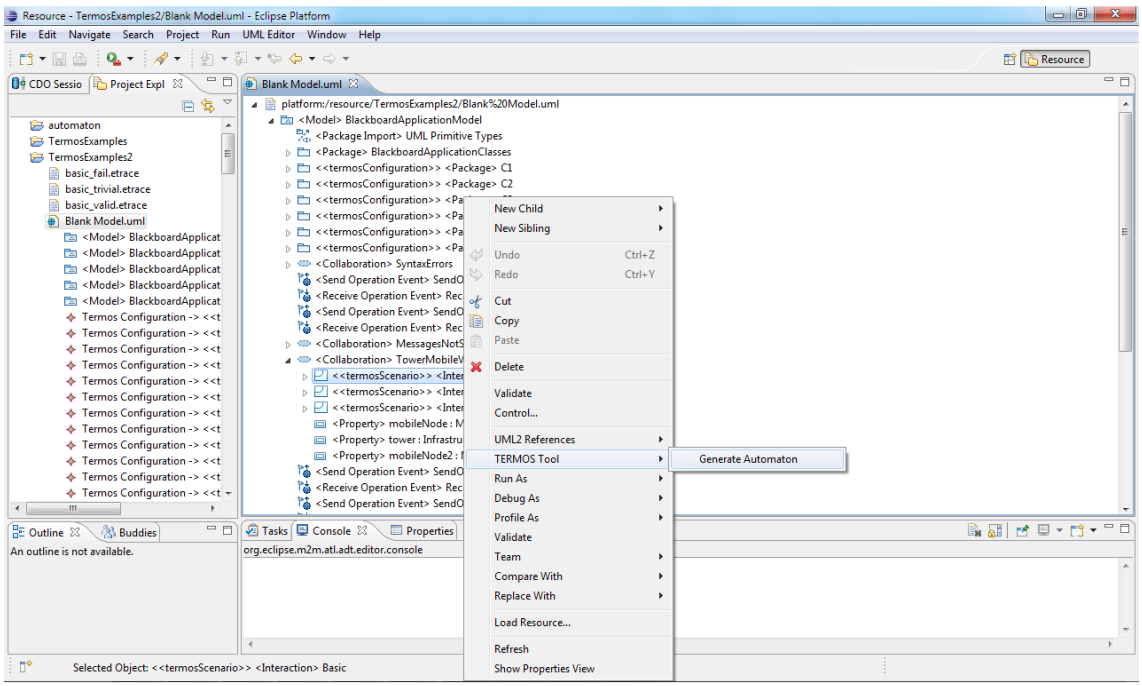


Figure 4.5: *FSA Generated From a TERMOS Diagram*

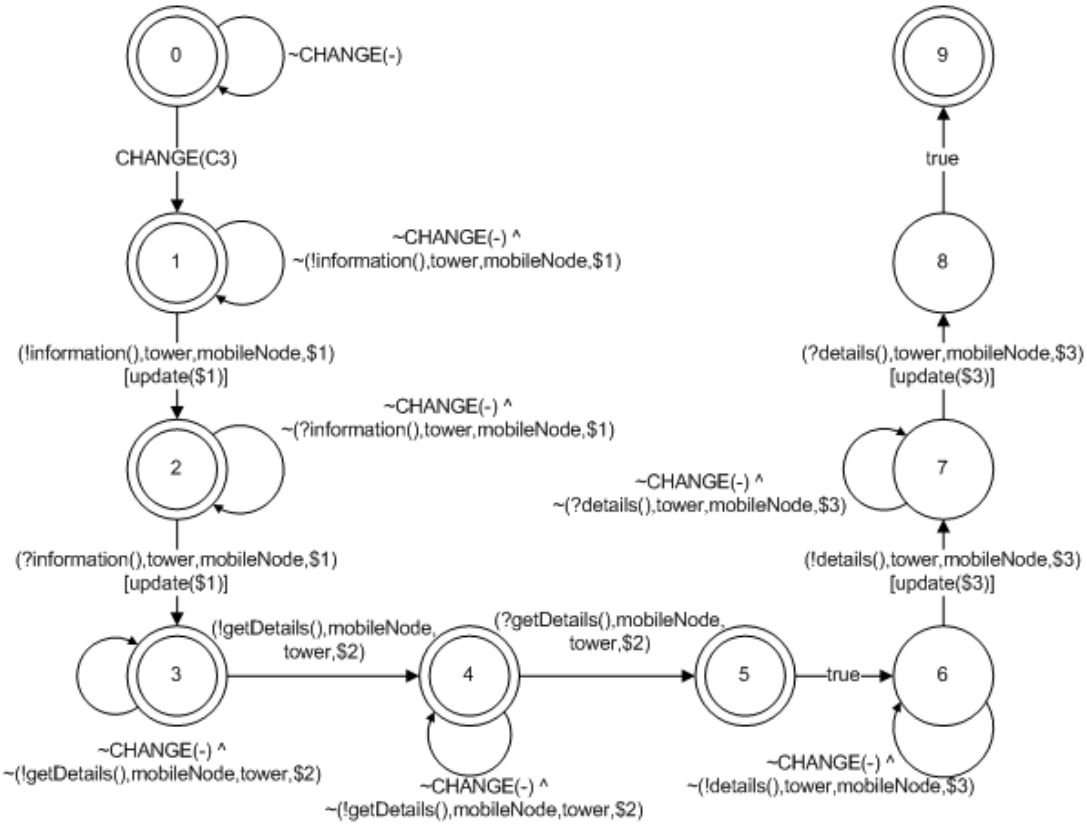
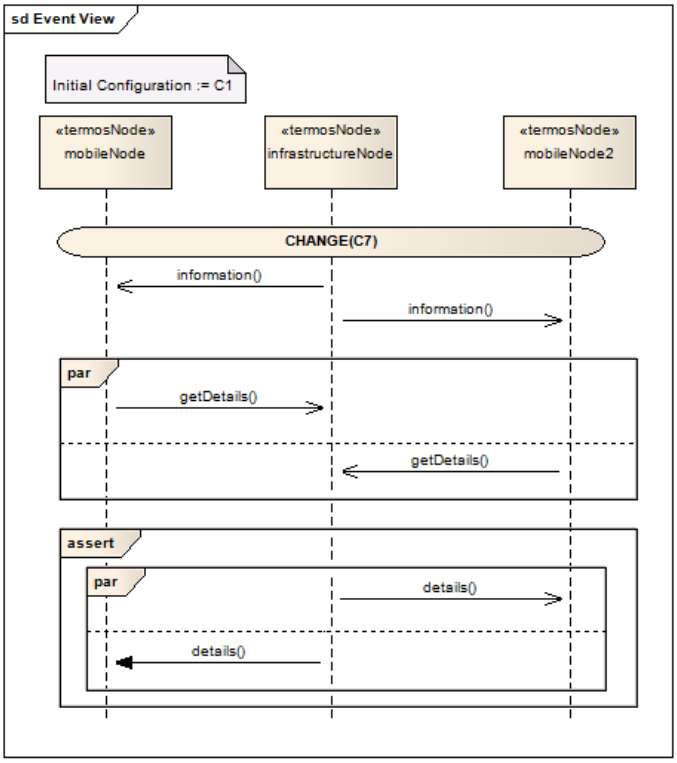


Figure 4.6: *A Scenario with* par *Fragments*



The trace validation can be started by clicking on a file with *.aut* exension in the *Project Explorer* view with the right mouse button and selecting the *Validate Trace* action from the *TERMOS Tool* submenu of the pop-up menu that appears as a result of the file selection. Obviously, we need to specify which trace file we would like to validate. This can be done when the file-selector window appears. Trace files have the *.etrace* extension and their format is as specified in the previous chapter. An example for the content of such trace files can be found in Section A.4. The example is a trace accepted by our previously presented automaton as non-trivially valid. The output of the trace validation action of the TERMOS Tool for that specific trace file can be seen in Section A.5.

4.3 Evaluation

The tool was tested with numerous other scenario descriptions and many traces, valid, trivially valid, and invalid ones. To check the capability of the tool to find syntax errors in a TERMOS scenario description and messages expected to be sent in configurations, where the connection is either unsafe or does not exists between the sender and the recipient nodes. The example scenarios are described in Table 4.1.

