Languages and frameworks for specifying test artifacts

PhD dissertation by

Zoltán Micskei

Advisors

István Majzik, PhD (BME)
Hélène Waeselynck, PhD (LAAS-CNRS)

MŰEGYETEM 1782

Zoltán Micskei
`http://mit.bme.hu/~micskeiz/`

March 2013

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

H-1117 Budapest, Magyar tudósok körútja 2.

Declaration of own work and references

I, Zoltán Micskei, hereby declare, that this dissertation, and all results claimed therein are my own work, and rely solely on the references given. All segments taken word-by-word, or in the same meaning from others have been clearly marked as citations and included in the references.

Nyilatkozat önálló munkáról, hivatkozások átvételéről

Alulírott Micskei Zoltán kijelentem, hogy ezt a doktori értekezést magam készítettem és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2013. 03. 18.

Micskei Zoltán

Acknowledgements

First and foremost I would like to express my gratitude to my advisors. I was fortunate enough to have been co-advised by two excellent researchers. István Majzik has guided me since my Master thesis in the world of academics. With Hélène Waeselynck I could taste the life at a research institute. Without their continued involvement and insights this dissertation would not have been possible.

I would like to sincerely thank Árpád Beszédes (University of Szeged) and Lionel Briand (University of Luxembourg) for taking the time to read the draft of my dissertation and helping me with their valuable feedback.

I would like to thank my former and present colleagues at BME's Department of Measurement and Information Systems and the Fault Tolerant Systems Research Group for the time we have spent together. I am especially grateful to András Pataricza for his trust and support.

I wish to thank the people in the Tolérance aux Fautes et Sûreté de Fonctionnement Informatique group at LAAS for the time I spent there, and Karama Kanoun and Jean Arlat for welcoming me. A special thanks to the doctorands of TSF for making my time in Toulouse even more memorable.

I would like to acknowledge the colleagues I worked with on the research presented in the dissertation (in no particular order): Minh Duc Nguyen, Nicolas Rivière, and Francis Tam.

I would like to thank the support of the ReSIST NoE project (IST 026764), coordinated by Jean-Claude Laprie, which made the co-supervision possible, and the support of the HIDENETS (IST 026979), R3-COP (Artemis 100233) and TÁMOP (4.2.2.B-10/1-2010-0009) projects.

Finally, I would like to express my sincere thanks to my family.

Köszönetnyilvánítás

Először is szeretném köszönetemet kifejezni konzulenseimnek. Abban a szerencsés helyzetben voltam, hogy kutatásaimat két kiváló konzulens is segítette. Majzik István már a diplomatervezésem óta irányította munkámat. Hélène Waeselynck mellett a LAAS-ban megismerhettem, hogy milyen egy kutatólaborban dolgozni. Kettőjük folyamatos támogatása és visszajelzései nélkül nem készülhetett volna el ez az értekezés.

Ezúton szeretném megköszönni Beszédes Árpádnak (Szegedi Tudományegyetem) és Lionel Briand-nak (University of Luxembourg), hogy átolvasták értekezésem előzetes verzióját, és értékes javaslataikkal segítették munkámat.

Köszönöm a BME Méréstechnika és Információs Rendszerek Tanszék és azon belül is a Hibatűrő Rendszerek Kutatócsoport volt és jelenlegi munkatársainak az együtt eltöltött időt. Külön köszönöm Pataricza Andrásnak a bizalmát és támogatását.

Köszönöm a Tolérance aux Fautes et Sûreté de Fonctionnement Informatique csoportjának a LAAS-ban eltöltött időt, valamint Karama Kanounnak és Jean Arlat-nak, hogy vendégül láttak. Külön köszönet a TSF doktoranduszainak, hogy még emlékezetesebbé tették a toulouse-i hónapjaimat.

Külön köszönet illeti kollégáimat, akikkel az évek során a disszertációban szereplő kutatásokon együtt dolgoztunk: Minh Duc Nguyen, Nicolas Rivière, Francis Tam.

Toulouse-i tartózkodásomat a ReSIST Network of Excellence projekt támogatása tette lehetővé. Kutatásaimat ezen kívül még a HIDENETS (IST 026979), R3-COP (Artemis 100233) és a TÁMOP (4.2.2.B-10/1-2010-0009) projektek támogatták.

Végezetül szeretném megköszönni családomnak támogatásukat és kitartásukat. Szüleimnek, akik azzá neveltek, aki vagyok. Testvéremnek, hogy átolvasta munkámat. Kedvesemnek, hogy mellettem állt és biztatott.

Summary

Testing is one of the most common verification and validation activities in software development. Testing is a complex process, where numerous test artifacts have to be developed that represent test requirements, test cases, etc. The precise construction of these test artifacts requires suitable languages. Together the languages describing the test artifacts and the tools supporting the various test tasks form a test framework that can support the entire test process. This dissertation is concerned with developing *languages* and *test framework*s for two specific, new application domains.

The first part of the dissertation presents research on testing *high availability middleware* systems. In case of such middleware not just the usual functional properties, but also the robustness of an implementation – the degree to which it can function correctly in the presence of invalid inputs – is of great concern. The first challenge the dissertation aimed for was the robustness testing of high availability middleware systems. One of the contributions of the dissertation is the systematic definition of languages and development of supporting tools that form a test framework, which uses a combination of API testing, state-based mutation testing and OS call interception to provide inputs that exercise the middleware to activate potential robustness faults. The test suite generated by the test tools in the framework was executed on three different middleware implementations comparing their robustness.

The second part of the dissertation focuses on *mobile systems*. The context-aware, highly dynamic nature of mobile systems makes it especially hard to describe them in test artifacts using the existing languages. The second challenge of the dissertation was centered on testing of mobile systems. The dissertation recommends language extensions to graphical scenario languages that can intuitively capture the frequently changing communication structures. Using these extensions a formal test requirement language, called TERMOS, was designed that is based on UML 2 Sequence Diagrams. An automaton-based operational semantics was defined for the new language. A test framework was developed that supports the evaluation of execution traces against the graphical scenarios and can identify violations of the requirements captured in the scenarios.

When we started to work on TERMOS, it turned out that the *semantics* of UML 2 Sequence Diagrams is a complex issue. The UML specification contains some explicit semantic variations, but there are many more subtle semantic choices. The third challenge of the dissertation was about the semantics of UML 2 Sequence Diagrams. The dissertation collects 13 proposed formal semantics and analyses the different choices taken by them. The contribution of the dissertation is a structured representation of the various semantic choices and options. The detailed discussions of the choices highlight the relations and consequences of each of the options. This categorization was later used to design the semantics of the TERMOS language.

Összefoglaló

A tesztelés az egyik leggyakrabban használt verifikációs és validációs tevékenység a szoftverfejlesztés során. A tesztelés egy összetett folyamat, ahol sokféle tesztelési terméket kell elkészíteni, amik az egyes tesztkövetelményeket, teszteseteket stb. tartalmazzák. A tesztelési termékek precíz tervezéséhez viszont szükség van megfelelő nyelvekre. A tesztelési termékeket leíró nyelvek és az egyes tesztelési feladatokat támogató eszközök együttesen egy tesztkeretrendszert alkotnak, amely képes a tesztelési folyamatot támogatni. Az értekezés témája nyelvek és tesztkeretrendszerek kidolgozása két új alkalmazási területhez.

Az értekezés első része egy, a *magas rendelkezésre állású köztesrétegek* tesztelésével kapcsolatos kutatást mutat be. Ilyen köztesrétegek esetén nem csak a szokásos funkcionális jellemzők, hanem az implementáció robusztussága – azaz az érvénytelen bemenetekkel szembeni ellenállósága – is fontos szempont. Az értekezés által tárgyalt első kihívás tehát a magas rendelkezésre állású köztesrétegek robusztusságának tesztelése. Az értekezés egyik eredménye egy olyan szisztematikusan kidolgozott tesztkeretrendszer (nyelvek és eszközök), ami az API tesztelés, állapot-alapú mutációs tesztelés valamint az OS hívások eltérítésének kombinációjával többféle robusztussági hibatípust tud felderíteni. A keretrendszer eszközei által generált tesztkészlet futtatásával három különböző köztesréteg robusztusságát hasonlítottuk össze.

Az értekezés második része *mobil rendszerekre* összpontosít. A mobil rendszerek kontextusra érzékeny, dinamikus jellege különösen megnehezíti leírásukat a különböző tesztelési termékekben a meglévő nyelvekkel. Az értekezés második kihívása a mobil rendszerek tesztelésére vonatkozott. Az értekezés kiegészítéseket javasol a grafikus forgatókönyv-leíró nyelvekhez, amiknek a segítségével intuitív módon lehet leírni a gyakran változó kommunikációs struktúrákat. Ezekre a kiegészítésekre építve bevezet egy TERMOS nevű, formális, tesztkövetelményeket leíró nyelvet, ami az UML 2 Szekvencia Diagramokra épül. Az új nyelvhez egy automatákon alapuló működési szemantikát definiál. Kidolgoztunk továbbá egy olyan tesztkeretrendszert is, amely képes futási nyomokat kiértékelni a grafikus forgatókönyvek alapján, és azonosítani tudja, hogy sérülnek-e a forgatókönyvekben megfogalmazott követelmények.

Amikor elkezdtünk dolgozni a TERMOS nyelven, kiderült, hogy az UML 2 Szekvencia Diagramok *szemantikája* sokkal összetettebb kérdés, mint ahogy gondoltuk. Habár az UML szabvány egyértelműen megjelöl néhány szemantikai változatot, annál sokkal több rejtett szemantikai döntési pont van valójában. Az értekezés harmadik kihívása az UML 2 Szekvencia Diagramok szemantikájához kapcsolódik. Az értekezés 13 korábban javasolt formális szemantika alapján vizsgálta a különböző választási lehetőségeket. Az értekezés eredménye az egyes döntési pontok és azok lehetőségeinek strukturált formában való megjelenítése és kiértékelése. Az egyes lehetőségek elemzése kitér az azok közötti kapcsolatokra, valamint az egyes választások következményeire. Ezt a kategorizálást használtuk fel később a TERMOS nyelv szemantikájának meghatározása során.

Contents

Contents **vii**

1 Introduction **1**
1.1 Existing test languages and approaches . . . 2
1.1.1 Examples of languages for describing test artifacts . . . 2
1.1.2 Using UML 2 for specifying test artifacts . . . 3
1.1.3 Test approaches utilized in the dissertation . . . 3
1.2 New application domains . . . 4
1.2.1 High availability middleware systems . . . 4
1.2.2 Context-aware mobile computing systems . . . 5
1.3 Summarizing the new challenges . . . 6
1.4 Research method . . . 6
1.5 Contributions and structure of the dissertation . . . 8

2 Robustness testing of HA middleware **11**
2.1 Robustness test techniques . . . 12
2.1.1 Injecting physical faults . . . 12
2.1.2 Using random inputs . . . 13
2.1.3 Using type-specific tests . . . 13
2.1.4 Testing object-oriented systems . . . 14
2.1.5 Applying mutation techniques . . . 14
2.1.6 Model-based robustness testing . . . 14
2.2 Activation modes of robustness faults and the test approach . . . 15
2.3 Robustness test framework and tools . . . 18
2.3.1 Type-specific testing . . . 18
2.3.2 Mutation-based sequential testing . . . 24
2.3.3 OS call interception . . . 27
2.4 Robustness testing case studies . . . 29
2.4.1 Case study design . . . 30
2.4.2 Case study results . . . 32
2.4.3 Conclusions of the case studies . . . 38
2.5 Summary . . . 39

3 Semantic choices in UML 2 Sequence Diagrams **41**
3.1 UML Sequence Diagrams in the OMG specification . . . 42
3.1.1 Syntax of Sequence Diagrams . . . 42

3.1.2 Semantics of Sequence Diagrams 45
3.2 Overview of proposed semantics 49
3.2.1 Trace semantics from Störrle 52
3.2.2 STAIRS approach 52
3.2.3 ASM-based semantics of Cavarra & Filipe 53
3.2.4 Trace-based semantics of Cengarle & Knapp 53
3.2.5 True-concurrency semantics from Küster-Filipe 53
3.2.6 M-net based semantics of the P-UMLaut project 54
3.2.7 Safety-liveness semantics from Grosu & Smolka 54
3.2.8 Branching time semantics from Hammal 54
3.2.9 Modal Sequence Diagrams 54
3.2.10 Operational semantics from Knapp & Wuttke 55
3.2.11 Thread-tag based semantics 55
3.2.12 Semantics based on CPN 55
3.2.13 Template semantics 55
3.3 Semantic choices in Sequence Diagrams 56
3.3.1 Interpretation of a basic Interaction 57
3.3.2 Introducing CombinedFragments 60
3.3.3 Computing partial orders 62
3.3.4 Introducing Gates 66
3.3.5 Interpretation of conformance-related operators 67
3.4 Summary 74

4 A test language and framework for mobile systems 77
4.1 Testing mobile systems 78
4.1.1 Modeling notations for mobile systems 78
4.1.2 Test platforms for mobile systems 79
4.1.3 A detailed case study: mobile GMP 79
4.2 Test approach and framework 81
4.2.1 The test approach for mobile systems 81
4.2.2 The test framework for mobile systems 82
4.3 The TERMOS language 84
4.3.1 Design choices for the language 84
4.3.2 Syntax of the language 87
4.3.3 Semantics of the language 92
4.4 Tools for the test framework 99
4.5 Summary 101

5 Conclusion and future work 103
5.1 Summary of the research results 103
5.2 Future work 104
5.2.1 Robustness testing of HA middleware 104
5.2.2 Semantic choices in UML 2 Sequence Diagrams 104
5.2.3 A test language and framework for mobile systems 105

A Details on the robustness test framework 107
A.1 XML Schema of the type-specific tool's configuration 107

A.1.1 XML Schema of the types' metadata 107
A.1.2 XML Schema of the functions' metadata 108
A.2 XML Schema of the mutant generator tool's configuration 108
A.3 XML Schema of the OS call wrappers tool's configuration 109

B Details on UML 2 Sequence Diagrams **111**
B.1 XMI of an example Interaction 111

C Details on the TERMOS language **113**
C.1 Abstract syntax of the event view 113
C.2 The unwinding algorithm 113
C.3 A more complex example for the unwinding 116

Bibliography **121**
Publication list 121

Chapter 1

Introduction

Testing is an essential but complex and resource-consuming task in software development. IEEE defines testing as an "activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component" [IEE10]. Figure 1.1 depicts a high-level view of testing: test cases are created from the specification of the system, these are executed, and verdicts are assigned describing the outcome, e.g., passed or failed. Of course many important questions need to be answered before testing can be carried out. What part or functionality of the system needs to be tested? In what manner are the outcomes of the tests evaluated? Who decides whether a test passed or failed? To answer these questions several other *test artifacts* are needed besides test cases.

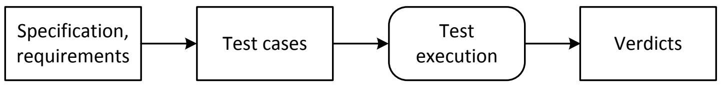


Figure 1.1: High-level view of the testing process

Figure 1.2 presents the artifacts used in testing in more detail[1]. From the high-level user requirements and the specification of the system *test requirements* are derived that define how the system should or should not behave in a certain situation. The *test approach* collects how and when the testing should be conducted, e.g., what processes, techniques, test levels and tools should be used. *Test purposes* describe what part or functionality of the system should be covered by testing. Later, *test case* specifications are created in which the inputs, predicted results, and set of execution conditions are specified. The expected output for a given input is obtained from a *test oracle*. The test cases are implemented, and with the help of *test adapters* they are executed in a *test execution environment*, which contains the *system under test* (SUT) and potentially some test doubles (drivers, stubs, etc.) that simulate the other components and the environment of the system. During the test execution *test traces* are recorded, which can contain the responses of the SUT, details about the changes in the test environment, etc. Finally, the outcomes of the test executions are evaluated, and *verdict*s are assigned. The set of possible verdicts is usually pass, fail, error (there was an error in the test execution environment itself), and inconclusive (neither a pass nor a fail verdict can be determined). These artifacts and the tools supporting them are combined into a *test framework*.

[1]As testing is such a general term, these basic concepts have been defined in many ways. The dissertation follows mainly the IEEE terminology [IEE10] extended with the ISTQB glossary [IST10].

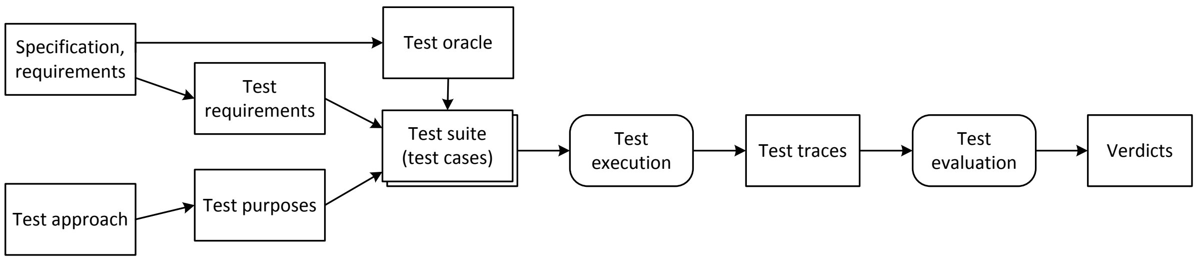


Figure 1.2: General test artifacts

Testing has an extensive literature (just to name a few well-known books [Bei90; Bec02; MS04]), numerous methods and techniques have been proposed to test different types of systems. In order to apply these methods, suitable languages are needed that can be used to precisely design and describe the above test artifacts.

The research presented in this dissertation was focused on (i) what *languages* can be used to describe these test artifacts especially in application domains in which a proper solution is missing, and (ii) using the test artifacts how can *test frameworks* be constructed that can be used for testing in specific application domains.

1.1 Existing test languages and approaches

This section first gives examples of the existing languages used for describing test artifacts. Next, it presents how the Unified Modeling Language (UML), one of the most commonly used languages to model software systems, can be used in modeling test artifacts. Finally, different test approaches are introduced, which will be used in the dissertation.

1.1.1 Examples of languages for describing test artifacts

Depending on the type of the test artifact to describe, several methods and notations have been proposed. For example, test purposes can be described with labeled transition systems [JJ05] or with temporal logic formulae [Hon+01]. Test requirements can be extracted from UML models [BL02]. Test cases can be defined using TTCN-3 [ITU07], test configurations in the ATML language [IEE11]. Test oracles can be expressed as automata [Hes+08] or in SDL [Koc+98].

For describing partial behavior like test requirements or test purposes, a very common approach is to use graphical *scenario languages* [GHN93; PJ04; KSH07]. They provide an intuitive yet powerful notation to express communication between different entities. Several language variants were proposed over the years. The International Telecommunication Union's (ITU) Message Sequence Chart (MSC) [ITU11] was one of the first of such languages. It is widely used, since its first introduction in 1993 it was updated several times. Live Sequence Chart (LSC) [DH01] concentrated on distinguishing possible and necessary behaviors. The dissertation focused on software systems, thus from the possible testing and modeling notations, the UML language was highly relevant.

1.1.2 Using UML 2 for specifying test artifacts

The Unified Modeling Language (UML) [OMG11b] developed by the Object Management Group (OMG) is one of the most commonly used languages to model software systems. UML has extensive tool support, and can be used in many aspects of software development from capturing requirements to specifying deployments. A recent paper by Cook [Coo12] presents a good overview of the history and evolution of the language.

To support the testing activities, a dedicated UML profile was developed. With the help of the *UML 2 Testing Profile* [OMG05] a UML model can specify (i) the test architecture, (ii) the behavior of test cases, and (iii) contents of the test data. The test architecture is modeled typically with stereotyped components for test context, arbiter etc. The behavior of test cases and test procedures are given usually with the scenario language found in UML, namely *Sequence Diagrams*, and the profile offers stereotypes to express default behavior or logging and validation actions. The test data stereotypes are used to define data partitions, and make expressing wildcards, omitted values possible.

The first version of Sequence Diagrams included in UML 1.x was similar to basic MSCs, i.e., it included lifelines representing communicating instances and messages going between lifelines. The next version introduced in UML 2.0 was a major rework; the language was extended with several complex, high-level elements. For example, new notations were added to express alternative or parallel flows. Moreover, what is even more significant from a testing perspective, language constructs were included to express mandatory and forbidden behavior, or messages that can be ignored. However, the meaning of these elements, i.e. their *semantics*, was described only in natural language text fragments, which allowed several different interpretations.

Thus in order to use Sequence Diagrams to describe test artifacts or extend the language to cope with the characteristics of new application domains, first it should be identified what semantic variations exist for the language, and which of them fits for testing related activities.

1.1.3 Test approaches utilized in the dissertation

Testing activities can be differentiated based on what level they operate. Typical categories include module or unit testing (dealing with only one module), integration testing (checking the cooperation of several modules), and system testing (analyzing the whole system possibly taking into account its environment). The dissertation focuses on methods for *system testing*.

There are many approaches that can be applied at system level to test the functionality. One typical categorization, which is common in the protocol testing community, differentiates *active* and *passive testing* [CGP03; AMN12]. In active testing the tests stimulate directly the SUT by providing inputs to it. However, this is not possible in some situations, e.g., when there is no direct interface to the SUT or the SUT operates in a complex environment. In these cases passive testing techniques offer an alternative, where the operation of the system is observed by recording *execution traces*, and then this trace is checked on-line or off-line to determine whether it conforms to the specification. This approach is common in testing distributed systems, where the test framework does not provide constant input to each of the nodes, instead it creates an initial test setup, and later observes the behavior of the nodes through their communication. In the application domains presented in this dissertation both active and passive testing were useful test approaches.

In system level testing usually not only the core functionality, but other *non-functional requirements* are considered. Non-functional requirements include performance or the different attributes of dependability [Avi+04], like robustness or availability. Such testing can be characterized with the following two components of the tests (stimuli); the *workload* triggers the (regular) operation of the system, while the *faultload* contains the different faults and stressful conditions applied on the system. Depending on how these two loads are balanced, different kinds of system properties can be tested, e.g., in API robustness testing only a faultload is executed against the public interfaces of the system, or in stress testing only a high level of workload is applied. One part of the research presented in this dissertation focused on robustness, which is the attribute of dependability that measures the behavior of the system under non-standard conditions. Robustness is defined by IEEE as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [IEE10].

As new application domains emerge and new system attributes become relevant to test, the classic methods have to be evaluated and one should identify whether new challenges have been arisen.

1.2 New application domains

The dissertation focuses on the following new and emerging application domains, which present several new challenges for the testing activities.

1.2.1 High availability middleware systems

Recently availability became a key factor even in common off-the-shelf computing platforms. High availability (HA) can be achieved by introducing manageable redundancy in the system. The common mechanisms to manage redundancy and achieve minimal system outage can be implemented independently from the application in a component called a HA middleware. To standardize the functionality of such middleware systems leading IT companies formed the *Service Availability Forum* (SA Forum) to elaborate the *Application Interface Specification* (AIS) [SAF07]. Different vendors implemented the common specification in their solutions.

With multiple middleware products developed from the same specification the demand to compare the various implementations naturally arises. The most frequently examined properties are performance and functionality, but especially in case of HA products dependability is also an important property to be considered.

The characteristics of HA middleware systems can be summarized as follows.

State-based nature The complexity in testing these middleware implementations comes from the highly state-based nature of these systems: without a proper setup code most of the calls in the public interface result in trivial error messages, this way the valid operation cannot be tested. For example, the health of a component is checked in a callback, which needs to be registered after a connection to the middleware is initialized.

Robustness is a key factor As the availability requirements towards these systems are extremely high, HA middleware systems should handle even the unexpected situations. Typically testing effort is focused on all the valid paths and some of the common invalid inputs. However, in a HA middleware preparing for erroneous inputs is especially important, because an error in one component can render the whole system inaccessible if the middleware is not robust enough.

Testing, among other verification and validation techniques, can be used also to assess the robustness of a system. Specifically the goal of robustness testing is to activate robustness faults (typically design or programming faults) by supplying invalid inputs or presenting stressful environmental conditions.

Although robustness testing of HA middleware was a new research topic, previous robustness testing results from other application domains can serve as guidance. Early robustness testing experiments on command line programs generated a large number of random inputs [MFS90] (a technique known as "fuzzing"). As HA middleware systems have a common interface specification, a more relevant technique is *type-specific* testing introduced in the Ballista project [KDD08]. Here robustness tests were generated for POSIX compliant operating systems using valid and invalid values defined for the data types in the API. This method shall be extended to be used in testing the robustness of state-based middleware systems.

The challenge in describing tests for HA middleware systems lies in that (i) existing test languages mostly focus on conformance and not robustness, (ii) an HA middleware is a complex system that has several inputs that may trigger robustness faults, and (iii) its API is state-based with numerous functions, types and parameters.

1.2.2 Context-aware mobile computing systems

Mobile computing systems involve devices (handset, PDA, laptop, intelligent car, etc.) that move within some physical areas, while being connected to networks by means of wireless links (Bluetooth, IEEE 802.11, GPRS, etc.). Such devices represent an integral part of our life now. The specific characteristics of these systems can be summarized as follows.

Context awareness Context might be "any information that can be used to characterize the situation that are considered relevant to the interaction between a user and an application" [BDR07]. For example, in mobile computing the context can be information collected by means of physical sensors such as location, time, speed of vehicle, or it can be information about network parameters, such as bandwidth, delay and connection topology. These systems have to take into account the actual context in their actions.

Dynamic, evolving environment In mobile computing systems the system structure, the number of mobile devices are not fixed. It varies over time, due to the dynamic appearance, suspension or stopping of nodes. Besides that, connectivity between nodes is also highly dynamic. As the nodes are free to move arbitrarily, they can join or leave the system in an unpredicted manner. Links may be established or destroyed, yielding an unstable connection topology.

Communication with unknown partners in local vicinity In ad hoc mobile networks, a natural communication is local broadcast. It is used as a basic step for the discovery layer in mobile applications (e.g., group discovery service for membership protocols, a route discovery in routing protocols, etc.). In this class of communication, a node broadcasts a message to its neighbors. As the topology of the system is unknown, the sending node does not know a priori the number and identity of potential receivers. Whoever is in transmission range of the sending node may listen and react to the message.

Existing languages presented in Section 1.1.1 were developed to describe mainly static configurations. Object creation or destruction can be depicted in some of the languages; however, these notations are not suitable to express frequent appearance or disappearance of other nodes or nearby objects. There were some works proposing extensions (e.g., [BM04; SE04]), but they concentrated mostly on mobile software agents and logical mobility. Moreover, existing languages focus on the communication between the entities, and do not offer an intuitive way to describe the actual context, i.e., the current state of the environment. Modeling language extensions and adaptation of existing test methods are needed that take into account the specificities of context-aware mobile systems.

1.3 Summarizing the new challenges

As the previous sections illustrated there are several relevant existing test approaches and languages, however they need to be adapted or extended to suit the new application domains. The following challenges summarize the *open research questions*, which have driven the work presented in the dissertation.

Challenge 1: Adapting robustness testing to HA middleware. How can relevant test inputs for a HA middeware be specified in test artifacts to support the automated testing of the robustness of such systems?

Challenge 2: Specifying mobile systems in test artifacts. How can dynamic, frequently changing communication structures and unknown partners be specified in test artifacts in a way that such systems can be later evaluated?

Instead of designing completely new test languages, we tried to reuse existing languages when possible. However, to incorporate new concepts into an existing language, that language should have a clear and precise semantics. From the available scenario languages, we focused on UML 2 Sequence Diagrams. But, as described previously, Sequence Diagrams can have several semantic interpretations. Thus in order to define testing related extensions, first the semantics of UML 2 Sequence Diagrams has to be studied.

Challenge 3: Analyzing the semantics of UML 2 Sequence Diagrams. What semantic choices are available in UML 2 Sequence Diagrams, and what options can be chosen when the language is extended to support the description of test artifacts in a specific application domain?

Therefore the goal of the dissertation was to define test frameworks addressing these challenges, and develop the necessary languages for expressing the various test artifacts in the frameworks.

1.4 Research method

According to [SS07], a research activity can be classified as *basic* ("research for the purpose of obtaining new knowledge") or *applied research* ("research seeking solutions to practical problems"). Moreover, *classical research* (using the scientific method of "formulate hypotheses then check or test these by means of experiments and observations") and *technology research* ("research for the purpose of producing new and better artefacts") can be differentiated. Technology research belongs usually to applied research, and it uses an iterative process: (i) starting with a problem analysis in which the potential needs for the new artifact are collected; (ii) the new artifact is constructed in an innovative way; (iii) the new artifact is evaluated against the initial needs.

The research presented in this dissertation can be categorized as applied, technology research, as its goal is to create better artifacts for solving practical problems. As the artifacts included new modeling languages, an important question was how modeling languages can be constructed.

Language construction To solve the identified challenges, it was required to design new modeling languages and extend existing ones. Engineering a new language is a complex task, in order to precisely define a new modeling language the following artifacts have to be specified [OMG11a].

- *Abstract syntax* defines the main conceptual elements of the language and their relationships. The abstract syntax is meant for automated processing, and nowadays it is usually given using metamodels.
- *Concrete syntax* defines the human interface of the language (e.g., visual or textual notation). The elements of the concrete syntax have to be mapped to the elements of the abstract syntax.
- *Well-formedness rules* define additional constraints on the abstract syntax, which capture more complex conditions that cannot be specified otherwise easily in the abstract syntax.
- *Semantics* define the meaning of the language elements, usually with the help of a mapping to a well-defined semantic domain.

Lets use the language of UML Sequence Diagrams as an example to demonstrate these concepts. According to the abstract syntax, Sequence Diagrams have concepts like message, guard and lifeline; there are several types of messages (synchronous, asynchronous, etc.); and a lifeline can send zero or more messages. The concrete syntax of a message in Sequence Diagrams is a graphical arrow symbol, a lifeline is depicted with a box and a dashed line. A well-formedness constraint defined for message is that the sending event of a self-message has to appear before its receiving event. The semantics is given by mapping diagram fragments to set of event traces.

Defining semantics The semantics of a language defines the language's semantic domain and a mapping from the syntactic elements of the language to constructs of the semantic domain [HR04]. The semantic domain can be anything that has a well-defined meaning, e.g., another modeling or programming language or some well-understood formalism. The semantics does not necessarily define behavior, e.g., the semantics of UML Class Diagrams should also be defined. Depending on the actual context and needs, the definition of the semantic domain and the semantic mapping can be given with different levels of formalization (e.g., natural text, mathematics or automatic tools). The following two styles are relevant for graphical modeling languages [LRS11]:

- *Denotational*: With a denotational semantics the meaning is given with a mathematical function that maps a model directly to its meaning, called its denotation.
- *Operational*: With an operational semantics the meaning is given as a sequence of computational steps that results from processing the input model.

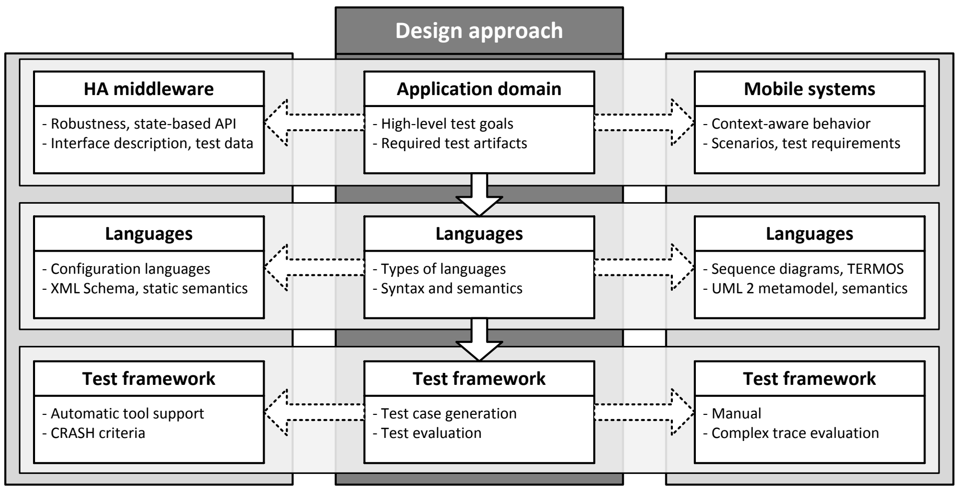


Figure 1.3: Common design approach

Design approach Although the characteristics and the challenges of the two applications domains are quite different, the approach used for designing the test frameworks was similar. Figure 1.3 illustrates the common steps utilized: first, the high-level goals and required test artifacts were collected; next, the required test languages were designed; finally, based on the languages a test framework was developed. Moreover, the figure depicts the focus points in each of the two application domains. For example, HA middleware systems required simpler configuration languages to describe interfaces, but the tools of the framework offer test generation. For mobile systems more complex languages were needed to express scenarios, however, the framework concentrated on evaluation of test traces. The specialties of the two application domains will be elaborated in the next chapters.

1.5 Contributions and structure of the dissertation

The main contributions of the dissertation are centered around the above challenges, and the dissertation's chapters follow this organization. The work presented here was part of joint collaborations in various research projects. After summarizing the general results I formulate my own contributions.

Chapter 2 deals with robustness testing of HA middleware (Challenge 1). We designed and implemented a robustness test framework, and conducted experiments on several middleware implementations to compare their robustness. This work was a joint research with Francis Tam from Nokia Research Center, whose contributions included the overall directions of the investigation and discussions on the tool concept [Tam09]. Related publications are the following: [4], [9], [12], [16], [19], [20], [21].

Thesis 1 *Following a systematic method I identified potential activation modes of robustness faults (including activation through stateless API, stateful API and underlying services), designed languages to represent the related test artifacts, and developed algorithms for tools that use these languages to generate test data. I implemented the languages and tools in a test framework, which can compare the robustness of standard specification-based middleware implementations.*

Chapter 3 presents the semantic choices of UML 2 Sequence Diagrams (Challenge 3). We surveyed the proposed formal semantics for Sequence Diagrams, categorized the semantic choices and options found in them, and developed a structured framework to represent these choices. This work was partially carried out in the ReSIST EU research project [ReS09]. Related publications are the following: [1], [23].

Thesis 2 *I identified and categorized the semantic choices and available options in UML 2 Sequence Diagrams. I gave a structured framework with an easy to use feature-model like representation of the available options that can be used to adapt the semantics of the language to a specific purpose.*

Chapter 4 contains the results on testing mobile systems (Challenge 2). We proposed the necessary language extensions for graphical scenario languages to represent the characteristics of mobile systems, designed a test requirement language called TERMOS using these extensions, and developed a test framework that can check execution traces against these requirements. This work was partially carried out in the HIDENETS EU research project [HID09]. Developing the test language and framework for mobile systems was a joint research with Nicolas Rivière and Minh Duc Nguyen from LAAS-CNRS. Designing and developing a tool called GraphSeq for matching parts of the test traces with test requirements was a contribution in Minh Duc Nguyen's PhD dissertation [Ngu09]. Áron Hamvas, an MSc student I supervised at BME, created a tool [Ham10] for TERMOS. Related publications are the following: [5], [8], [13], [18], [24].

Thesis 3 *I designed a test requirement language that can be used in the domain of mobile systems. I defined the syntax of the language using extensions to the UML Sequence Diagrams' metamodel, and its semantics using an automaton-based formal operational semantics. The language is capable of expressing local broadcasts and changes in the communication topology, and has the necessary syntactic and semantic choices to make the specified requirements checkable.*

Chapter 2

Robustness testing of HA middleware

A high availability (HA) middleware is a software component that implements common techniques to protect applications from failures resulting in outages. Typical techniques include introducing and managing redundancy in the system, e.g., adding warm standby computers or duplicated communication channels. The Service Availability Forum (SA Forum) consortium was formed to develop open specifications on the interfaces of such middleware systems to increase the interoperability of products and applications from different vendors [TT12]. SA Forum's Application Interface Specification (AIS) [SAF07] defines the interface between the HA middleware and the custom application. It is a C language interface partitioned into a number of services. For example, the Cluster Membership Service (CLM) provides a consistent view of the computing nodes, while the Availability Management Framework (AMF) manages the life-cycle of redundant components. Several implementations have been developed for the specifications, e.g., the open-source OpenAIS and OpenSAF or commercial ones like GoAhead's SAFfire and Fujitsu Siemens Computers SAFE4TRY[1].

In case of HA products the *robustness of the implementation* is also an important property to be considered. Robustness failures in the middleware can be activated by poor quality application components, and one such component may render the whole application inaccessible. The common API used by different implementations makes it possible to develop a single robustness test framework and compare the different implementations. However, generating an effective test suite, executing it and evaluating the results usually needs a lot of manual work. As AIS provides a semi-formal description of the interfaces, which can be used to gather the possible inputs and output acceptance conditions, it allows automated test construction and test execution.

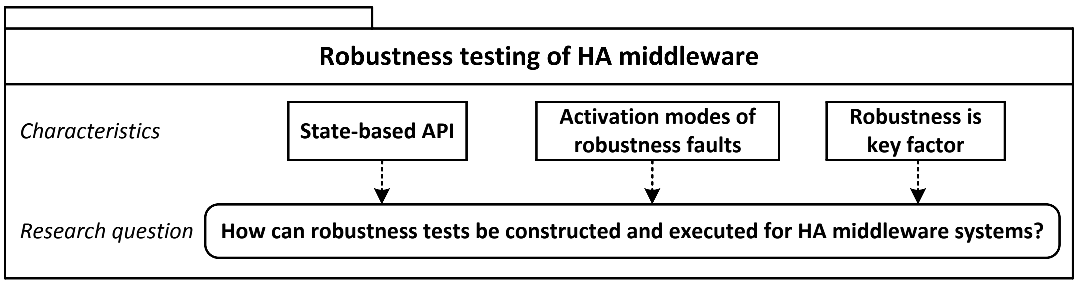

Robustness testing of HA middleware
Characteristics
State-based API
Activation modes of robustness faults
Robustness is key factor
Research question
How can robustness tests be constructed and executed for HA middleware systems?

Figure 2.1: Research question of the chapter

[1]Note that some of these products are not available anymore, e.g., GoAhead was acquired by Oracle in 2011.

Thus, the goal of the research presented in this chapter was to define a test approach for evaluating and comparing the robustness of different HA middleware implementations (Figure 2.1).

The chapter starts with an overview of existing robustness testing experiments, and collects the different test techniques (Section 2.1). Next, Section 2.2 describes the developed test approach: based on the potential inputs for activating the robustness faults, different test techniques are identified for each of the activation modes. Section 2.3 presents the design and implementation of the test framework, including the languages for expressing the test artifacts and the tools for generating tests. Section 2.4 evaluates the test approach and framework using case studies on several middleware implementations. Finally, Section 2.5 summarizes the contributions of the chapter.

2.1 Robustness test techniques

Robustness is an attribute of resilience that measures the behavior of the system under non-standard conditions. Robustness is defined in IEEE Standard 24765:2010 as the degree to which a system operates correctly in the presence of *exceptional inputs or stressful environmental conditions* [IEE10]. To further refine the difference between robustness and resilience Avizienis *et al.* defined robustness as "dependability with respect to external faults, which characterizes a system reaction to a specific class of faults" [Avi+04].

The goal of *robustness testing* is to activate those faults (typically design or programming faults) or vulnerabilities in the system that result in incorrect operation, i.e., robustness failure, affecting the resilience of the system. Robustness testing mostly concentrates on the internal design faults that can be activated through the system interface. Usually, the results of robustness tests are not checked against a detailed functional specification, just against a set of simplified failures modes. One such classification is the CRASH criteria [KD00]: *Catastrophic* (the whole system crashes or reboots), *Restart* (the application has to be restarted), *Abort* (the application terminates abnormally), *Silent* (invalid operation is performed without error signal), and *Hindering* (incorrect error code is returned — note that returning a proper error code is considered as robust operation). Typically robustness is expressed as a collection of measures; such measures can include the ratio of test cases that expose robustness faults, or the number of robustness faults exposed by a given test suite.

If the robustness of a complex systems is tested, then usually the tests consists of two components: the *workload* triggers (regular) operation of the system, while the *faultload* contains the different effects of external faults and stressful conditions applied on the system.

In the past decades many research projects were devoted to the robustness testing of a specific application or application type. The early methods were mainly based on hardware fault injection, but later the research focus moved to software-implemented techniques. This section introduces the main milestones, which can be connected to the introduction of new *test techniques.* Note the techniques presented here were usually used much earlier for other purposes (e.g., physical fault injection was developed before the '90s), this section concentrates on using a technique specifically for testing robustness.

2.1.1 Injecting physical faults

Early work on robustness testing used *fault injection* (FI) tools to induce or simulate the effects of various hardware related faults. Here a clear distinction shall be made between the purposes of general FI and FI for robustness testing. The general technique assesses the ability of a system or component to handle internal hardware or software faults. In a robustness test framework, FI can be used to assess the ability of a component to handle the effects of external faults (that reach the system through its interactions with its environment) that are triggered by injecting faults into the environment (e.g., interacting components or underlying layers) while keeping the tested component intact. In this way also the robustness of error detection and error handling mechanisms (considered as components to be tested) can be investigated. FIAT [Bar+90] or FTAPE [TIJ96] are examples for FI tools that are reported to be used for such robustness testing purposes.