The automaton generated for the *Basic valid scenario* was tested with a valid, a trivially valid, and an invalid trace file, whereas the *Valid scenario with Par* was tested with nine different trace files. Seven of these traces were valid; the difference between them was the sequencing of the messages. Some of these valid traces contain trace records that are irrelevant to the scenario, still, TERMOS Tool was not disturbed by these message events.

Table 4.1: *Requirement Scenarios Checked With TERMOS Tool*

Name	Description
No termosScenario stereotype	In this scenario, the UML Interaction element does not have the termosScenario stereotype applied. TERMOS Tool identifies this as a syntax error and does not generate an automaton.
No Assert	This scenario description has no Assert operator at the end of the diagram. TERMOS Tool identifies this as a syntax error and does not generate an automaton.
No initial configuration	This TERMOS diagram does not specify an initial configuration. TERMOS Tool identifies this as a syntax error and does not generate an automaton.
Invalid fragment	The scenario description contains two CombinedFragments that are not valid in TERMOS. TERMOS Tool identifies this as a syntax error and does not generate an automaton.
Tower with MobileNode	In this scenario an InfrastructureNode instance tries to communicate with a MobileNode instance, but their current configuration does not make it possible. TERMOS Tool identifies this event and, therefore, it does not generate an automaton.
MobileNode with MobileNode	This scenario is the same as the previous one; the only difference is that the two nodes are of the same type. No automaton is generated due to the lack of semantical soundness.
Basic valid scenario	This is the TERMOS scenario presented previously in this chapter to show how the tool works. The automaton generated from this scenario can be seen on Figure 4.5.
Valid scenario with Par	This is also a previously presented scenario with three nodes participating in it. The diagram contains *Par* fragments to be able to test the capability of the tool to identify every valid sequence of message events. TERMOS Tool identifies all of the valid sequences and builds the automaton accordingly.

Another trace was trivially valid, as one of the messages that participate in the prefix of the TERMOS diagram was not sent during the execution, and a trace file was invalid due to the system losing a message.

The primary goal of the implementation of this tool was to show that the algorithms presented in the previous chapter work in practice as well and can actually be used to perform a trace validation. This goal has been achieved as the tool is able to transform TERMOS diagrams into automatons and validate execution traces using these automatons, and the automaton generation and the trace validation functionalities are completely based on those algorithms.

The tool can use some fine tuning for sure, with regards to user experience, for example, but it is capable of processing most TERMOS diagrams and validate traces against requirements defined in those diagrams. Also, a great success would be the integration of the graph-matching tool called *GraphSeq* [17] in order to create a tool that covers the whole trace validation process. The automation of this process can be further supported

by making the user have to specify the traces to validate and the model that contains all the TERMOS diagrams and letting the tool deal with these inputs itself by executing the complete trace validation process and generating a report at the end.

Chapter 5

Conclusion

Working on my diploma thesis project, I have had the chance to become familiar with the state-of-the-art testing methodologies used for the testing of traditional distributed systems as well as with the ones addressing the challenges raised by mobile ad hoc networking, the prevailing form of networking used in the context of mobile distributed systems. This mobile setting has specificities that require system designers, as well as test architects and system testers, to handle issues unseen in traditional systems. The diploma thesis has introduced a test platform and a testing framework that may be used for the testing of mobile computing systems.

To be able to use this testing framework, in which requirement scenarios play a vital role, a graphical scenario language had to be selected that is able to denote the continuous evolution of the network topology. The most widely used scenario languages have been examined and introduced in this diploma thesis, and they were found to be unable to express everything that would be required when describing the behaviour of a mobile distributed system. Therefore, a new scenario language, TERMOS was chosen for this purpose.

As part of this diploma thesis project, a tool has been built to contribute to the realization of the previously mentioned testing framework. This is a small contribution, but it definitely plays an essential role in the framework. The tool is capable of performing a significant part of execution trace validation. It also helps creating TERMOS diagrams with a valid syntax, as most of the syntax rules defined for the language are checked during the first phase of the trace validation process. From some aspects, the tool is also able to check the semantical soundness of TERMOS diagrams.

The tool could probably be further developed after a more extensive testing. Clearly, there has not been enough time and capacity to create really complex and big scenarios that could point out all the weaknesses of the tool. On the other hand, it is also worth noting that requirement scenarios usually focus on very specific parts of the behaviour, thus, they are not very big in size, generally.

Also, TERMOS Tool is only capable of validating traces where the participating nodes are already matched with the symbolic nodes of the requirement scenario. By integrating it with another application that performs this graph matching task, the whole validation

process could be covered.

One really positive and important thing about the tool is that it has been created as an extension to one of the most popular integrated development environments, therefore, it can easily be integrated with other tools that are implemented as plug-ins of the same environment. Thus, if the graph matching tool is implemented as a separate plug-in of Eclipse, the IDE TERMOS Tool extends, the two tools can easily be orchestrated to cover the whole trace validation process.

Only a few issues have been mentioned here that show how the tool could be further developed; obviously, some more could be found after an extensive testing. Nevertheless, the main goal of the development of TERMOS Tool have been achieved: TERMOS scenario language and the algorithms that existed only in theory before have been implemented and shown to have the capability to contribute to the testing of mobile distributed systems.

Acknowledgement

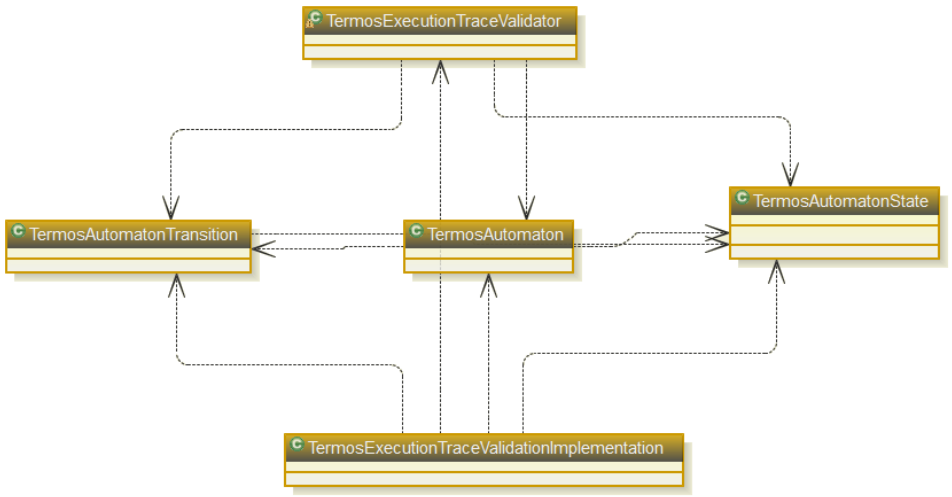
It is a pleasure to thank to those who made my thesis possible with their support and advices.

Special thanks to Zoltán Micskei, my thesis advisor, for his patience and for the great academic help I received from him during the past three years.