2.1.2 Using random inputs

One of the first robustness test techniques was the generation of *random input* for the system. Random inputs are easy to generate, there is a chance that robustness faults are activated by them, and due to the simple acceptance criteria (crash/hung is checked) there is no need to generate reference output.

Fuzz [MFS90] was one of the first tools supporting this technique. It was utilized in three series of experiments to test the reliability and robustness of various applications. In 1990, utility programs on seven variants of Unix operating systems were tested. In 1995, the tests were repeated to check whether robustness of these utilities had been improved and support to test X Window applications were added. Lastly, in 2000, Fuzz was used to test 30 GUI applications on Windows NT. Although the method used was really simple, it detected numerous robustness errors, namely 40% of the Unix command line programs and 45% of the Windows NT programs crashed (terminated abnormally) or hung (stopped responding to input within a reasonable length of time) when called with random input data.

Although random testing is a basic technique, it proves to be useful even for modern COTS software systems. The tests in Fuzz were reapplied to MacOS in a study prepared in 2007 [MCM07] with the following results: 10 command line utilities crashed out of the 135 utilities that were tested (a failure rate of 7%), 20 crashed and 2 hung out of the 30 GUI programs tested (a failure rate of 73%). Thus, it turns out that robustness testing using random inputs is still a viable technique as robustness of common software products has not been significantly improved in general in the last fifteen years.

Fuzzing is extensively applied to security related testing, as presented in the book [TDM08].

2.1.3 Using type-specific tests

The robustness tests can be further refined by using specific invalid inputs as recommended by the classical test design techniques [MS04]. To minimize the amount of manually created test cases a *type-specific method* was introduced. The basic idea is that valid and invalid values, or ranges of values are defined for the data types used in the system's interface functions, and the robustness tests are generated by combining the values for the different parameters. The size of the invalid input domain can be further reduced by utilizing inheritance between the types to test.

The Ballista tool [KD00] introduced this approach to compare the robustness of 15 POSIX operating systems using a test suite for 233 function calls. The general goal of the research was to implement methods to measure the robustness of the exception handling mechanism of systems. The results could be used to evaluate the dependability of a system and characterize how it responds to the failures of other components. In an experiment performed on the Safe Fast I/O library, the performance drawback of robustness hardening was also measured. The tests showed that the performance penalty of proper data validation and parameter checking was fewer than 2%. A good summary of the experiences gained using Ballista can be found in [KDD08].

2.1.4 Testing object-oriented systems

The type-specific technique mentioned above can be enhanced in object-oriented (OO) systems with the help of automatically building a parameter graph with the type structure. The parameter graph describes how the specific object types used as parameters in method calls can be generated as results of calling constructors or public methods of other classes. This way the generation of an invalid object (needed to test a given method) can be traced back to the call of another method (possibly having parameters of simpler input types).

The JCrasher tool [CS04] creates robustness tests for Java programs automatically by analyzing which methods could return a type needed for the actual parameters. It examines the type information of the set of Java classes constituting the application and constructs code fragments that will create instances of different types to test the behavior of public methods with random or invalid data.

In OO applications the testing of exception handling is an important aspect of assessing the robustness of the fault handling and recovery code. Exception flow analysis and testing exception-catch paths is presented in [Fu+05].

2.1.5 Applying mutation techniques

Code mutation techniques [DeM+88] can be also applied to generate robustness tests. Starting from a valid code, e.g., a functional test or an application using the system's interfaces, mutation operators can be applied, which resemble the typical faults causing robustness problems (e.g., omitting calls, interchanging calls, replacing normal values in parameters with invalid values). One such approach is [DM02], where the machine code of device drivers were mutated according to typical programming errors.

Mutation and extension of valid test sequences may also help in state-based systems or components to cover more states and transitions than in case of stateless API testing. In [Lei+10], first a set of paths is generated to cover state transitions of the tested component, and normal test cases are applied to traverse these paths and bring the component into specific states. In each state, the available methods are called with invalid inputs to test the robustness in that state. This approach is motivated by the fact that complex components may fail differently in different states.

2.1.6 Model-based robustness testing

The increasingly popular model-driven development paradigm led to the idea of model-based testing (using models as formal or semi-formal specification for testing purposes) [Bro+05; Net+07] and also model-based automated test generation. Naturally, model-based test generation can be tailored to create robustness tests by looking for extreme values and conditions on the basis of interface definitions, pre- and post-conditions, invariants, and constraints fixed in the design model. Model-based testing is currently a very active field of research; here we mention only a few techniques and tools that are relevant to testing robustness.

The first test generation approaches utilized formal specifications and functional models (B, Z, LOTOS, etc.). Constraint-solving techniques were applied to generate boundary values of input domains as well as the corresponding test cases. In state-based formalisms, e.g., in IOLTS [FMP05], path searching and model mutation (on the basis of fault models) were applied in order to find tests for concrete robustness criteria. Timed behavior was modeled and tested using timed automata [FRT10] or extended interoperability models [Mat+09].

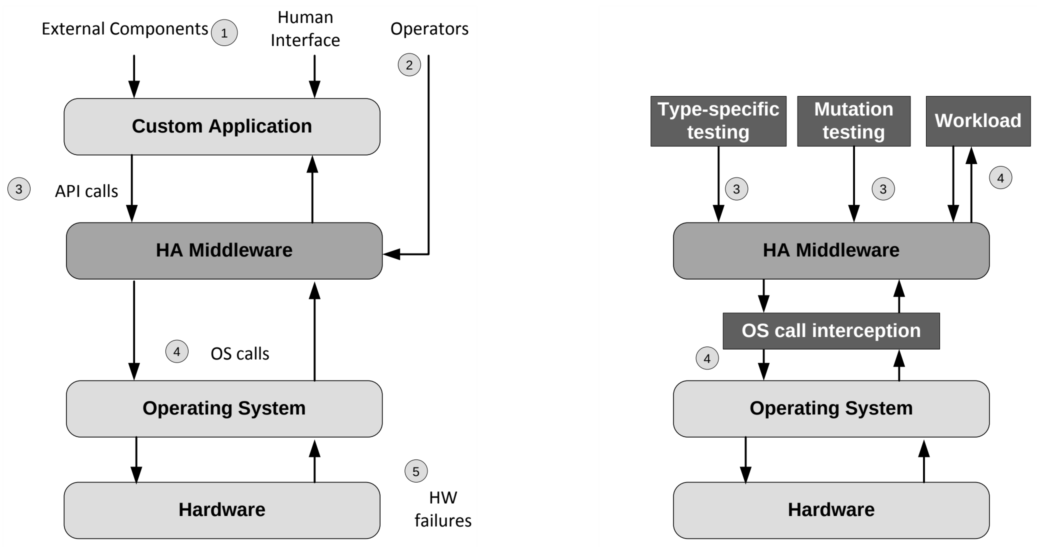
In protocol testing FSM-based models are used for a long time. The robustness of an implementation were tested by supplying messages representing undefined transitions in the model [BP94]. In a more recent experiment SDL was the primary modeling language used for generating robustness tests in case of communication protocols [SRC07]. In another work [PK07], finite state machine models of communication protocols were extended and faulty protocol data units were generated on the basis of a stress operational profile in a statistical approach to model-based robustness testing.

In UML-based designs the Object Constraint Language (OCL) was used to specify valid domains, this way providing input information also for robustness testing. Typical examples of UML-based test generator tools that support (a subset of) OCL were LTG/UML [UL06] and ParTeG [WS08]. Adding all robustness-related information (e.g., error-handling, invalid transitions) could result in a complex, hard to read model. To counter this [ABH12] recommended a methodology that uses aspect-oriented modeling; robustness behavior is modeled as separate aspect state machines that are woven together with the behavioral model later.

Model-based configuration and execution of robustness testing is complementary to model-based testing. In [OM09], a framework was presented that fits to the model-based development approach by offering to the tester the model of the tested application (using UML class diagram model elements) and domain-specific extensions that allow the configuration of fault injection and robustness testing experiments. The modifications that are required in the environment of a tested component for robustness testing are implemented automatically (using a Java bytecode manipulation technology) on the basis of the model extensions.

2.2 Activation modes of robustness faults and the test approach

The first step of developing the test approach in the case of a "black box" AIS middleware was to identify the possible sources of inputs that can activate robustness faults. These inputs are depicted in Figure 2.2(a), considering a typical computing node of a HA distributed system.



(a) Sources for activating robustness faults (b) Robustness test techniques

Figure 2.2: Overview of the test approach for HA middleware systems

1. *External components and human interface*: They affect the operation of the application, thus their effects reach the HA middleware only indirectly (through normal, erroneous or missing

API calls).

2. *Operators*: In general, operator errors appear as erroneous configuration of the middleware and erroneous calls using the specific management interface.
3. *API calls*: The calls of the application components using the public interfaces of the HA middleware can lead to failures if they use exceptional values in parameters or try to call a certain function when the component is not in the required state.
4. *OS calls*: The robustness of a system is also characterized by its ability to handle the exceptions or error codes returned by the OS services it uses.
5. *Hardware failures*: The most significant hardware failures in a HA system are host and communication failures (that have to be tolerated in the normal operating mode of the HA middleware) and lack of system resources.

These sources can be categorized as direct sources (API calls, OS calls, operators) and indirect sources (externals components, human interface, HW failures). The effects of the indirect sources could only reach the middleware through one of the direct ones. Regarding the direct sources the following decisions were made:

- The standardized middleware API calls are considered as a potential source of activating robustness faults as they represent faults in the applications, in external components used by the applications and human interaction also. The challenge in testing the API calls was that most of the AIS interface functions are state-based, i.e., a proper initialization call sequence, middleware configuration and test arrangement is required, otherwise a trivial error code is returned.
- The failures of the OS system calls were included as they do not only represent the faults of the OS itself (which has lower probability for mature operating systems), but failures in other software components, in the underlying hardware and in the environment could also manifest in an error code returned by a system call. Possible examples of such conditions are writing data to a full disk, communication errors when sending a message, etc.
- Operator errors cause also a significant part of service unavailability, however, the configuration of the HA middleware and the system management interface were still under standardization by the SA Forum, thus they were not included in the robustness test framework.

The developed test approach focused on the direct sources, as the thorough testing of the potential failures caused by the direct sources would cover a significant part of the failures induced by the indirect sources.

Test approach Generally, testing the robustness of a component could have three different targets: testing the robustness of (i) the component's stateless API, (ii) the component's state-based API, and (iii) handling the failures of the used lower-level services. As we could see from the description of the selected sources, which can activate robustness faults, all these three targets are relevant in case of a HA middleware.

The following test approach was developed that utilizes a combination of three techniques to cover these targets, as depicted on Figure 2.2(b). Table 2.1 illustrates the relationship between the activation sources, test targets and the selected test techniques.

- *Type-specific testing*: The robustness of the middleware's API functions should be tested in case of invalid values are used as parameters. This requires calling all the functions in the API of the middleware with a thorough combination of the possible valid and invalid values. The type-specific testing technique is used to construct such a robustness test suite, as this technique offers a systematic method to define the necessary valid and invalid test values.

Table 2.1: Relationships of sources, targets and techniques in the test approach

Activation source	Test target	Test technique
API calls	stateless and state-based API	type-specific testing
API calls	state-based API	mutation-based testing
OS calls	lower-level services	OS call interception

- *Mutation-based sequential testing*: Some specific states of the middleware can only be reached by complex call sequences; nevertheless, the robustness of the API functions should be tested even from these states. Thus first the middleware should be directed to these states, then its functions should be called with invalid inputs. The functional test suites provided by the vendors of the HA middleware could be used to reach these states as these test suites usually cover all the important states of the middleware. Therefore exceptional test sequences are constructed by using mutation operators on functional test suites that represent typical faults (e.g., changing the sequence of test calls, modifying parameters or function names).

- *OS call interception*: In order to test how the middleware reacts to the failures of the consumed lower-level services, the calls to the OS services should be intercepted and their return values should be modified. This interception can be realized with the help of a wrapper component that is placed between the middleware and the operating system libraries. Furthermore, this kind of test activity requires a workload application that drives the middleware in a way that the full range of the utilized OS calls could be observed and intercepted.

Thus the robustness faults activated by the API calls are covered by type-specific and mutation-based testing, while the faults activated by OS calls are tested by OS call interception.

The recommended test approach uses the following method to evaluate of the outcomes of the tests. Recall, that the results of robustness tests can be categorized according to the CRASH criteria (Section 2.1). A widely accepted simplified approach is that first only the obvious robustness failures are recognized: Catastrophic, Restart and Abort. These classes of results can be easily detected in the current setting also, e.g., the middleware crashes, a segmentation fault occurs in the tested application or the test does not finish after a reasonably long timeout. The other two classes (Silent, Hindering) would require much more effort to recognize, as they need the proper definition of expected return value for every test case.

To develop a test framework that can implement the above defined test approach, the following systematic method was designed that makes it possible to create the necessary languages and automatic tools for a given test technique.

1. First, the necessary test artifacts and their requirements are collected.

2. Next, the required test languages describing the test artifacts are constructed.

3. Finally, automatic tools are developed that can generate the test artifacts from descriptions given using the test languages.

The next section presents how this method was used to construct a robustness test framework that uses type-specific testing, mutation-based testing and OS call interception.

2.3 Robustness test framework and tools

Taking into consideration the potential sources of activating robustness faults, we developed a set of tools to assist in the activation of these faults by generating proper test values and performing the test calls. As the AIS specification is quite low-level, i.e. it specifies C functions and types directly, the test framework is aligned to this abstraction level. For example, the languages used in the tools contain references to function calls or include code fragments, and they have simple textual syntax. We selected XML as the basic language format, as it can be conveniently processed, and the abstract syntax can be easily specified with XML Schema (XSD).

2.3.1 Type-specific testing

The part of the test framework presented in this section concentrates on calling the functions in the API of the HA middleware with systematic combinations of exceptional values. In type-specific testing instead of defining the exceptional cases one by one for each API function, the exceptional values are defined with regard to the parameter types that are used in the functions. The section first collects the required test artifacts, then designs the languages describing these tests artifacts, and finally presents a tool that can generate an executable test suite based on these languages.

2.3.1.1 Required test artifacts

The final test artifacts required in type-specific testing are the *test programs* implementing the test cases that call the API functions of the middleware with a given combination of exceptional values. In a type-specific approach the following additional artifacts are needed to generate these test programs.

- *Functions to test*: the list of the functions and their signature define the subset of the API to test.
- *Types used in the functions*: as type-specific testing focuses on the individual types present in the functions to test, a list of these types and their properties are required.
- *Test values for each type*: the list of the values used for each type in the test cases should also be collected.

Thus the required test artifacts are determined in the end by the requirements of the different exceptional test values to specify.

Test values In case of an AIS-based HA middleware the test values can be analyzed and designed as follows.

- *Exceptional values*: For simple types, e.g., numbers and enumerations, values recommended by traditional test techniques [MS04] were selected, like nominal values, boundary values and values outside the domain of the given type. For more complex types the exceptional values can be extended with (i) syntactically incorrect values, e.g., string not in the format A.B.C.D for an IPv4 address, (ii) semantically incorrect values, e.g., non-existing version number, and (iii) values used in invalid context, e.g., not initialized handle[2].
- *Valid values*: If only exceptional values are used in the testing, then the effect of a simple parameter can be masked by the exceptional values of the other parameters. Thus it is important to include valid values for each type, and test the possible combinations of valid and invalid test values. (Note that this has the additional benefit that it simplifies later fault localization.)

[2]In AIS a *handle* is a reference between the application and the middleware, which is used in every subsequent invocation after the initialization.

- *Chaining of types*: It was often the case that a valid value for some types required valid instances of other types (e.g., to define a registered component in AIS, first a valid handle is needed). Thus the definition of the test values had to include references to other types or non-trivial initialization code.
- *Complex structures*: Some types used in API functions are complex structures. For example, `SaAmfProtectionGroupMemberT` is a structure consisting of a component name (`SaNameT`), a HA state (`SaAmfHAStateT`) and a rank (`SaUint32T`). Constructing exceptional values from all possible combinations of the basic types in these structures would result in far too many values, because many structures are built from more than four basic types and the AIS functions have on average two or three parameters. Thus, in the case of complex structures the following systematic method was used. For each member one invalid value is selected. Test values are assigned to the complex structure, where all members are valid and where only one member has an invalid value while the others have valid values.
- *Inheritance*: Inheritance among the types helped to reduce the number of type-specific exceptional values to be defined. Namely, exceptional values of an ancestor type are not defined again in an inherited type, because the tool will use them automatically: the exceptional values specified in the ancestors are also applied recursively as test values in the descendants. In this way, in each type only the specific values shall be defined. The example in Table 2.2 illustrates how this method can be applied in the case of AIS types. In this case `SaAmfHandleT` would inherit the values specified for `BaseType` and `SaUint64T` also.

Having collected the necessary test artifacts and their requirements, the languages used for specifying these artifacts can be designed.

Table 2.2: Using inheritance when defining the test values for types

Type	Parent	Test values
`BaseType`	-	not initialized
`SaUint64T`	`BaseType`	0; MAXINT
`SaAmfHandleT`	`SaUint64T`	initialized AMF handle; already finalized handle
`SaClmHandleT`	`SaUint64T`	initialized CLM handle; handle with no callbacks registered

2.3.1.2 Defining the languages for the test artifacts

As described in the previous section a language is required to define the metadata of the functions to test, the metadata of the types used in those functions, and the test values for each type.

Function metadata The requirements for the description of the API functions are quite simple, it should include the return value and the parameters for a given function. The language was specified in a way to simplify the subsequent automatic processing. The full abstract syntax of the language can be found in Appendix A.1.2. The following elements can be specified in the language:

- *ReturnType*: the type of the function's return value;
- *Parameters*: contains a *Parameter* child element for each of the function's parameters;
- *ParameterOrder*: the position of the parameter (included to make the processing easier);
- *IsPointer*: identifies directly whether the parameter is a pointer or not;

```xml
<Function name="saAmfFinalize">
  <ReturnType>SaAisErrorT</ReturnType>
  <Parameters>
    <Parameter>
      <ParameterOrder>1</ParameterOrder>
      <ParameterName>amfHandle</ParameterName>
      <ParameterType>SaAmfHandleT</ParameterType>
      <IsPointer>true</IsPointer>
      <Type>in</Type>
    </Parameter>
  </Parameters>
</Function>
```

Listing 2.1: Example function metadata

- *Type*: can have the value of “in”, “out” or “in/out”.

An example for using the language can be seen on Listing 2.1.

```xml
<Type>
  <Name>SaAmfCallbacksT</Name>
  <ValidValueMethod generate="true" validValueIndex="2"/>
  <PointerMethod generate="true"/>
  <ParentName value="BaseType"/>
  <IncludeFile fileName="CallBackMethods.c" />
</Type>
```

Listing 2.2: Example type metadata

Type metadata The following requirements were identified for the language to describes the types used in the API functions. It should be able to

- define the inheritance between types;
- include complex initialization code;
- offer a way to return an instance of the given type representing a valid value;
- make it possible to access the test values for the given type from the test values of other types.

The full abstract syntax of the language used to describe types can be found in Appendix A.1.1 as an XML Schema. The definition of the elements of the language are the following:

- *Name*: the name of the type;
- *ValidValueMethod*: designates that a method should be generated that returns a valid value for this type;
- *PointerMethod*: initiates the construction of a method to access test values via pointers, which can be used in other types;
- *ParentName*: if this element is present, then all test cases of the given ancestor type are re-used in this type;
- *IncludeFile*: specifies a source file that contains common initialization code for the test values of the type.

Listing 2.2 shows an example for describing the `SaAmfCallBacksT` type.

Test values The most important requirement against the test value definition was that it should offer a very flexible description, where even complex logic or even calls to other parts of the middleware can be specified. Thus instead of trying to design a complicated high-level modeling notation, the valid and exceptional test values are stored in stand-alone files as C code snippets. Storing test values as C code snippets makes it easy to define values that use complex call sequences for initialization. An example for such a test value can be seen on Listing 2.3. Note that %I% is a placeholder, that will be later replaced when these test artifacts are processed.

```c
// values[%I%]: test value representing a registered component name
char compName[15] = "comp_b_in_su_x";
values[%I%].length = strlen( compName );
memcpy( values[%I%].value, compName, values[%I%].length );

SaAmfHandleT handle = generateValidSaAmfHandleT();
saAmfComponentRegister(handle, &(values[%I%]), NULL );
```

Listing 2.3: Example test value as a C code snippet

2.3.1.3 Template-based type-specific test generator tool

The template-based type-specific test generator (TBTS-TG) tool should process the test artifacts defined with the above languages and automatically generate an executable robustness test suite. The TBTS-TG tool consists of two components (see Figure 2.3). The first component creates a library of test values for each of the types. The second one creates the actual test case programs, which will call the API function with combinations of the test values. These two components together generate the full test suite, that can be later executed on the middleware implementations.

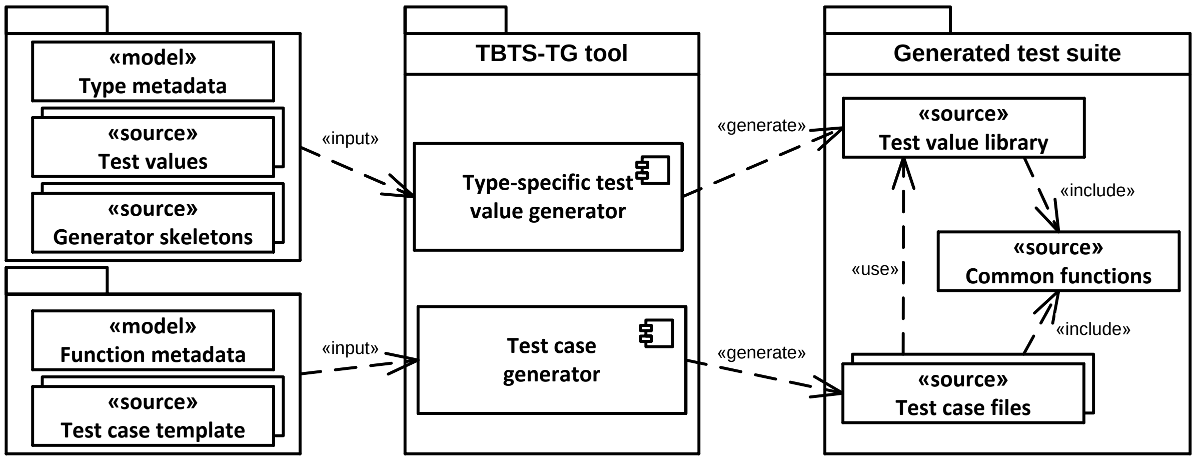


Figure 2.3: Architecture of the type-specific test tool

Test value generator The first component uses the type metadata, the definitions of the test data a C code skeleton to generate the test value library. Algorithm 2.1 contains the pseudo-code of the test value generator's behavior.

Algorithm 2.1: Generating the test value library

```
Input: $T$ type metadata description
Input: $V_1, \dots V_n$ set of test values for each type
foreach $t \in T.Types$ do
    create the «t»* generate«t»() function
    foreach $v \in V_t$ do
        insert $v$ in the function's skeleton
    if $t.ValidValueMethod.generate$ then
        create the «t» generateValid«t» function
    if $t.PointerMethod.generate$ then
        create the «t»** generatePointer«t» function
```

Listing 2.4 shows an example of a method in the test value library created by the TBTS-TG tool for the `SaNameT` type. It contains invalid values representing typical programming errors (0 and 1), a syntactically invalid value (2), a semantically invalid value (3), a value used in an invalid context (4) and a valid value (5). Note that the tool is flexible enough to create complex test values, e.g., in (5) a custom code gets the actual component's name using the middleware's API.

```c
SaNameT* generateSaNameT(int * numberOfGenerated)
{
  static SaNameT values[6];
  *numberOfGenerated = 6;

  // 0 - not initialized

  // 1 - not valid length
  values[1].length = strlen( "string1" ) + 20;
  memcpy( values[1].value, "string1", values[1].length );

  // 2 - name not specified in LDAP DN format
  values[2].length = strlen( "aisRobustnessComp" );
  memcpy( values[2].value, "aisRobustnessComp", values[2].length );

  // 3 - not defined component name
  values[3].length = strlen( "safComp=UnknownComponent" );
  memcpy( values[3].value, "safComp=UnknownComponent", values[3].length );

  // 4 - service unit name instead of a component name
  values[4].length = strlen( "safSu=aisRobustnessSU" );
  memcpy( values[4].value, "safSu=aisRobustnessSU", values[4].length );

  // 5 - component name belonging to this process
  SaAmfHandleT handle = generateValidSaAmfHandleT();
  saAmfComponentNameGet( handle, &(values[5]) );

  return values;
}
```

Listing 2.4: Example type-specific test value generator

If a new type is added to the API, only its XML type metadata and the C code snippets with the test values have to be added, and all the other test codes can be regenerated with the tool.

Test case generator In the next step the tool creates the actual test programs for each of the function defined in the API to test. It is a straightforward step, which uses the function metadata and a C code test case template as inputs.

Figure 2.4 illustrates the behavior of generated test program using a UML 2 Activity Diagram. Before starting the test calls the test program waits for a predefined time to let the middleware finish all initialization. Next, the test value combinations of the function under test's parameters are enumerated (note, because some of the AIS functions have a very large number of parameters, there is a configurable upper limit for the calls to execute). The actual call is performed in a newly forked child process. The test program waits for (i) the child process to finish, (ii) a signal representing a segmentation fault in the child process, or (iii) a timeout. After that, it logs the outcome, and gets the next parameter value combination.

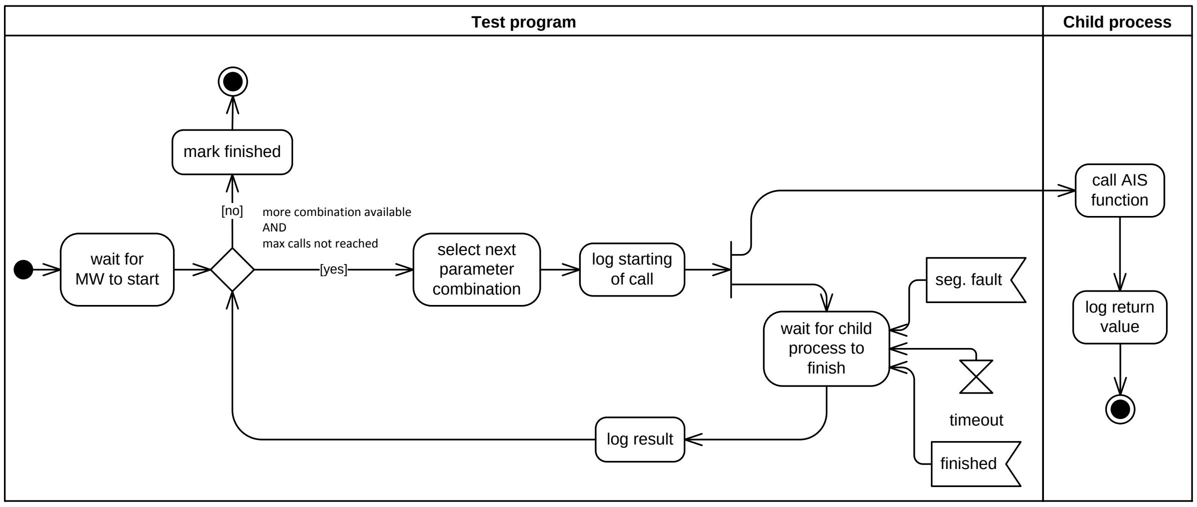


Figure 2.4: Overview of the generated test program's behavior

Once the input generators and the test program sources are created by the TBTS-TG tool, they are compiled and linked with a utility library, which contains functions for logging the results. The generated test suite is self-contained, i.e., it does not depend on the TBTS-TG tool; it is pure C code, which can be easily executed on the different middleware implementations.

Executing the test suite The first version of the test suite consisted of standalone C programs that called the AIS API functions directly (outside of the AMF framework). However, in more recent middleware implementations the AMF functions cannot be called from a process that was not started by AMF. Thus, in the current version the AMF service of the middleware starts the test programs configured as SA-aware components. The test cases were executed by forking a new process for each test case – in order to minimize impact of the test cases on each other. To further minimize the impact of test suites on each other (here test suites consist of a group of tests that focus on the same API function), the middleware was restarted between two suites.

To support the automatic execution of the test suite a test execution engine was prepared. This engine runs the same test programs on each HA middleware, only the following tasks are implementation-dependent (as these are not standardized by the SA Forum): (i) construction of an implementation-specific configuration file on the basis of a common abstract configuration (which consists of one service group and one service unit containing the actual test case as a single component), and (ii) restarting the middleware between the runs of the test cases.

2.3.2 Mutation-based sequential testing

While the TBTS-TG tool tests mostly individual functions, mutation-based techniques could be used to generate tests with complex call sequences. The basic idea is that *mutation operators* representing external effects (including the faults of the external components) that can activate robustness faults in the tested component, like omitting a call or changing the specified order of calls, are applied to valid functional test programs that use the HA middleware. In this way a large number of complex robustness test cases can be obtained automatically.

The advantage of using mutation-based techniques in the robustness testing of HA middleware is that it can handle the *state-based* nature of the middleware. The original call sequences in the source code to mutate can lead the middleware to a state, which is otherwise not trivial to reach, e.g., a component is initiated and a restart was already triggered on it once. From this state an exceptional condition can be simulated with the applied mutation that otherwise, if not encountered in this specific state, would not have an effect.

2.3.2.1 Required test artifacts

To develop this mutation-based sequential test approach, the following test artifacts are required:

- *source code to mutate*: source code that contains complex call sequences utilizing the services of the middleware;
- *mutation operators*: definition of suitable mutation operators to create robustness test cases;
- *configuration of mutations*: configuring how and how many mutants to create.

In the current version of the robustness test framework five types of mutation operators are implemented: omission, relocation and swapping of calls, modifying conditions, replacing parameters with a `NULL` value. The meaning of the operators are described in Table 2.3. Note that the approach can be extended with other mutation operators (e.g., replace parameters with other invalid values).

Table 2.3: Implemented mutation operators

Operator	Description
OmitCall	Removes a statement from the source code in which a call to a given function is found.
RelocateCall	Moves a statement containing a call to a given function to another random position in the source code.
SwapCalls	Swaps two function calls.
ModifyIfCondition	Replaces a logical operator in the condition of an `if` statement with a randomly chosen another operator.
ReplaceParameterWithNull	Replaces a parameter in a function call with a `NULL` value.

With respect to applying the mutation operators it is important to define constrains. If the operators are applied completely random, then the effectiveness of the generated mutants could be low (e.g., omitting calls that only produce debugging information or modifying functions that are not related to the middleware). For this reason the configuration of the mutations shall allow to define which calls or parts of the source to mutate.

2.3.2.2 Defining the mutant configuration language

Based on the requirements outlined in the previous section a configuration language was specified (Appendix A.2 contains the abstract syntax of the language in an XML Schema format). The language makes it possible to define the source files used for mutation, the mutation operators to use, and constrains on applying the operators. The elements of the language are the following:

- *Inputs*: root element having child elements describing the sources to mutate;
- *Input*: a source file or a directory containing source files to mutate, its *Location* child element specifies its absolute or relative path;
- *Mutations*: definition of the type and number of mutations to apply;
- *NumberOfMutants*: specifies how many mutants should be created from each source file,
- *NumberOfOperatorToApply*: describes how many times operators should be applied to create one mutant (it can be the same or different operators),
- *Operators*: the set of possible operators that can be used to generate mutants (it can be a subset of operators defined in the previous section).

The semantics of the configuration language is the following. Without any parameters, the operators are applied in a random manner (e.g., the *OmitCall* operator will search randomly for a function call to delete in the input source file). Parameters can be supplied to the operators to constrain this random choice: the *inFunction* parameter describes that modification should happen inside a given function, while the *call* parameter directs that only calls to a certain function should be modified.

Listing 2.5 shows an example for the configuration. Here for all source files located in the `saftest` directory ten mutants will be generated, in each mutant one operator will be applied.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<Inputs>
  <Input type="directory">
    <Location>saftest</Location>
    <Mutations>
      <NumberOfMutants>10</NumberOfMutants>
      <NumberOfOperatorToApply>1</NumberOfOperatorToApply>
      <Operators>
        <OmitCall call="*" inFunction="main" />
        <OmitCall call="saAmfComponentRegister" inFunction="*" />
        <ModifyIfCondition inFunction="*" />
        <RelocateCall call="*" inFunction="*" />
      </Operators>
    </Mutations>
  </Input>
</Inputs>
```

Listing 2.5: Example mutant configuration

2.3.2.3 Mutation-based sequential test generator tool

The inputs of the MBST-TG tool (Figure 2.5) are the source files to be mutated and a configuration file that describes the parameterization of the mutation operator, e.g., the filters to be used when searching for a call to apply the mutation.

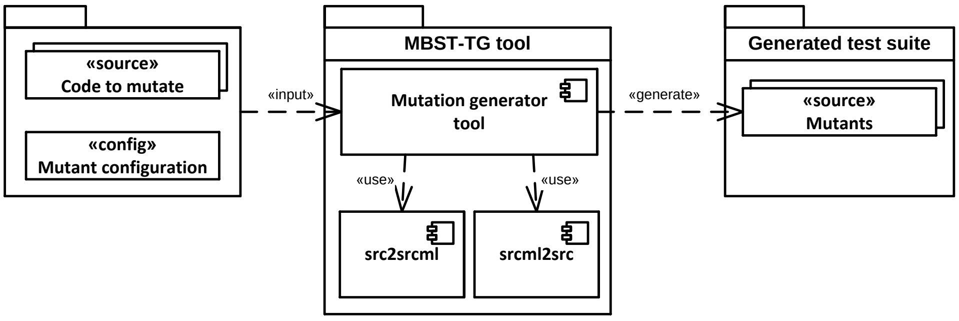


Figure 2.5: Architecture of mutation-based test tool

Algorithm 2.2 presents the strategy implemented in the tool to apply the mutation operators. The configuration for a given input source file prescribes the applicable operators, the number of mutants to create and number of operators to apply. If one operator shall be applied to create a mutant for a source file, then the possible operators are applied in a round-robin manner in the different mutants. If more than one operator shall be applied, then for each mutant the tool chooses randomly the given number of operators from the possible ones. If an operator cannot be applied, e.g., there is no call that was specified, then the tool tries to apply another operator if there are some left.

Algorithm 2.2: Applying mutation operators

```
foreach $input \in Inputs$ do
    if $NumberOfOperatorsToApply$ = $1$ then
        $rr := \text{RoundRobinList}(Operators)$
        for $i = 1$ to $NumberOfMutants$ do
            $m := \text{NewMutant}()$
            $o := \text{next}(rr)$
            $\text{ApplyOperator}(m, o)$
    else
        for $i = 1$ to $NumberOfMutants$ do
            $m := \text{NewMutant}()$
            $l := \text{List}(Operators)$
            for $i := 1$ to $NumberOfOperatorsToApply$ do
                repeat
                    $o := \text{random}(l)$
                    $\text{remove}(l, o)$
                    $success := \text{ApplyOperator}(m, o)$
                until $success \neq \text{true} \land l \neq \emptyset$
```

The challenge of implementing the mutation-based sequential test generator tool (MBST-TG) tool was the parsing and modification of the test programs' C source files. As the available free parsers encountered various problems when system header files were included in the input files, we followed a light-weight approach instead of obtaining the full parse tree (that is required for compilation). The srcML tool [SDM12] was used to build an XML file representing only the *syntactic structure* of the input source files. This syntactic structure is enough to implement the common mutation operators.

The mutant candidates came from two sources. The first one was the SAF Test [SAF06] project, which is an open-source conformance test suite for SA Forum specifications. Because the test cases in SAF Test are redundant, 10 source files were selected that cover the functionality of the others as well. The source files had to be slightly modified, because the B.01.01 version of SAF Test did not use the required LDAP Distinguished Name format for component names. The second source was the functional test suite provided by OpenAIS, from which the testamf test file was used for mutation. The MBST-TG tool was configured to generate mutants using one operator each time, and in a second run using two random operators each time. Note that occasionally the mutation may result in such source code that cannot be compiled (data flow analysis is not performed, this way, for example, changing of function calls may result in using variables that were not assigned a value before). Altogether from these mutants 92 valid mutants were included in the test suite.

2.3.3 OS call interception

The third part of the robustness framework intercepts OS system calls executed by the HA middleware and injects exceptional values into their return values. This section first collects the typical use cases for diverting OS calls, next it defines the language used to configure the interception, and finally presents the implemented OS call wrapper tool.

2.3.3.1 Required test artifacts

By intercepting OS calls, different types of stressful environmental conditions can be simulated. The most common ones include the followings:

- Call the original system call and return its result, but with a given probability include a delay. This mode can simulate network delays or overloaded subsystems.
- With a given probability return one of the predefined error codes of the system call. This mode can simulate various failures in the environment, like faulty authorization configurations or hardware failures.
- Instead of calling always the requested system calls, omission failures can be simulated by sometimes intercepting the calls but not forwarding them to the operating system.

Thus in case of OS call interception the test artifact that characterizes the robustness testing is the configuration of the diversion of the different OS calls. The configuration language of the OS call interception shall support the following modes of diversion:

- inserting a variable delay;
- changing the return value of the system call;
- configuring whether the original call should be forwarded to the OS or not;
- the interception should occur only with a given probability to represent rare faults.

Among this information the basic properties of the OS call (return type, list of possible error codes) are required additionally.

2.3.3.2 Defining the configuration language

Based on the requirements outlined in the previous section an XML-based configuration was designed (its abstract syntax can be found in Appendix A.3 as an XSD). Listing 2.6 presents an example for its usage. It defines that the `bind` function returns an `int`, indicates an error with the `-1` return value, and if this function is called, then it should be intercepted with 0.5 probability and the return value from the OS should be delayed with 10 milliseconds.

```xml
<function name="bind">
  <signature>
    <returnType>int</returnType>
    <standardErrors>
      <error type="int" value="-1" />
    </standardErrors>
  </signature>
  <interception>
    <logCall>true</logCall>
    <forwardCall>true</forwardCall>
    <detourChance>50</detourChance>
    <returnValue>
      <mode>normal</mode>
      <delay>10</delay>
    </returnValue>
  </interception>
</function>
```

Listing 2.6: Example function configuration in OS call interception

The description of the elements in the configuration are the following:

- *function*: the name of the function to intercept is specified in the *name* attribute;
- *signature*: the return type and the return values representing errors are listed in the signature element;
- *logCall*: it defines whether the calls should be logged;
- *forwardCall*: it instructs whether the original function should also be called;
- *detourChance*: it specifies a percentage of how often the interception of the call of the given function shall happen;
- *returnValue*: the *mode* element in the *returnValue* node has three valid values:
  - *normal*: the return value from the call to the original OS function is returned;
  - *desiredReturn*: the value specified in the *desiredReturn* element is returned by the wrapper;
  - *standardError*: one of the errors defined in the *standardErrors* element is returned randomly;
- *delay*: this element can instruct the wrapper to include a further delay (specified in milliseconds) before returning the value.

This configuration offers a *flexible* way of intercepting or delaying system calls to simulate different types of exceptional situations.

2.3.3.3 OS call wrapper tool

The developed OS call wrapper tool (see Figure 2.6) uses the configuration language described in the previous section. The wrapper is implemented in a library containing functions exactly matching the signatures of the system calls to intercept. The wrapper is injected using the Unix `LD_PRELOAD` variable, which can be used to load predefined libraries instead of system libraries.

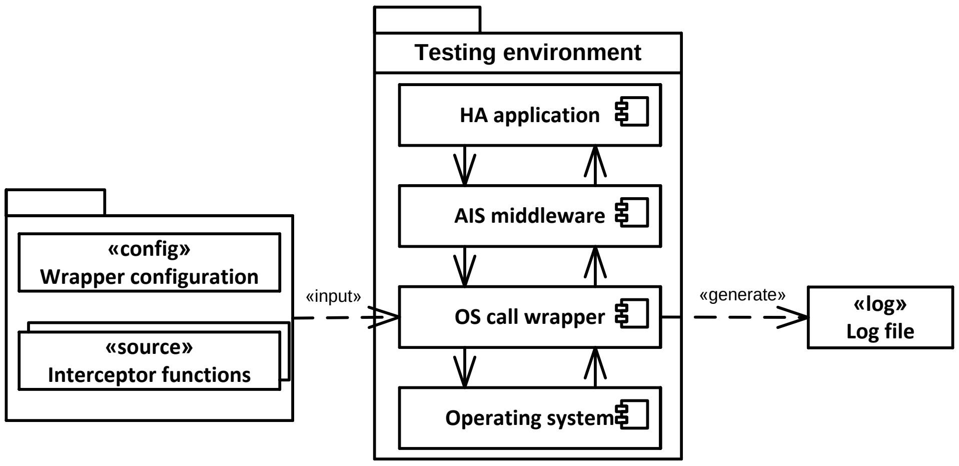


Figure 2.6: Architecture and integration of the OS call wrapper tool

The algorithm of handling an incoming system call is given in Algorithm 2.3.

Algorithm 2.3: Handling a system call in the OS call wrapper

```
Input: $s$ system call
Data: $r$ return value
if $detourChance \geq \text{rand}(0, 100)$ then
    switch mode do
        case normal
            $r := \text{call}(s)$
        case standardError
            if forwardCall then
                $\text{call}(s)$
            $r := (e \in standarErrors)$
        case desiredReturn
            if forwardCall then
                $\text{call}(s)$
            $r := desiredReturn.value$
    if delay > 0 then
        $\text{sleep}(delay)$
else
    $r := \text{call}(s)$
return $r$
```

Since the middleware is tested here as a black box, the system calls can be triggered only indirectly, by starting a workload application. As a *workload* to trigger OS calls from the middleware, a synthetic HA application was prepared that resembles a search and index engine. The application utilizes the AMF and checkpoint service of the middleware. Using the *strace* utility all system calls of the middleware were logged during the execution of the workload application on two implementations, and the intersection of the two sets of OS calls was included in the test suite, namely the functions `accept`, `bind`, `close`, `gettimeofday`, `munmap`, `poll`, `sendmsg`, `setsockopt` and `socket`.

2.4 Robustness testing case studies

As the final results of the research presented in this chapter are a new test framework and tools, a natural way to evaluate them is to execute the robustness test suite generated by the tools on different HA middleware implementations. The *objective* of this evaluation could be summarized in the following way.

Objective Can the test framework be used to assess the robustness of HA middleware systems?

The evaluation consisted of executing the test suites on middleware implementations and collecting data on the results. Testing was conducted on system level, i.e. the real middleware running on a computer node was studied. The results of the evaluation were used to make *qualitative* observations, and the results influenced the next set of test executions resulting in a flexible design. Thus the evaluation method can be categorized as *case study* according to the taxonomy of [Woh+12]. The next sections describe the case study design and the obtained results.

2.4.1 Case study design

The presentation of the case study design follows the guidelines of [RH09]. First, the general objective is refined into detailed research questions. Next, the planning of the case study is detailed (e.g., unit of analysis or data collection procedures).

Research questions Three research questions were formulated based on the objective.

RQ1 Can the more complex robustness testing techniques detect additional failures?
RQ2 Can the three different test technique be used to uncover different failures?
RQ3 Can the robustness of the different middleware implementations compared?

These research questions directed the design of the case studies. Three related case studies were performed, each of them focusing on one of the above research questions. The case studies used similar contexts, but they differed in the types of the test techniques applied and the middleware implementations tested (the case studies were performed in subsequent years with the continuously improved test framework and new versions of the middleware systems).

Case The object of study was in each case study the AIS-based HA middleware systems.

Unit of analysis The units of analysis were the different middleware implementations. Throughout the case studies the following ones were studied.

- *OpenAIS*: OpenAIS was one of the first open source AIS implementation started by MontaVista Software around 2002. Frequent versions were released until 2010, later the development of some of the components were migrated to the Corosync project. In the case studies several versions were studied: release 0.69 (2005-09-20), release 0.80.1 (2006-08-16), trunk[3] 2006-12-11, trunk 2007-10-02.

[3]The trunk is a development version obtained directly from the source control system of the project at a specified date.

- *SAFE4TRY*: SAFE4TRY was a package from Fujitsu Siemens Computers, which consisted of the SAF AIS implementation RTP-SAF-L V2.1A and the PRIMECLUSTER cluster.
- *OpenSAF*: OpenSAF is another open source implementation launched in 2007 by Motorola. Later several major vendors joined the development. The project is still releasing new versions, the last one was in October 2012.

Data collection The main method of data collection was indirect, the results of the test calls, test cases and log files were saved in each case study. The test framework was responsible to implement the data collection protocol, the following information was persisted automatically for each test execution.

- *Exit code*: The exit code of the test case process, possible values include *success*, *segmentation fault* or *timeout*.
- *Error code*: For each of the calls made to the middleware API an AIS error code is returned. The error codes are defined in the AIS specification, e.g., `SA_AIS_OK` or `SA_AIS_ERR_VERSION`.
- *Log files*: Operational logs of the middleware were saved, which could contain information about additional internal errors.

Analysis The test framework saved the exit and error codes for each test case in CSV files, which were later analyzed with spreadsheet applications or in the first case study with data mining tools. The log files were analyzed manually by searching for possible signs of errors or warnings, as the format of these logs are different for every implementation.

Validity The counter threats to validity several data sources (different middleware implementations) were used in the case studies. Both commercial and open source solutions were selected to cover different types of products. As the case studies did not involve human subjects or interviews, typical interpretation problems were not relevant. However, reliability of the case studies was a major concern. Scripts automated every aspects starting from test execution, middleware restarts between tests to data collection to obtain reproducible, reliable results.

Three case studies were developed to answer the three research questions defined. The case studies differed in the tools used from the test framework and the middleware implementations tested.

- *Case study 1*: The first case study addressed *RQ1*. It used prototype testing tools implementing random and type-specific techniques, and executed tests on an early version of OpenAIS to find out whether more complex techniques could find extra failures.
- *Case study 2*: The second one used all the three test tools (type-specific, mutation-based and OS call interception) on SAFE4TRY and two versions of OpenAIS, and tried to tackle *RQ2* (regarding the types of failures uncovered by the three techniques).
- *Case study 3*: The final case study used only the type-specific tool, but it executed tests on all middleware to answer *RQ3* by comparing the robustness of different implementations.

2.4.2 Case study results

The next section presents the results of the three robustness testing case studies.

2.4.2.1 Case study 1: Comparing generic and type-specific testing

The first case study (first reported in [16]) assessed the efficiency of the type-specific approach compared to more simple methods of using the same valid and invalid values for all types. The tests were executed on the AMF (17 functions) and CLM module (7 functions) of OpenAIS 0.69, which implemented the early A.0.1.01 version of the AIS specification.

Three different test suites were used. A generic tool created two test suites, where each test case called one AIS function with all the possible combinations of a set of predefined input values. The same set of input values was used for every type in every function (this was possible as most of the parameters were pointers in the AIS functions). The first suite used only invalid addresses, while the other used `NULL` and a valid address as an input to the tested functions. Note, that this was a very crude technique, but it represents coding errors when handling pointers that are frequent in C programs. The initial version of the generic test tool was implemented in approximately three days.

The third test suite was generated by the type-specific testing tool (Section 2.3.1). In case of functions with more than five complex parameters the number of the generated test cases for that function was limited to 4000. The design and implementation of the type-specific tool itself required about two weeks. The main advantage of the automated test approach is that the type-specific testing of a new function requires only the completion of the metadata, and supplying the test values and logging code for the new types used in the function. When adding a new function these activities required usually only 1-2 hours (or even less, if the types of the function's parameters were already defined).

Table 2.4: Results from the different exceptional input generation and test techniques. Legend: calls resulting in robustness failures / total number of calls.

Technique	OpenAIS AMF	OpenAIS CLM
Generic testing with invalid addresses	2406 / 2456	60 / 424
Generic testing with `NULL` and valid address	87 / 136	0 / 44
Type-specific testing	8001 / 13640	65 / 2280

Table 2.4 lists the ratio of API calls that resulted in robustness failures and the number of test calls executed. The result of the call was initially considered as a robustness failure if the call produced a segmentation fault in the test program. CLM was more resilient to generic testing since it used less pointers than AMF. Just the number of observed robustness failures are not enough to compare the effectiveness of the three test suites, thus we refined the characterization of the results of the tests using the following methods.

First, we assigned the possible error codes (as potential results) to test inputs values. The test outputs were then filtered and only those test runs were inspected, in which the output was not among the expected error codes. In this way hindering failures (i.e., when an incorrect error code is returned) can be differentiated from robust behavior (where the proper error code is returned in response to an invalid input).

Next, instead of the number of the observed robustness failures a much better measure is the number of the robustness faults causing those failures. The first step is toward this goal is to identify the exact parameter combination that causes a failure (note that in case of closed source implementation faults can be traced back only to this level). However, even in our early tests thousands of robustness test cases were generated, thus an automated method was needed to analyze the results. Following the method presented in [Pin+05], the decision tree method of a data mining tool was used to trace back robustness failures to the parameter combinations causing those failures in each of the AIS functions (e.g., if the *version* is valid but the *handle* is `NULL` in `saAmfInitialize`, then it causes always a segmentation fault). Table 2.5 lists the obtained results. In case of several functions, type-specific testing identified additional robustness faults in comparison with generic testing, while in case of three functions only type-specific testing was effective.

Table 2.5: Faults found in OpenAIS by functions. X + Y means that generic testing found X faults while type-specific identified Y more. The star denotes a critical error, which caused segmentation fault not only in the test program, but in the middleware executive also.

Function name	Faults	Function name	Faults
saAmfCompNameGet	1	saAmfProtectionGroupTrackStop	2
saAmfComponentCapabilityModelGet	1	saAmfReadinessStateGet	1 + 1
saAmfComponentRegister	2	saAmfResponse	1*
saAmfComponentUnregisterRegister	2	saAmfSelectionObjectGet	1 + 1
saAmfDispatch	1	saAmfStoppingComplete	1*
saAmfErrorCancelAll	1	saClmClusterNodeGet	0 + 1
saAmfErrorReport	3	saClmClusterTrack	0 + 1
saAmfFinalize	1	saClmClusterTrackStop	0
saAmfHAStateGet	2	saClmDispatch	0
saAmfInitialize	0 + 2	ClmFinalize	0
saAmfPendingOperationGet	1	saClmInitialize	0
saAmfProtectionGroupTrackStart	2	saClmSelectionObjectGet	0

Conclusions of the first case study This first case study addressed research question *RQ1* (*Can the more complex robustness testing techniques detect additional failures?*). The analysis of the data collected showed that there could be failures that could not be detected by simple techniques. The use of more complex robustness testing techniques is further motivated by the fact that the type-specific testing in the case study found two additional critical faults affecting not just the test application but the middleware itself. Although results of the case study are limited by the fact that it was performed only on one implementation, the obtained results are consistent with the experiences of other robustness testing projects (e.g., the Ballista project [KDD08]).

2.4.2.2 Case study 2: Comparing the three testing techniques

The second case study (first reported in [12]) analyzed the failures found by all three tools in SAFE4TRY and two newer versions of OpenAIS.

Results from the type-specific tests Just by trying to compile the test suite generated by the TBTS-TG tool on the system under test, several discrepancies were found. The header files used in OpenAIS differed in eight places from the official header files of the AIS specification, and thus from the header files used by the test suite. There was also one misspelling in SAFE4TRY's header files. Moreover, there were several types in the specification that were mapped to different types in the implementations, e.g., `SaInt32T` is mapped to `long` in SAFE4TRY and to `int` in OpenAIS.

Table 2.6 summarizes the exit codes of the test cases that were logged when executing the test suite. Segmentation faults definitely indicate robustness failures, since in a HA middleware (that has to be prepared for the erroneous behavior of the managed components) even invalid inputs should be handled correctly. Timeouts could indicate normal behavior, because some of the API functions could be parameterized to wait for an event to dispatch. However, after examining the concrete values used in the tests, it turned out that the large number of timeouts in openais-trunk-2006-12-11 and openais-0.80.1 is not reasonable. Note for openais-0.80.1 the sum of the calls is lower than for the other two implementations, because in case of `saAmfProtectionGroupTrack` the test program and the middleware crashed at the beginning of the test and no calls were executed for that functions.

Table 2.6: The number of test cases that exited with the given status code in case of type-specific testing of the different platforms.

Status code	openais-0.80.1	openais-trunk-2006-12-11	SAFE4TRY
0 (success)	24568	26019	29663
11 (seg. fault)	1110	1468	0
14 (timeout)	467	2178	2

Segmentation faults occurred in 13 functions of openais-trunk-2006-12-11 and in 12 functions of openais-0.80.1. Timeouts were observed in 7 functions of openais-trunk-2006-12-11, in 7 different functions of openais-0.80.1, and in one function of SAFE4TRY (namely, in `saAmfDispatch` when specifying a flag representing blocking; here timeout is the correct behavior). Table 2.7 lists the details.

Table 2.7: Functions that produced robustness failures in case of type-specific testing

Failure	openais-0.80.1	openais-trunk-2006-12-11
seg. fault	saAmfComponentErrorClear saAmfComponentErrorReport saAmfComponentNameGet saAmfComponentRegister saAmfComponentUnregister saAmfHAStateGet saAmfHealthcheckConfirm saAmfHealthcheckStart saAmfHealthcheckStop saAmfInitialize saAmfProtectionGroupTrackStop saAmfSelectionObjectGet	saAmfComponentErrorClear saAmfComponentErrorReport saAmfComponentNameGet saAmfComponentRegister saAmfComponentUnregister saAmfHAStateGet saAmfHealthcheckConfirm saAmfHealthcheckStart saAmfHealthcheckStop saAmfInitialize saAmfProtectionGroupTrack saAmfProtectionGroupTrackStop saAmfSelectionObjectGet
timeout	saAmfComponentErrorClear saAmfComponentNameGet saAmfCSIQuiescingComplete saAmfDispatch saAmfInitialize saAmfHealthcheckConfirm saAmfProtectionGroupTrackStop	saAmfComponentErrorClear saAmfComponentNameGet saAmfComponentUnregister saAmfCSIQuiescingComplete saAmfDispatch saAmfProtectionGroupTrack saAmfProtectionGroupTrackStop

Some of the test cases caused fatal error in the middleware. The tests for 14 functions in openais-0.80.1 and for 6 functions in openais-trunk-2006-12-11 produced an internal *assertion violation*, and the middleware exited. The following two assertion violations were observed (the assertions show that the code detected that some input values are not valid; however, the problem was not correctly handled inside the middleware as it resulted in a crash):

```
aisexec: amf_lib_exit_fn: Assertion ‘comp != ((void *)0)’ failed.

aisexec: amfcomp.c:1142: amf_comp_register: Assertion ‘0’ failed.
```

In the first case study version 0.69 of OpenAIS was used. In comparison with this previous experiment, the following could be observed: in the current versions of OpenAIS the simple method of using only invalid pointers and integer values as exceptional parameters did not activate as many robustness failures as previously. One of the reasons for this is that moving to version B.01.01 of AMF the number of pointer parameters decreased significantly. In the type-specific robustness test suite 58.6% of the tests resulted in segmentation fault for version 0.69, while this number was only 4.2% and 4.9% for the 0.80.1 and trunk versions, respectively. Thus, the robustness of OpenAIS was definitely improved, although it still lags behind the robustness of SAFE4TRY.

Results from the mutation-based testing The mutant test sequences obtained from SAF Test and testamf were executed on the three implementations. The number of observed robustness failures is summarized in Table 2.8.

Table 2.8: The number of observed robustness failures / the total number of executed test cases in case of mutation-based testing of the different platforms.

Input	openais-0.80.1	openais-trunk	SAFE4TRY
SAF Test	8 / 63	0 / 63	1 / 63
testamf	22 / 29	28 / 29	0 / 29

The robustness failures discovered by the SAF Test mutants were the following. In case of eight mutants, openais-0.80.1 exited with one of the previous (Section 2.4.2.2) or with the following assertion:

```
./aisexec: symbol lookup error: /opt/openais-0.80.1/exec/
/service_amf.lcrso: undefined symbol: assert
```

In SAFE4TRY, when stopping the middleware after one of tests the following error occurred:

```
Error in communication! ERROR: Stopping AMF subsystem was not successful
```

Note that the SAF Test programs are constructed in such a way that the return value is checked after each function call, and if it does not match the predefined value then the program is aborted with an error message. This feature of the SAF Test programs makes them difficult to be used in robustness tests, because the subsequent calls are not executed if a wrong return value is detected. thus the tests terminate usually very early after one or two calls.

When the testamf mutants were executed as AMF components in openais-trunk and openais-0.80.1 the CPU utilization increased to 100% and a hard reset had to be performed. Thus, Table 2.8 contains the results from running the testamf mutants as standalone programs. During the experiments with the mutants the above detailed assertions were also observed.

It could be observed that mutation-based robustness testing highlighted additional robustness failures that were not detected by the type-specific tests. It gives reasons for applying such complex test sequences.

Results from the OS call wrapper For each of the nine selected system calls (see Section 2.3.3) a separate test case was executed by starting the workload application and after a while forcing a failover. The system calls were forwarded to the OS, and with a predefined probability a random error code was returned (the probability depended on the frequency of the call, which was determined in probe runs).

Table 2.9: The system calls that provided the given outcome using the OS call wrapper.

Outcome	openais-0.80.1	openais-trunk	SAFE4TRY
No failure observed	accept, close, gettimeofday, munmap, sendmsg, setsockopt	accept, bind, close, gettimeofday, sendmsg	accept, close, gettimeofday, sendmsg, setsockopt
Application failed	-	munmap, setsockopt	poll
Middleware failed	bind, poll, socket	poll, socket	bind, munmap, socket

The first row of Table 2.9 lists the system calls in which case the workload application was executed successfully in spite of the injected fault. The second row shows such cases when the application exited but the middleware did not fail. The last row indicates the test cases when also the middleware exited (typically silently, without error messages). Note that due to the random injection of error codes, these latter cases just indicate potential robustness faults without objectively comparing the implementations.

Conclusions of the second case study This second case study addressed research question *RQ2* (*Can the three different test technique be used to uncover different failures?*). The case study showed that type-specific testing was useful to uncover problems in the definition of the API, and test a wide range of parameter combinations. Mutation-based testing was able to detect a failure in SAFE4TRY, where the type-specific test suite have not found any obvious robustness failures. The interception of OS calls was able even to crash SAFE4TRY, which did not happen with the other two test techniques. Thus different kinds of robustness failures were observed with the three test techniques. However, the case study also pointed out limitations in the current test framework, e.g., the SAF Test programs are not the most adequate input for mutation-based testing or that tracing back robustness failures detected with the OS call wrapper is not trivial.

2.4.2.3 Case study 3: Comparing the different middleware implementations

The last case study (reported in [9] and [12]) compared the results of the type-specific test suite from all middleware implementations. (Note, the testing of OpenSAF was performed by András Kövi on a two node OpenSAF 3.0.FC cluster). The test suite generated by the type-specific test tool were executed on all the tested implementations, thus comparing their results can offer insights on the robustness of the different middleware systems. To be more precise, as the implementations supported different versions of the AIS specification, different versions of the test suite were also executed. However, the difference between the versions was relatively small (e.g., adding two new type and removing four old ones from the approximately 60 defined types, removing one function from the 20 tested ones), thus *quantitative* observations can be deduced.

Two metrics of the executed robustness testing can be compared:

- the number of given *exit codes* of the test cases;
- the number of given *error codes* returned by the middleware.

As it can be seen from Table 2.10 most of the tests returned with success (exit code 0), a few returned with timeout (exit code 14), and only the early development branch of OpenAIS suffered segmentation violations (exit code 11). It is also notable that in the second OpenAIS version all of these bugs were corrected and no segmentation violations happened. Comparing the number of timeout errors reveals that SAFE4TRY caused significantly less of these than the other two. Even though it cannot be easily verified, as the source code of this implementation is not publicly available, the probable reason behind it can be the different way of memory handling and early verification of invalid parameters.

Table 2.10: Exit code counts by middleware implementations

Exit code	OpenAIS		OpenSAF	SAFE4TRY
	trunk-2006-12-11	trunk-2007-10-02		
0 (exit)	26019	21228	28309	29663
11 (segv)	1468	0	0	0
14 (timeout)	2178	1578	1356	2

Table 2.11 contains the distribution of returned error codes by middleware implementations. Although all implementations are based on the same high-level API, their parameter and error handling mechanism can differ significantly. The two most frequently returned errors are the `BAD_HANDLE` and the `INVALID_PARAM` codes. The `BAD_HANDLE` means that the handle is invalid or corrupted. The `INVALID_PARAM` covers most of the cases when parameters do not conform to the requirements. The number of `SA_AIS_ERR_INVALID_PARAM` codes show that SAFE4TRY and OpenSAF detect much more invalid parameter combinations. Moreover, OpenSAF mostly returns `INVALID_PARAM` instead of `BAD_HANDLE`. This can probably mean that OpenSAF checks the library handle in most cases only if the other parameters conform to the requirements. When an assertion was violated in OpenAIS, all the remaining calls for the given test program resulted in library error, that is the reason for the high number of `SA_AIS_ERR_LIBRARY` codes (which could not be considered as robust behavior). Finally, as there are significant differences in the types and numbers of the returned error codes, probably the answers to invalid calls should be more precisely defined in the AIS specification.

Conclusions of the third case study The third case study was conducted to answer research question *RQ3* (*Can the robustness of the different middleware implementations compared?*). The case study showed that even just the analysis of the error and exit codes of the executed test suite can point out the following differences.

- The robustness test framework can quickly identify implementations with significantly lower robustness (e.g., the early versions of OpenAIS, where tests produced segmentation faults).
- The test suite can identify whether an implementation could return invalid error codes (as in the case of OpenAIS). Invalid error codes hinder the error handling in the applications and can signal further bugs in the implementation.

Table 2.11: Error code counts by middleware implementations

Error code	OpenAIS		OpenSAF	SAFE4TRY
	trunk-2006-12-11	trunk-2007-10-02		
(invalid error code)	1279	0	0	0
139	0	1260	0	0
`SA_AIS_ERR_BAD_FLAGS`	0	0	576	384
`SA_AIS_ERR_BAD_HANDLE`	20408	16628	7056	20708
`SA_AIS_ERR_EXIST`	0	0	0	1
`SA_AIS_ERR_INIT`	0	0	295	6
`SA_AIS_ERR_INVALID_PARAM`	226	64	19755	6073
`SA_AIS_ERR_LIBRARY`	2316	2644	0	52
`SA_AIS_ERR_NOT_EXIST`	1296	1458	0	1786
`SA_AIS_ERR_NOT_SUPPORTED`	0	0	0	144
`SA_AIS_ERR_TRY_AGAIN`	30	0	224	0
`SA_AIS_ERR_VERSION`	336	336	294	294
`SA_AIS_OK`	128	98	109	215

- Finally, the results can exhibit the differences between the parameter and error handling mechanisms of the implementations. An implementation that identifies invalid parameters early could signal better robustness.

A more detailed analysis that traces back robustness failures to faults (like the one performed in the first case study) would offer a more precise comparison, but it remains future work as it was not performed for the data of the third case study.

2.4.3 Conclusions of the case studies

The case studies were performed to evaluate whether the developed test framework can be used to assess the robustness of HA middleware systems. The three case studies tried to find out whether (i) can more complex techniques find additional failures, (ii) the different test techniques would find the same robustness failures, and (iii) the test framework could be used for comparison. The results of the case studies were constructive, i.e., they showed additional failures found only by different test techniques or identified more robust implementations. Note, the case studies focused on qualitative and on quantitative data.

The case studies used several middleware implementations to increase the trustworthiness of the results. Moreover, all test execution and data collection were automated to counter human errors. The type-specific test suite and all detailed results on OpenAIS were made available online [BME07] to enable the reproduction and analysis of the case studies.

Finally, the case studies helped to identify limitations and future work in the robustness test framework, which will be summarized in the next section.

2.5 Summary

In this chapter a test approach for HA middleware systems was presented. Figure 2.7 summarizes the research problem and the developed languages and test framework. Based on the possible activation modes of the robustness faults of the HA middleware the following methods were developed:

- adapting the classical type-specific API function testing;
- state-based testing using mutated functional tests;
- diverting the calls going to and return values coming from the OS services using a wrapper.

The novelty of the approach is (i) the *method* of identifying test artifacts and designing the supporting languages and tools, and (ii) the application of *automatic tools* that construct the test cases systematically on the basis of the *standard interface specification* (API functions) and existing functional test suites. The robustness testing of the HA middleware implementations demonstrated that these tools can be used efficiently and their test results are complementary as they detect distinct failure types. It is important to emphasize that robustness testing was used only to observe these problems, and further work is needed to find the causes and to turn the observations into dependability benefits, e.g., by identifying the wrong implementation approaches that shall be corrected.

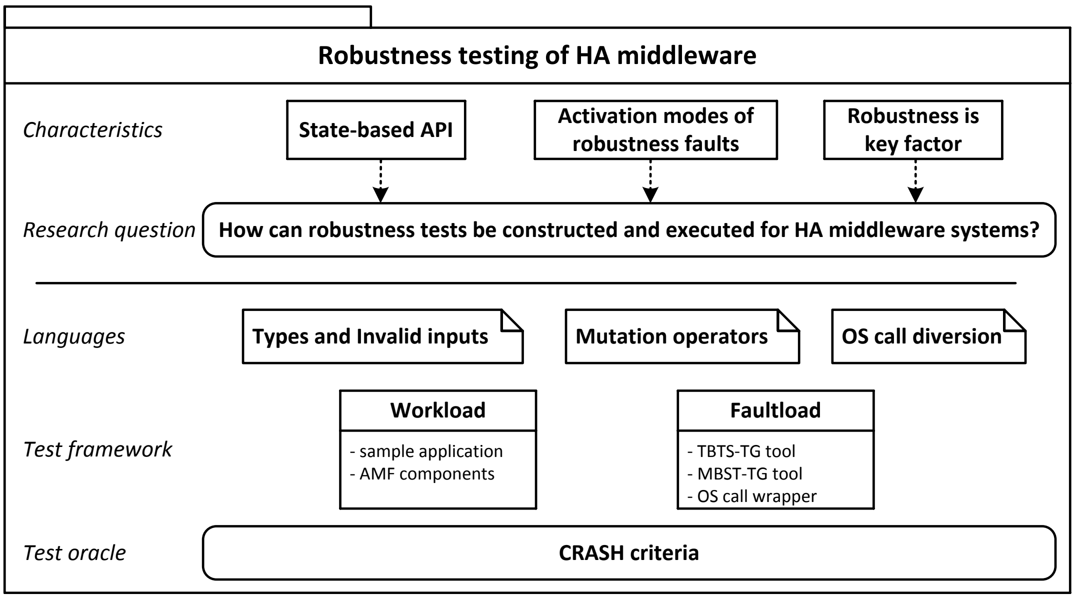


Figure 2.7: Summary of problem, languages and framework for robustness testing

The developed test tools and test suite could form the basis of a *robustness benchmark* for HA middleware. To be useful as a means to obtain a fair comparison of various implementations, the following required properties of a benchmark suite [KS08] were taken into consideration.

- *Representativeness*: A benchmark must reflect the typical use of the target system. The robustness testing is complete from the point of view of the set of AMF functions that can be called from the potential applications. In the case of the set of operating system calls that should be intercepted – since no specific application was known at the time of constructing the robustness test suite – a synthetic application was used as a workload that exercised the AMF services.
- *Repeatability*: The results obtained for a specific execution of the benchmark should be repeatable (within acceptable statistical margins) for the same system setup, even if executed by different end users. Since the test suite is available as a ready-to-use set of programs, and not as a high-level specification that can be interpreted in various ways, the implementation of the test suite is unambiguous and its execution is repeatable.

- *Portability*: Most of the developed tools are used only in the construction phase of the benchmark (namely, the type specific test generator and the mutant based test generator) and the resulting test files (that are used during benchmarking) are available as standard C files. This way the portability of these tools is not an issue. The OS call wrapper tool is available as a standard C program which can be compiled in different AIS application environments.

Chapter 3

Semantic choices in UML 2 Sequence Diagrams

Graphical scenario languages are frequently used to express various test artifacts. They offer an intuitive visual notation to communicate test requirements or test cases, or other artifacts depicting messages sent between different entities. In our research, we concentrated on the scenario language defined in the Unified Modeling Language (UML), namely Sequence Diagrams.

The 2.0 version of UML changed Sequence Diagrams significantly. Several elements were borrowed from MSC, many new complex elements were added to the language, and the semantics and the underlying metamodel were rewritten. Due to the increased expressiveness of the language, interpreting a complex diagram that uses the new constructs is a difficult task; thus, having a precise formal semantics becomes even more critical. But the many different purposes Sequence Diagrams are used for, e.g., showing the flows of method calls inside a program, or giving a partial specification of interactions in a distributed system, require quite different interpretations of the language. Indeed, many different semantics have been proposed for Sequence Diagrams. For a practitioner wanting to use Sequence Diagrams for a given purpose, it is not easy to select a suitable semantics.

We faced exactly this problem when we were working on the definition of test languages for mobile computing systems. When we tried to define the semantics of the new language, we encountered the problem that the various formal semantics for Sequence Diagrams handle even the most basic diagrams quite differently. It turned out that there are several subtle choices in the interpretation of language constructs. Moreover, these choices and all their consequences are often not obvious. A structured representation of all these choices was needed.

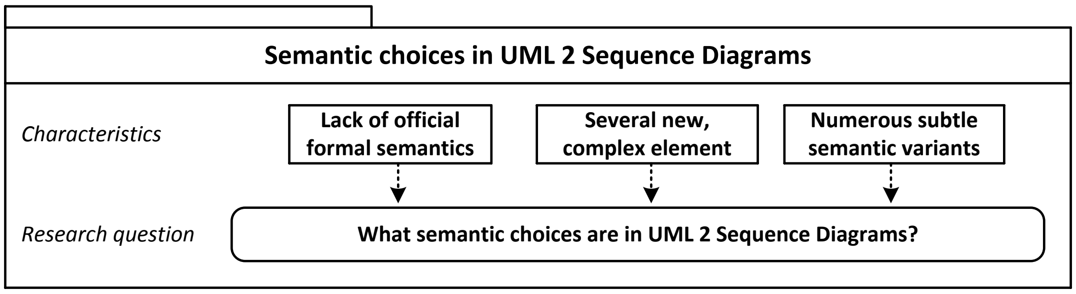

Semantic choices in UML 2 Sequence Diagrams
Characteristics
Lack of official formal semantics
Several new, complex element
Numerous subtle semantic variants
Research question
What semantic choices are in UML 2 Sequence Diagrams?

Figure 3.1: Research question of the chapter

Based on our experience, our aim was to (i) give an overview about the proposed *formal semantics*,

(ii) *collect and categorize* the semantic choices faced by them, and (iii) present the *different options* for the collected choices and the relations between these options in a structured format.

The chapter is divided into the following parts. Section 3.1 presents the syntax and semantics as defined in the OMG specification. We tried to highlight those parts, which are usually missing from published overviews about Sequence Diagrams. Section 3.2 presents a survey of 13 proposed formal semantics for Sequence Diagrams. We selected semantics created for different purposes (e.g., for using in high-level specifications or in verification tools), to have a broad coverage of language variants and usage modes. Section 3.3 collects and categorizes the semantic choices for Sequence Diagrams and describes what are the consequences of choosing one or the other options. Finally, Section 3.4 summarizes the contributions of the chapter.

3.1 UML Sequence Diagrams in the OMG specification

Sequence Diagrams are defined in the UML Superstructure specification [OMG11b]. More precisely, scenarios in UML are modeled with *Interactions*[1]. Interactions can be illustrated on several diagram types: Sequence Diagrams, Interaction Overview Diagrams, Communication Diagrams, Timing Diagrams, and Interaction Tables. Thus, the syntax and semantics are defined for Interactions; Sequence Diagrams are just a concrete notation to depict them.

3.1.1 Syntax of Sequence Diagrams

The syntax defined in the specification consists of (i) a concrete syntax defining the graphical notation, and (ii) an abstract syntax given with a metamodel defining the relationships between the elements.

3.1.1.1 Concrete syntax

This section summarizes the elements of Interactions and their notations on Sequence Diagrams. Figure 3.2 illustrates a basic *Interaction*. *Lifeline*s represent the individual participants in the Interaction, which communicate via *Message*s.

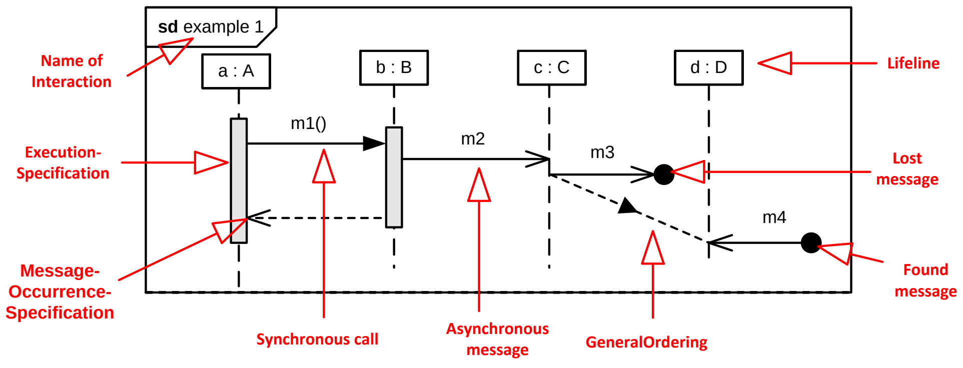


Figure 3.2: Example Sequence Diagram

Message is a general term: it can be a synchronous or an asynchronous communication; it can mean calling an *Operation* or sending a *Signal* (specified by its *MessageSort* attribute). *MessageKind* defines whether the sender or receiver of the message is known (complete, lost or found messages).

[1]Throughout the text, elements of the UML metamodel are written with CamelCase.

Messages have two *MessageEnds*. *GeneralOrdering* can constrain the ordering of otherwise unrelated occurrences. *ExecutionSpecification* is a specification of the execution of a unit of behavior or action within a Lifeline. *OccurrenceSpecification* (and its descendants) is the basic unit of semantics. Sending and receiving messages are marked with *MessageOccurrenceSpecification*; starting and ending of ExecutionSpecifications are represented with *ExecutionOccurrenceSpecification*.

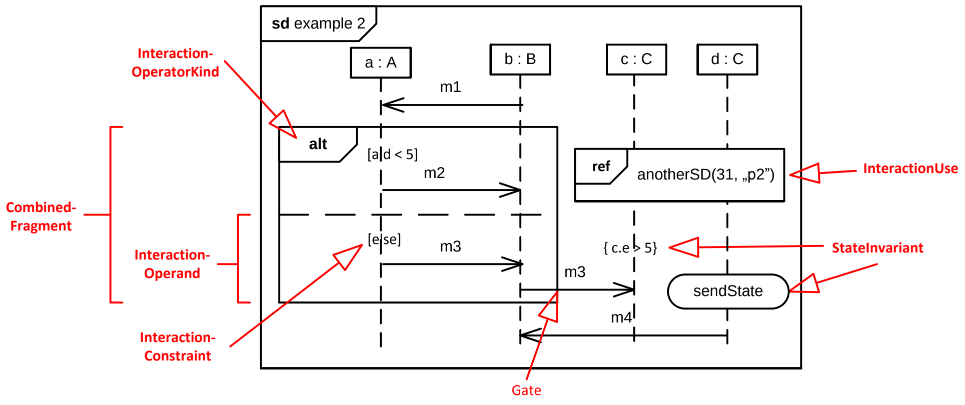


Figure 3.3: Example for CombinedFragment

More complex Interactions can be created with *CombinedFragment*. A CombinedFragment consists of one or more *InteractionOperands*. An *InteractionOperatorKind* specifies the purpose of the fragment. *InteractionConstraints* can guard each InteractionOperand. Messages on their own cannot cross the boundaries of CombinedFragments: they need a *Gate* which links the two parts of the message. An *InteractionUse* refers to another Interaction. It can be passed parameters and can have a return value.

StateInvariant is a run-time constraint on one of the participants of the Interaction. StateInvariants have two kinds of notation: it can be an expression of attributes and variables, or it can refer to a state of the Lifeline's instance (both notations are used on Figure 3.3). Further constructs exist, e.g., for specifying time and duration constraints, for a complete list see the specification [OMG11b].

3.1.1.2 Abstract syntax

The abstract syntax of Interactions is defined with metamodeling; the model is presented in *Section 14.2 Abstract Syntax* of [OMG11b]. The abstract syntax is depicted in several separate diagrams, which makes it sometimes hard to see all the connections between the important elements. Thus, we illustrate the abstract syntax of the most important elements of the BasicInteractions package on one diagram in Figure 3.4. (Note that the various Events classes, the MessageSort and MessageKind classes, attributes and some of the association names are not depicted on the picture for readability.)

Figure 3.5 illustrates the abstract syntax of the Fragments package. (Again, attributes and some of the association names are not depicted on the picture for readability.) *InteractionFragment* is an abstract class for Interaction, CombinedFragment, InteractionOperand, InteractionUse and Continuation, and also for OccurrenceSpecification, ExecutionSpecification and StateInvariant.

From the abstract syntax we can see for example, that a StateInvariant belongs to one Lifeline; thus it is a local constraint, or that there are three kinds of Gates, each for different purposes.

The specification contains a simple example illustrating an Interaction's concrete and abstract syntax; however that diagram does not contain CombinedFragments. It is really helpful to see how

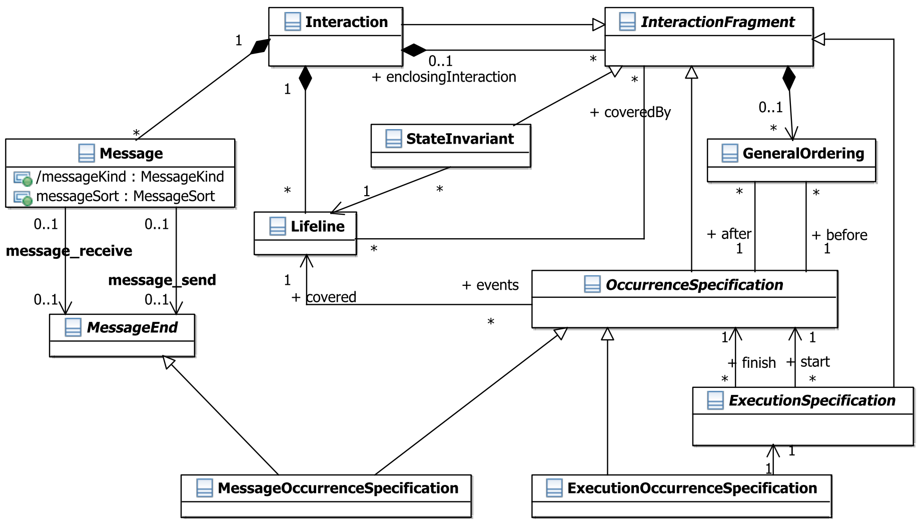


Figure 3.4: The abstract syntax of the BasicInteractions package (fragment)

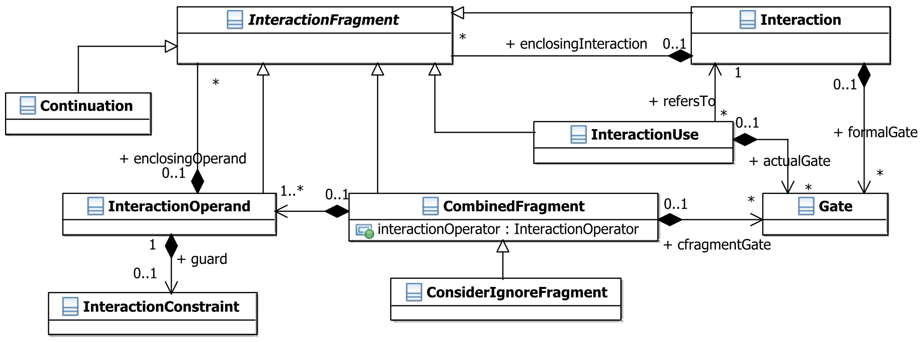
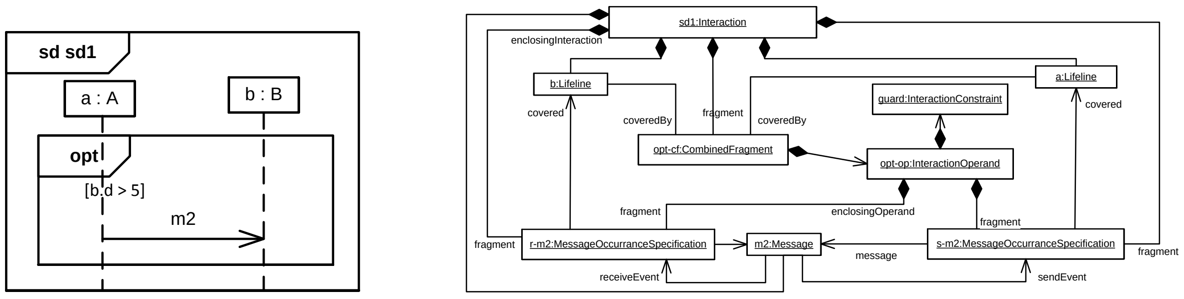


Figure 3.5: The abstract syntax of the Fragments package (fragment)

the different elements relate to each other; thus a more complex example is included here.

The right side of Figure 3.6 contains the metamodel elements of *sd1*. The Interaction is a container for all other elements. The OccurrenceSpecifications are linked to the appropriate Lifelines and Messages. The Lifelines are connected to the CombinedFragments that cover them. The InteractionOperand contains the InteractionFragments (OccurrenceSpecifications, StateInvariants, other CombinedFragments, etc.) which are enclosed by this operand. An InteractionFragment can be enclosed only by one operand; thus when an InteractionFragment is nested in several operands, only the bottom-most containment is illustrated in the model explicitly.

Finally, an Interaction can be stored in XMI (XML Metadata Interchange) format to exchange models between different tools. The XMI contains the Interaction's abstract representation, e.g., for Figure 3.6 the XML in Appendix B.1 is generated.



(a) Concrete syntax (b) Abstract syntax

Figure 3.6: A complex Interaction's concrete and abstract syntax (fragment)

3.1.2 Semantics of Sequence Diagrams

There are two major challenges when dealing with the semantics given in the OMG specification.

- The description of the semantics is *scattered* throughout the text. Some parts are in the introduction of the chapters, while some information is only in the constraints defined in the detailed description of a class.
- The specification uses so-called *semantic variation points* [Sel04], i.e., part of the semantics is not specified in detail to allow using UML in many domains. When UML is used in a concrete domain, the modeler has to choose from the different possible variations. However, sometimes these variation points are not marked explicitly.

This section summarizes the parts of [OMG11b] that deal with the semantics of Interactions. (Note, in the remaining part of this section page numbers refer to [OMG11b].)

3.1.2.1 Common run-time semantics

UML introduced a common run-time semantics for its different notations, which defines basic elements, e.g., Behavior, Actions, and Event. Section 6.2 of [OMG11b] (*On the Run-Time Semantics of UML*) summarizes the basics. All behavior is caused by actions executed by active objects. It describes also the basic causality model.

> "The causality model is quite straightforward: Objects respond to messages that are generated by objects executing communication actions. When these messages arrive, the receiving objects eventually respond by executing the behavior that is matched to that message. (page 11)"

The *CommonBehavior*s package (Chapter 13) deals with the fundamentals of behavior. A *Behavior* describes how the states of the objects, as reflected by their structural features, change over time. Behavior is an abstract metaclass; its subtypes are Interactions (Chapter 14), Activities (Chapter 12), State Machines (Chapter 15) and Use Cases (Chapter 16). These subtypes differ on how they model a behavior, e.g., what level of detail is captured. A Behavior is attached to a *BehavioredClassifier* (e.g., to a Class or to a Collaboration). A Behavior is the implementation of a *BehavioralFeature*, which can be an *Operation* or a *Reception* of a *Signal*. Behaviors can be invoked by *Action*s. An Action (Chapter 11) is the fundamental unit of behavior specification. "*An action takes a set of inputs and converts them into a set of outputs, though either or both sets may be empty* (page 225)." Examples for actions are *CallOperationAction*, *SendSignalAction* or *WriteVariableAction*. The execution of Actions can result in an *Event*. Possible Events include *CallEvent* or *SignalEvent*. For Events caused by *InvocationActions* a request (e.g., a Message) is generated and sent, which contains the arguments of the Action and the identities of the sender and receiver object. Similarly, at the receiving of the request an Event will occur. Events, through *Triggers*, may cause the execution of Behaviors. Figure 3.7 summarizes the inheritance relations between these elements.

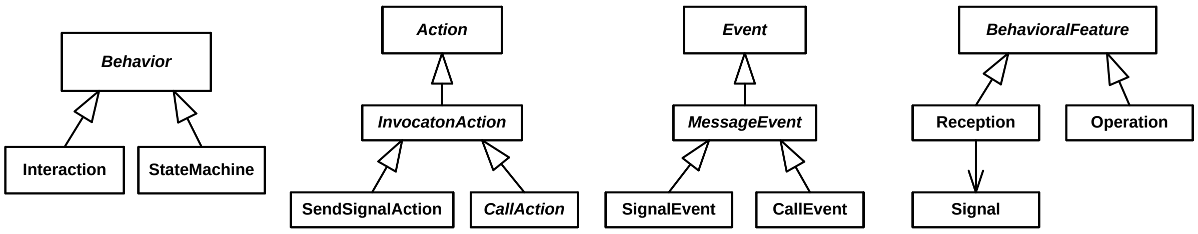


Figure 3.7: The inheritance relationship between some of the fundamental elements

Figure 3.8 presents the relations of these fundamental concepts. Associations in red are not part of the specification; they illustrate only possible implicit relationships. For example, a CallBehaviorAction Action causes a direct invocation of a Behavior, while the CallOperationAction does this via a BehavioralFeature. Likewise, Actions do not necessarily cause an Event, and not all Events are caused by Actions. These relationships and the semantic domain of behaviors are detailed in *Section 13.1 Overview of the Common Behaviors* chapter.

The run-time semantics of UML is also described in [Sel04], where not only the concepts, but also the design decisions behind them are explained.

Finally, in *Relation of trace model to execution model* (page 496) the relation between this general semantic domain and the elements of Interactions is described. Invocation occurrences and receive occurrences are modeled by OccurrenceSpecification. Actions are generally not described in Interactions. A request is modeled by a Message; an execution of a behavior is by ExecutionSpecifications. Although the CommonBehaviors package tries to unite the behavior described in the several different notations, sometimes this is not achieved yet completely. For example, both ReceiveSignalEvent (from BasicInteractions) and SignalEvent (from Communications) describe the receipt of a Signal, they both are descendants of MessageEvent, but have no relations with each other.

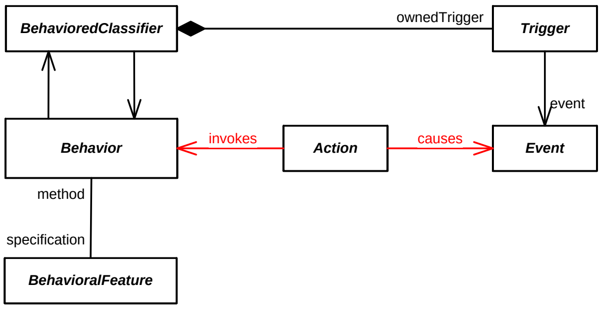


Figure 3.8: Relations of the basic behavioral concepts (red associations are not part of the specification)

3.1.2.2 Semantics of basic Interactions

Interactions describe behavior with messages between participants. The focus is on the order and the types of the messages, although Interactions can contain reference to data in message parameters and constraints. The central concept of the semantics is a trace.

> "A trace is a sequence of event occurrences, each of which is described by an Occurrence-Specification in a model" (page 495).

A central question is what part of the behavior is modeled by the Interactions.

> "There are normally other legal and possible traces that are not contained within the described interactions" (page 473).

Interactions can model also invalid traces, and there could be traces that are not described by the Interaction:

> "The semantics of an Interaction is given as a pair of sets of traces. The two trace sets represent valid traces and invalid traces. The union of these two sets need not necessarily cover the whole universe of traces. The traces that are not included are not described by this Interaction at all, and we cannot know whether they are valid or invalid" (page 495).

Collecting all the references yields that invalid traces are defined by assert and negative fragments, and constraints such as StateInvariant, DurationConstraint and TimeConstraint. The semantics of Interactions is explained with an interleaving semantics, i.e., two events may not occur at exactly the same time.

Producing the traces of a diagram is constrained by the following rules:

- Occurrences on the same Lifeline must occur in the same order as they are specified, even for the receiving of messages sent by different objects (page 483).
- Receiving a message should occur after the sending of the message (page 507).
- GeneralOrdering can add further constraints to OccurrenceSpecifications, which are not related.

Thus the semantics defines partial orders on OccurrenceSpecifications, and valid traces are those, which can be generated satisfying these orders.

3.1.2.3 Semantics of fragments

If no operator is explicitly given, then the InteractionFragments of a diagram should be combined using a form of sequential composition, weak sequencing (page 500). As Figure 3.5 shows, OccurrenceSpecifications are also InteractionFragments; thus this default composition applies also to basic Interactions. The rules for weak sequencing are the following (page 483):

1. "The ordering of OccurrenceSpecifications within each of the operands is maintained."
2. "OccurrenceSpecifications on different lifelines from different operands may come in any order."
3. "OccurrenceSpecifications on the same lifeline from different operands are ordered such that an OccurrenceSpecification of the first operand comes before that of the second operand."

The other sequencing construct, strict sequencing, has a stronger version of the second rule: OccurrenceSpecifications on different Lifelines from different operands become ordered as in the third rule (that is, the content of the first operand comes before that of the second operand).

For the Interactions that use elements from the Fragments package, the semantics is mainly defined in the description of the CombinedFragment element when detailing the various operators (pp. 482–485). We grouped the operators into the categories of Table 3.1.

The first category contains operators that introduce choice and iteration. The operators in the second category are for parallelization and sequencing. Operators in the last category are related to the conformance relation, i.e., the way a trace is categorized as valid, invalid or inconclusive according to a diagram. For example, an *assert* describes a mandatory behavior, while a *neg* one that should not happen. The operators *consider* and *ignore* change the set of message names from which valid and invalid traces can be built (see later in Section 3.3.5.2).

Table 3.1: Operators in CombinedFragment

Operators that introduce choice and iteration	
alt	"alt designates that the CombinedFragment represents a choice of behavior."
opt	"opt designates that the CombinedFragment represents a choice of behavior where either the (sole) operand happens or nothing happens."
break	"break designates that the CombinedFragment represents a breaking scenario in the sense that the operand is a scenario that is performed instead of the remainder of the enclosing InteractionFragment."
loop	"loop designates that the CombinedFragment represents a loop. The loop operand will be repeated a number of times."
Operators for parallelization and sequencing	
par	"par designates that the CombinedFragment represents a parallel merge between the behaviors of the operands."
seq	"seq designates that the CombinedFragment represents a weak sequencing between the behaviors of the operands."
strict	"The semantics of strict sequencing defines a strict ordering of the operands on the first level within the CombinedFragment."
critical	"critical designates that the CombinedFragment represents a critical region. A critical region means that the traces of the region cannot be interleaved by other OccurrenceSpecifications."
Operators that are related to the conformance relation	
neg	"neg designates that the CombinedFragment represents traces that are defined to be invalid."
assert	"assert designates that the CombinedFragment represents an assertion. The sequences of the operand of the assertion are the only valid continuations. All other continuations result in an invalid trace."
ignore	"ignore designates that there are some message types that are not shown within this combined fragment."
consider	"consider designates which messages should be considered within this combined fragment."

Other important classes defined in the Fragments package are *InteractionConstraint* (guards on CombinedFragment) and *variables* (local attributes and parameters of Interactions, arguments of Messages).

The semantics presented in the OMG specification gives a basic idea how Sequence Diagrams should work. However, this natural language semantics is incomplete and ambiguous; thus we need to look into existing formal semantics to understand how Sequence Diagrams are interpreted in practice.

3.2 Overview of proposed semantics

Many formal semantics were proposed for UML 2 Sequence Diagrams over the years. We selected thirteen approaches, listed in Table 3.2. As the UML 2.0 specification completely changed how Interactions are defined (different semantics, introduction of invalid traces and CombinedFragments, etc.), the table does not contain approaches for UML 1.x Sequence Diagrams.

Table 3.2: Summary of proposed semantics

Name	References	Years	Comments/Tools
Störrle	[Stö03a; Stö03b; Stö04]	2003–2004	
STAIRS	[HS03; Hau+05; RHS05a; RHS05b; LS06; Run07; Lun08]	2003–2008	Implemented in Maude
Cavarra & Filipe	[CK04a; CK05a]	2004	
Cengarle & Knapp	[CK04b; CK05b; CGW06; CK08]	2004–2008	
Küster-Filipe	[Küs06; Bow06]	2005–2006	
P-UMLaut	[Eic+05]	2005	P-UMLaut tool
Grosu & Smolka	[GS05]	2005	
Hammal	[Ham06]	2006	
MSD	[HKM07; HM08]	2006–2008	Synchronous, S2A tool
Knapp & Wuttke	[KW07]	2006–2007	HUGO/RT tool
Thread-tag based	[DHC07]	2007	
CPN	[Fer+07]	2007	Synchronous
Template semantics	[SVN08a; SVN08b]	2008	

There are many other papers proposing a semantics for UML 2 Sequence Diagrams (e.g., [Bro+08; Cen07]), and it is impossible to include all of them. The selected 13 approaches contain both pioneering works which influenced most of the others, and less referenced ones which concentrated on specific usages of Sequence Diagrams. It is thus hoped that they are representative for the different possible choices and options, at least to some extent.

Table 3.3 collects which constructs are mentioned in the different approaches. Note that the different approaches sometimes redefine the meaning of the original constructs, and handle the given elements at very different levels of detail. Thus, the goal of this table is not to calculate a percentage of how much of the specification is covered by each work; instead, it may serve as a reference to search which publication mentions a given element.

Table 3.3: Overview of the mentioned elements in each approach

	Störrle	STAIRS	Cavarra & Filipe	Cengarle & Knapp	Küster-Filipe	P-UMLaut	Grosu & Smolka	Hammal	MSD	Knapp & Wuttke	Thread-tag	CPN	Template semantics
Interaction													
Local attributes		●				●							
GeneralOrdering	●			●		●				●	●		
Message													
argument		●				●				●			
asynchronous	●	●	●	●	●	●	●	●		●	●	●	●
lost, found					●	●							
creation, destruction						●							
CombinedFragment													
guard		●	●		●	●		●	●	●			●
alt	●	●	●	●	●	●	●	●	●	●	●	●	●
opt	●	●		●		●		●	●	●	●	●	●
loop	●	●		●		●	●	●	●	●	●	●	●
break	●					●		●	●				●
par	●	●	●	●	●	●		●		●	●	●	●
seq	●	●	●	●	●	●	●	●	●	●	●	●	●
strict	●	●	●	●		●	●	●	●	●	●		
critical	●					●							●
neg	●	●	●	●	●		●		●	●			
assert	●	●	●	●	●				●				
ignore	●			●					●	●			
consider	●			●					●				
Other elements													
Gate		●			●	●							
StateInvariant		●	●		●	●			●	●			
DurationConstraint	●	●						●					
TimeConstraint	●	●						●					
InteractionUse (ref)	●	●			●	●	●						
argument						●							

If we look through the table, the following observations can be made:

- Conformance-related operators were not considered in one third of the approaches. Even if it is one of the most important aspects of the language, it is hard to formalize it and solve its issues. Moreover, *consider* and *ignore* were not mentioned in four of the eight that dealt with conformance.
- Gates were handled explicitly in only a small number of papers.
- Variables and arguments were also not mentioned in several approaches. It is understandable, because they are not in the focus of Sequence Diagrams, and not easy to express in some of the formalisms.
- Handling time and time constraints was also not common.
- Some of the elements (ExecutionSpecification, Continuation, PartDecomposition) were not explicitly handled in any of the approaches; thus we left them out from the table.

The rest of the section gives a brief description of each of the approaches. As there are 13 approaches, the overall content is quite long. Readers more interested in the different semantics choices can jump directly to the discussion in Section 3.3, and return to some of the approaches later.

3.2.1 Trace semantics from Störrle

Störrle was one of the firsts to propose a semantics for UML 2 Sequence Diagrams in [Stö04] (previously published in [Stö03a; Stö03b]). It is a trace-based semantics, which contains much of the elements of the OMG specification. The semantics defined the set of valid and invalid traces for "plain InteractionFragments", i.e., ones without CombinedFragment. Later, for CombinedFragment the semantics of each operator is presented. At that time, the OMG specification was still in a draft version; since then a few element names have changed. Section 3.1 in [Stö04] analyzes the semantic approach used in the OMG specification and finally categorizes it as an interleaving, linear-time semantics of complete traces with abstract real time. Section 5 in [Stö04] deals with *assert* and *neg* in detail, and gives several potential meanings for the *neg* operator. It also points out many issues with the OMG specification.

3.2.2 STAIRS approach

In [HS03] the authors introduce STAIRS (Steps To Analyze Interactions with Refinement Semantics). They define a denotational, trace-based semantics for Sequence Diagrams, where the focus is on the precise definition of refinement for Interactions. Three types of refinement are defined:

- *Supplementing*: inconclusive traces are categorized as either positive or negative;
- *Narrowing*: some of the positive traces are categorized now as negatives;
- *Detailing*: introducing a more detailed description without significantly altering the externally observable behavior.

In [Hau+05] the approach is extended to Timed STAIRS; the semantics is modified in a way that the reception and consumption of messages are differentiated (this leading to three event types: transmission, reception, consumption). In [RHS05a] the interpretation of the *neg* operator is analyzed, and new operators (*refuse*, *veto*) are proposed instead of it. The dissertation [Run07] summarizes the denotational STAIRS and its extensions.

In [LS06] (and later greatly extended in [Lun08]) an operational semantics is given complying with the above denotational semantics. In Section 7.3 of [Lun08] a good overview is given of the challenges when defining semantics for UML Sequence Diagrams. The operational semantics uses a reduced abstract syntax given by a grammar to represent Sequence Diagrams. The model of the operational semantics consists of an execution system, which stores the state of the communication channels and the sequence diagram, and a projection system, which finds the enabled events. The operational semantics is also implemented in the Maude language.

3.2.3 ASM-based semantics of Cavarra & Filipe

In [CK05a] the authors proposed a technique using Object Constraint Language (OCL) templates to express liveness properties in UML Sequence Diagrams, based on results of LSC [DH01]. Using concepts from LSC, several problematic parts of the OMG specification were addressed. May and must behavior, universal, and existential diagrams can be differentiated. In Fig. 2 in [CK05a] the authors give a nice example that certain liveness properties cannot be expressed with $assert$ or neg. Therefore, they propose an after/eventually OCL template, which says that after a condition becomes true there is a guarantee that eventually another condition will become true. Moreover, they introduce global constraints and methods for synchronization at the beginning or end of CombinedFragments.

In [CK04a], the authors defined a semantics to this liveness-enriched Sequence Diagrams using abstract state machines (ASM). Locations are associated with each important point on the Lifelines. For each instance, a separate process is assigned. ASM rules are defined to specify the progress of one instance depending on what kind of fragment the instance currently is in. In the conclusion, several good observations are made on the challenges of UML Sequence Diagrams.

3.2.4 Trace-based semantics of Cengarle & Knapp

In [CK04b] the authors define a denotational semantics for the traces of Interactions using pomsets (partially ordered multisets). Later, in [CK05b] an operational semantics is given for Sequence Diagrams. The semantics of the positive fragments is similar to the one defined by Störrle. The authors concentrate on the interpretation and definition of negative fragments. Rules are given for each of the operators specifying whether a trace positively or negatively satisfies a fragment with that operator. The authors point out that with the basic interpretation of negative fragments it is easy to construct *overspecified* Interactions, i.e., an Interaction that can be positively and negatively satisfied from the same trace. In the paper [CK04b] the operator not is introduced instead of neg and $assert$ to overcome some of the problems with negative satisfaction. Later, the work is extended in [CK08] to define the semantics using a different formalism (namely institutions), and in [CGW06] to handle variability expressed on a diagram.

3.2.5 True-concurrency semantics from Küster-Filipe

Küster-Filipe defined a true-concurrent semantics based on event structures in [Küs06]. In [Bow06] the semantics is extended to handle the InteractionUse construct. It considers only a smaller number of operators and constructs (alt, par, seq, and StateInvariant), but gives them a well-defined semantics.

The semantics uses the temperature (hot and cold messages) concept from LSC to express mandatory or possible behavior. For example, hot messages *must* be received, while cold messages *may* be received after sending. Furthermore, it uses LSC's location concept to mark occurrences on a Lifeline.

The approach constructs for every Lifeline a labeled prime event structure. The model takes into account the possible nesting of CombinedFragments and gives a very clear definition for the predecessors of every event. Finally, the event structures for the different Lifelines are combined according to the Messages sent between them. In the end of [Küs06] a two-level temporal logic is presented, which can be used to specify Interactions.

3.2.6 M-net based semantics of the P-UMLaut project

In [Eic+05] a semantics is given for Sequence Diagrams based on M-nets (multivalued nets), which is an algebra based on high-level Petri nets. The method handles basic data types (Boolean and integers); thus, it can include the local attributes of Interactions, the arguments of Messages, and the evaluation of conditions in the semantics. M-net fragments are given for basic constructs, like starting of a Lifeline or sending and receiving of a message. These are then connected by composition operators according to the enclosing CombinedFragment's operator. The semantics defined in the paper assumes that all behavior is explicitly specified in the diagrams and no conformance-related operator is used.

3.2.7 Safety-liveness semantics from Grosu & Smolka

In [GS05] the authors propose to interpret valid and invalid parts of an Interaction as liveness and safety properties, respectively. The Sequence Diagrams are first transformed to hierarchic, nondeterministic automata, then the high-level automata are flattened, and finally liveness Büchi automata are constructed from the positive automata, and safety Büchi automata from the negative ones. Based on the languages these automata accept, refinement of Sequence Diagrams is defined.

The paper only treats the combination of basic diagrams with no CombinedFragment and bounded high-level Interaction Overview Diagrams. In this way, their trace language is regular, but it is a restriction of the OMG specification.

3.2.8 Branching time semantics from Hammal

The author of [Ham06] presents a denotational semantics based on partial orders. It assigns to each fragment a graph containing the OccurrenceSpecifications and their relations. The structures are later enriched with timing information using the timing constraints on the diagram.

3.2.9 Modal Sequence Diagrams

Modal Sequence Diagrams (MSD) [HKM07; HM08] are an extension to UML Sequence Diagrams by Harel and Maoz, which adapts LSCs to the notation of UML. LSC is a language inspired from MSC that allows the specification of possible and mandatory scenarios.

The authors point out that the root of all the challenges regarding $assert$ and neg are that these were introduced as simple operators, while they are rather modalities. UML Sequence Diagrams do not have a clear definition of the modalities of the diagrams and thus the authors apply the model of LSC to UML. The *modal* stereotype is attached to InteractionFragments to specify whether it describes a hot (universal) or cold (existential) behavior. A hot fragment represents a behavior that is mandatory, while the cold represents only a possible behavior. The operators $assert$ and neg are used then just as syntactic notation to show whether the constructs inside them have hot or cold modality. The authors also treat the question how multiple diagrams should be handled, one point that is often missing from others.

In the Appendix, a formal semantics based on weak alternating automata is sketched. First, the diagram is transformed into an intermediate format, an unwinding structure, from which the states (the cuts of the diagrams) and the transitions (message sending) of the automaton are derived. The current semantics considers only synchronous messages; the sending and receiving is treated as one event.

3.2.10 Operational semantics from Knapp & Wuttke

The paper [KW07] proposes an operational semantics, where an interaction automaton is produced by unwinding the Interaction. One single interaction automaton is created for the entire Interaction. The authors apply some restrictions to ease the processing of Sequence Diagrams (e.g., replace *neg* with the binary logic variant *not* introduced in [CK04b], restrict the use of *not* only to basic interactions, restrict loops to only allow basic interactions, etc.). Later, this interaction automaton is used as an observer process in the SPIN model checker to check the communication produced by UML State Machines.

3.2.11 Thread-tag based semantics

In [DHC07] a trace semantics was proposed for specifying object-oriented programs with multiple threads on the same Lifelines. The authors claim that if the instances of the Interaction are multi-threaded objects, then the ordering should not be specified for messages originating from the same Lifeline; instead, only for those messages which are from the same Lifeline and from the same thread of the Lifeline. For this reason, they extend messages with "thread tags", i.e., identifiers specifying which the sender and receiver threads for that message are. Later, a *trace-based semantics* is given for the operators, where the ordering rules are defined with respect to thread tags. Conformance-related operators are not considered in the paper.

In our opinion, some of the problems presented in the paper can be solved without modifying the original semantics with the help of inline PartDecompositions, i.e., when an instance is decomposed to multiple Lifelines representing its inner connectable elements, like the threads of an object.

3.2.12 Semantics based on CPN

In [Fer+07] the authors propose a translation that produces a Colored Petri Net from UML use cases and Sequence Diagrams. For the basic operators (*opt*, *alt*, *par*, *loop*, and *ref*), templates are assigned to show what kind of CPN fragment should be created. The translation does not consider conformance-related operators. It seems, although it is not stated explicitly in the paper, that each diagram contains initially only one active instance (it can later fork into several executions with a *par*). Only synchronous messages are handled, because the sending and receiving are represented by the same transition.

3.2.13 Template semantics

In [SVN08a] a formalization using template semantics is proposed for UML 2 Sequence Diagrams. The formalization is described in more detail in the technical report [SVN08b]. The approach gives an operational semantics for which the basic computation model is hierarchical transition systems (HTS). First, the maximal sequence fragments of the diagram are computed, i.e., the maximal sequences of consecutive Messages that do not contain CombinedFragments. Then, for each Lifeline a complex HTS is formed by composing the maximal blocks of the Lifeline using the InteractionOperators. Finally, the HTSs for the Lifelines are composed using interleaving operators.

3.3 Semantic choices in Sequence Diagrams

Section 3.1.2 presented the informal semantics defined in the OMG specification. As it could be seen from the overviews in Section 3.2, several approaches were proposed to formalize the semantics of UML Sequence Diagrams. This section *collects* and *categorizes* the different choices taken by these approaches. Table 3.4 presents our categorization of the semantic choices collected.

Table 3.4: Categorizing semantic choices in UML 2 Sequence Diagrams

Interpretation of a basic Interaction	What is a trace? Categorizing traces Complete or partial traces
Introducing CombinedFragments	Combining fragments
Computing partial orders	Processing the diagram Underlying formalisms Choices and predicates
Introducing Gates	Gates on CombinedFragments Formal and actual Gates
Interpretation of conformance-related operators	Assert and negate Ignore and consider Conformance-related operators in complex diagrams Traces being both valid and invalid

First, the various interpretations of the basic concepts are listed (Section 3.3.1). Next, the methods for handling the concept of CombinedFragment are analyzed (Section 3.3.2). Section 3.3.3 collects how the partial orders of a diagram are computed, and how the related operators and elements (alternatives, guards, etc.) are handled. Section 3.3.4 is about the different types of Gates. Finally, Section 3.3.5 details the handling of conformance-related operators.

When necessary, the discussion is illustrated by example diagrams containing traces that show the difference between the options listed in the given section. Since the approaches differ quite heavily in their formalization (basic definitions, symbols to use, etc.), we present each option without its respective formal definition. Some of the subsections do not list every approach, as some UML elements are not considered by all the approaches.

Furthermore, each subsection ends with a diagram summarizing the different choices and options. Figure 3.9 illustrates the notation used in these diagrams, which was inspired by feature models [Kan+90]. For example, for A both B and C has to be selected, for D only one of E or F can be chosen, while H is an optional choice, which may or may not be selected. The † symbol marks an option, which departs from the OMG specification. Note, however, that there is no negative connotation associated with this symbol, as several "non-standard" options proved to be really useful for specific applications.

3.3.1 Interpretation of a basic Interaction

Let us start the discussion with a simple diagram without any explicit operator. Because the semantics of an Interaction is defined as the valid and invalid traces produced by the diagram, first the content

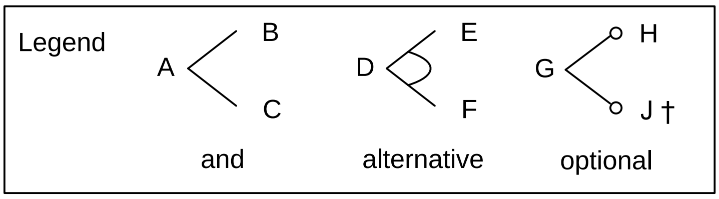


Figure 3.9: Notation used in the summary diagrams

of a trace has to be specified.

3.3.1.1 What is a trace?

Since the purpose of the semantics is to categorize traces, a definition of traces is needed. Usually to simplify the representation, the notation of $!m1.!m2.?m1.?m2$ is used for traces, where $!m$ denotes sending and $?m$ denotes receiving the message m. However, in a formal semantics, one has to be more explicit, e.g., because there can be several Lifelines in the Interaction sending messages with the same name, it should be specified who sent or received the message.

Thus some of the semantics (STAIRS, Cengarle & Knapp, Grosu & Smolka, Template semantics) represents elements of the trace with tuples, e.g., (action, sender, receiver, message name).

However, on a diagram, where the same message name appears twice between the same Lifelines, the above notation cannot describe the ordering that the receiving of the first m message should come before the receiving of the second one (Figure 3.10).

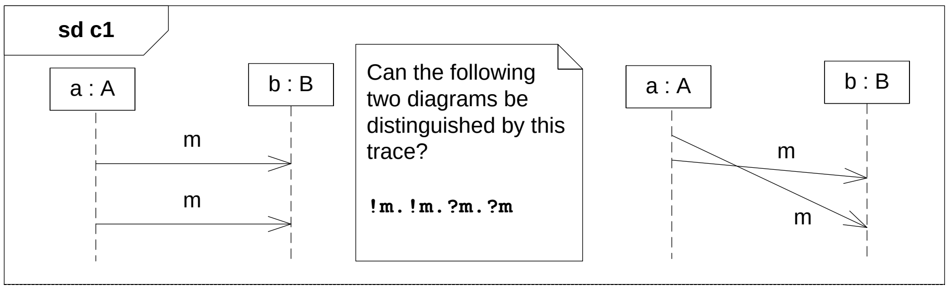

sd c1
a : A
b : B
m
m
Can the following two diagrams be distinguished by this trace?
!m.!m.?m.?m
a : A
b : B
m
m

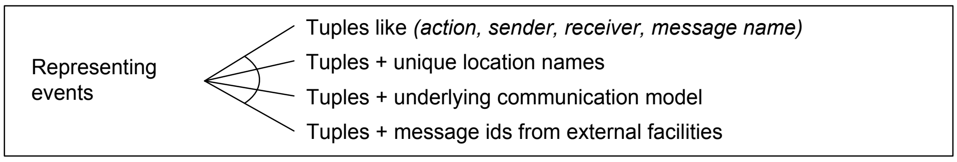
Figure 3.10: Handling ordering constraints from duplicate messages

Using explicit locations can help this: each OccurrenceSpecification is assigned a unique location name; thus the two receptions of Signal m can be differentiated. The location names are symbolic labels that usually conform to the visual position of the location. Approaches using locations are Störrle, Cavarra & Filipe, Küster-Filipe, P-UMLaut, Hammal, and MSD.

Another option can be to specify the underlying communication model. For example, in the operational version of STAIRS, it can be specified whether the execution model should use a global FIFO or one FIFO for each Lifeline, etc.