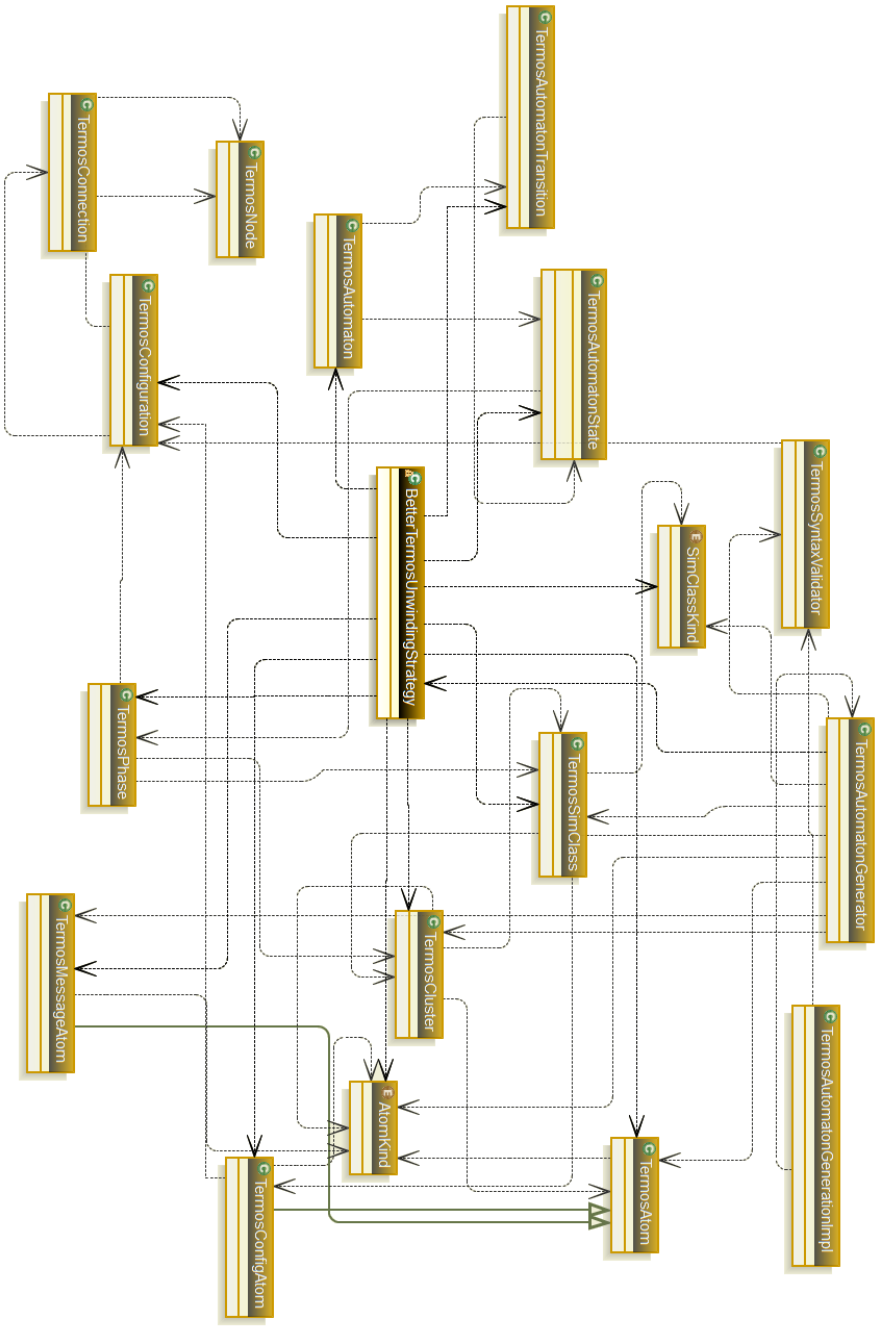
I would also like to thank Tibor Bende, my friend, for supporting me throughout the semester and for spending his nights with checking my grammar.

Appendices

A.1 TERMOS Tool Implementation Classes Participating in the Execution Trace Validation Action



A.2 TERMOS Tool Implementation Classes Participating in the Automaton Generation Action



A.3 XML Representation of an Automaton Generated by TERMOS Tool

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<TermosAutomaton name="Basic">
  <TermosStates>
   <TermosState stateId="0" stateKind="INITIAL">
    <TermosVariable name="tower" value=""/>
    <TermosVariable name="mobileNode" value=""/>
   </TermosState>
   <TermosState stateId="1" stateKind="ACCEPT_TRIVIAL">
    <TermosVariable name="tower" value=""/>
    <TermosVariable name="mobileNode" value=""/>
   </TermosState>
   <TermosState stateId="2" stateKind="ACCEPT_TRIVIAL">
    <TermosVariable name="tower" value=""/>
    <TermosVariable name="mobileNode" value=""/>
    <TermosVariable name="$1" value=""/>
   </TermosState>
   <TermosState stateId="3" stateKind="ACCEPT_TRIVIAL">
    <TermosVariable name="tower" value=""/>
    <TermosVariable name="mobileNode" value=""/>
    <TermosVariable name="$1" value=""/>
   </TermosState>
   <TermosState stateId="4" stateKind="ACCEPT_TRIVIAL">
    <TermosVariable name="tower" value=""/>
    <TermosVariable name="mobileNode" value=""/>
    <TermosVariable name="$1" value=""/>
    <TermosVariable name="$2" value=""/>
   </TermosState>
   <TermosState stateId="5" stateKind="ACCEPT_TRIVIAL">
    <TermosVariable name="tower" value=""/>
    <TermosVariable name="mobileNode" value=""/>
    <TermosVariable name="$1" value=""/>
    <TermosVariable name="$2" value=""/>
   </TermosState>
   <TermosState stateId="6" stateKind="REJECT">
    <TermosVariable name="tower" value=""/>
    <TermosVariable name="mobileNode" value=""/>
    <TermosVariable name="$1" value=""/>
    <TermosVariable name="$2" value=""/>
   </TermosState>
   <TermosState stateId="7" stateKind="REJECT">
    <TermosVariable name="tower" value=""/>
    <TermosVariable name="mobileNode" value=""/>
    <TermosVariable name="$1" value=""/>
    <TermosVariable name="$3" value=""/>
    <TermosVariable name="$2" value=""/>
   </TermosState>
   <TermosState stateId="8" stateKind="REJECT">
    <TermosVariable name="tower" value=""/>
    <TermosVariable name="mobileNode" value=""/>
    <TermosVariable name="$1" value=""/>