The above solutions are defined for symbolic traces. In order to analyze concrete system traces, the receiving events should be matched with the sending event that caused it, which is only possible if each message in the trace can be uniquely identified. Thus, each message should have a unique identifier obtained from some external monitoring facility. See, e.g., [Hal+06] for such a definition of a trace, and for applications to monitoring distributed systems.

Note that in our example diagrams, for the sake of simplicity, we will use the shorthand $!m$ instead of $(send, lifeline, m, id)$.

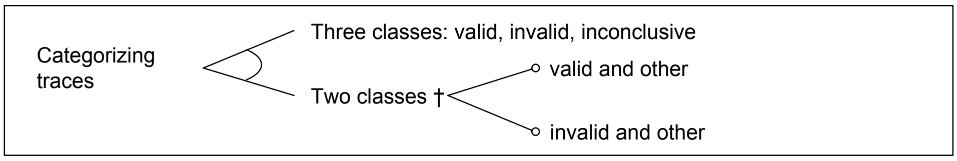

Representing events
Tuples like (action, sender, receiver, message name)
Tuples + unique location names
Tuples + underlying communication model
Tuples + message ids from external facilities

3.3.1.2 Categorizing traces

Once it is defined how to represent a trace, it should be decided how to categorize the traces. The UML specification gives the semantics of an Interaction as a set of valid and a set of invalid traces. However, it states that there can be other traces, for which we cannot know whether they are valid or invalid.

For the approaches that use Sequence Diagrams for specification and refinement (Störrle, STAIRS, Cengarle & Knapp) using all the three classes is convenient. Usually, they define the valid and invalid traces explicitly, and all other traces are considered inconclusive. Cavarra & Filipe and Küster-Filipe do not explicitly mention inconclusive traces, but they have a separate "aborted" mode for invalid traces; thus they are able to differentiate invalid and inconclusive traces. In MSD, locations and messages have cold or hot temperature assigned. A cold message depicts a potential behavior; thus, a trace violating it is considered as an inconclusive one. A hot message represents a mandatory behavior; its violation results in an invalid trace.

Some approaches differentiate only two classes of traces. The ones using Sequence Diagrams for verification purposes (Grosu & Smolka, Knapp & Wuttke) separate the traces into either valid/other or invalid/other classes. The focus on validity or invalidity depends on whether the property to be checked is a liveness property (a valid trace is exhibited) or a safety one (no invalid trace is exhibited). The other approaches (P-UMLaut, Hammal, Thread-tag, CPN, Template semantics) are not dealing with conformance-related operators; hence, they do not have invalid traces and may be classified into the valid/other category.



3.3.1.3 Complete or partial traces

According to the OMG specification, basic Sequence Diagrams specify complete, potential behaviors, meaning that the traces represented by the Interaction are examples for valid traces, and all the other traces are inconclusive with respect to the given diagram. Thus, the standard interpretation of the diagram in Figure 3.11 is that $!m1.?m1.!m2.?m2.!m3.?m3$ is valid and all other traces are inconclusive. Most of the approaches use this interpretation.

Sometimes this interpretation is not convenient, e.g., when one would like to specify requirements [HM08], safety properties [GS05] or test purposes [Pic03]. For this reason, two of the semantics use an interpretation with partial traces, i.e., the diagram depicts only parts of the valid traces; other

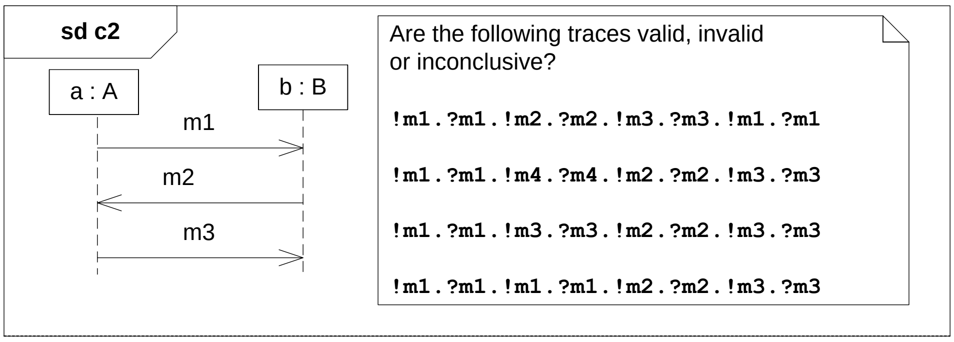

sd c2
a : A
b : B
m1
m2
m3
Are the following traces valid, invalid or inconclusive?
!m1.?m1.!m2.?m2.!m3.?m3.!m1.?m1
!m1.?m1.!m4.?m4.!m2.?m2.!m3.?m3
!m1.?m1.!m3.?m3.!m2.?m2.!m3.?m3
!m1.?m1.!m1.?m1.!m2.?m2.!m3.?m3

Figure 3.11: Interpretation of a basic Interaction

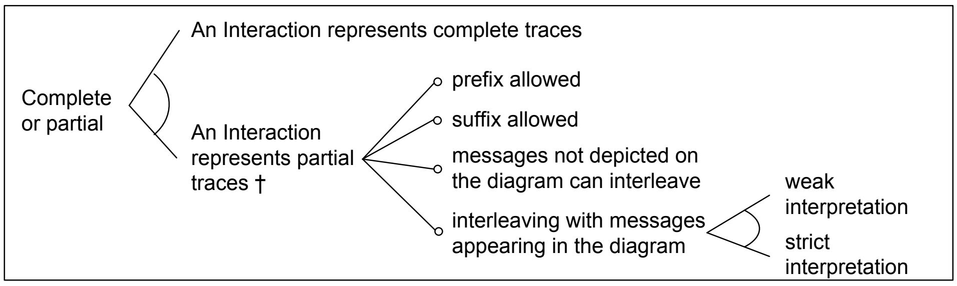
messages can interleave with them to form the complete, valid traces. These extra messages usually come from two sources:

- There can be a prefix or suffix for the diagram.
- During the processing of the diagram, messages can interleave with the ones depicted explicitly on it (e.g., messages coming from other Lifelines, other message types not used on the diagram, or duplicate messages).

Grosu & Smolka use an interpretation where a valid trace can have any suffix, but no prefix. In MSD there may be any prefix or any suffix in the system trace before the shown behavior occurs (e.g., in Figure 3.11 $!m1.?m1.!m2.?m2.!m3.?m3.!m1.?m1$ is a valid trace for both approaches).

For handling interleaving messages, several interpretations are possible (see the discussion conducted in [Klo03]). The shown events usually may interleave with other events that are not explicitly mentioned in the diagram (e.g., in Figure 3.11 $!m1.?m1.!m4.?m4.!m2.?m2.!m3.?m3$ is a valid trace for both Grosu & Smolka and MSD). The difference is for the messages appearing on the diagram. For example, after receiving $m1$, are $m1$ or $m3$ allowed to appear? The *strict interpretation*, which is used in MSD, is that the diagram is complete with respect to occurrence specifications that are given in it explicitly. Therefore, neither $m1$ nor $m3$ are allowed right after an $m1$ message is matched. The *weak interpretation* (a form of which is used by Grosu & Smolka) is less restrictive with respect to the shown occurrence specifications. It only requires that the trace events occur in the specified order (e.g., an $m2$ message is in the future of $m1$) and may as well accept duplicates. Hence, the trace $!m1.?m1.!m3.?m3.!m2.?m2.!m3.?m3$ is valid for Grosu & Smolka, but not for MSD. Note that the trace $!m1.?m1.!m1.?m1.!m2.?m2.!m3.?m3$ is valid for both approaches; however, for MSD it is valid only if the first $m1$ message is considered as a prefix, and only the second $m1$ message is matched for the diagram.

Intuitively, interpretations with partial traces amount to filter out the behavior that is irrelevant to the categorization of traces: trace prefix, suffix, or extra interleaving events are ignored and categorization is based on the remaining part of the trace. It turns out that two operators of the UML 2.0 Sequence Diagrams, $ignore$ and its dual operator $consider$, allow us to further manipulate the set of events appearing in the traces. Not surprisingly, the decision of whether to work with partial or complete traces will have a strong impact on how these operators are interpreted. We will come back to this issue in Section 3.3.5.2 discussing $ignore$ and $consider$.


Complete or partial
An Interaction represents complete traces
An Interaction represents partial traces †
prefix allowed
suffix allowed
messages not depicted on the diagram can interleave
interleaving with messages appearing in the diagram
weak interpretation
strict interpretation

3.3.2 Introducing CombinedFragments

The OMG specification defines weak sequencing as the default composition operator for fragments. Accordingly, most semantics retain this operator to compose a CombinedFragment with the rest of the diagram (Störrle, STAIRS, Cengarle & Knapp, Küster-Filipe, Knapp & Wuttke, Thread-Tag based, Template semantics).

Due to the weak sequencing, events that do not belong to the same lifeline can occur independently if they are not related by a path of messages. Figure 3.12 exemplifies this. Message $m1$ is located above the *opt* fragment, but there is actually no precedence relation: $m1$ can occur independently of messages $m2$ and $m3$. Similarly, placing something below a CombinedFragment does not necessarily mean that it comes after the messages inside the CombinedFragment. In Figure 3.12, there is an ordering constraint between $m2$ and $m3$ only because they share lifeline *c*. If the optional message $m2$ does not occur, then there is no constraint on $m3$. For example, trace $!m3.!m1.?m1.?m3$ is valid.

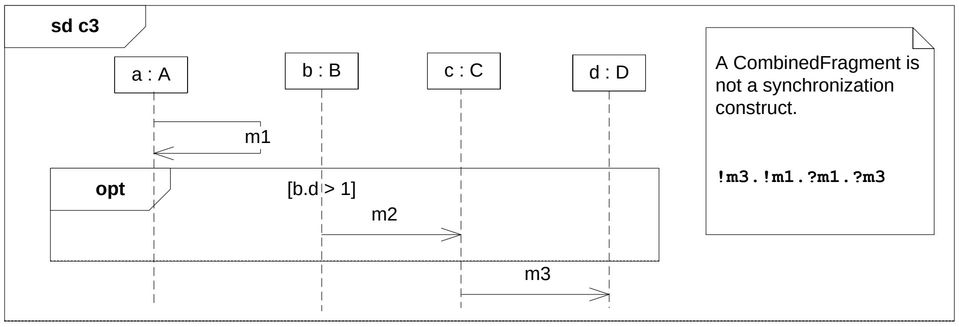


Figure 3.12: Composition with weak sequencing: above/below positions do not imply before/after relations

For such semantics, there is no synchronization point for crossing the borders of an operator. Technically, entering or exiting an operator is not an *OccurrenceSpecification.* As far as we understand the OMG specification, the only OccurrenceSpecifications are (1) sending and receiving of Messages and (2) start and end of an ExecutionSpecification. These are the events that can appear in traces, and ordering constraints are defined for them only.

Also, the spatial extension of operators has no specific meaning if weak sequencing is used. In Figure 3.12, the *opt* box expands to all lifelines, but the meaning would be the same if it covered lifelines *b* and *c* only. This interpretation enjoys the property that an empty box is equivalent to no box (except for conformance-related operators, to be discussed in Section 3.3.5).

As a last example of how weak sequential composition determines the interpretation of diagrams, let us take an example with a loop (Figure 3.13). The meaning of the *loop* operator is given as the recursive application of the *seq* operator. Because weak sequencing is used between the successive iterations of the loop, the trace where all the sending of *m1* and *m2* happens first, and all the receiving comes after it, is a valid trace.

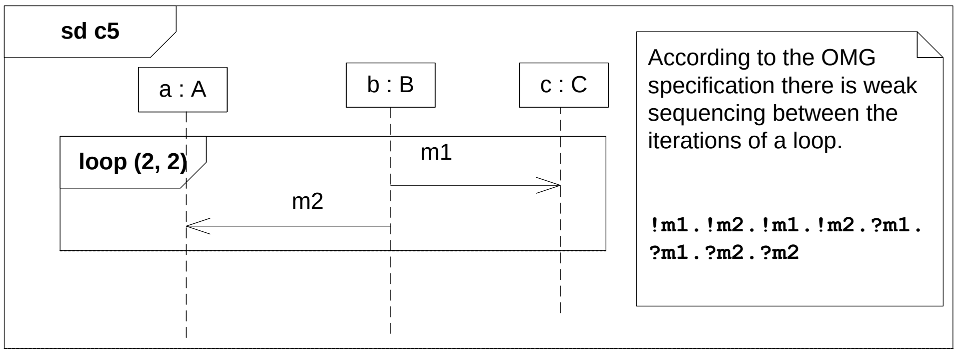

sd c5
a : A
b : B
c : C
loop (2, 2)
m1
m2
According to the OMG specification there is weak sequencing between the iterations of a loop.
!m1.!m2.!m1.!m2.?m1.
?m1.?m2.?m2

Figure 3.13: Loop means a weak sequencing between the iterations of the loop

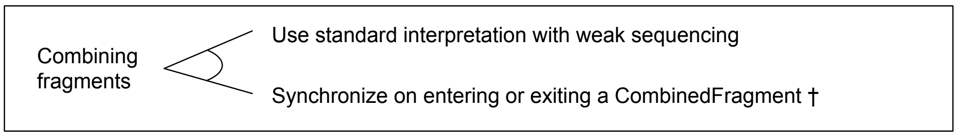
While weak sequencing is the default according to the OMG specification, five semantics we reviewed introduce synchronization on entering and exiting fragments (Cavarra & Filipe, P-UMLaut, Hammal, MSD, CPN). This nonstandard interpretation is usually adopted for work using Sequence Diagrams for verification purposes. It is well known from previous work on MSCs that such graphical scenario languages are neither regular nor context-free, which raises decidability issues [MP05]. The synchronization then allows a reduction of the described partial orders of events, and makes properties easier to check. For example, assume the loop in Figure 3.12 can have an arbitrarily high number of iterations. The language is not regular with the standard interpretation, while it becomes regular if synchronization is enforced at each iteration.

In addition to reducing the expressive power of the language, the consequences of the synchronization are the following:

- Above/below positions now imply before/after relations, making the interpretation of the diagram close to the visual intuition;
- the spatial extension of boxes does matter, forcing each involved lifeline to synchronize;
- an empty box is no longer equivalent to no box; and
- the loop construct has an interpretation that is similar to the one of loops in programming languages.

None of the traces shown in Figure 3.12 and Figure 3.13 is valid for the semantics enforcing synchronization.

Some authors have proposed to retain the weak sequencing as the composition operator, except for loops where a new construct (*sloop*) makes it possible to consider strict sequencing of loop iterations [KW07].


Combining fragments
Use standard interpretation with weak sequencing
Synchronize on entering or exiting a CombinedFragment †

3.3.3 Computing partial orders

The UML 2 specification defines the rules for computing the orderings between the OccurrenceSpecification on a simple diagram (see Section 3.1.2.2). This is usually a partial order because there can be independents events in the Interaction. For example, in the leftmost diagram on Figure 3.14, $!m1$ and $!m2$ are not related while $?m1$ has to come before $?m2$.

With CombinedFragments this default ordering can be modified, e.g., in the middle diagram on Figure 3.14 the ordering between $?m1$ and $?m2$ is also relaxed, and we no longer have a complete ordering for events on the same Lifeline. An important thing to note is that when using par, the immediate predecessor and successor of OccurrenceSpecifications become sets. For example, in diagram $c7$ the predecessor of $!m3$ can be $!m1$ or $!m2$. Likewise, there is no such concept as the immediate next event; instead, there is a set of events. Finally, alternate fragments define several partial orders, one for each of their operands. In the rightmost diagram on Figure 3.14, there are two partial orders, one over the set of events $\{!m1, ?m1, !m3, ?m3\}$ and the other one over the set $\{!m2, ?m2, !m3, ?m3\}$.

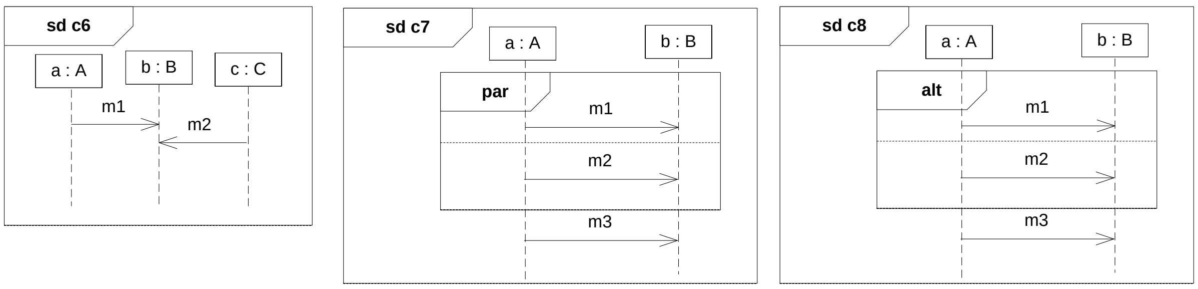


Figure 3.14: Partial orders in diagrams

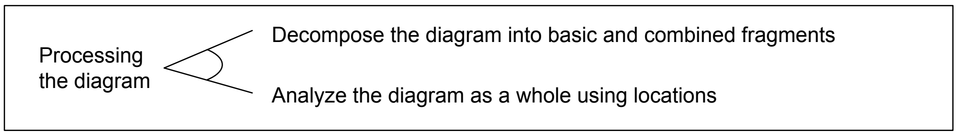
When a diagram contains several CombinedFragments their effects combine. It may result in complex orderings, which are not trivial to calculate. Thus, a significant question about a semantics is how it computes the orderings for an Interaction.

3.3.3.1 Processing the diagram

The approaches in the proposed semantics can be categorized in the following two main categories.

The semantics in the first category parse the diagram and decompose it. The CombinedFragments and the basic fragments in the diagram are identified (Störrle, P-UMLaut, Hammal, Thread-tag, CPN, Template semantics); some approaches even build a syntax tree from the elements of the diagram based on an abstract syntax (STAIRS, Cengarle & Knapp, Knapp & Wuttke). Usually, the parsing from a diagram's concrete syntax to this intermediate representation is not given in detail (some rules can be found in [Eic+05] or in [SVN08a] based on maximal independent sets). After the parsing, the semantics is computed by recursively unfolding the fragments and gluing them together based on rules defined for each of the operators.

The semantics in the second category analyze the diagram as a whole. The locations in the diagram are labeled, and the constraints about the relative ordering of locations are computed. The semantics connected to LSC use this approach (Cavarra & Filipe, Küster-Filipe, MSD). Küster-Filipe computes the event sequences leading to each of the locations. In MSD the first step is to obtain the valid cuts of the diagram from the analysis of locations.


Processing the diagram
Decompose the diagram into basic and combined fragments
Analyze the diagram as a whole using locations

3.3.3.2 Underlying formalisms

In a semantics, the underlying formalism has a significant impact on how the orderings are computed and expressed. Table 3.5 summarizes the formalisms used in the surveyed approaches.

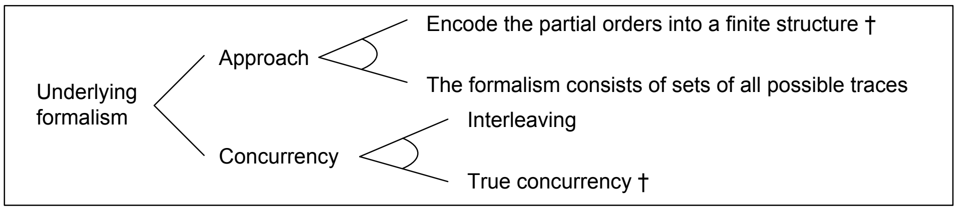
The diversity of formalisms in the approaches is the consequence of the *diversity of interests* for using Sequence Diagrams. Some authors define the semantics to check traces (e.g., Knapp & Wuttke), some to compute all possible traces of a diagram (e.g., Störrle), some use the semantics to support refinement-based development (e.g., STAIRS), or translate the diagrams into behavior models in order to connect to existing simulation or verification tools (e.g., P-UMLaut). The different purposes can be supported in either one or the other formalism more easily.

Table 3.5: Underlying formalisms in the semantics

Name	Type of semantics	Concurrency
Störrle	Denotational semantics, rules for computing the set of traces	Interleaving
STAIRS	Denotational semantics, rules for computing the set of traces; operational semantics based on transitional systems	Interleaving
Cavarra & Filipe	Building Abstract State Machines, the ASMs accept or reject a trace	True concurrency
Cengarle & Knapp	Denotational semantics based on pomsets; operational semantics based on pomsets	True concurrency
Küster-Filipe	Denotational semantics based on event structures	True concurrency
P-UMLaut	Translating to M-nets	True concurrency
Grosu & Smolka	Translating to Büchi automaton, semantics is defined by the traces accepted by the automaton	Interleaving
Hammal	Denotational semantics based on graphs representing all traces	Interleaving
MSD	Building an alternating Büchi automaton, the automaton defines the trace-language accepted by the diagram	Interleaving
Knapp & Wuttke	Building an interaction automaton, the automaton observes traces and accepts or rejects them	Interleaving
Thread-tag based	Denotational semantics based on pomsets	True concurrency
CPN	Translating to Colored Petri nets	True concurrency
Template semantics	Operational semantics using Hierarchical Transition Systems	Interleaving

As a general comment, the underlying formalisms can be differentiated depending on whether they encode the partial orders into a finite structure (Cavarra & Filipe, Küster-Filipe, P-UMLaut, Grosu & Smolka, Hammal, MSD, Knapp & Wuttke, CPN, Template semantics), or they consist of sets of all possible traces (Störrle, STAIRS, Cengarle & Knapp, Thread-tag). With the first approach it is easier to verify traces, but it is usually feasible only with some syntactic restrictions (like Knapp & Wuttke allowing only basic fragments nested in a *neg*) and an interpretation that reduces the described partial orders (e.g., by synchronizing lifelines at the borders of fragments).

In MSD the model of not just one Interaction, but a system consisting of several Interactions is also defined. This is consistent with the fact that in MSD Interactions define only partial traces.


Underlying formalism
Approach
Encode the partial orders into a finite structure †
The formalism consists of sets of all possible traces
Concurrency
Interleaving
True concurrency †

3.3.3.3 Choices and predicates

The definition of choices and predicates in the OMG specification is very permissive. As will be seen in this section, there are numerous options *what* and *when* to choose and *who* chooses.

What An *alt* offers much more flexibility than an *if* construct in traditional programming languages would: several of its operands can have implicit true guards, from which one is non-deterministically chosen. Some approaches try to reduce this non-determinism. Cavarra & Filipe prescribe that the operands of the *alt* are evaluated from top to bottom, and the first one evaluated as true be chosen (a similar concept, *deterministic alt* was introduced in the UML 2 Testing Profile [OMG05]).

Who The UML 2 specification does not define *who* should make the choice between the operands of an *alt*. This can lead to *non-local choices*, a problem well studied in MSC [MGR05]. An example for non-local choice can be seen on Figure 3.15, where either instance *a* sends *m1* or instance *b* sends *m2*, but not both. For semantics working with complete traces, non-local choices raise implementation problems: it may be impossible to implement a system, which shows the valid traces of the diagram. Most of the semantics accept non-local choice as a consequence of having a high-level, powerful specification language.

When With the introduction of synchronization at the beginning and end of CombinedFragments (Section 3.3.2) some approaches specify a common point in time when all Lifelines have to make the choice.

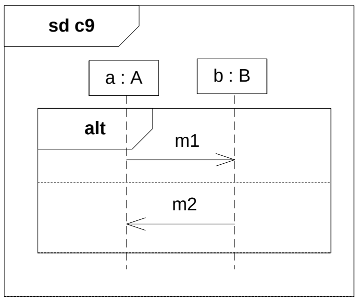


Figure 3.15: Simple non-local choice

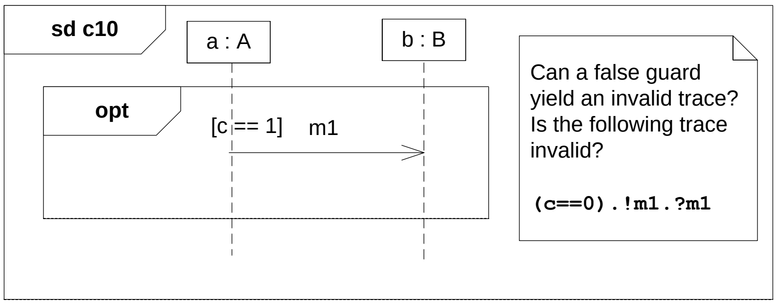

sd c10
a : A
b : B
opt
[c == 1]
m1
Can a false guard yield an invalid trace? Is the following trace invalid?
(c==0).!m1.?m1

Figure 3.16: Handling of explicit guards

Thus handling choices is a complex issue. The main approaches used in the different semantics are the followings.

- *No explicit time point for the choice*: The sets of traces from each operand are computed independently and are combined with the rest of the diagram using the default weak sequencing to obtain all the possible traces of a diagram (Störrle, STAIRS, Cengarle & Knapp, Thread-tag).

- *Explicit time points for the choice on each Lifeline*: Lifelines process the diagram separately and choose between operands independently (Cavarra & Filipe, Küster-Filipe, Template semantics). Therefore, each Lifeline could make its choice at different times, but the semantics guarantees that all Lifelines choose the same operand (e.g., by fixing the evaluation order of operands).
- *Explicit global time point for the choice*: All involved lifelines synchronize before entering a choice, and only one global choice is made (P-UMLaut, Hammal, Grosu & Smolka, MSD, Knapp & Wuttke, CPN). These approaches typically use an automaton-based formalism, where one transition represents the taken choice for all Lifelines.

So far we only tackled implicit guards. Several semantics do not handle explicit guards. For the ones that do, a difference is how a false guard is interpreted. STAIRS processes guards similarly to constraints; thus a trace with a false guard is invalid, while for the other approaches (Cavarra & Filipe, Küster-Filipe P-UMLaut, Hammal, MSD, Knapp & Wuttke) a guarded choice cannot yield invalid traces. For example, the trace given in Figure 3.16 is invalid for STAIRS, while for the others it is not.

There are several options regarding who should evaluate the guard. The evaluation could be local to one Life-line (STAIRS, Küster-Filipe), all Lifelines could interpret the guard separately (Cavarra & Filipe), or the guard could be evaluated globally (P-UMLaut, Hammal, MSD, Knapp & Wuttke). The latter option is consistent with an explicit global time point for the choice.

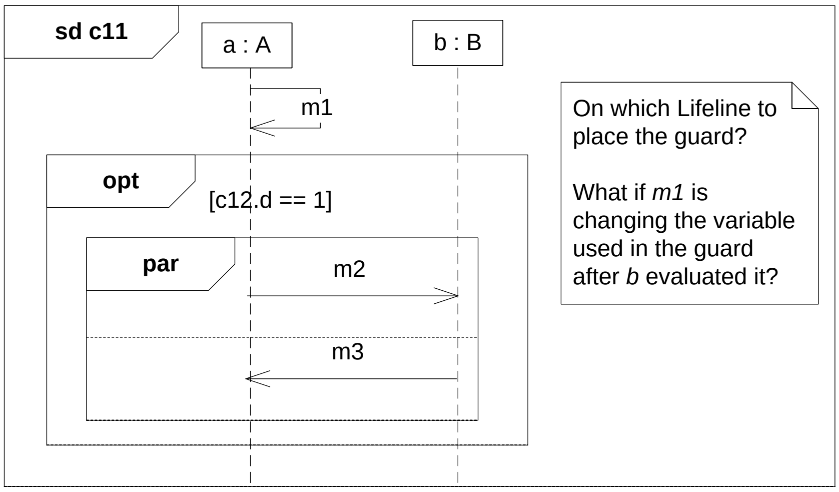
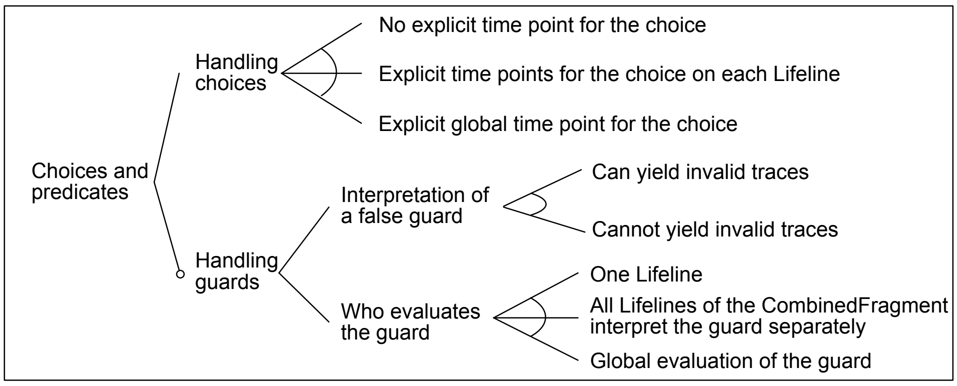

sd c11
a : A
b : B
m1
opt
[c12.d == 1]
par
m2
m3
On which Lifeline to place the guard?
What if *m1* is changing the variable used in the guard after *b* evaluated it?

Figure 3.17: Data used in guards

Evaluating the guards separately or referring to global data may lead us to *scope* and *well-definedness* problems. As pointed out by Section 4.4 in [Eic+05] if Lifelines can evaluate the guards at different times, the value of the guard can change in the meantime. The UML 2 specification prescribes that the guard should be placed "on the lifeline where the first event occurrence will occur, positioned above that event, in the containing Interaction or InteractionOperand". However, as we mentioned before, "the" first event in an operand is not well defined, e.g., as in Figure 3.17.

3.3.4 Introducing Gates

As recalled in Figure 3.5, Gates allow Messages to go inside and outside of Interactions (*formalGate*), InteractionUses (*actualGate*) and CombinedFragments (*cfragmentGate*).


Choices and predicates
Handling choices
No explicit time point for the choice
Explicit time points for the choice on each Lifeline
Explicit global time point for the choice
Handling guards
Interpretation of a false guard
Can yield invalid traces
Cannot yield invalid traces
Who evaluates the guard
One Lifeline
All Lifelines of the CombinedFragment interpret the guard separately
Global evaluation of the guard

3.3.4.1 Gates on CombinedFragments

With the *cfragmentGate* type of Gate, messages can cross the boundaries of CombinedFragments (see Figure 14.9 of [OMG11b]). Since *cfragmentGate* is allowed for any operator, it can yield problems.

As reported by Pickin in [Pic03], this will cause issues with loops. If a message goes into a loop, then it will have one sending end, but multiple receiving ones (see Figure 3.18). The *loop* operator is defined as a recursive application of the *seq* operator. Thus if the loop is unfolded, the result is a Message which has more than one receiving MessageEnds, which violates its constraints.

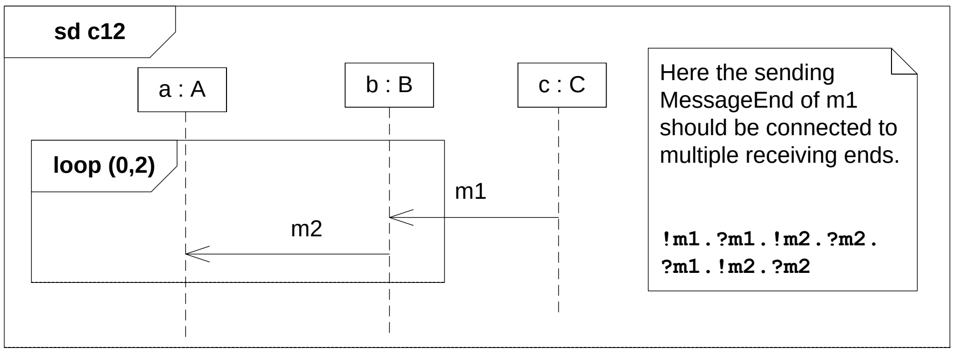

sd c12
a : A
b : B
c : C
loop (0,2)
m1
m2
Here the sending MessageEnd of m1 should be connected to multiple receiving ends.
!m1.?m1.!m2.?m2.
?m1.!m2.?m2

Figure 3.18: Message going into a loop

Most of the semantics do not consider *cfragmentGates*, or disallow it by their redefined abstract syntax (STAIRS, Cengarle & Knapp, Knapp & Wuttke). Our recommendation is also to remove it from the specification or heavily restrict its use, e.g., only to *critical* regions and *co-regions*.

3.3.4.2 Formal and actual Gates

The other two types of Gates (formal and actual) introduce a convenient facility for expressing complex scenarios: when a diagram includes a reference to another diagram (see Figure 3.19), Gates make it possible to model the passing of mes-sages. The referenced diagram has formalGates placed on its boundaries, allowing the representation of messages that come from, or go to, its environment. The environment is determined by the including diagram, where actualGates are placed at the borders of the *ref* box. Gates are MessageEnds that connect the Messages inside and outside the referenced diagram.

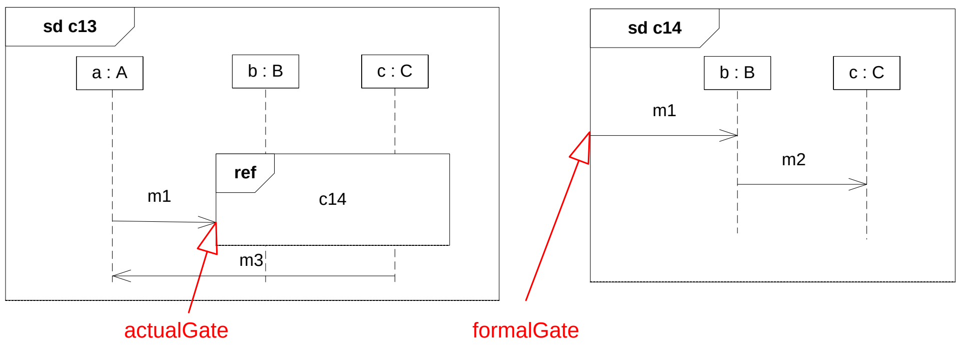
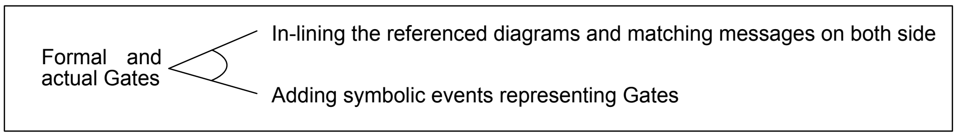


Figure 3.19: Formal and actual Gates

The surveyed semantics handle Gates in the following way. In STAIRS the set of Gates is defined as a subset of Lifelines, and events are defined when Gates receive or send Messages. Küster-Filipe adds symbolic events representing Gates, and extra orderings are added to the event structure accordingly. In P-UMLaut the referenced fragments are inlined before processing the Interaction.


Formal and actual Gates
In-lining the referenced diagrams and matching messages on both side
Adding symbolic events representing Gates

3.3.5 Interpretation of conformance-related operators

The interpretation of conformance-related operators (see our classification in Table 3.1) is a central issue in the definition of the semantics. Many papers about UML 2 Sequence Diagrams deal with this issue (or at least mention it). Indeed, quoting from [PJ04], "[*assert/negate/ignore/consider*] constructs open up a veritable pandora's box of expressions whose meaning is obscure." We provide here an overview of how the various semantics handle these constructs.

3.3.5.1 Assert and negate

The operators *assert* and *neg* allow the specification of mandatory and forbidden behavior. Störrle was the first to discuss their interpretation in a formal semantics, and he identified several possible meanings for both operators [Stö04]. Further discussion of the *neg* construct can be found in [CK04b; RHS05a].

In practice, the chosen interpretation of *assert* is consistent in all the semantics we reviewed. Let S be the fragment contained in an *assert* box.

- The expression $assert(S)$ defines the same valid traces as S;
- Every trace that is not valid for S is invalid for $assert(S)$.

The *neg* construct is more controversial, and several interpretations have been adopted. Table 3.6 illustrates some of the differences between them. They may be described as follows:

- For Störrle, the preferred interpretation is that $neg(S)$ flips the valid and invalid traces of S. Inconclusive traces are left unchanged. This interpretation enjoys the property that $neg \circ neg = Id$.

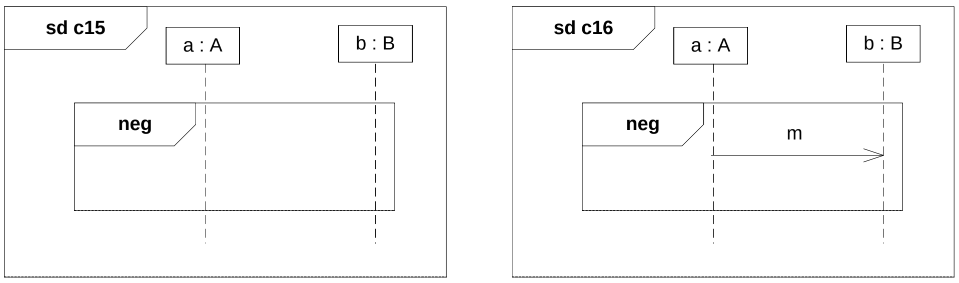


Figure 3.20: Negative fragments

Table 3.6: Interpretation of negative fragments on Figure 3.20 (Σ^* is the universe of traces)

	c15			c16		
Approach	Valid	Invalid	Inconclusive	Valid	Invalid	Inconclusive
Störrle	$\emptyset$	$\{\epsilon\}$	$\Sigma^* - \{\epsilon\}$	$\emptyset$	$\{!m.?m\}$	$\Sigma^* - \{!m.?m\}$
STAIRS	$\{\epsilon\}$	$\{\epsilon\}$	$\Sigma^* - \{\epsilon\}$	$\{\epsilon\}$	$\{!m.?m\}$	$\Sigma^* - \{\epsilon, !m.?m\}$
Cengarle & Knapp	$\{\epsilon\}$	$\emptyset$	$\Sigma^* - \{\epsilon\}$	$\{\epsilon\}$	$\{!m.?m\}$	$\Sigma^* - \{\epsilon, !m.?m\}$
Grosu & Smolka	$\Sigma^* - \{\epsilon\}$	$\{\epsilon\}$	$\emptyset$	$\Sigma^* - \{!m.?m\}$	$\{!m.?m\}$	$\emptyset$
Cavarra & Filipe, Küster-Filipe	$\emptyset$	Σ^*	$\emptyset$	$\emptyset$	$\{!m.?m\}$	$\Sigma^* - \{!m.?m\}$

This table focuses on the interpretation of negative fragments, and ignores diagram-wide issues that will be discussed in Section 3.3.5.3, such as whether an invalid prefix always makes an invalid trace. Hence, an invalid trace $!m.?m$ for the fragment may eventually yield a set of invalid traces $!m.?m.\Sigma^*$ for the diagram in Figure 3.20

Table 3.7: Interpretation of alternative negation operators (assuming each neg in Figure 3.20 is replaced by this operator)

	c15			c16		
Approach	Valid	Invalid	Inconclusive	Valid	Invalid	Inconclusive
Refuse	$\emptyset$	$\{\epsilon\}$	$\Sigma^* - \{\epsilon\}$	$\emptyset$	$\{!m.?m\}$	$\Sigma^* - \{!m.?m\}$
Not	$\Sigma^* - \{\epsilon\}$	$\{\epsilon\}$	$\emptyset$	$\Sigma^* - \{!m.?m\}$	$\{!m.?m\}$	$\emptyset$

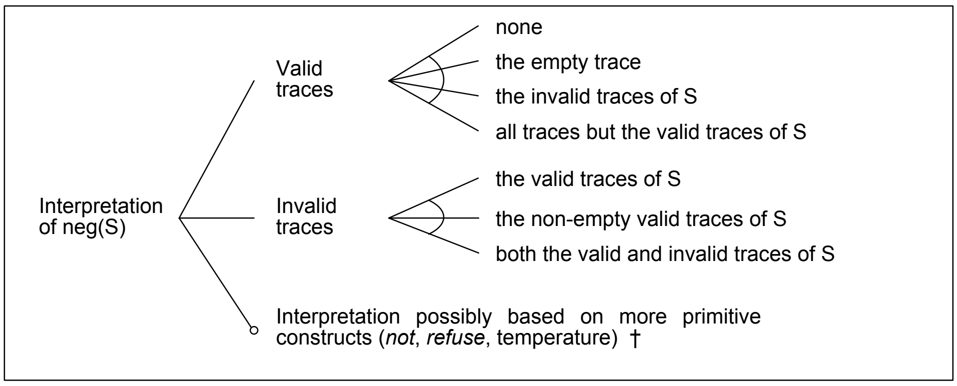
Operator $refuse(S)$ from STAIRS: all valid and invalid traces of S are invalid, there is no valid trace. Operator $not(S)$ from Cengarle & Knapp and Knapp & Wuttke: anything but S, there is no inconclusive trace.

- In STAIRS, the empty trace is the only valid trace of $neg(S)$. Both valid and invalid traces of S are invalid for $neg(S)$. This interpretation ensures that neg is monotonic with respect to the refinement relation chosen by the authors: if S is refined by S', then $neg(S)$ is refined by $neg(S')$. It is worth noting that neg is not primitive for this semantics. It is interpreted as a choice between $skip$ and $refuse(S)$, where $refuse$ is the primitive concept. The meaning of $refuse$ is shown in Table 3.7.
- For Cengarle & Knapp, the empty trace is also the only valid trace of $neg(S)$. Non-empty traces that are valid for S are invalid for $neg(S)$. All other traces are inconclusive. This interpretation enjoys the property that $neg(skip) = skip$, that is, an empty neg box is equivalent to no box.

Like in STAIRS, monotonicity with respect to refinement is considered, but with a different notion of refinement (and a different monotonicity property). Also, the semantics introduces a new operator, not, that is more primitive than neg and upon which the meaning of neg is built (see Table 3.7).

- For Grosu & Smolka, $neg(S)$ is interpreted as "anything but S"[2]. All traces, but the valid traces of S, are valid for $neg(S)$. This interpretation is relevant to verification purposes when the aim is to check that a system never exhibits the forbidden behavior.
- In Küster-Filipe, Cavarra & Filipe and MSD, $neg(S)$ is syntactic sugar for a global false predicate put at the end of S. Table 3.6 shows the interpretation of Küster-Filipe and Cavarra & Filipe. Note that the MSD interpretation would be different because the diagrams would describe partial traces (see Section 3.3.1.3). These three semantics are actually specific in their expression of mandatory and forbidden behavior because they inherit from modalities previously defined for LSCs. Inside a diagram, individual locations are assigned a hot (mandatory) or cold (possible) temperature. The operator $assert$ is syntactic sugar to turn all inside locations to hot, and neg adds a (hot) false predicate. In our interpretation of Figure 3.20, we assumed that the locations inside the neg have a cold temperature (otherwise, in the righthand diagram, the system would be required to exhibit $?m.!m$ and reach the false predicate, so that all traces would be invalid as in the empty neg).

To sum up, all semantics agree that $neg(S)$ should turn valid traces of S to invalid. However, there are differences in the way invalid traces of S are handled. Also, the empty trace is sometimes assigned a specific treatment.


Interpretation of neg(S)
Valid traces
none
the empty trace
the invalid traces of S
all traces but the valid traces of S
Invalid traces
the valid traces of S
the non-empty valid traces of S
both the valid and invalid traces of S
Interpretation possibly based on more primitive constructs (not, refuse, temperature) †

3.3.5.2 Ignore and consider

The operators $ignore$ and $consider$ affect the notion of conformance to a diagram, by changing the alphabet from which the valid and invalid traces are built. They make it possible to account for the sending and receiving of messages not explicitly represented in the diagram. The description of these operators is unclear in the OMG specification, and few semantics address them. For the ones that do, the proposed interpretation depends on whether the semantics works with complete or partial traces.

Störrle, Cengarle & Knapp and Knapp & Wuttke fall in the first category, using an interpretation with complete traces. For them, by default, a valid trace can only contain OccurrenceSpecifications shown in the diagram. The operators $ignore$ and $consider$ both allow the extension of traces with

[2]The neg is thus interpreted like the not operator mentioned just above.

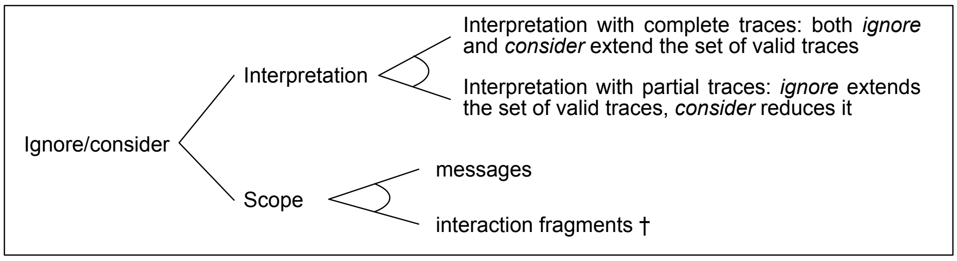
additional OccurrenceSpecifications. Ignoring a message m means that occurrences of $?m$ and $!m$ may interleave with the explicitly specified behavior. Assume that S is a basic interaction fragment involving a single message n and having one valid trace $!n.?n$. Then, $ignore(\{m\}, S)$ means that all traces of the form $(?m|!m)^*.!n.(?m|!m)^*.?n.(?m|!m)^*$ are valid. Note that the set of ignored messages could contain n, in which case we would accept traces with multiple occurrences of n. The dual operator $consider(\{m\}, S)$ is interpreted as $ignore(M - \{m\}, S)$, where M is the set of all possible messages. Its intuitive meaning is thus “ignore everything but m”.

Semantics considering partial traces, like MSD, cannot have the same interpretation. Messages not shown in a diagram are already “ignored” by default, and a system trace may contain an arbitrary prefix (resp. a suffix) before (resp. after) the shown behavior occurs. The ignore operator is useful only in the case where we want to allow multiple occurrences of the shown messages, in the temporal window of the shown behavior. If interaction S involves message n and does not involve message m, then

- $ignore(\{m\}, S)$ is equivalent to S;
- $ignore(\{n\}, S)$ has a larger set of valid traces than S.

As regards the *consider* operator, its interpretation departs from the one given by semantics that work with complete traces. In MSD, *consider* is a means to ***reduce*** the set of valid traces. It is useful when the “considered” messages would be ignored by default. Hence, if interaction S does not involve message m, $consider(\{m\}, S)$ results in a more restrictive interaction than S: it no longer allows occurrences of m in the temporal window of S.

MSD exhibits additional specificities in the way *ignore* and *consider* are handled. First, the operators are changed to specify interaction fragments (not just messages) to be ignored or considered. For example, we may “consider” a fragment consisting of message m_1 sent by lifeline l_a to lifeline l_b, followed by message m_2 from l_b to l_a. This increases the expressiveness compared with just considering m_1 and m_2. Second, the introduced fragments are assigned a temperature, offering an opportunity to distinguish cold and hot violations when the “considered” behavior inopportunely occurs.


Ignore/consider
Interpretation
Interpretation with complete traces: both *ignore* and *consider* extend the set of valid traces
Interpretation with partial traces: *ignore* extends the set of valid traces, *consider* reduces it
Scope
messages
interaction fragments †

3.3.5.3 Conformance-related operators in complex diagrams

Up to now, we discussed the interpretation of each conformance-related operator taken in isolation. But in complex diagrams, conformance-related operators can include, or be nested into, other constructs.

Let us consider the nesting of conformance-related operators into each other. This raises issues such as the interpretation of multiple assertions, multiple negations, assertions of negations, negation of fragments with considered and ignored messages, and messages that are both ignored and considered. Of course, the semantics dealing with these constructs will assign a precise meaning to such cases. However, the assigned meaning may defeat intuition, with the risk of producing diagrams that users do not properly understand. To illustrate this point, double negation is a good example (see

Figure 3.21). It is striking that the various semantics offer all possibilities for the categorization of trace $!m.?m$:

- The trace is valid for Störrle;
- It is invalid for STAIRS and all interpretations adding a false global predicate (MSD, Küster-Filipe, Cavarra & Filipe);
- It is inconclusive for Cengarle & Knapp.

Whatever the chosen semantics, care must be taken that it really captures the meaning intended by the specifier.

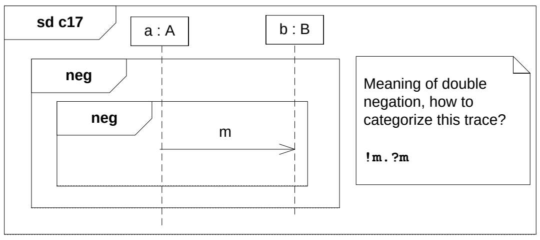


Figure 3.21: Nesting conformance-related operators

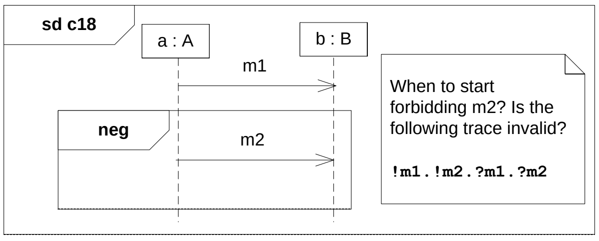

sd c18
a : A
b : B
m1
neg
m2
When to start forbidding m2? Is the following trace invalid?
!m1.!m2.?m1.?m2

Figure 3.22: Borders of conformance-related operators

Syntactic restrictions may reduce the risk of counterintuitive interpretations. In [Stö04], Störrle came to the conclusion that *neg* should not be used as an ordinary operator. It should be used only at the top level of a diagram, to indicate that the diagram describes a forbidden scenario. This avoids intriguing cases where *neg* is nested into other operators. Other authors (Knapp & Wuttke, Grosu & Smolka) forbid the nesting of operators into a negation, so that negated fragments can only contain basic interactions. Their motivation, however, is not to preserve intuition but to ease verification of conformance: they want the detection of invalid traces to be kept decidable. As a general rule, one may put syntactic restrictions on conformance-related operators for either purpose, for keeping diagrams intuitive or for facilitating their usage in verification activities.

Apart from the case where the operator is used only at the top level (as recommended by Störrle for the *neg* operator), an important semantic issue is how conformance-related operators are combined with the rest of the diagram. The general decision on whether to synchronize or not on the borders of boxes has an impact on the categorization of traces. In Figure 3.22, the shown trace is not invalid if synchronization is enforced. This raises the issue of when to start requiring (*assert*), accepting (*ignore*) or forbidding (*neg*, and also *consider* in its interpretation using partial traces) the communication events appearing in the operator.

Another issue is how to interpret sequencing (whether weak or strong) when the prefix of a trace completely traverses a negative region. Figure 3.23 exemplifies this issue. Will a trace starting with prefix $!m1.?m1.!m2.?m2$ be categorized as invalid whatever the suffix? Almost all the semantics answer by yes. This has the advantage of facilitating the identification of invalid traces: decision can be taken locally, independently of what will happen subsequently (see the discussion conducted in [CK04b]). However, a different interpretation is chosen in STAIRS: the trace is inconclusive if the suffix does not match. Note that Figure 3.23 involves a *neg*, but a similar example could be built with a trace prefix violating an *assert*.

What about then the continuation of traces that do not completely traverse a negative region? For example, in Figure 3.23, is the trace $!m1.?m1.!m2.!m3.?m3$ valid or inconclusive? In the semantics

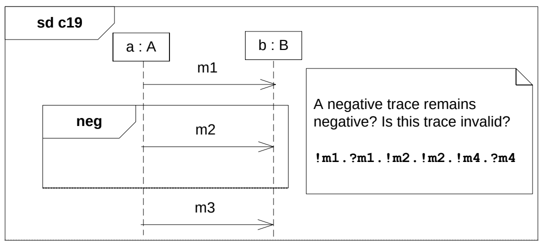

sd c19
a : A
b : B
m1
neg
m2
m3
A negative trace remains negative? Is this trace invalid?
!m1 . ?m1 . !m2 . !m2 . !m4 . ?m4

Figure 3.23: Negative trace remains negative?

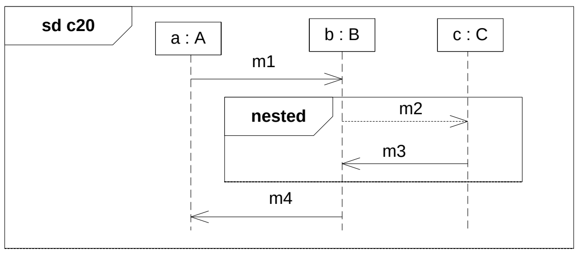
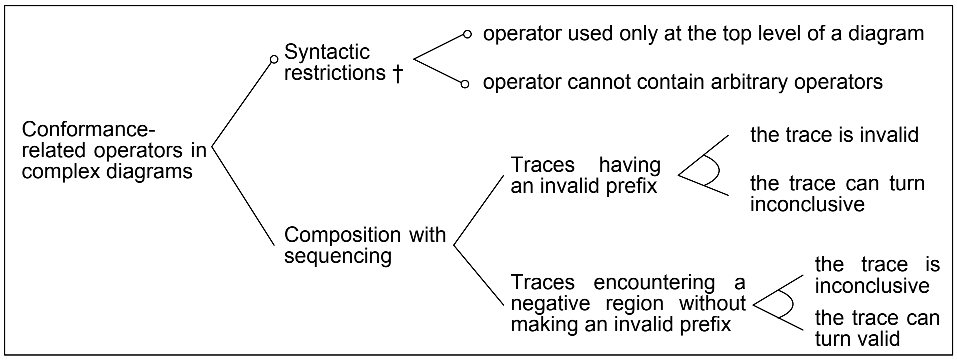


Figure 3.24: Nested operator in MSD

of Grosu & Smolka, the trace would be valid. For Cengarle & Knapp and STAIRS, the only valid traces would be $!m1.?m1.!m3.?m3$ and $!m1.!m3.?m1.?m3$. For semantics adding a global false predicate (MSD, Küster-Filipe, Cavarra & Filipe), there can be no valid traces: only invalid or inconclusive ones.

Technically, for the latter three semantics, completely traversing the *neg* yields a hot violation, while failing to traverse the *neg* is a cold violation. In both cases, the local violation determines the categorization of the overall trace (invalid, inconclusive) whatever the suffix. Hence, the scope of the local violation is actually global. The authors of MSD have proposed a new operator, *nested*, allowing them to restrict the scope of cold violations. It is then possible to have valid continuations of traces that do not completely traverse a fragment. Let us recall that in MSD, the primitive concept to express mandatory/forbidden behavior is the temperature. In Figure 3.24, message *m2* is cold (indicated by a dotted line) while message *m3* is hot (indicated by a solid line). It means that *m2* may occur, and if it does, then *m3* is required. Since MSD only considers synchronous messages, we do not distinguish the sending and receiving of messages. In a nested fragment, a cold violation is confined: the trace continues in the enclosing fragment. In Figure 3.24, the trace *m1.m4* is indeed valid. It would be inconclusive if *m2* and *m3* were in the plain fragment.


Conformance-related operators in complex diagrams
Syntactic restrictions †
operator used only at the top level of a diagram
operator cannot contain arbitrary operators
Composition with sequencing
Traces having an invalid prefix
the trace is invalid
the trace can turn inconclusive
Traces encountering a negative region without making an invalid prefix
the trace is inconclusive
the trace can turn valid

3.3.5.4 Traces being both valid and invalid

Ambiguous diagram can be constructed, where a given trace is both valid and invalid. In the example of Figure 3.25, this is due to non-determinism. A given event in the trace can be considered as occurring either inside or outside the scope of the conformance-related operator, depending on some non-deterministic choice. Parallel constructs come with similar ambiguities.

It may seem that the problem comes with the nesting of conformance-related operators into non-deterministic constructs. But the example of Figure 3.26, borrowed from [CK04b], only involves weak sequencing. Intuitively, it is not clear whether the occurrence of *m1* in the trace falls into the scope of

neg, or may be considered as posterior to the neg. Note that the interpretations given by the various semantics are not necessarily ambiguous. For semantics adding a false predicate at the end of the neg fragment, trace $!m1.?m1$ is clearly invalid. For STAIRS, it is valid (only a double occurrence of $m1$ would be invalid). However, for Cengarle & Knapp, the trace is both valid and invalid.

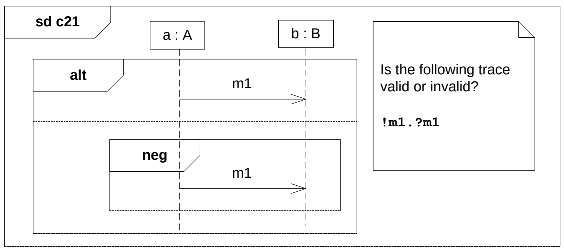


Figure 3.25: Ambiguity due to non-determinism

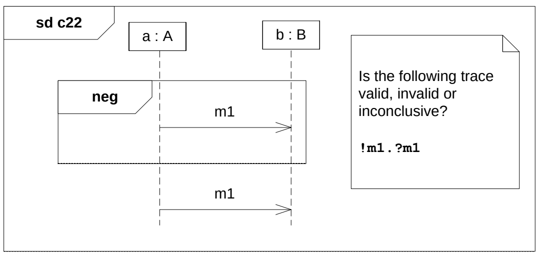


Figure 3.26: Ambiguous scope of a conformance-related operator

Figure 3.27 illustrates yet another possibility for introducing ambiguity. Here, the ambiguous case comes from the consideration for the values of message parameters, in a diagram where the two occurrences of message m cannot be distinguished. The shown trace may be categorized as valid or invalid, depending on whether the final assertion is $2 > 1$ or $1 > 2$.

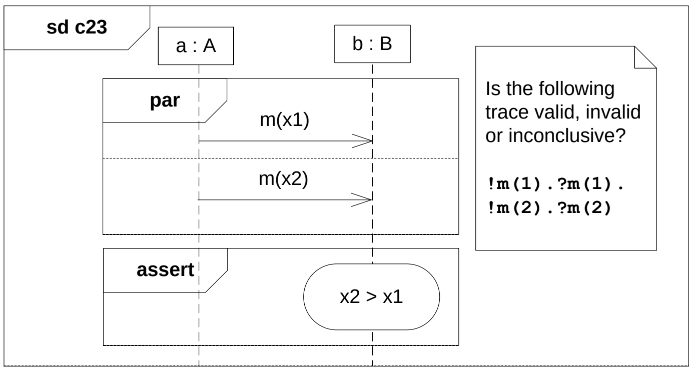
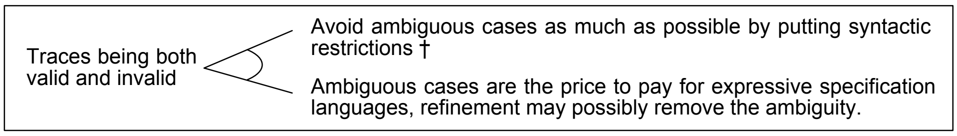


Figure 3.27: Ambiguous cases due to the consideration of data values

These various examples show that it is extremely difficult to get rid of ambiguous cases. We may put syntactic restrictions to avoid some of the cases (e.g., use only a deterministic if-then-else form of alt constructs, or use neg only at the top level of the diagram), but avoiding all of them by construction would probably require us to sacrifice too much in terms of language expressiveness. Indeed, from our analysis of work dealing with conformance-related operators, all surveyed approaches face cases where a trace can be both valid and invalid. In general, checking whether a diagram is ambiguous is an undecidable problem. From previous work on MSCs [MP05], we know that language complementation and intersection are not decidable for graphical scenarios. Then, if ambiguous cases are to be avoided, it is probably wise not only to put syntactic restrictions, but also to adopt an interpretation that brings the semantics of diagrams to a regular language (synchronize on entering and exiting fragments, encode the partial orders into an automaton). In this way, the ambiguous cases not covered by the syntactic restrictions can be detected and reported to the user.

Not all authors explicitly mention the existence of ambiguous diagrams. For the ones who do [CK04b; LS06; Stö04], this is not necessarily considered as a problem. In [CK04b], ambiguous diagrams are called *overspecified* interactions. There may exist refinements that remove the ambiguity, so that an overspecified interaction may indeed have an implementation. For example, both Figure 3.25 and Figure 3.26 are overspecified interactions according to Cengarle & Knapp. Figure 3.26 is not implementable, but Figure 3.25 is implementable by the *skip* process.


Traces being both valid and invalid
Avoid ambiguous cases as much as possible by putting syntactic restrictions †
Ambiguous cases are the price to pay for expressive specification languages, refinement may possibly remove the ambiguity.

3.4 Summary

Due to the variety of usage for scenarios, there is nothing such as "the" semantics of UML 2 Sequence Diagrams. The OMG group has always insisted on the fact that the standard enables specialization of parts of UML for a particular situation or domain. As regards Sequence Diagrams, it would probably be an impossible task to exhibit an "all-in-one" semantics fitting purposes as diverse as the description of example interactions, of test cases, or of checkable properties.

Flexibility to assign different interpretations to diagrams leaves UML practitioners with a difficult problem: the one of selecting a semantics well suited for their purpose. There is a lack of a clear picture of available options. Since the pioneering work of Störrle for giving a formal meaning to UML 2 Sequence Diagrams, a number of alternative semantics have flourished, and the research presented in this chapter is an attempt to gain a synthetic view of the choices that underlie them.

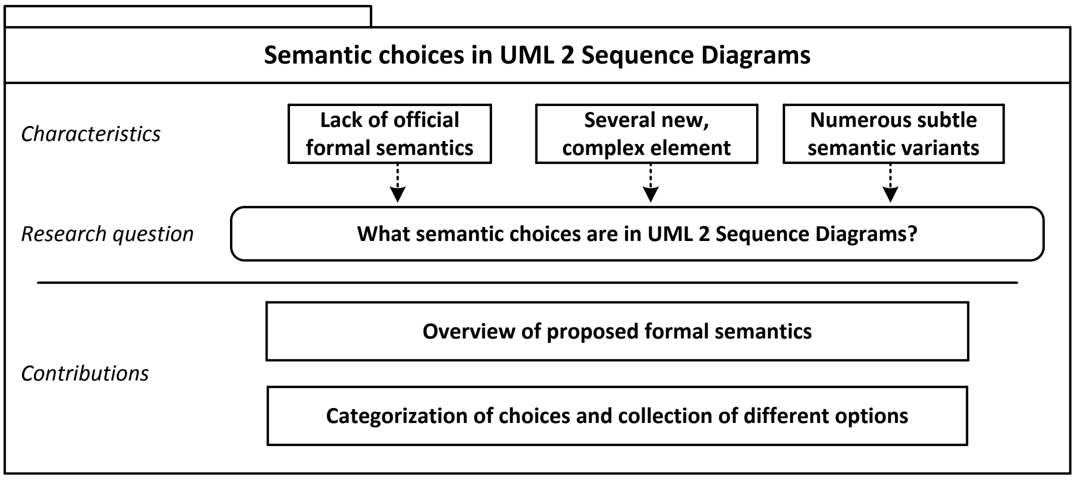

Semantic choices in UML 2 Sequence Diagrams
Characteristics
Lack of official formal semantics
Several new, complex element
Numerous subtle semantic variants
Research question
What semantic choices are in UML 2 Sequence Diagrams?
Contributions
Overview of proposed formal semantics
Categorization of choices and collection of different options

Figure 3.28: Summary of research question and contributions

Figure 3.28 summarizes the research problem and our contributions. Our approach was to select a sample of 13 semantics and to systematically identify the points in which they differ. We took care to include widely referenced work, as well as less-referenced one that may be representative of more specialized concerns. We created a categorization of choices, ranging from the interpretation of basic diagrams to the interpretation of advanced constructs such as conformance-related operators.

For each choice, we listed the options encountered in the analyzed sample. Our discussion of options tried to be very practical, by showing their concrete consequence on examples of diagrams. We ended up with a structured representation of the various choices and options, inspired by feature models. We selected a simple, visual notation on purpose; in this way a deep background in theoretical computer science is not required to use the results in practical projects (which is often a barrier in other works on formal semantics).

The next chapter presents how we used the categorization of semantics choices for developing a new language and designing its syntax and formal semantics. Our hope is that this categorization will help others similarly in interpreting or adapting Sequence Diagrams in their own application domain.

Chapter 4

A test language and framework for mobile systems

Mobile ad-hoc networks propose new challenges for software development and verification and validation activities. In addition to the issues found in fixed distributed systems, fresh ones are introduced in the new environment: high dynamicity and context awareness. New nodes are constantly joining and leaving, the application running on the host has to be aware of these changes. Nodes are moving out of each other's communication range frequently, hence the failure of sending a message is not a rare event any more. The state of an application depends not only on the messages it receives from the others, it should also take into account its context, e.g., its current location coordinates supplied by a GPS unit or other information from the environment. Thus the test approach of these systems should take into account these specificities.

A common test approach used in traditional distributed systems is some form of passive testing, i.e., when the tester can only monitor the behavior of the system under test (SUT) through its execution traces, but cannot directly interact with it. In such cases instead of defining test cases with exact inputs and expected output (which would not be useful, as the tester could not directly feed input into the SUT), the logs of the messages in the system are checked against some invariant properties. These properties are expressing test requirements, e.g., that whenever the SUT receives a request it shall send the requested data or an error code. But even if all the requirements are satisfied in the collected execution traces, it does not guarantee that all the important behaviors were observed. Therefore test purposes can be specified that express partial behaviors that should be covered during the executions. The goal of testing is then to execute the SUT in different situations until all test purposes are covered and all the requirements are satisfied.

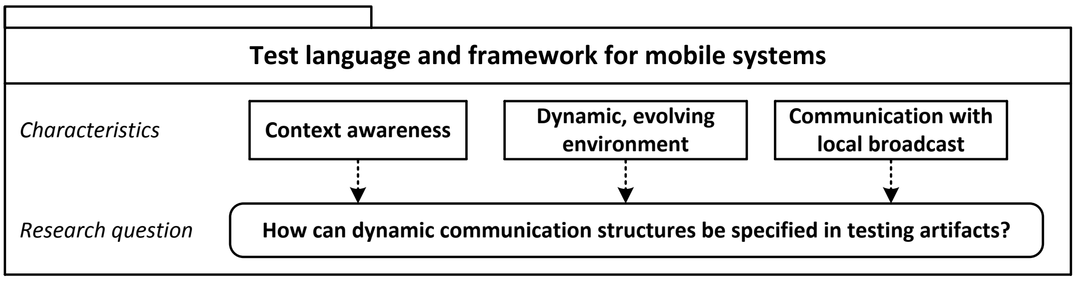

Test language and framework for mobile systems
Characteristics
Context awareness
Dynamic, evolving environment
Communication with local broadcast
Research question
How can dynamic communication structures be specified in testing artifacts?

Figure 4.1: Research question of the chapter

Test purposes and test requirements can be conveniently specified using scenario languages. However, current modeling languages have to be adapted to the characteristics of mobile systems, as we will see that they do not offer intuitive notations to capture frequent changes in the topology or conditions on the context of the system. Thus the goal of the research presented in this chapter was to (i) extend existing modeling languages to cope with mobile settings, and (ii) to propose a test framework that uses these extensions to test mobile systems.

The structure of the chapter is the following. Section 4.1 reviews existing languages, and presents our case study on a mobile group membership protocol to identify missing language features. Section 4.2 describes (i) our extensions for scenario languages to model mobile systems, (ii) our test approach that uses requirements expressed as graphical scenarios to check execution traces, and (iii) the components of our test framework. Section 4.3 defines the syntax and semantics of our modeling language called TERMOS. Section 4.4 illustrates the tools developed for the test framework. Finally, Section 4.5 summarizes the contributions of the chapter.

4.1 Testing mobile systems

Testing of distributed systems offers several challenges, e.g., the behavior of the system is highly asynchronous or the final verdict has to be assigned using the partial verdicts returned by the different test components. However, ad-hoc mobile networks introduce further issues to test development and execution. Ad-hoc networks are by nature very dynamic, which implies that (i) test modeling notations have to cover scenarios where nodes are appearing or disappearing, (ii) test platforms should be able to simulate the specialties of mobile networks (e.g., frequent disconnects or high latencies).

4.1.1 Modeling notations for mobile systems

According to our research, currently there is no standard for modeling mobile systems yet, but in the recent years several approaches have emerged. A number of publications focus on logical mobility, i.e., when code moves between devices. One example of logical mobility is a mobile agent, a software component that executes specific tasks on behalf of someone with some autonomy. Mobile Ambients [CG00] is a low-level formalism based on process calculus, which offers mobility primitives. Mobile Agent Modeling with UML is a UML profile recommended by Belloni and Marcos in [BM04]. The stereotypes and tagged values of the profile are organized into views that describe the different aspects of the mobile agent. In [AMS03] a modeling notation called Mobicharts is proposed for applications in mobile environments. Mobicharts describe state changes of a task using services like migration to a new host or disconnected operations. Some approaches offer mechanisms to express some form of basic physical mobility also, i.e., when the host moves from one location to the other. Usually in this cases locations are predefined, hierarchical places with some attributes, e.g., wireless towers or access points with different outbound connections, a home or an office network with different firewall rules, or rooms in a house with different local resources. Grassi *et al.* proposed an UML profile to support physical mobility of the computing nodes and the logical mobility of software elements [GMS04]. The behavior of mobility was expressed on so-called mobility manager statecharts. The paper included examples to show how the profile can be applied to describe basic mobile code paradigms. In [Bau+03] a UML extension called Mobile UML was proposed to model mobile systems in global computing. The extensions consisted of (i) a UML profile to express mobility concepts (location, mobile, mobile location) and (ii) new diagram types. According to the authors, the problem with UML Sequence diagrams when modeling mobile scenarios is that movement of an entity can be expressed only indirectly by adding a new object box. Thus, to overcome the complexity of this approach, a new diagram type Sequence Diagram for Mobility (SDM) was recommended.

The protocol testing community reported several case studies of analyzing mobile protocols. In [NV05], the authors study MIPv6, a protocol enabling nodes to remain reachable while moving around in the IPv6 Internet. In [Cav+04], the studied protocol is in the ad hoc domain: the Dynamic Source Routing protocol (DSR) that allows routes to be discovered and maintained in multi-hop wireless ad hoc networks. Both [NV05] and [Cav+04] had to tackle the problem of not having mobility-related concepts in the protocol modeling language SDL. They had to add specific components (one for each node, plus a centralized controller) to capture the notion of communication with neighbors.

Several approaches have been proposed, that contain many similar elements, however, each of them are specialized for a specific aspect of mobile systems, and no general standard is available at the moment. Moreover, these extensions mainly consider logical mobility or physical mobility from one infrastructure point to the other, and they do not offer a solution to ad-hoc networks.

4.1.2 Test platforms for mobile systems

Several test platforms have been proposed for mobile systems. Some approaches try to recreate realistic environments. For example, in [LNY04], a telephony application is tested by having human operators carrying handsets in an urban area. In [Bru+04], testing a car-to-car application involves three prototype vehicles driven on a road. Although this type of testing is useful in validation exercises, its high cost and restricted controllability or observability limits its application. In practice, a major part of the testing activities will preferably be performed using emulation or simulation facilities.

For mobile computing systems a usual approach is to simulate wireless communication (in both infrastructure and ad hoc modes) by integrating a network simulator as a part of the test platform. Such simulators have been originally developed to support networking research, but they are now also used to experiment with the application level. For example, the ns-2 simulator is used in [MD04] to evaluate a health-monitoring application, and in [Sch+05] to evaluate a car-to-car messaging system. In both cases, the network simulator is only part of the complete platform, which also includes a context controller. The context controller is used to simulate contextual information in mobile settings, like location-based data. The context controller typically uses a mobility model, like random waypoint or pathway model, to obtain location data.

There have been several toolkits for simulating contexts in recent years. Car-to-car applications may use traffic simulators that simulate the movement of vehicles along roads. For example, TraNS [Pió+08] connects the ns-2 network and the SUMO traffic simulator, STRAW [CB05] uses a Java event-based simulator, and GrooveNet [Man+06] even offers to connect simulated and real cars.

All these tools, network simulators and context controllers, were mainly developed for evaluation purposes, but they may serve verification purposes as well. However, they have to be usually be customized to the actual application domain and test framework, e.g., typically adapters have to be written to drive the controllers according to the test inputs and collect the logs for test evaluation.

4.1.3 A detailed case study: mobile GMP

To better understand the new testing related challenges we performed a detailed case study [13][24], the analysis of a mobile Group Membership Protocol (GMP) [HJR04]. This case study was performed in the framework of the HIDENETS project (Highly dependable IP-based networks and services) [HID09], which analyzed end-to-end resilience solutions for mobile-based applications and services.

One intended application domain for Hidenets was the automotive one, emphasis being put on car-to-car communication scenarios. Hence, we chose a case study exemplifying ad hoc networking issues between mobile entities. The protocol is used to form groups from mobile nodes (which are close enough that they communicate with each other), and provide a consistent view of the actual members of the group to everyone. The nodes periodically broadcast discovery messages with their positions to inform others. Each group has a leader, which decides whether the group should be merged with a nearby, newly discovered group or it should be split if some node is moving away from the other members. The protocol calculates a so called safe distance from the velocity and communication range of the nodes, and uses this metric to decide whether a merge or a split should be started. The analysis of the protocol was conducted by (1) reviewing the specification, (2) creating a UML model for the implementation of the protocol, (3) comparing the specification to the implementation, and (4) performing test experiments with nodes moving randomly.

The protocol specified 8 safety and progress properties. We reverse-engineered a detailed UML model from the implementation (static structure diagrams, state machines and sequence diagrams), and tried to capture these properties in UML's recommended constraint language (OCL). However, some of the properties could not be captured as constraints on the classes and objects (e.g., the *conditional eventual integration* property stating that two nearby groups should merge if they remain at safe distance long enough). Next, we tried to express the properties as graphical scenarios in UML, but we faced again several challenges (e.g., a scenario could not easily refer to the state of the topology or global constraints could not be expressed in UML SD). We created several variants of these scenarios with alternative notations, but the results were not satisfactory. Finally, the testing of the implementation revealed scenarios that violate the safety properties of the protocol.

The insights gained from this case study can be summarized as follows. Standard UML was appropriate to model the structure and behavior of one node, but it was inconvenient for modeling complex scenarios including several nodes. For example, Figure 4.2 illustrates how modeling broadcast messages or topology changes is problematic in pure UML.

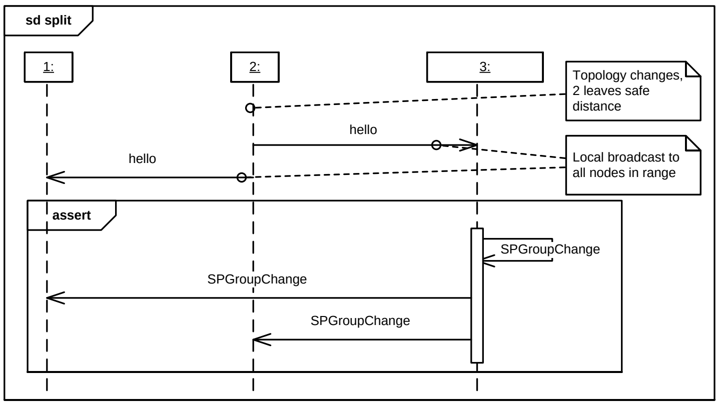

sd split
1:
2:
3:
Topology changes, 2 leaves safe distance
hello
hello
Local broadcast to all nodes in range
assert
SPGroupChange
SPGroupChange
SPGroupChange

Figure 4.2: An example complex scenario from the GMP case study

The analysis showed general challenges that are relevant for any mobile application dealing with mobility and cooperation of hosts.

- Services and applications in mobile settings rely heavily not just on user input but also on context information, like current location data.
- It is not easy to model mobile system instances. Without a suitable notation and modeling methodology serious design defects could be introduced.

- The definition of properties containing spatial and temporal information is a complex task, but the correct formulation is essential to the later verification steps.
- Moreover, a test execution engine should be able to feed the SUT not only the test messages, but also these contextual data.

The detailed properties and requirements from the GMP protocol, as well as the fail scenarios we observed, served as examples for the typical concepts that should be expressed in context-aware mobile systems. The GMP study helped us to identify the challenges that had to be solved, and served as a testbed for the new test language and framework we have developed.

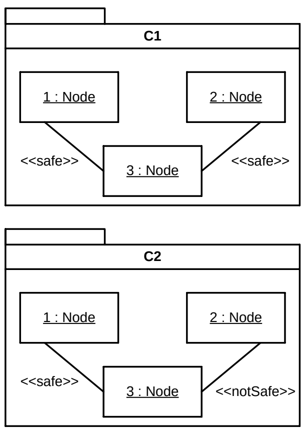
4.2 Test approach and framework

As Section 4.1 showed some extensions are needed in UML to model mobile scenarios conveniently. To solve this issue, we propose to integrate the description of spatial relationship between nodes into UML 2 Sequence Diagrams. This section presents these extensions, how they can be used in test artifacts, and the components of a test framework that uses these artifacts to test mobile systems.

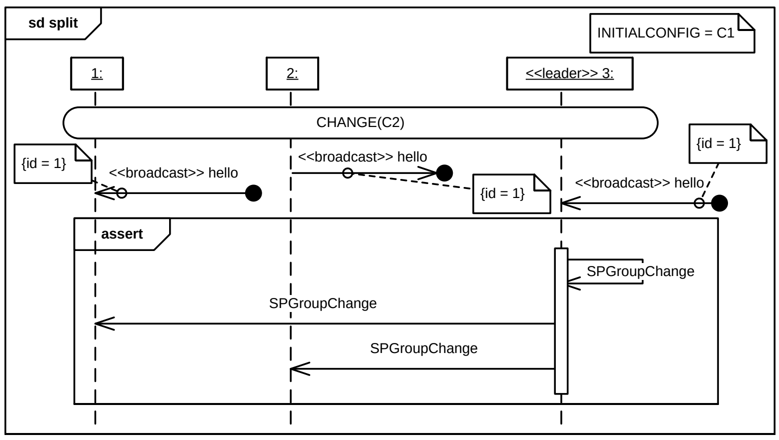
4.2.1 The test approach for mobile systems

The extended Sequence Diagrams include two connected views, the spatial view (describing the topological configurations of the system nodes, as well as some contextual information) and the event view (describing communication events, and their causal dependencies on configuration change events). More precisely:

- The *spatial view* consists of a set of labeled graphs, corresponding to the various configurations that occur in the scenario. For a given configuration, the labels attached to vertices and edges represent relevant attributes of system nodes and of communication links between nodes.
- The *event view* makes it explicit which communication event occurs in which spatial configuration, and configuration changes are introduced as global events.



(a) Spatial view



(b) Event view

Figure 4.3: Example requirement scenario for a mobile system

Figure 4.3 exemplifies how these two views can be applied to the scenario presented previously on Figure 4.2. The scenario says that whenever the leader detects that a node in the group is not at safe distance anymore, it has to split the group and notify the other nodes. In the event view, note the global configuration change event that identifies the actual configuration for each message. Moreover, the introduction of a special «broadcast» stereotype helps to express local broadcast in the *hello* messages. In the spatial view, it is the responsibility of the designer to determine convenient abstractions for the concrete configurations, depending on the target application. Here, the GMP behavior is governed being at safe distance or not. This explains the chosen edge labels. Nodes are merely characterized by their symbolic identifiers (see label variables 1 and 2), but tuple of labels are allowed for applications needing a richer representation of node attributes.

The three proposed mobility-related extensions can be used in scenarios expressing different test artifacts. Figure 4.4 shows examples for test requirements and a test purpose. The leftmost figure presents a positive requirement capturing an invariant property, e.g. whenever a given behavior happens in the trace, then something, which is contained in the *assert* fragment, always follows. The middle scenario depicts a negative requirements, i.e. forbidden behaviors that should never occur in the trace. The rightmost figure is a test purpose; it describe behaviors to be covered by testing, that is, we would like these behaviors to occur at least once in the trace.

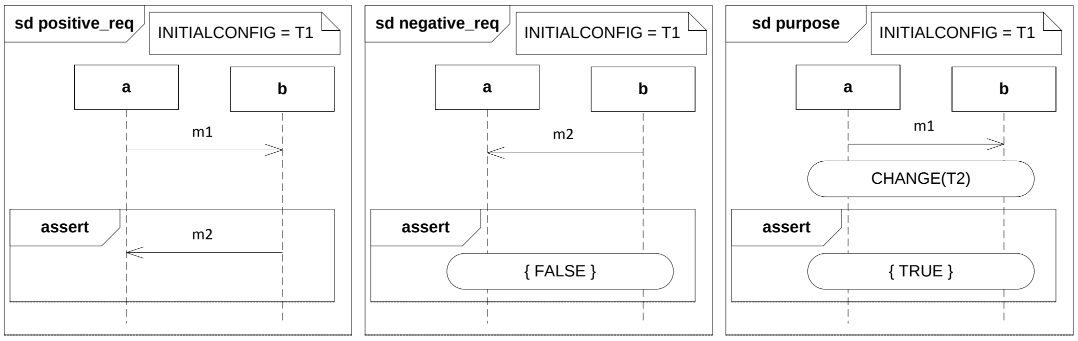


Figure 4.4: Requirement and test purpose scenarios (event view)

Using these extensions we developed (i) a *test requirement language* (called TERMOS, TEst Requirement language for Mobile Setting) and (ii) a *test framework*, which focuses on specifying application requirements as graphical scenarios and evaluates actual execution traces with respect to these requirements.

4.2.2 The test framework for mobile systems

The developed test framework is depicted on Figure 4.5. Its primary purpose is to check the requirements captured as graphical scenarios against traces coming from an execution environment, where the SUT is exercised in various, complex situations. Thus the framework can be seen as a form of passive testing, where requirements represent invariants. The components of the test framework are the followings.

Requirements Requirements are captured using the proposed TERMOS language, which includes the mobility related extensions presented in the previous section. The syntax of the language includes elements for representing spatial configurations, changes in the communication structure and broadcast messages.

Execution environment In order to exercise the mobility-related behavior of the SUT, the usual test components that drive the interfaces of the SUT have to be combined with a network simulator

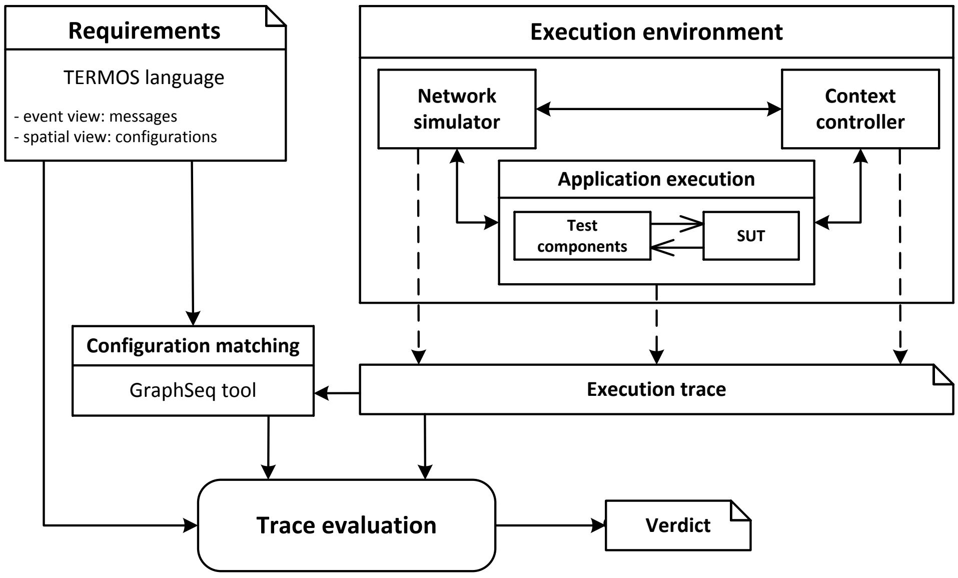


Figure 4.5: The test framework for mobile systems

and a context controller. The network simulator could simulate delays or communication errors on wireless or fixed links. The context controller based on mobility patterns can adjust the communication topology in the network simulator dynamically, and it can provide location-based data, like actual position, to the application execution component.

Execution traces During the runs in the execution environment detailed execution traces are collected. These include the messages exchanged between the SUT and its environment, the changes in the communication topology, and the evolution of the relevant context, e.g., the actual positions of the nodes periodically.

Configuration matching There is a gap between the abstract configurations defined in the requirement scenarios and the configurations observed in the trace. It should be decided which node from the execution environment can play the roles represented in the scenarios. Based on their types and connections the abstract nodes have to be mapped to the concrete ones found in the trace. However, usually there are several possible matching possible. Moreover, the matching should take into account not only one configuration, but the changes in a series of configurations. To solve this task, a method and a tool called *GraphSeq* [NWR10] was developed, which can reason on a series of abstract and concrete configuration graphs, and can return the set of possible matching and valuations.

Trace evaluation The obtained traces can be evaluated with respect to the requirements. Using the set of configuration matching the messages in the trace have to be analyzed, whether their types, parameters and order conform to the ones specified in the scenarios. This requires a precise formal semantics for the event view of TERMOS, one that makes checking traces possible. As we could see in Chapter 3 there are many semantics choices in UML 2 Sequence Diagrams, a consistent set of options have to be carefully selected that assigns an unambiguous meaning to diagrams. Finally, pass, fail or inconclusive verdicts are be assigned to the *(trace, requirement, matching)* combinations. Note that to evaluate a trace both the configuration matching and the processing of the scenario is needed.

From this test framework my contributions focused on the TERMOS language, thus the rest of the section presents the new language in detail.

4.3 The TERMOS language

This section discusses (i) the design choices of the language, (ii) its abstract and concrete syntax, (iii) its automaton-based formal operational semantics, and (iv) the developed tool support.

4.3.1 Design choices for the language

The most important design goal of TERMOS was that it shall allow the evaluation of traces. Every decision about the syntax and semantics supported this goal. Moreover, in order to make TERMOS an easily usable test requirement language, we made the following high-level design decisions:

- the language should help to capture only those details that are relevant to the given requirement;
- a requirement is preferred to be simple, complex properties should be decomposed into smaller ones;
- a scenario represents an independent, self-contained check, no hierarchical description or referencing is recommended.

Using these guidelines we selected the syntactic and semantic options summarized in Table 4.1 for TERMOS from the choices presented in Chapter 3. The rational behind each of the decisions is the following.

Interpretation of a basic Interaction Our purpose with the language is to evaluate traces against partial behavior fragments captured in the requirement scenarios, thus the interpretation of a basic Interaction is the following.

- *What is a trace?* Because real execution traces should be checked, it should be made possible to exactly identify the sending and receiving events of all messages; thus the definition of a trace contains unique identifiers. Accordingly, a concrete trace will be a tuple containing $(?m, receiver, id)$ or $(!m, sender, id)$, where m is the name of the message sent or received, and id is an identifier generated by execution environment of the test framework. Note, that a send event may be aimed to several receivers (e.g., in the case of a broadcast message), but a receive event involves only one receiver. The id serves the purpose to match the sending and receiving events of a given message.
- *Categorizing traces* Not all traces are relevant for a requirement; hence the trace universe is partitioned into three classes (valid, invalid and inconclusive traces).
- *Complete or partial traces* A test requirement scenario is a partial description because it captures just a fragment of the system's behavior (subset of nodes, subset of messages). We wanted a very flexible language; thus in TERMOS both a prefix and suffix are allowed, messages not depicted on the diagram can interleave, and the weak interpretation is used.

Introducing CombinedFragments Recall, that in the UML specification there is no synchronization mechanism amongst lifelines when entering or exiting fragments. This could present several challenges when verifying traces (e.g., there is no common point to evaluate guards or the scope of the operator is unclear). For this reason, in TERMOS, entering and exiting a CombinedFragment is treated as a synchronization point in order to make checking a trace easier.

Table 4.1: Design choices for the TERMOS language

Interpretation of a basic Interaction	What is a trace?	Definition of a trace contains unique message Ids
	Categorizing traces	Valid, invalid, inconclusive
	Complete or partial traces	Partial traces (prefix/suffix allowed, extra messages can interleave, weak interpretation for duplicates)
Introducing CombinedFragments	Combining fragments	Synchronization on entering or exiting a CombinedFragment
Computing partial orders	Processing the diagram	Process the diagram as a whole using locations
	Underlying formalisms	Interleaving semantics, encode the partial orders into a finite automaton
	Choices and predicates	Explicit global time point for the choice, a false guard does not yield an invalid trace, guards are evaluated globally
Introducing Gates	Gates on Combined-Fragments Formal and actual Gates	Gates were disallowed completely
Interpretation of conformance-related operators	Assert and negate	Instead of *neg* as an operator, a global false predicate can be put at the end of the diagram
	Ignore and consider	Using partial traces and the weak interpretation makes *ignore* redundant, *consider* reduces the set of valid traces
	Conformance-related operators in complex diagrams	Nesting is restricted, traces having an invalid prefix are invalid
	Traces being both valid and invalid	Syntactic restrictions avoid some of the ambiguous cases

Computing partial orders The defined formal semantics was inspired by LSC's semantics, as the goals of TERMOS are similar to LSC (like expressing requirements or depicting partial traces).

- *Processing the diagram* TERMOS uses also the location concept and processes the diagram as a whole.
- *Underlying formalism* A state-based formalism was chosen because it makes checking of a given trace feasible. An automaton is built for the whole diagram, which represents all lifelines.
- *Choices and predicates* In TERMOS, to make verification possible, there is a global time point when all the participating lifelines evaluate the guards and choose one alternative (this will be represented by a common transition in the formal semantics). If the choice is guarded, the explicit guards appear as transition labels. Moreover, only a deterministic form of guarded choice is allowed (similar to an if-then-else construct). Finally, variables in guards and state invariants can only refer to message parameters previously sent or received and to node attributes in the

current configuration. This guarantees that unrepresented nodes and messages cannot change the valuation of a predicate.

Introducing Gates Requirement scenarios should be kept simple; thus all types of Gates were disallowed. This means, that scenarios cannot reference each other, every requirement scenario is analyzed independently.

Interpretation of conformance-related operators The conformance-related operators (assert, neg, consider, ignore) modify the categorization of a trace as valid, invalid or inconclusive. Their usage is heavily restricted in order to make the checking of a trace feasible.

- *Assert and negate* The diagram can have only one *assert* box at the end of the diagram, which should cover all lifelines. Handling the *neg* operator combined the approaches from MSD (using a global false predicate instead of *neg* as an operator) and Störrle (negation is for the whole Interaction, it can appear only at the end of the top-level fragment).
- *Ignore and consider* Because of partial traces and weak interpretation the *ignore* operator is not needed. The operator *consider* is used to reduce the set of valid traces, i.e., to indicate that some of the extra messages are not allowed.
- *Conformance-related operators in complex diagrams* To ease the detection of valid traces the nesting of operators is heavily restricted. Only one level of nesting is allowed for conformance-related operators (e.g., *assert* into a top-level *consider*). Table 4.2 summarizes the allowed combinations of nested operators. However, when using nesting, the containing operator should be at the main level of the diagram.
- *Traces being both valid and invalid* We want to avoid ambiguous cases as much as possible. Thus apart from the syntactic restrictions presented before, further checks are defined on the generated automaton, to detect some remaining cases of non-deterministic categorization into valid and invalid traces.

Table 4.2: Can the operator in the row be nested in the operator in the column?

	alt	*opt*	*par*	*assert*	*consider*
assert					●
consider				●	

Summary The collection of the semantic choices from Chapter 3 helped to identify what should be decided for the new language. As can be seen from this example, the categorization of choices provides a *structured framework* to consider the various options and to make design decisions that suit the purpose of the newly defined language.

However, it should be noted that these design decisions influenced the expressiveness of the new language. As the usage of several elements were forbidden or heavily restricted (e.g., using gates or nesting operators), several UML 2 SD scenarios cannot be expressed in TERMOS. We think that this is the price one have to pay to have a language that can be later used for verification purposes. Nevertheless we took care to have sufficient expressiveness to be able to represent the GMP scenarios. We also took inspiration from the language elements offered by other formalisms for requirement scenarios in (non mobile) distributed systems, like LSC [HM03] and its adaptation to UML 2 [HM08].

4.3.2 Syntax of the language

In the description of the language's syntax, we put emphasis on the new elements we propose to allow description of scenarios in mobile settings. The new elements concern (i) the introduction of a spatial view for the scenario, (ii) the accounting for spatial configurations in the event view, (iii) and the representation of broadcast communication. The rest of the section then provides an overview of the syntax of the event view, recapitulating the syntactic constraints put on the core UML elements to facilitate the definition of the semantics.

4.3.2.1 Syntax of the spatial view

The spatial view may contain several spatial configurations. Each configuration is given a name, e.g., Figure 4.6 shows a configuration named $C3$.

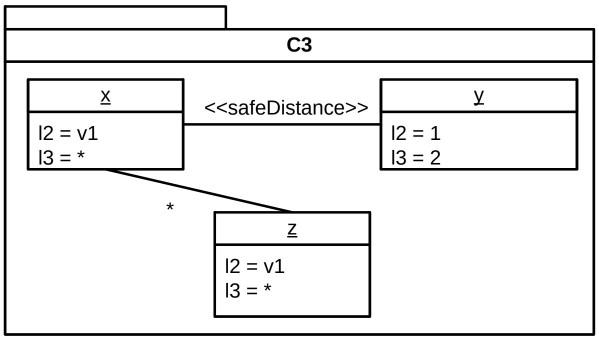


Figure 4.6: Example of spatial configuration

Abstract syntax A configuration is a labeled graph, where vertices represent system nodes and edges represent different kinds of connection between nodes. Each node has a symbolic identifier. For example, Figure 4.6 shows three nodes having identifiers x, y and z. This means that any scenario referring to $C3$ must involve lifelines for nodes x, y and z. In order to allow for a richer representation of configurations, nodes can have two additional attributes of integral types (i.e., integers or enumeration types). The corresponding vertex labels in the graph can take different forms:

- A *constant value* from the integral type. For example, in Figure 4.6, the two attributes of node y have constant values 1 and 2.
- A *variable name*, denoting a value from the type. For example, the first attribute of nodes x and z must be identical, but their precise value is let unspecified (variable $v1$). This value is intended to remain stable in the configuration. Moreover, if a scenario involves several graph configurations containing label variable $v1$, it must be substituted for a single value. Thus, $v1$ can be seen as a symbolic global constant for the scenario.
- A *wildcard* indicating a don't care value, e.g., the second attribute of node x. Don't care values do not need to remain stable in the given configuration.

Edges can be labeled by constant values or wildcards. In Figure 4.6, it is assumed that the connection type is an enumerated type *safeDistance, communicationDistanceOnly*, like in the GMP testing case study. Nodes x and y have a *safeDistance* connection, nodes y and z are disconnected, and we do not care about the connection of nodes x and z, they may exhibit unstable connections/disconnections during the configuration.

Concrete syntax To be as compatible with the original UML specification as possible, a spatial configuration is depicted using object diagrams. A package with the name of the configuration contains all elements. Nodes are represented as instances, slots named $l2$ and $l3$ contain the additional labels defined for the given node. Labels for edges are represented as stereotypes, because they characterize the given connection between the two nodes.

4.3.2.2 Spatial elements in the event view

The event view of a scenario uses UML 2.0 Sequence Diagrams, with some extensions to explicitly account for the spatial configurations defined in the spatial view.

Abstract syntax An Interaction can be tagged with the *termosScenario* stereotype (Figure 4.7) to show that it is a requirement scenario in TERMOS. The *termosScenario* stereotype has an association named *initialConfiguration* giving the initial configuration of the Interaction.

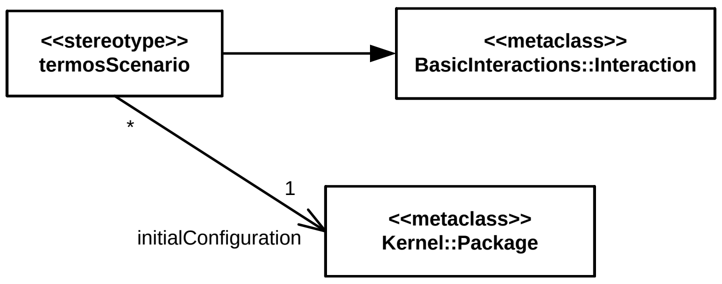


Figure 4.7: The termosScenario stereotype

Configuration changes are represented by global events of the form CHANGE($newConfigName$) that induce a global synchronization for all lifelines. Configuration changes cannot be nested into operators, except into a consider operator that is at the main level. Configuration changes are "decided" by the environment. Configuration changes arise deterministically and involve all lifelines at the same time. In this way, the diagram can be decomposed into fragments, where each fragment takes place in a well-defined spatial configuration. This makes it explicit which communication event occurs in which configuration. Predicates (guards of alt operands, state invariants) may refer to variables of their current or past configurations (i.e., node label variables).

Concrete syntax Representing the initial configuration with a stereotype's tagged value fits well into the UML framework, the only drawback is that because Interactions are the abstract concepts representing scenarios, they visually do not appear on a Sequence Diagram. In most of the modeling tools, assigning a stereotype to an Interaction is only reflected in the textual properties view, but not on the diagram itself. For this reason, in the examples used in this chapter the initial configurations of the diagrams are depicted in a comment box containing the text INITIALCONFIG. These comments are not part of the semantic model, rather they ease the readability of the examples.

For the graphical element of a configuration change the symbol of Continuation was reused. The original Continuation element is not allowed in TERMOS, thus it could not cause a misunderstanding.

The configuration changes may involve the dynamic creation, shutdown and restart of nodes. For example, a scenario may have three successive configurations $C4$, $C5$, $C6$ (see Figure 4.8(a)), where:

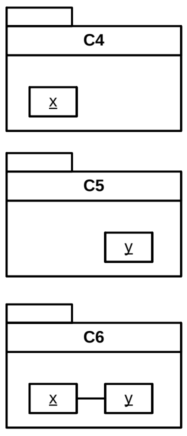
- $C4$ contains a node with identifier x;

- $C5$ does not contain a node with identifier x, but contains a node with identifier y that was not present in C4;
- $C6$ contains both nodes x and y.

There is no convenient way to illustrate such a dynamic structure in sequence diagrams. For example, a lifeline can be stopped, but then it is not possible to restart it. Also, dynamic creation can only occur as the result of an action performed by an existing lifeline. To solve this problem, we take the convention that the spatial configuration determines which node is alive/dead at some point of the scenario. There is a lifeline for every nodes mentioned in any one of the configurations.

Well-formedness rules If a node is not active at some point of the scenario, then it is not supposed to participate to any communication interaction. Figure 4.8(b) shows an example for such invalid messages. Checks can be provided to warn the scenario specifier whenever communication is not compatible with the spatial view:

- dead nodes sending and receiving messages;
- active nodes exchanging messages while there is no path connecting them in the current configuration.



(a) Spatial configurations

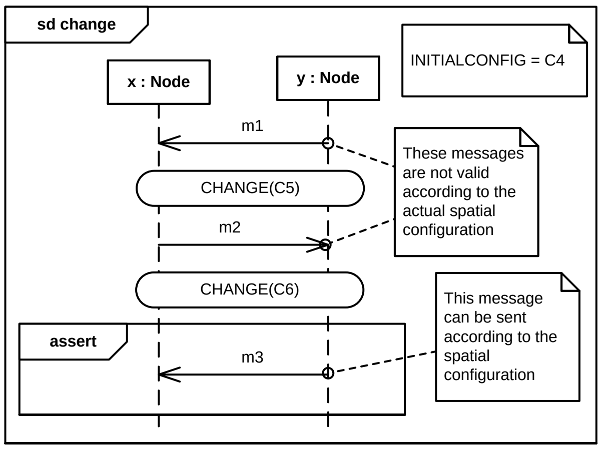

sd change
INITIALCONFIG = C4
x : Node
y : Node
m1
These messages are not valid according to the actual spatial configuration
CHANGE(C5)
m2
CHANGE(C6)
This message can be sent according to the spatial configuration
assert
m3
(b) Configuration changes and messages

Figure 4.8: Combining the spatial and event view

4.3.2.3 Broadcast communication

UML 2 Sequence Diagrams focus on point to point communication. There is no element dedicated to the representation of broadcasts or multicasts. This is a serious drawback for representing local broadcasts, i.e., communication with unknown partners in local vicinity.

Abstract syntax We propose to use the concepts of lost and found messages to represent such broadcasts. Lost messages are messages with no explicit receiver. Similarly, found messages do not have an explicit sender. Lost and found messages offer flexibility to represent partial behavior, where not all lifelines and not all communication events are of interest. Such flexibility is quite useful when specifying requirement scenarios; hence we need lost and found messages independently of our consideration for local broadcasts.

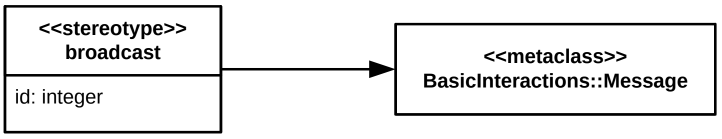


Figure 4.9: The broadcast stereotype

In order to distinguish broadcasts from "usual" lost/found messages, we assign them the *«broadcast»* stereotype. A broadcast involves one send event followed by zero or more receive events. A tagged value is attached to the corresponding lost/found messages, so that each receive event of the diagram can be paired to the send event that caused it. Figure 4.9 presents the definition of the broadcast stereotype.

Concrete syntax For representing local broadcast the usual notation of lost and found messages is used. Figure 4.10 shows an example how this can be combined with the broadcast stereotype. There are two broadcast messages on the diagram, one sent by node x (identified by $id = 1$) and one by node z (identified by $id = 2$). Every other node receives the broadcasts messages, as depicted by the found messages.

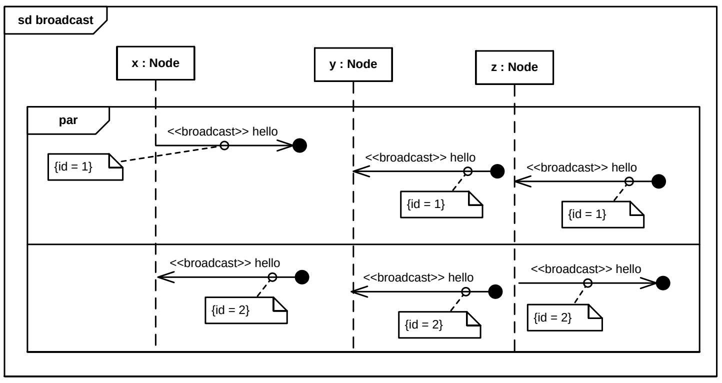


Figure 4.10: Example of broadcast messages

4.3.2.4 Syntax of event view

Abstract syntax The abstract of the event view is derived from the syntax of UML 2.0 Sequence Diagrams. According to the decisions described in Section 4.3.1, some of the elements were removed and some additional constraints were added to adopt it our environment.

Appendix C.1 contains the complete abstract syntax of TERMOS given with a metamodel. The changes to the original abstract syntax are collected in Table 4.3.

Concrete syntax Apart from the configuration change and the global StateInvariant no changes were made to the concrete syntax. This should help using existing UML modeling tools to create TERMOS scenarios.

Table 4.3: Changes to the original Sequence Diagram syntax

Type	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 4.2
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.

Well-formedness rules Apart from the simple constraints presented in Table 4.3, there are other, more complex checks that could be done to validate whether a requirement scenario is also semantically well-formed.

- Check, whether messages depicted on the diagram can be sent and received in the current spatial configuration.
- Check, whether predicates refer only to message parameters received so far and to configuration labels from the current or past configurations.

Automated checks can be implemented to verify that a diagram conforms to the above changes and constraints, which will be presented later in this section.

4.3.3 Semantics of the language

We defined an automaton-based operational semantics for the TERMOS language. The semantics has been inspired by the semantics proposed for LSC, more specifically the one defined by Klose [Klo03]. The approach builds an automaton from the diagram, the states of the automaton being determined by the valid cuts of the diagram. Informally, a cut is intended to represent a consistent global state characterized by the events occurred so far, and it is meaningful to reason about the past or the future of this state. The automaton's transitions then stand for the successor relation among the cuts. Klose's approach has been extended (i) to incorporate UML SD elements not present in LSC, e.g., *alt* or *par* combined fragments, and (ii) to handle the mobile settings related elements, e.g., broadcast messages and configuration changes. Also, the details of the construction of the automaton differ in several aspects:

- Klose builds a Büchi automaton to accommodate infinite traces. Since we are dealing with finite test traces, we are building a standard automaton.
- Klose has a separate treatment for the pre-chart (for us, the analogous would be everything before the assert fragment) and chart (for us, would be the content of the assert fragment). Our semantics builds a single automaton for the whole diagram.
- We have an interleaving semantics, while Klose allows several events to occur at the same time.

As regards the last two points, our choices are similar to the ones made for MSD, which also has an interleaving semantics captured in one automaton. However, as LSC does not have a concept of compound fragments and MSD was defined only for synchronous messages, their semantics have to be extended and adapted.

Our definition of the semantics consists of the following steps.

1. *Pre-processing*: the diagram is parsed, its basic building blocks and the orderings between them are identified.
2. *Unwinding*: the automaton is constructed using the structures built in the first step.
3. *Checking well-formedness*: as stated in Section 4.3.2.4 the diagram has to conform also to complex well-formedness rules, which can be checked based on the formal semantics.
4. *Connecting to the spatial view*: the automaton built for the event view has to be connected to the spatial view.

The next sections will describe each step. The details of the semantics will be illustrated by taking example scenarios and defining their semantics.

4.3.3.1 Pre-processing the diagram

To create an automaton capturing the semantics of a diagram, first the elements of the diagram are identified.

Definition 1 (Atom) The basic building block of a TERMOS diagram is called an atom. The following elements are atoms:

- Lifeline heads, denoted by $\bot_l$ for Lifeline l;
- Lifeline ends, denoted by $\top_l$ for Lifeline l;
- MessageOccurrenceSpecifications, i.e., sending a message or receiving a message;
- StateInvariants (for global StateInvariants every Lifeline has a separate StateInvariant atom);
- configuration changes;
- entering a CombinedFragment;
- exiting a CombinedFragment;
- guards. □

The orderings of the atoms on one Lifeline are defined by their *position*. Klose uses an integer as the position of atoms, however this is not sufficient in our case. In the case of parallel or alternate fragments, the visual positioning of atoms does not necessarily mean a temporal relation between them, i.e., the elements inside the second operand of a *par* fragment are drawn below the elements inside the first operand, but they should not necessarily happen after the atoms in the first operand.

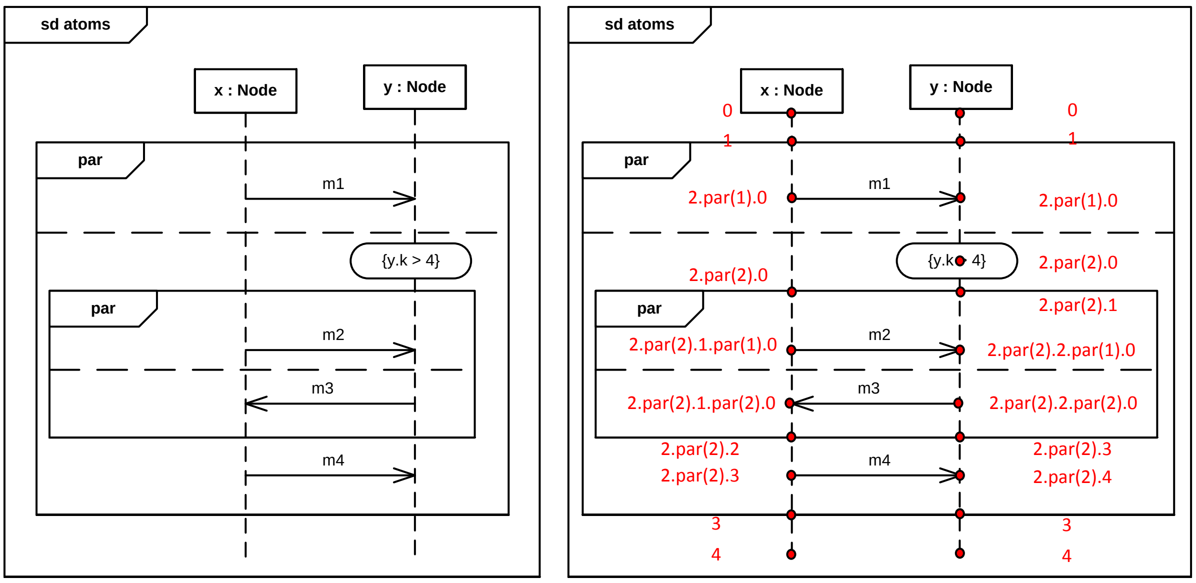


Figure 4.11: Example for assigning atom position to *par* fragments

To solve this issue, instead of an integer value a path expression is assigned to each atom, similarly to the approach used in [Küs06]. The method is illustrated by the following example.

The left side of Figure 4.11 contains the example diagram, while the right side is annotated with the atom positions. The idea is that for the elements inside the main fragment or for the elements inside one operand, every atom is assigned a number according to their visual position starting from zero. If we enter a CombinedFragment, then a path expression is added to the position quantifying in which operand the current atom resides. In the current example this translates to the following positions.

- The head of Lifeline x is assigned position 0.
- The atom for entering the first *par* fragment still belongs to the main fragment, thus it gets position 1.
- The *par* gets the next position, which is 2.
- Elements inside the *par* inherit the position of the *par* fragment (namely 2 in the current example), and an expression describing in which operand of the *par* they reside. Thus, sending *m1* on x gets 2.par(1).0, meaning that it is in the *par* identified by position 2, it is in the first operand of the *par*, and it is the first atom of that operand.
- Entering the second *par* fragment is in the second operand of the outer *par*, thus it is assigned 2.par(2).0. Sending of *m2* is inside the nested *par*, its position reflects this nesting: 2.par(2).1.par(1).0. The second *par* has the position 2.par(2).1, this position is prefixed to every elements inside that fragment.
- Exiting a fragment belongs to the same level as the fragment itself, thus exiting the first *par* gets the position 3, showing that it is at the main diagram fragment.
- Atom positions are only unique per lifelines, and atoms representing the same event (e.g., entering the same fragment), can have different positions assigned. This is illustrated with the help of positions on lifeline y.

Thus the definition of the atom position is the following.

Definition 2 (Position) The *atom position* identifies the position of an atom on one lifeline. It has the form [path]id, where path is a string identifying in which CombinedFragment the atom is, and id is an integer giving the order of the atom compared to the other atoms inside that fragment. The string path is empty if the atom is in the main fragment of the diagram, otherwise it is in the form p.opr(op)., where p is the position of the CombinedFragment the atom is in, opr is name of the operator of the fragment, and op is the number of the operand the atom is in. □

The example on Figure 4.12 shows how atom positions can be assigned to an alt fragment. Notice, that for all guards an atom is assigned on the Lifeline, where the guard is placed.

Guards of operands are grouped to the next atom on the Lifeline, forming a *cluster* with that atom. However, care must be taken, because sometimes there is no next atom inside the guard's operand (e.g., in an empty [else] operand coming from an *opt* fragment). In this case, the cluster contains only the guard. In the original UML specification, there can be several immediate successor of an atom also (e.g., if the atom is right before a *par* with several operands). With the introduction of a separate atom for the beginning of a fragment, this is not the case in TERMOS.

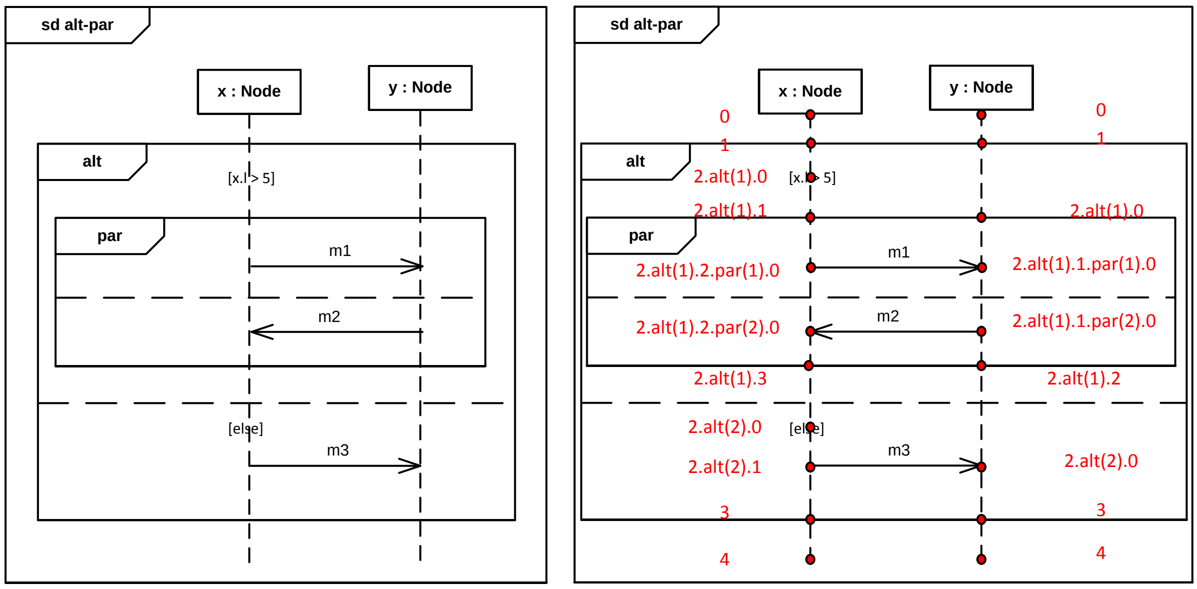


Figure 4.12: Example for assigning atom position to *alt* fragments

Definition 3 (Cluster) If the a atom is a guard with a position `p.i` and there exists an atom with a position `p.(i+1)`, then the two form a cluster. Every other atom forms a cluster with only that atom in it. □

The $\mathrm{Clusters}(l)$ function returns all the clusters on the lifeline l. To handle the positions of clusters with multiple elements, the concept of location is defined.

Definition 4 (Location) The $\mathrm{Location}(cl)$ function returns the minimum of the positions of the atoms inside the cluster, where $\min(p.i, p.(i+1)) = p.i$. □

Several elements provide synchronization across Lifelines, e.g., configuration changes or entering a fragment, the clusters corresponding to these elements have to be mapped together. Simultaneous classes, *SimClass*es, serve this purpose.

Definition 5 (SimClass) A simultaneous class is a set of clusters from separate lifelines. The clusters representing the following elements form a SimClass together, every other cluster forms a SimClass with only that cluster as its member:

- the beginning of the same CombinedFragment;
- the end of the same CombinedFragment;
- the same configuration change;
- the same global StateInvariant. □

Figure 4.13 illustrates how atoms, clusters and SimClasses are defined for a diagram. To sum up: atoms are "points" on lifelines; clusters are used to group simultaneous atoms on a given lifeline; SimClasses group clusters that are simultaneous at a diagram-wide level, that is, non singleton Simclasses represent synchronization of several lifelines.

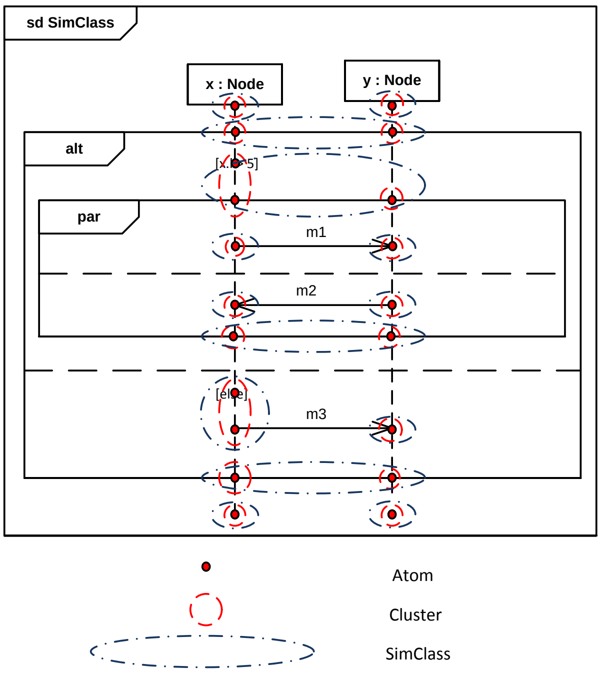


Figure 4.13: Atoms, clusters and Simclasses on a diagram

Two relations are defined between clusters on one lifeline. Causality, denoted by $\prec$, defines a partial order between clusters. Conflict, denoted by $\#$, defines which events cannot appear in the same trace, e.g., atoms from different operands of an *alt*.

Definition 6 (Local causality) Let cl_1, cl_2 be two clusters on lifeline l with their location in the form: $\mathrm{Location}(cl_1) = p_1.i.p_2$ and $\mathrm{Location}(cl_2) = p_1.j.p_3$, where if p_1 is the empty string then the $p_1.$ prefix, if p_2 or p_3 is the empty string, then the $.p_2$ or $.p_3$ postfix is removed respectively. Local causality is defined then as

$$cl_1 \prec cl_2 \quad \text{iff } j > i.$$

□

Definition 7 (Local conflict) Let cl_1, cl_2 be two clusters on Lifeline l with their location in the form: $\mathrm{Location}(cl_1) = p_1.\mathsf{alt}(i).p_2$ and $\mathrm{Location}(cl_2) = p_1.\mathsf{alt}(j).p_3$, then local conflict is defined as

$$cl_1 \# cl_2 \quad \text{iff } i \neq j.$$ □

Definition 8 (Predecessors) The predecessors function calculates the immediate predecessor(s) of a cluster cl on its Lifeline l:

$$\mathrm{Predecessors}(cl) \overset{def}{=} \left\{cl' \in \mathrm{Clusters}(l) \mid cl' \prec cl \wedge \nexists cl'' \in \mathrm{Clusters}(l) : cl' \prec cl'' \prec cl\right\}.$$ □

For example, on Figure 4.12, the predecessor of the cluster with location 2.alt(1).0 is 1, while the predecessors of the cluster 2.alt(1).3 are 2.alt(1).2.par(1).0 and 2.alt(1).2.par(2).0.

For handling the causality between clusters on different lifelines, the message sending and receiving events have to be mapped. To achieve this, every message is assigned a unique symbolic identifier in the form $\$i$, where i is an integer. Let ID be the set of identifiers generated that way, and let $\mathrm{MessageSends}(sd)$ and $\mathrm{MessageReceives}(sd)$ be the sets of all message sending and receiving atoms of the diagram sd.

Definition 9 (MessageID) The $\mathrm{MessageID}\colon \mathrm{MessageSends}(sd) \cup \mathrm{MessageReceives}(sd) \rightarrow ID$ function returns, for each sending or receiving atom, the identifier of the corresponding message. □

The Predecessors function can be extended to SimClasses to contain also the causality relations implied by the connection between message sending and receiving. Let $\mathrm{SimClasses}(sd)$ represent the set of all SimClasses of diagram sd.

Definition 10 (Prerequisites) The immediate predecessors of a SimClass scl in the diagram sd are given by the Prerequisites function:

$$\begin{aligned}\mathrm{Prerequisites}(scl) \overset{def}{=} \Big\{ & scl' \in \mathrm{SimClasses}(sd) \mid \\ & \exists cl \in scl, \exists cl' \in scl' : cl' \in \mathrm{Predecessors}(cl) \vee \\ & \big(\exists a \in cl \cap \mathrm{MessageReceives}(sd), \exists a' \in cl' \cap \\ & \mathrm{MessageSends}(sd) : \mathrm{MessageID}(a) = \mathrm{MessageID}(a')\big)\Big\}.\end{aligned}$$ □

The conflict relation is extended to SimClasses using the Conflicts function.

Definition 11 (Conflicts) The $\mathrm{Conflicts}(scl) : \mathit{SimClass} \rightarrow \mathcal{P}(\mathit{SimClass})$ function returns the SimClasses which have clusters that are in different operands of an *alt* fragment than the operand in which the clusters of scl are. □

We then have $scl_2 \in \mathrm{Conflicts}(scl_1)$ if and only if the two SimClasses respectively contain a cluster cl_1 and a cluster cl_2 such that:

- the location of cl_1 on its lifeline l_1 has a form $\mathit{prefix}_1.k_1.\mathsf{alt}(i).\mathit{suffix}_1$ (i.e., cl_1 is in an *alt* operand)
- the location of cl_2 on its lifeline l_2 has a form $\mathit{prefix}_2.k_2.\mathsf{alt}(j).\mathit{suffix}_2$ with $j \neq i$ (i.e., cl_1 is in an *alt* operand having a different number)
- the *alt* coincide, that is, the following clusters ()representing the beginning of the fragment) belong to the same SimClass:

– $cl'_1 \in \text{Clusters}(l_1)$ having location $prefix1.(k_1 - 1)$
– $cl'_2 \in \text{Clusters}(l_2)$ having location $prefix2.(k_2 - 1)$

Note that the local conflict relation (#) corresponds to a special case of the global conflict, when $l_1 = l_2$.

4.3.3.2 Unwinding algorithm

The aim of the unwinding algorithm is to build a *symbolic automaton* that characterizes traces as valid or invalid according to the requirement scenario. Inspired from [Klo03], the principle is to gradually unwind the SimClasses of the diagram, until all of them have been processed.

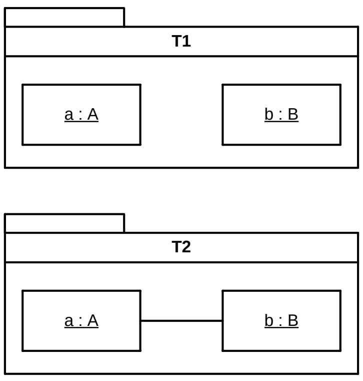
A state of the automaton is a global state of the scenario, capturing the progress of all lifelines. The algorithm starts in an initial state with the lifelines heads unwound. Then, it uses the precedence and conflict relations to search for the enabled classes of atoms and to compute the successor states. Each transition is labeled according to the currently unwound class. A label can be an event expression, consuming a trace event that matches it, or a predicate to be evaluated without consuming an event. Both kinds of labels may involve variables, and event consumption may trigger update actions. If the trace analysis reaches a state where no transition can be fired, the automaton exits and returns a verdict that depends on the category of the state.

The automaton has three categories of states. Trivial accept states are used to categorize traces that do not exhibit the potential behavior before the $assert$. Reject states represent traces that are invalid (e.g., something in the $assert$ is violated), and stringent accept states denote traces successfully reaching the end of the $assert$.

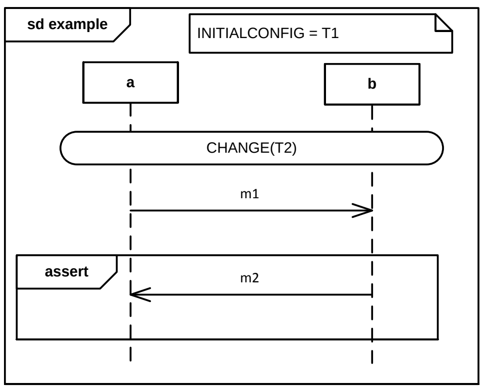
The definition of a symbolic automaton and the unwinding algorithm is detailed in Appendix C.2.

The example on Figure 4.14 is used to illustrate the unwinding algorithm. On Figure 4.14(a) the two configurations referenced in the scenario, while on Figure 4.14(b) the event view is presented. Figure 4.15 depicts the generated symbolic automaton. Trivial accept states are marked with double circles, reject states with single circles and stringent accept traces with triple circles. The example shows how a simple diagram with configuration changes and messages is handled.

A more complex example with an alt and a par fragment can be found in Appendix C.3.



(a) Spatial view



(b) Event view

Figure 4.14: Example scenario used for generating automaton

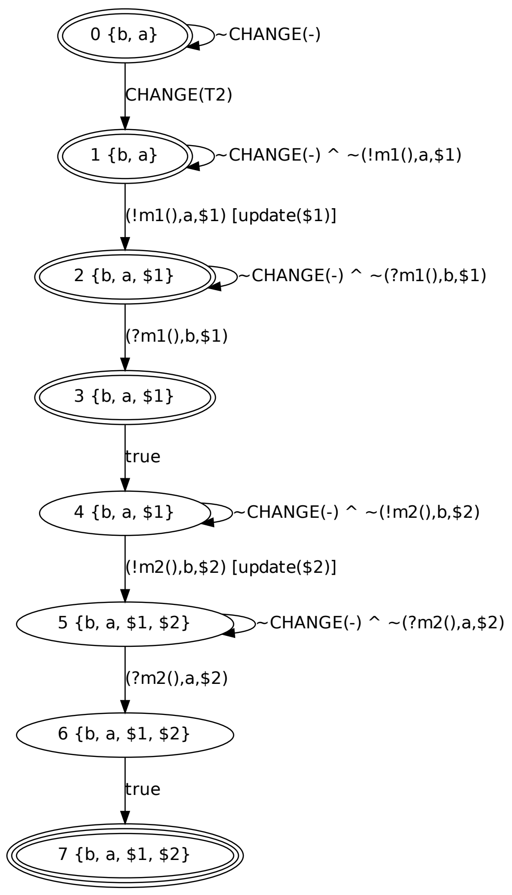

0 {b, a}
~CHANGE(-)
CHANGE(T2)
1 {b, a}
~CHANGE(-) ^ ~(!m1(),a,$1)
(!m1(),a,$1) [update($1)]
2 {b, a, $1}
~CHANGE(-) ^ ~(?m1(),b,$1)
(?m1(),b,$1)
3 {b, a, $1}
true
4 {b, a, $1}
~CHANGE(-) ^ ~(!m2(),b,$2)
(!m2(),b,$2) [update($2)]
5 {b, a, $1, $2}
~CHANGE(-) ^ ~(?m2(),a,$2)
(?m2(),a,$2)
6 {b, a, $1, $2}
true
7 {b, a, $1, $2}

Figure 4.15: Automaton generated from the scenario on Figure 4.14

4.3.3.3 Well-formed scenarios

Well-formedness is not a purely syntactic issue. Some checks depend on the semantics. At the end of Section 4.3.2.4, two checks were mentioned:

- Check, whether messages depicted on the diagram can be sent and received in the current spatial configuration.
- Check, whether predicates refer only to message parameters received so far and to configuration labels from the current or past configurations.

The first check can be performed as soon as the preprocessing step, when the orderings are computed. Using the positions it is straightforward to determine the current spatial configuration for a communication atom. Moreover, the message identifier allows us to identify the sender of receiving events. It suffices then to verify that for each lifeline l:

- Each communication atom of l occurs in a configuration where l exists, i.e., there is a vertex with symbolic identifier l in the configuration graph.

- If the atom is a receive event and lifeline l' of the diagram was the sender, then there is a path connecting l and l' in the current configuration graph such that all edge labels in the path have constant values.

The second check can only be performed on the automaton. It suffices to verify that, for each state q, the predicates appearing on the outgoing transitions do not refer to free variables that are undefined in q. The implementation is straightforward since, by construction, we know which variables are defined for which state. The check could be integrated into the unwinding algorithm, i.e., when the transition labels for guard and state invariant atoms are computed.

4.3.3.4 Combining the spatial and event view

The analysis of the event view of a TERMOS scenario produces a symbolic automaton with variables that depend on the spatial configuration. The automaton must be instantiated in the framework of the concrete configurations that occurs during system execution. Checking whether a system trace satisfies or violates the scenario is done under the following conditions:

- Analysis is started in a state where the system is in a concrete configuration that matches the initial configuration of the scenario.
- The concrete values for the configuration variables, including the concrete identifiers of nodes participating to the scenario, are known.
- The trace includes configuration change events.

Such conditions are fulfilled by using the GraphSeq tool. It returns a set of matches for the desired sequence of spatial configurations where a match includes: (i) a valuation for all configuration variables, (ii) the temporal window for each individual configuration. This allows the verification of the trace against the scenario requirement.

4.4 Tools for the test framework

The following prototype tools have been developed to demonstrate and analyze the TERMOS language and the test framework.

The *GraphSeq* tool was developed at LAAS by Minh Duc Nguyen [Ngu09]. The tool searches for the matches of spatial configurations in an execution trace. GraphSeq uses a graph matching tool and reasons on sequences of graphs. It returns the possible matches of the configurations defined in the spatial view of the scenario to the concrete configurations found in the execution traces.

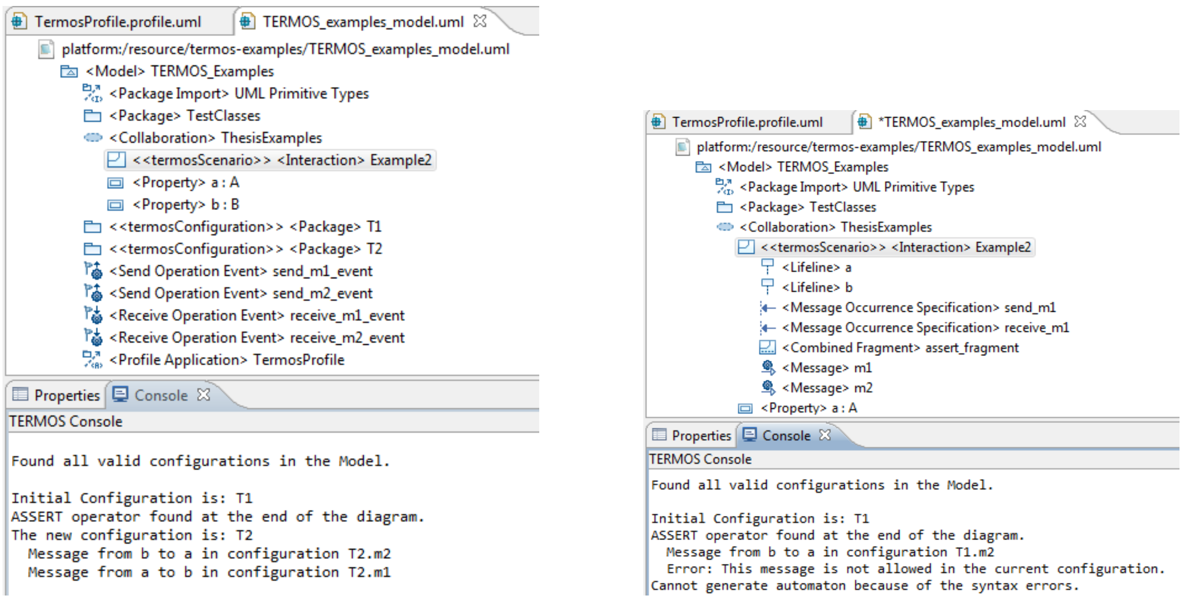
The first prototype for TERMOS, developed at LAAS by Irina Nitu, focused on the study of the unwinding algorithm itself. It served as a quick feedback on the algorithm, as e.g., the tool could provide a graphical visualization of the generated automaton structure using the Graphviz open source package. The graphical visualization allowed to manually check the result of the algorithm for a sample of diagrams, illustrating the various TERMOS constructs.

The second prototype for TERMOS, developed at BME as the Master thesis of Áron Hamvas [Ham10][1], focused on integrating TERMOS in a UML tool. The tool is implemented as an Eclipse plug-in, as the Eclipse platform has extensive built-in support to manipulate UML models. The plug-in loads an Eclipse UML2 compliant model, and operates on the UML elements directly. The tool performs the following tasks on UML 2 models tagged with the TERMOS profile stereotypes defined in Section 4.3.2:

[1]I was the advisor for the Master thesis, thus gave directions and feedback, but Áron Hamvas implemented the tool.

- checks whether the scenario conforms to the syntactic constraints specified in Section 4.3;
- checks whether the messages in the scenario conform to the semantic restrictions in Section 4.3.3.3;
- builds an automaton from a scenario and the accompanying configurations;
- can evaluate traces with respect to the generated automaton.

Figure 4.16 shows the TERMOS plug-in in action. The left side shows a successful check, while on the right side the scenario is not well-formed, one of the messages is not allowed according to the actual spatial configuration. Once the checks are successful and an automaton is generated, traces can be evaluated with the TERMOS plug-in.