    <TermosVariable name="$3" value=""/>
    <TermosVariable name="$2" value=""/>
   </TermosState>
   <TermosState stateId="9" stateKind="ACCEPT_STRINGENT">
    <TermosVariable name="tower" value=""/>
    <TermosVariable name="mobileNode" value=""/>
    <TermosVariable name="$1" value=""/>
    <TermosVariable name="$3" value=""/>
    <TermosVariable name="$2" value=""/>
   </TermosState>
  </TermosStates>
```

The XML code continues on the next page...

```
  <TermosTransitions>
   <TermosTransition from="0" label="CHANGE(C3)" to="1"/>
   <TermosTransition from="0" label="~CHANGE(-)" to="0"/>
   <TermosTransition from="1" label="(!information(),tower,mobileNode,$1)
   [update($1)]" to="2"/>
   <TermosTransition from="1" label="~CHANGE(-) ^
    ~(!information(),tower,mobileNode,$1)" to="1"/>
   <TermosTransition from="2" label="(?information(),tower,mobileNode,$1)
    [update($1)]" to="3"/>
   <TermosTransition from="2" label="~CHANGE(-) ^
    ~(?information(),tower,mobileNode,$1)" to="2"/>
   <TermosTransition from="3" label="(!getDetails(),mobileNode,tower,$2)
    [update($2)]" to="4"/>
   <TermosTransition from="3" label="~CHANGE(-) ^
    ~(!getDetails(),mobileNode,tower,$2)" to="3"/>
   <TermosTransition from="4" label="(?getDetails(),mobileNode,tower,$2)
    [update($2)]" to="5"/>
   <TermosTransition from="4" label="~CHANGE(-) ^
    ~(?getDetails(),mobileNode,tower,$2)" to="4"/>
   <TermosTransition from="5" label="true" to="6"/>
   <TermosTransition from="6" label="(!details(),tower,mobileNode,$3)
    [update($3)]" to="7"/>
   <TermosTransition from="6" label="~CHANGE(-) ^
    ~(!details(),tower,mobileNode,$3)" to="6"/>
   <TermosTransition from="7" label="(?details(),tower,mobileNode,$3)
    [update($3)]" to="8"/>
   <TermosTransition from="7" label="~CHANGE(-) ^
    ~(?details(),tower,mobileNode,$3)" to="7"/>
   <TermosTransition from="8" label="true" to="9"/>
  </TermosTransitions>
</TermosAutomaton>
```

A.4 The Contents of an Execution Trace File

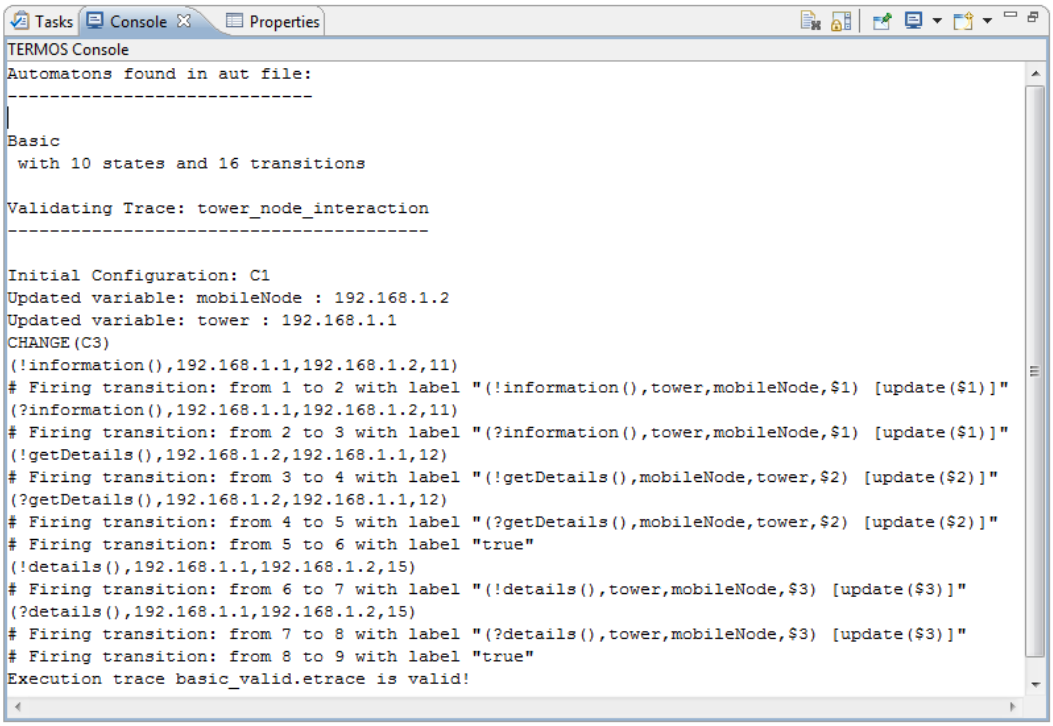
```
termos_execution_trace tower_node_interaction
init_config C1
mobileNode 192.168.1.2
tower 192.168.1.1