```
platform:/resource/termos-examples/TERMOS_examples_model.uml
  <Model> TERMOS_Examples
    <Package Import> UML Primitive Types
    <Package> TestClasses
    <Collaboration> ThesisExamples
      <<termosScenario>> <Interaction> Example2
      <Property> a : A
      <Property> b : B
    <<termosConfiguration>> <Package> T1
    <<termosConfiguration>> <Package> T2
    <Send Operation Event> send_m1_event
    <Send Operation Event> send_m2_event
    <Receive Operation Event> receive_m1_event
    <Receive Operation Event> receive_m2_event
    <Profile Application> TermosProfile
```

```
Found all valid configurations in the Model.

Initial Configuration is: T1
ASSERT operator found at the end of the diagram.
The new configuration is: T2
  Message from b to a in configuration T2.m2
  Message from a to b in configuration T2.m1
```

```
platform:/resource/termos-examples/TERMOS_examples_model.uml
  <Model> TERMOS_Examples
    <Package Import> UML Primitive Types
    <Package> TestClasses
    <Collaboration> ThesisExamples
      <<termosScenario>> <Interaction> Example2
        <Lifeline> a
        <Lifeline> b
        <Message Occurrence Specification> send_m1
        <Message Occurrence Specification> receive_m1
        <Combined Fragment> assert_fragment
        <Message> m1
        <Message> m2
      <Property> a : A
```

```
Found all valid configurations in the Model.