CHANGE(C3)
(!information(), 192.168.1.1, 192.168.1.2, 11)
(?information(), 192.168.1.1, 192.168.1.2, 11)
(!getDetails(), 192.168.1.2, 192.168.1.1, 12)
(?getDetails(), 192.168.1.2, 192.168.1.1, 12)
(!details(), 192.168.1.1, 192.168.1.2, 15)
(?details(), 192.168.1.1, 192.168.1.2, 15)

#end of trace
```

The first line assigns a name with the execution trace (optional). The second line specifies the initial configuration (optional). The third and fourth commands are required to match the symbolic nodes represented by the lifelines in the TERMOS diagram with the actual node IDs, which are, in this case, IP addresses. Then, the next few lines are the events recorded during the execution of a test case. The last lines in the trace file is a comment. Comment lines begin with a *'#'* and can be placed anywhere within the file, TERMOS Tool does not process these lines, only prints them in the console. Moreover, empty lines, extra whitespaces do not influence the operation of TERMOS Tool.

A.5 Output of the Trace Validation Action of the TERMOS Tool for Valid Traces



List of Figures

List of Tables

Bibliography

[1] José Antonio Arnedo, Ana Cavalli, and Manuel Núñez. Fast testing of critical properties through passive testing. In *TestCom'03: Proceedings of the 15th IFIP international conference on Testing of communicating systems*, pages 295–310, Berlin, Heidelberg, 2003. Springer-Verlag.

[2] John J. Barton and Vikram Vijayaragharan. *Ubiwise: A Simulator for Ubiquitous Computing Systems Design, Technical report HPL-2003-93.* Hewlett-Packard Lab, 2003.

[3] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[4] Matthew Clegg and Keith Marzullo. A low-cost processor group membership protocol for a hard real-time distributed system. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium*, page 90, Washington, DC, USA, 1997. IEEE Computer Society.

[5] U2TP Consortium. *UML 2.0 Testing Profile Specification.* Object Management Group Inc., 2004. Version 1.0, formal/05-07-07.

[6] Zhen Ru Dai. Model-Driven Testing with UML 2.0. 2nd European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations, 2004.

[7] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. In *Formal Methods in System Design*, pages 293–312. Kluwer Academic Publishers, 1998.

[8] Manfred Reitenspieß et al. Experimental proof-of-concept set-up hidenets (description of hidenets test-bed implementation). *HIDENTES D6.3 Deliverable*, December 2008. http://www.hidenets.aau.dk/.

[9] Apache Software Foundation. Apache log4j 1.2. http://logging.apache.org/log4j/1.2/.

[10] The Eclipse Foundation. Eclipse Modeling - MDT (UML2 Tools). http://www.eclipse.org/modeling/mdt/?project=uml2tools, 2010.

[11] The Eclipse Foundation. Eclipse Platform Overview (Project Charter). http://www.eclipse.org/eclipse/eclipse-charter.php, 2010.

[12] The Eclipse Foundation. Eclipse.org. http://www.eclipse.org, 2010.

[13] The Eclipse Foundation. EMF. http://www.eclipse.org/emf/, 2010.

[14] The Eclipse Foundation. MDT. http://www.eclipse.org/mdt/, 2010.

[15] The Eclipse Foundation. PDE. http://www.eclipse.org/pde/, 2010.

[16] The Eclipse Foundation. Rich Client Platform. http://www.eclipse.org/home/categories/rcp.php, 2010.

[17] Gábor Huszerl, Hélène Waeselynck (ed.), Zoltán Égel, András Kövi, Zoltán Micskei, Minh Duc N'Guyen, Gergely Pintér, and Nicolas Rivière. Refined design and testing framework, methodology and application results. *HIDENTES D5.3 Deliverable*, December 2008. http://www.hidenets.aau.dk/.

[18] Abu Zafer Javed, Paul Anthony Strooper, and G. N. Watson. Automated generation of test cases using model-driven architecture. In *AST '07: Proceedings of the Second International Workshop on Automation of Software Test*, page 3, Washington, DC, USA, 2007. IEEE Computer Society.

[19] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison - Wesley Professional, 2003.

[20] Jochen Klose. *Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior.* C. v.O. Universitat Oldenburg, 2003. PhD thesis.

[21] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[22] Zoltán Micskei and Hélène Waeselynck. The many meanings of UML 2 Sequence Diagrams: a survey. *Software and Systems Modeling*, April 2010.

[23] Minh Duc Nguyen, Hélène Waeselynck, and Nicolas Rivière. Testing mobile computing applications: toward a scenario language and tools. In *WODA '08: Proceedings of the 2008 international workshop on dynamic analysis*, pages 29–35, New York, NY, USA, 2008. ACM.

[24] Object Management Group. *Unified Modeling Language (OMG UML), Superstructure, V2.1.2*, 2007. Version 1.0, formal/05-07-07.

[25] Marcus Radimirisch and et al. Use case scenarios and preliminary reference model. *HIDENTES D1.1 Deliverable*, December 2006. http://www.hidenets.aau.dk/.

[26] Ina Schieferdecker, Zhen Ru Dai, Jens Grabowski, and Axel Rennoch. The UML 2.0 testing profile and its relation to TTCN-3. In *TestCom'03: Proceedings of the 15th IFIP international conference on Testing of communicating systems*, pages 79–94, Berlin, Heidelberg, 2003. Springer-Verlag.

[27] Bikram Sengupta and Rance Cleaveland. Triggered Message Sequence Charts. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 167–176, New York, NY, USA, 2002. ACM.

[28] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Elsevier Inc., 2007.

[29] Hélène Waeselynck, Zoltán Micskei, Minh Duc N'Guyen, and Nicolas Rivière. Preliminary testing framework and methodology. *HIDENTES D5.2 Deliverable*, December 2007. http://www.hidenets.aau.dk/.

[30] Larry D. Wittie. Computer networks and distributed systems. *Computer*, 24(9):67–76, 1991.

[31] ITU-T Recommendation Z.120. Message Sequence Chart (MSC), November 1999.