Initial Configuration is: T1
ASSERT operator found at the end of the diagram.
  Message from b to a in configuration T1.m2
  Error: This message is not allowed in the current configuration.
Cannot generate automaton because of the syntax errors.
```

(a) Successful check (b) Check failed

Figure 4.16: Syntactic and semantic checks in the TERMOS plug-in

Note, that integration of these tools was not completed initially (it was only completed just recently in 2013 at LAAS [AWR13]). Thus at first the components of the framework were analyzed separately, e.g., the mapping returned by GraphSeq had to be manually specified in the input file of the TERMOS plug-in.

GraphSeq was evaluated using the traces from the GMP case study and by connecting it to a mobility simulator [NWR10]. The TERMOS language and plug-in was mainly demonstrated using scenarios created for the GMP protocol. The final deliverable of the Hidenets project [HW08] added a few illustrative scenarios inspired from other uses cases (a blackboard application, a platooning application and a distributed black box system).

Moreover, an important lesson learned from implementing the TERMOS plug-in was that there are issues regarding the conformance and interoperability of UML modeling tools.

Conformance to the standard We have not found a tool, which conforms 100% to the UML 2.x Sequence Diagrams specification. For example, the graphical symbol of Continuation, which was chosen for representing configuration changes, were not available in several tools; some tools placed guards always on the first Lifelines; or an empty guard was displayed even for an *assert*.

Import/export Most of the tools support exporting to XMI, but because of version incompatibilities and interoperability issues they are often not able to import each other's output. This makes it hard do create the models in one tool, and then process them separately in Eclipse UML2 (which has a limited, tree-view based user interface for manipulating models). Recently, the Model Interchange Working Group (MIWG) of OMG has started to work on this issue [EL12], but it takes time until all the issues will be resolved.

Handling profiles Some of the TERMOS extensions were defined as a UML profile, as this is the recommended extension mechanism for UML. However, the modeling tools handle profiles in various ways, which makes it again difficult to reuse profiles in different tools or even in different major versions of the same tool.

For this reason, the XML representation of some of the example scenarios had to be manually modified before they could be processed by the TERMOS plug-in.

4.5 Summary

This chapter presented an approach for testing context-aware mobile systems with dynamic communication structures (Figure 4.17).

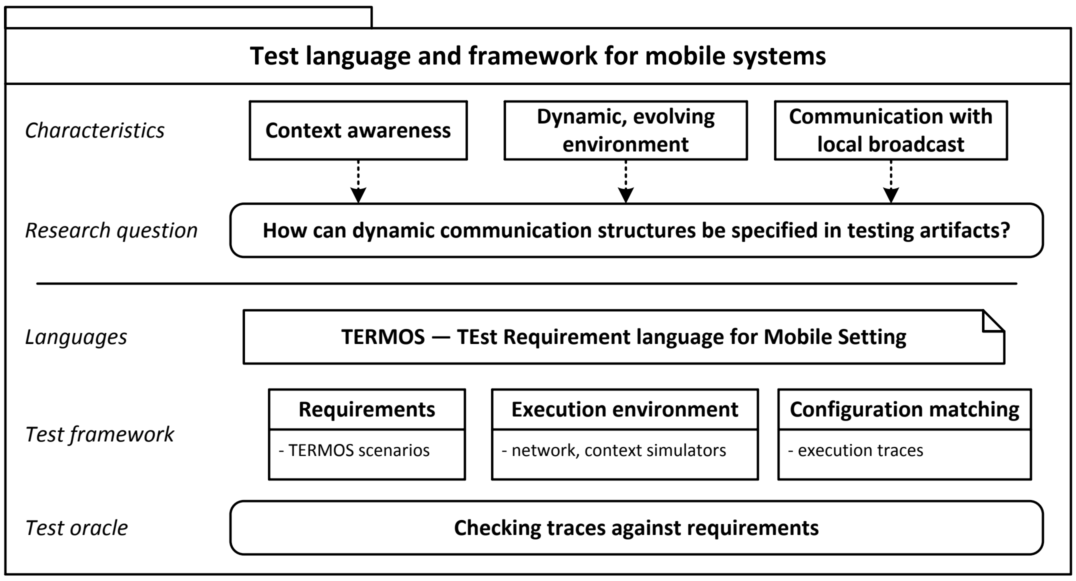

Test language and framework for mobile systems
Characteristics
Context awareness
Dynamic, evolving environment
Communication with local broadcast
Research question
How can dynamic communication structures be specified in testing artifacts?
Languages
TERMOS — TEst Requirement language for Mobile Setting
Test framework
Requirements
- TERMOS scenarios
Execution environment
- network, context simulators
Configuration matching
- execution traces
Test oracle
Checking traces against requirements

Figure 4.17: Summary of problem, languages and framework for mobile system testing

We proposed to extend classical graphical scenario languages with a spatial view, and represent explicitly the connection between the messages and the actual context and communication topology in which the messages were sent. The proposed extensions could be incorporated in several test artifacts, e.g. test purposes or requirements. Using these extensions we developed a test requirement languages called TERMOS. TERMOS is based on UML 2 Sequence Diagrams, and it was carefully designed to have several syntactic restrictions to avoid ambiguity and make checking of execution traces possible. The structured categorization of semantic choices and options presented in Chapter 3 was used to design the semantics of TERMOS. We defined an automaton-based operational semantics for the language, and implemented two tools when developing the proposed algorithms. The tools were tested with scenarios from a mobile group membership protocol, exemplifying ad hoc networking issues. The tools showed that the language is capable of expressing requirements for mobile systems, the operational semantics is working, however, there are still issues with the model interchange and interoperability in current modeling tools.

Chapter 5

Conclusion and future work

5.1 Summary of the research results

To close the dissertation this section revisits the initial open research questions in the identified new challenges (Section 1.3), and discusses the achieved results and their applications.

Challenge 1: Adapting robustness testing to HA middleware. *How can relevant test inputs for a HA middeware be specified in test artifacts to support the automated testing of the robustness of such systems?*

Chapter 2 presented the developed languages and robustness test approach for HA middleware. We identified the possible inputs to activate robustness faults and based on this information designed a robustness test framework that uses the combination of type-specific testing, mutation-based testing and OS call diversion. The framework was developed by identifying the required test artifacts, next creating the necessary languages, and finally implementing tools that generate the test artifacts. The robustness test suite was executed in several case studies on three different middleware implementations: on OpenAIS, on OpenSAF, and on Fujitsu-Siemens' SAFE4TRY. The faults identified in OpenAIS were reported to the open source community, and the whole robustness test suite was made publicly available [BME07]. The robustness testing results for OpenAIS and OpenSAF were uploaded to the public AMBER Data Repository (ADR) [AMM10]. ADR is an open repository for storing, analyzing and sharing benchmark and measurement data. Our robustness testing results can be downloaded or analyzed in ADR. (Note, the test results were prepared for the ADR repository and uploaded by András Kövi.)

Challenge 2: Specifying mobile systems in test artifacts. *How can dynamic, frequently changing communication structures and unknown partners be specified in test artifacts in a way that such systems can be later evaluated?*

Chapter 4 introduced the recommended language extensions for representing dynamic, frequently changing communication structures and unknown partners. Using these extensions we defined a test requirement language based on UML 2 Sequence Diagrams called TERMOS, and a test framework that can check test traces with respect to TERMOS requirements. The TERMOS language was developed based on the analysis of a mobile group membership protocol (GMP). The syntax and semantics of the language was carefully designed to support evaluation of traces. Prototype tools were created to validate the recommended semantics and test approach, and they were tested using the scenarios from the GMP.

Challenge 3: Analyzing the semantics of UML 2 Sequence Diagrams. *What semantic choices are available in UML 2 Sequence Diagrams, and what options can be chosen when the language is extended to a specific application domain?*

Chapter 3 presented the work on the semantics of UML 2 Sequence Diagrams. Based on an analysis of 13 proposed formal semantics we identified the possible semantic choices in UML 2 Sequence Diagrams, and presented the different options using a notation inspired by feature models. This presentation illustrates alternative, conflicting or non-standard choices. The detailed discussions of the choices highlight the consequences of the different options and recommend one or the other for different purposes. This categorization of semantic choices was used to define the syntax and semantics of TERMOS.

5.2 Future work

The last section lists for each of the major chapters the remaining open questions, which can direct future work.

5.2.1 Robustness testing of HA middleware

The case studies showed that the developed tools and test framework can uncover robustness failures in HA middleware systems, and qualitative observations can be made about the different implementations. However, tracing back the robustness failures to the robustness faults, and in this way obtaining a *quantitative metric*, was only performed in the first case study. This case study showed that with the help of data analysis techniques identifying the possible root causes is possible; however, the automatic processing of the robustness testing results is still an open question.

The robustness framework was designed for AIS-based HA middleware, however results of the research can be exploited in other similar systems. On one hand, if the new target environment resembles the current one (e.g., C language API or POSIX-compliant OS), then even the tools can be reused as they can be extensively configured. For type-specific testing, all the templates and metadata files have to be redeveloped, however the generator tool can be utilized. The mutation tool can work on other C sources, but possibly new operators should be implemented to cover the important fault types of the new system. The OS wrapper does not depend on the HA middleware, when using with a different system only the OS calls to divert should be reselected. On the other hand, even if the tools cannot be used directly for some reason, then the presented methodology – first identifying inputs to activate robustness faults and required test artifacts, next designing the necessary languages and automatic tools – could serve as guidance.

5.2.2 Semantic choices in UML 2 Sequence Diagrams

Our work may admittedly need future extensions, to account for the large number of semantics existing today or the ones that will continue to emerge in the next years. Moreover, there is currently significant work in progress inside the OMG with respect to UML. On one hand, a formal semantics for a base subset of UML, called fUML [BC11], was proposed. It does not cover high-level constructs like Interactions yet, but hopefully it will evolve. On the other hand, the 2.5 version of UML is underway, with the main goal to simplify and restructure the specification [Coo12]. This activity can make the semantic variations more visible.

Nevertheless, we believe that the feature-model-like representation offers a convenient support for documenting the semantic variants, and for updating the existing categorization. In the long run, it could be imagined that the OMG standard for Sequence Diagrams includes a model similar to ours, so as to make the semantic variation points more explicit. If such is the case, we would recommend that some of the options marked as non-standard (like working with partial traces, considering a categorization into valid/other or invalid/other, or synchronizing on the borders of fragments) be explicitly mentioned in the specification. We repeatedly found them in several of the surveyed semantics, and feel that they are very useful to address some recurring needs related to verification and testing purposes.

5.2.3 A test language and framework for mobile systems

The work on testing mobile systems concentrated more on the language side than the framework. For example, connecting the GraphSeq tool and the TERMOS plug-in or implementing the execution environment fully functionally was not performed initially (however, it was just recently finished in LAAS). Regarding the language, the combination of a spatial and event view proved to be a viable idea. Implementing the operational semantics in automatic tools made it possible to experiment with various example scenarios, which helped to refine algorithms of the semantics. However, as it is common with research prototype tools, the thorough and systematic testing of the construction of the automaton was not carried out. The validation of the language and the framework in an end-to-end case study is now possible and will be done in the next future.

The work on TERMOS focused on the change in the communication structure. Some context information could be specified with label variables, however the examples concentrated on the state of the communication links. In other application domains, e.g., autonomous systems, the context is more detailed, and the objects in the environment have more attributes and relations that should be taken into account in the scenarios. In the context of the R3-COP research project [R3C11] we are working currently on testing autonomous systems [6], which will extend the results of TERMOS. For example, a more fine-grained context model including inheritance or complex attributes is used, and test setups used in the execution environment are now systematically generated based on combining the different scenarios. The lessons learned with TERMOS will influence the tooling also, e.g., UML profiles will not be used.

Finally, the contributions of the dissertation can be combined, e.g., robustness could be also an important characteristics of mobile systems. The mutation-based testing approach can utilize the scenarios defined in TERMOS to reach important states, where the invalid values or messages can be submitted to the system under test. In a recently started national research project one of our subtasks' topic is validation of distributed, mobile middleware systems, which offers a natural opportunity to unite the results of the dissertation.

Appendix A

Details on the robustness test framework

A.1 XML Schema of the type-specific tool's configuration

A.1.1 XML Schema of the types' metadata

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Types">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="Module">
          <xs:complexType>
            <xs:sequence>
            <xs:element maxOccurs="unbounded" name="Type">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Name" type="xs:string" />
                  <xs:element name="ValidValueMethod" minOccurs="0" maxOccurs="1">
                    <xs:complexType>
                      <xs:attribute name="generate" type="xs:boolean" use="required" />
                      <xs:attribute name="validValueIndex" type="xs:unsignedByte" use="optional" />
                    </xs:complexType>
                  </xs:element>
                  <xs:element name="PointerMethod" nillable="false">
                    <xs:complexType>
                      <xs:attribute name="generate" type="xs:boolean" use="required" />
                    </xs:complexType>
                  </xs:element>
                  <xs:element name="ParentName" minOccurs="0" maxOccurs="1">
                    <xs:complexType>
                      <xs:attribute name="value" type="xs:string" use="required"/>
                    </xs:complexType>
                  </xs:element>
                  <xs:element name="IncludeFile" minOccurs="0" maxOccurs="1">
                    <xs:complexType>
                      <xs:attribute name="fileName" type="xs:string" use="required" />
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

A.1.2 XML Schema of the functions' metadata

```xml
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="functions">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="module">
          <xs:complexType mixed="true">
            <xs:sequence minOccurs="0">
              <xs:element maxOccurs="unbounded" name="function">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="returnType" type="xs:string" />
                    <xs:element name="parameters">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element maxOccurs="unbounded" name="parameter">
                            <xs:complexType>
                              <xs:sequence>
                                <xs:element name="parameterOrder" type="xs:unsignedByte" />
                                <xs:element name="parameterName" type="xs:string" />
                                <xs:element name="parameterType" type="xs:string" />
                                <xs:element name="isPointer" type="xs:boolean" />
                                <xs:element name="type" maxOccurs="1" minOccurs="0">
                                  <xs:simpleType>
                                    <xs:restriction base="xs:string">
                                      <xs:enumeration value="in"/>
                                      <xs:enumeration value="out"/>
                                      <xs:enumeration value="in/out"/>
                                    </xs:restriction>
                                  </xs:simpleType>
                                </xs:element>
                              </xs:sequence>
                            </xs:complexType>
                          </xs:element>
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name" type="xs:string" use="required" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

A.2 XML Schema of the mutant generator tool's configuration

```xml
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Inputs">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="Input">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Location" type="xs:string" />
              <xs:element name="Mutations">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="NumberOfMutants" type="xs:unsignedByte" />
                    <xs:element name="NumberOfOperatorToApply" type="xs:unsignedByte" />
                    <xs:element name="Operators">
                      <xs:complexType>
                        <xs:choice maxOccurs="unbounded">
                          <xs:element minOccurs="0" maxOccurs="unbounded" name="SwapCalls">
                            <xs:complexType>
                              <xs:attribute name="call1" type="xs:string" use="required" />
                              <xs:attribute name="call2" type="xs:string" use="required" />
                            </xs:complexType>
                          </xs:element>
                          <xs:element minOccurs="0" maxOccurs="unbounded" name="RelocateCall">
                            <xs:complexType>
                              <xs:attribute name="call" type="xs:string" use="required" />
                              <xs:attribute name="inFunction" type="xs:string" use="required" />
                            </xs:complexType>
                          </xs:element>
                          <xs:element minOccurs="0" maxOccurs="unbounded" name="ReplaceParameterWithNull">
                            <xs:complexType>
                              <xs:attribute name="call" type="xs:string" use="required" />
                              <xs:attribute name="parameterNumber" type="xs:unsignedByte" use="required" />
                              <xs:attribute name="inFunction" type="xs:string" use="required" />
                            </xs:complexType>
                          </xs:element>
                          <xs:element minOccurs="0" maxOccurs="unbounded" name="OmitCall">
                            <xs:complexType>
                              <xs:attribute name="call" type="xs:string" use="required" />
                              <xs:attribute name="inFunction" type="xs:string" use="required" />
                            </xs:complexType>
                          </xs:element>
                          <xs:element minOccurs="0" maxOccurs="unbounded" name="ModifyIfCondition">
                            <xs:complexType>
                              <xs:attribute name="inFunction" type="xs:string" use="required" />
                            </xs:complexType>
                          </xs:element>
                        </xs:choice>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="type" use="required">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:enumeration value="file"></xs:enumeration>
                  <xs:enumeration value="directory"></xs:enumeration>
                </xs:restriction>
              </xs:simpleType>
            </xs:attribute>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

A.3 XML Schema of the OS call wrappers tool's configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="functions">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="function"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="function">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="signature"/>
        <xs:element ref="interception"/>
      </xs:sequence>
      <xs:attribute name="name" use="required" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="signature">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="returnType" type="xs:string"/>
        <xs:element ref="standardErrors"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="standardErrors">
```

```
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="1" ref="error"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="error">
    <xs:complexType>
      <xs:attribute name="type" use="required" type="xs:string"/>
      <xs:attribute name="value" use="required" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="interception">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="logCall" type="xs:boolean"/>
        <xs:element name="forwardCall" type="xs:boolean"/>
        <xs:element ref="detourChance"/>
        <xs:element ref="returnValue"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="detourChance">
    <xs:simpleType>
      <xs:restriction base="xs:integer">
        <xs:minInclusive value="0"/>
        <xs:maxInclusive value="100"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
  <xs:element name="returnValue">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="mode"/>
        <xs:element name="delay" type="xs:nonNegativeInteger"/>
        <xs:element ref="desiredReturn" minOccurs="0" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="mode">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="normal"/>
        <xs:enumeration value="desiredReturn"/>
        <xs:enumeration value="standardError"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
  <xs:element name="desiredReturn">
    <xs:complexType>
      <xs:attribute name="type" use="required" type="xs:string"/>
      <xs:attribute name="value" use="required" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Appendix B

Details on UML 2 Sequence Diagrams

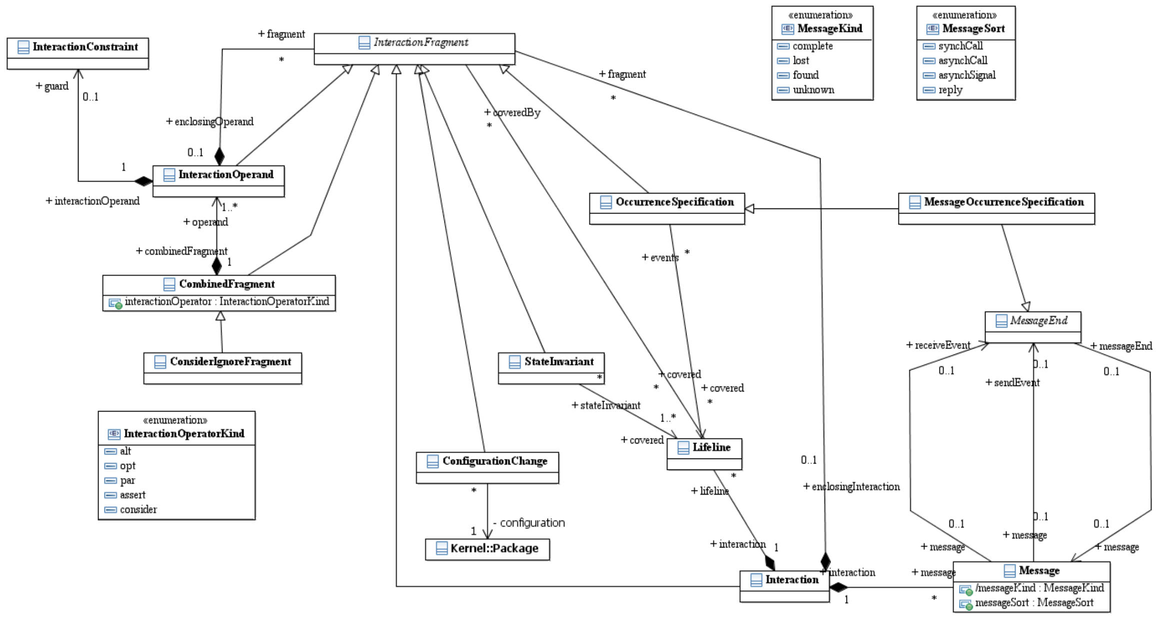
B.1 XMI of an example Interaction

```
<packagedElement xmi:type="uml:Collaboration" xmi:id="collaboration_id" name="Collaboration1">
 <ownedBehavior xmi:type="uml:Interaction" xmi:id="sd1_id" name="sd1">
  <ownedConnector xmi:type="uml:Connector" xmi:id="connector_id">
   <end xmi:type="uml:ConnectorEnd" xmi:id="connectorend-1_id" role="a_id" />
   <end xmi:type="uml:ConnectorEnd" xmi:id="connectorend-2_id" role="b_id" />
  </ownedConnector>
  <lifeline xmi:type="uml:Lifeline" xmi:id="a-ll_id" name="a"
    represents="a_id" coveredBy="opt-cf_id r-m2_id" />
  <lifeline xmi:type="uml:Lifeline" xmi:id="b-ll_id" name="b"
    represents="b_id" coveredBy="opt-cf_id s-m2_id" />
  <fragment xmi:type="uml:CombinedFragment" xmi:id="opt-cf_id"
    covered="b-ll_id a-ll_id" interactionOperator="opt">
   <operand xmi:type="uml:InteractionOperand" xmi:id="opt-op_id">
    <guard xmi:type="uml:InteractionConstraint" xmi:id="guard_id">
     <specification xmi:type="uml:OpaqueExpression" xmi:id="guard-expression_id">
      <body>b.d > 5</body>
     </specification>
    </guard>
    <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="s-m2_id"
      covered="b-ll_id" event="sendEvent_id" message="m2_id" />
    <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="r-m2_id"
      covered="a-ll_id" event="receiveEvent_id" message="m2_id" />
   </operand>
  </fragment>
  <message xmi:type="uml:Message" xmi:id="m2_id" name="m2" messageSort="asynchCall"
    receiveEvent="r-m2_id" sendEvent="s-m2_id" connector="connector_id" />
 </ownedBehavior>
 <ownedAttribute xmi:type="uml:Property" xmi:id="a_id" name="a" type="A_id" end="connectorend-1_id"/>
 <ownedAttribute xmi:type="uml:Property" xmi:id="b_id" name="b" type="B_id" end="connectorend-2_id"/>
</packagedElement>
<packagedElement xmi:type="uml:SendOperationEvent" xmi:id="sendEvent_id"
  name="SendOperationEvent3" operation="m2-operation_id"/>
<packagedElement xmi:type="uml:ReceiveOperationEvent" xmi:id="receiveEvent_id"
  name="ReceiveOperationEvent3" operation="m2-operation_id"/>
<packagedElement xmi:type="uml:Class" xmi:id="A_id" name="A">
 <ownedOperation xmi:type="uml:Operation" xmi:id="m2-operation_id" name="m2"/>
</packagedElement>
<packagedElement xmi:type="uml:Class" xmi:id="B_id" name="B"/>
```

Appendix C

Details on the TERMOS language

C.1 Abstract syntax of the event view



C.2 The unwinding algorithm

In the definition of the operational semantics of TERMOS, the unwinding algorithm builds the symbolic automaton by gradually unwinding the classes of atoms, until all of them have been processed.

Definition 12 (Symbolic automaton) A symbolic automaton can be defined with a tuple $(\Sigma, Q, q_0, F_T, F_S, \rightarrow, Var, Def)$, where:

- Σ is a set of transition labels with possibly symbolic variables in Var,
- Q is the set of states,
- q_0 is the initial state,

- $F_T \subseteq Q$ and $F_S \subseteq Q$ are two disjoint subsets of accept states. They are used to distinguish trivial satisfaction of the requirements (the trigger before the $assert$ did not match) and stringent satisfaction (the content of the both the $assert$ and the trigger did match),
- $\rightarrow \subseteq Q \times \Sigma \times Q$ is the set of transitions,
- Var is the set of variables extracted from the TERMOS scenario. It includes all variables appearing in the spatial view (symbolic labels of vertices), in the event view (message parameters, free variables in OCL expression of guards and state invariants), and all symbolic message identifiers $\$i$ produced by the preprocessing of the diagram.
- $Def \subseteq Q \times \mathcal{P}(Variables)$ gives the subset of variables that are defined for each state. If $(q, \{v_1, v_2\})$ belongs to Def, then variables v_1, v_2 have a value in q, and all other variables are undefined in q. □

The unwinding algorithm is based on the notion of *phase*. The computed phases will correspond to automaton states.

Definition 13 (Phase) A phase is a tuple $(History, Ready, Cut, Variables)$, where:

- $History$ is the set of SimClasses which have already been unwound,
- $Ready$ is the set of SimClasses which are currently enabled to be unwound,
- Cut is a tuple $(cl_1, \ldots, cl_n)$, where n is the number of Lifelines in the diagram and each cl_j is a cluster from Lifeline j. The current cut is intended to represent the borderline between already unwound elements and those that are currently enabled.
- $Variables$ is the set of variables which are currently valuated. □

Like Klose, we assume that there is a function $\textsc{State}: phase \rightarrow Q$ assigning a unique state name for a phase. Note that if the same phase is encountered several times, the function is able to return the name already assigned at the previous steps of the unwinding algorithm.

Initialization The unwinding algorithm begins with an initialization step (see Algorithm C.1 for the definition).

Definition 14 (Initial phase) The initial phase considered by the algorithm is $(History_0, Ready_0, Cut_0, Variables_0)$ defined as follows:

- $History_0 = \{\{\{\perp_1\}\}, \{\{\perp_2\}\}, \ldots, \{\{\perp_n\}\}\}$[1],
- $Ready_0 = \{scl \in \text{SimClasses}(sd) \mid \text{Prerequisites}(scl) \subseteq History_0\}$,
- $Cut_0 = (\{\perp 1\}, \{\perp_2\}, \ldots, \{\perp_n\})$,
- $Variables_0$ set of variables appearing in the initial spatial configuration of the scenario (this includes the symbolic identifiers of the nodes participating to the scenario). □

That is, at the initial phase, only the lifeline heads have been unwound. We also assume that we start analysis in a state where the system is in the initial spatial configuration. This is ensured by connecting the GraphSeq graph matching tool to the TERMOS scenario. The graph matching tool also supplies concrete values for the identifiers of nodes and all other variables defined by the initial configuration, which are then marked as valuated. Finally, $\textsc{State}(Phase_0)$ is added to the set Q of the automaton states and to the F_T trivial satisfaction subset.

[1] $History$ is a set of SimClasses, a SimClass is a set of clusters, a cluster is a set of atoms; hence this notation.

Unwinding From the initial phase, the algorithm proceeds by computing successor phases. Given a phase, the STEP function returns the next phase obtained by firing a ready SimClass.

Definition 15 (STEP function) Given a $ph = (History_i, Ready_i, Cut_i, Variables_i)$ phase and an scl SimClass to fire, the $\textsc{Step}(ph, scl)$ function returns a new phase $ph' = (History_{i+1}, Ready_{i+1}, Cut_{i+1}, Variables_{i+1})$ defined as follows:

- $History_{i+1} = History_i \cup scl \cup \mathrm{Conflicts}(scl)$, that is, both the fired scl and its conflicting SimClasses are considered unwound,
- $Ready_{i+1} = \Big\{scl' \in \big(\mathrm{SimClasses}(sd) \setminus \{\{\{\top_1\}\}, \ldots, \{\{\top_n\}\}\}\big) \mid \mathrm{Prerequisites}(scl') \subseteq History_{i+1} \wedge scl' \notin History_{i+1}\Big\}$
- $Cut_{i+1} = \{cl'_1, \ldots, cl'_n\}$ is produced from $Cut_i = \{cl_1, \ldots, cl_n\}$ by letting $cl'_j = cl_j$ if lifeline j is not concerned by any cluster of the unwound SimClass. Other elements cl'_k are replaced by the corresponding cluster of the unwound SimClass, for each involved lifeline k.
- $Variable_{i+1}$ is the union of $Variables_i$ and of the set of newly valuated variables. Note that there are newly valuated variables only if ph contains communication events or configuration change events. □

Moreover, the unwinding algorithm performs the following tasks (see Algorithm C.2 for the definition).

Mode The new phase may, or not, correspond to an accept state. In the algorithm, this is governed by the $CurrentMode$ variable. While the trigger is being matched, current mode is AcceptTrivial and the produced states are put in F_T. When the entering of an $assert$ is unwound, current mode switches to Reject. It switches to AcceptStringent when the $assert$ is exited, and the successor is put in the set F_S.

Transition labels Roughly speaking, transition labels are obtained by conjoining the individual labels obtained from the atoms (of the clusters) of the unwound SimClass. If there are newly valuated variables, then the transition label also contains an explicit update action. For example, let us assume that we are currently unwinding a guard $x > 3$ and a send event $(!m(x), n_1, \$4)$. Let us also assume that the values for x and n_1 are currently defined; but the symbolic message identifier \$4 have not been assigned by the preliminary analysis. Then, the corresponding transition will be labeled: $x > 3 \wedge (!m(x), n_1, \$4)[\text{update}(\$4)]$, which can be interpreted as follows: if $x > 3$ holds with the current valuation, and the next event of the trace can match $(!m(x), n1, \$4)$ with an appropriate assignment of \$4, then the transition can be taken. Taking the transition consumes the event of the trace, and the current valuation is updated with the concrete message identifier of this event.

CombinedFragments Entering and exiting boxes is simply represented by a true transition. Note that there could be further optimization to remove the unnecessary states.

Self loops Because TERMOS uses an interpretation with partial traces, self loops must be added as soon as at least one of the exiting transitions contains a trace event, be it a communication event or a configuration change event. For example, if the next event of the trace does not match $(!m(x), n_1, \$4)$, are we allowed to consume this event and remain in the same state? Conversely if it matches, do we still have the choice to remain in the same state? The answer to the latter question is negative, hence the self-loop is labeled $\neg(!m(x), n_1, \$4)$. The answer to the former question is generally positive according to the partial traces interpretation, but can be negative if the event is in the scope of a $consider$. Let us also remind that our interpretation of $consider\{m\}$ is: the sending of m is forbidden for all lifelines, but it is allowed to receive a message m, if its sending was not forbidden. Accordingly, the self-loop is concerned by sending events only. Finally, note that unexpected configurations change events are always forbidden, hence the self-loop label always contains $\neg\mathsf{CHANGE}(-)$.

C.3 A more complex example for the unwinding

Figure C.1 contains a complex scenario example. In the *T3* configuration all nodes are connected. Notice that the first *m1* and *m3* messages are only ordered partially. The scenario contains a *par* and an *alt* fragment, moreover, the *par* is nested inside the *alt*. The generated automaton can be seen on Figure C.2. The automaton was created by the TERMOS plug-in and it is visualized with the help of Graphviz.

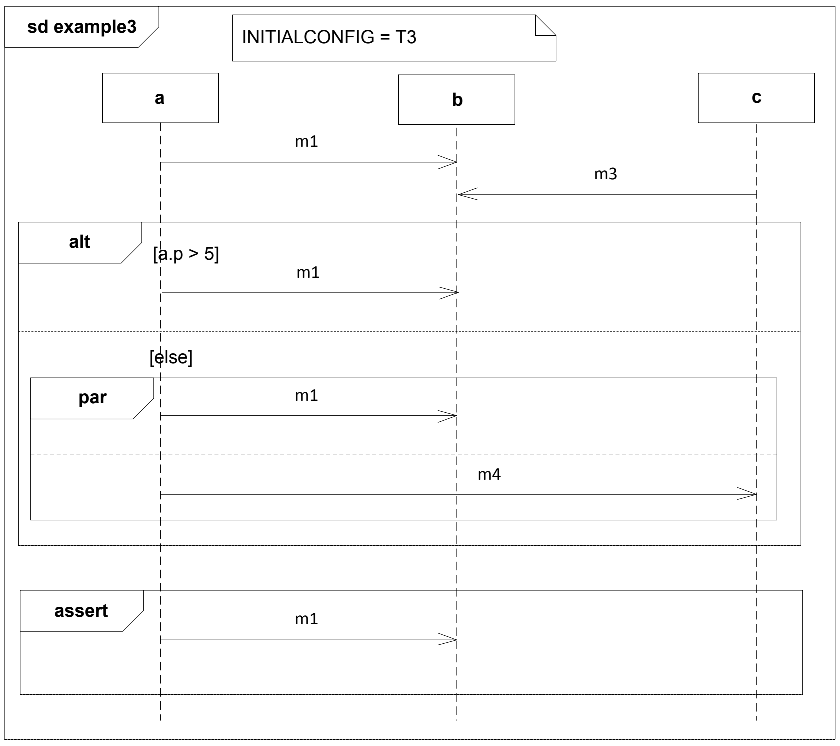


Figure C.1: A complex example scenario

Algorithm C.1: Unwinding algorithm

```
// Initialization
$Phases := \{Phase_0\}, Q := \{Phase_0\}, q_0 := \textsc{State}(Phase_0)$
$F_T := \{\textsc{State}(Phase_0)\}, F_S := \emptyset$
$CurrentMode :=$ AcceptTrivial
$SelfLoopLabel :=$ "¬ CHANGE(-)"                    // no unexpected config changes

// Unwinding loop
while $Phases \neq \emptyset$ do
    Extract $ph = (History_i, Ready_i, Cut_i, Variables_i)$ from $Phases$
    if $Ready_i \neq \emptyset$ then
        $SelfLoop :=$ false, $AddedSelfLabel :=$ ""
        foreach $sc \in Ready_i$ do
            $sucessor := \textsc{Step}(ph, sc)$
            // Compute the label of the triggered transition
            $UpdatedVariables := \emptyset, Label :=$ ""
            foreach $cl \in sc$ do                              // process clusters
                foreach $a \in cl$ do                           // process atoms
                    $\textsc{ProcessAtom}(a)$
            // Update the transition set
            if $UpdatedVariables \neq \emptyset$ then
                Build a label $ll$ of the form [list of updated variables]
                Append $ll$ to $Label$
            $\rightarrow := \rightarrow \cup \{(\textsc{State}(ph), Label, \textsc{State}(successor))\}$
            // Put successor in automaton states
            $Q := Q \cup \{\textsc{State}(sucessor)\}$
            if $CurrentMode$ = AcceptTrivial then
                $F_T := F_T \cup \{\textsc{State}(sucessor)\}$
            else if $CurrentMode$ = AcceptStringent then
                $F_S := F_S \cup \{\textsc{State}(sucessor)\}$
            // Put successor in phases
            Put $successor$ in $Phases$
        // Add a self-loop if needed
        if $SelfLoop$ = true then
            if $AddedSelfLabel$ is not empty then
                Build label $ll$ conjoining $AddedSelfLabel$ and $SelfLoopLabel$
            else
                $ll := SelfLoopLabel$
            $\rightarrow := \rightarrow \cup \{(\textsc{State}(ph), ll, \textsc{State}(ph))\}$
```

Algorithm C.2: *ProcessAtom(a)*: Processing atoms in the unwinding loop

```
Input: $a$ atom to process
switch $a$ do
    case entering an assert
        if Label = "" then
            $CurrentMode$ := Reject
            Label := "true"
    case exiting an assert
        if Label = "" then
            $CurrentMode$ := AcceptStringent
            Label := "true"
    case entering a consider
        if Label = "" then
            foreach considered message name $m$ do
                foreach symbolic node id $l_i$ in the current valuation do
                    build label $ll$ of the form: $\neg(!m(-), l_i, -)$
                    SelfLoopLabel := SelfLoopLabel conjoined with $ll$
            Label := "true"
    case exiting a consider
        if Label = "" then
            SelfLoopLabel := "¬ CHANGE(-)"
            Label := "true"
    case exiting an alt or a par
        if Label = "" then
            Label := "true"
    case guard or StateInvariant
        Make a label $ll$ with the predicate
        if $ll$ does not already appear in Label then
            Label := Label conjoined with $ll$
    case configuration change
        if Label = "" then
            UpdatedVariables := {variables in new config $C_i$} \ $Variables_i$
            SelfLoop := true
            Make a label $ll$ of the form "CHANGE($C_i$)"
            Label := $ll$
    case send or receive event
        UpdatedVariables := {variables in message parameters or message id} \ $Variables_i$
        SelfLoop := true
        Make a label $ll$ of the event
        Label := Label conjoined with $ll$
        if (the atom is a receive event) $\vee$ (the atom is a send event that does not appear under the
        form $\neg(!m(-), l_i, -)$ in SelfLoopLabel) then
            AddedSelfLabel := AddedSelfLabel conjoined with $\neg ll$
```

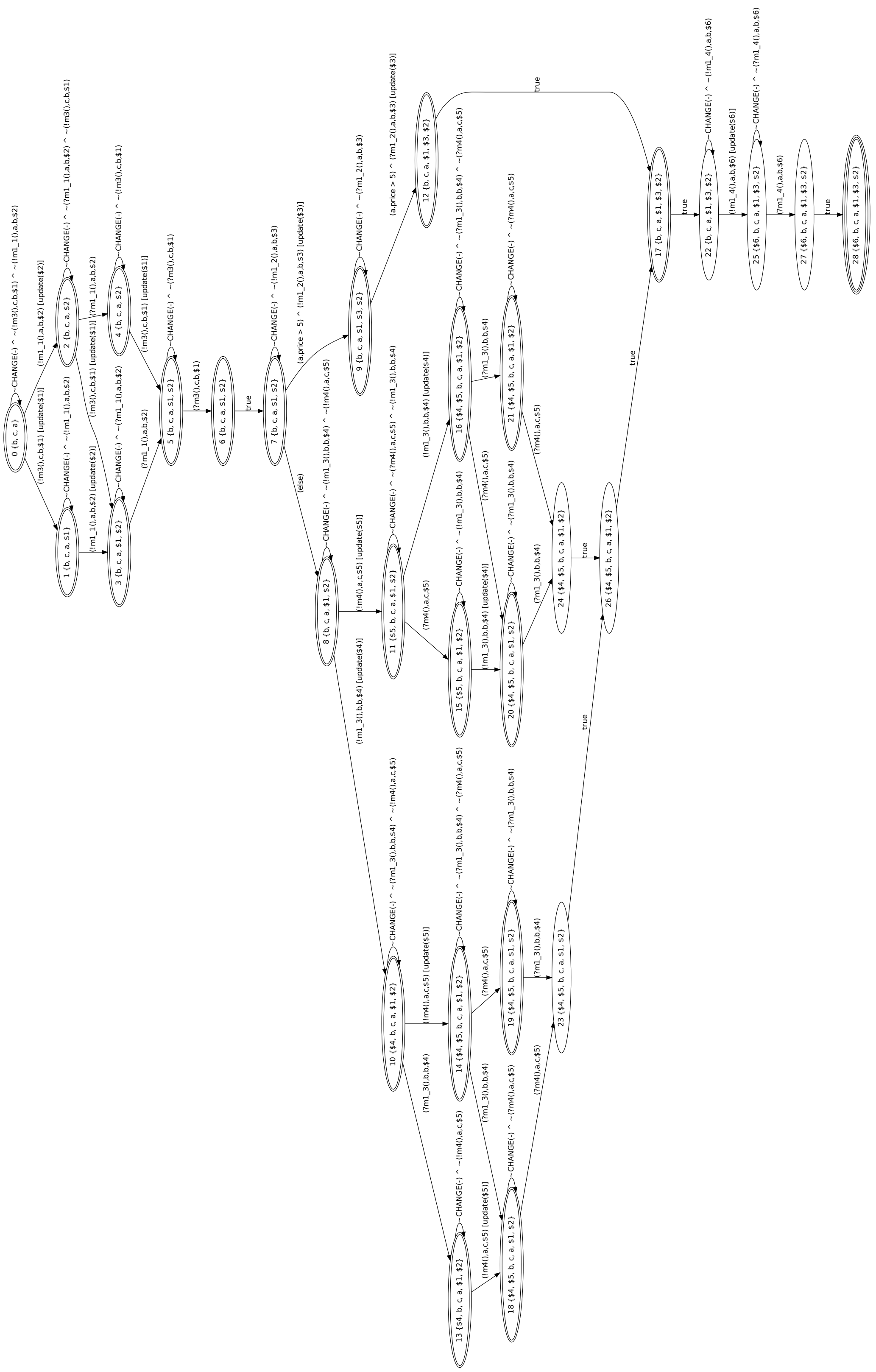


Figure C.2: The automaton generated from the scenario on Figure C.1

Bibliography

Publication list

Journal papers

[1] Z. Micskei and H. Waeselynck. “The many meanings of UML 2 Sequence Diagrams: a survey”. In: *Software and Systems Modeling* 10.4 (2011), pp. 489–514. DOI: 10.1007/s10270-010-0157-9

[2] G. Pintér, Z. Micskei, and I. Majzik. “Supporting design and development of safety critical applications by model based tools”. In: *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös nominatae Sectio Computatorica* XXX (2009), pp. 61–78. ISSN: 0138-9491

[3] I. Kocsis, A. Pataricza, Z. Micskei, A. Kövi, and Z. Kocsis. “Analytics of Resource Transients in Cloud Based Applications”. In: *Int. Journal of Cloud Computing* (). To appear. URL: http://www.inderscience.com/info/ingeneral/forthcoming.php?jcode=ijcc

Chapters in edited books

[4] Z. Micskei, H. Madeira, A. Avritzer, I. Majzik, M. Vieira, and N. Antunes. “Robustness Testing Techniques and Tools”. In: *Resilience Assessment and Evaluation of Computing Systems.* Ed. by K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel. Springer, 2012, pp. 323–339. DOI: 10.1007/978-3-642-29032-9_16

[5] G. Pintér, Z. Micskei, A. Kövi, Z. Égel, I. Kocsis, G. Huszerl, and A. Pataricza. “Model-Based Approaches for Dependability in Ad-Hoc Mobile Networks and Services”. In: *Architecting Dependable Systems V.* Springer, 2008, pp. 150–174. DOI: 10.1007/978-3-540-85571-2_7

International conferences

[6] Z. Micskei, Z. Szatmári, J. Oláh, and I. Majzik. “A Concept for Testing Robustness and Safety of the Context-Aware Behaviour of Autonomous Systems”. In: *Agent and Multi-Agent Systems. Technologies and Applications.* Ed. by G. Jezic, M. Kusek, N.-T. Nguyen, R. Howlett, and L. Jain. Vol. 7327. LNCS. Springer, June 2012, pp. 504–513. ISBN: 978-3-642-30946-5. DOI: 10.1007/978-3-642-30947-2_55

[7] I. Kocsis, A. Pataricza, Z. Micskei, I. Szombath, A. Kövi, and Z. Kocsis. “Cloud Based Analytics for Cloud Based Applications”. In: *1st International IBM Cloud Academy Conference.* Research Triangle Park, USA, Apr. 2012, pp. 1–23

[8] H. Waeselynck, Z. Micskei, N. Rivière, Á. Hamvas, and I. Nitu. “TERMOS: a Formal Language for Scenarios in Mobile Computing Systems”. In: *Mobile and Ubiquitous Systems: Computing, Networking, and Services (MobiQuitous 2010).* Ed. by P. Sénac, M. Ott, and A. Seneviratne. Sydney, Australia, Dec. 2010, pp. 285–296. DOI: 10.1007/978-3-642-29154-8_24

[9] A. Kövi and Z. Micskei. “Robustness Testing of Standard Specifications-Based HA Middleware”. In: *30th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, June 2010, pp. 302–306. ISBN: 978-1-4244-7471-4. DOI: `10.1109/ICDCSW.2010.73`

[10] F. Bouquet, R. Breu, J. Jurjens, F. Massacci, V. Meduri, Z. Micskei, F. Piessens, K. Stolen, and D. Varró. “SecureChange: Security Engineering for Lifelong Evolvable Systems”. In: *European Future Technologies Conference and Exhibition (FET09)*. Poster Session. Prague, Czech Republic, Apr. 2009, pp. 101–102

[11] L. Gönczy, I. Majzik, A. Horváth, D. Varró, A. Balogh, Z. Micskei, and A. Pataricza. “Tool Support for Engineering Certifiable Software”. In: *Electronic Notes in Theoretical Computer Science* 238.4 (2009). Proc. of 1st Workshop on Certification of Safety-Critical Software Controlled Systems (SafeCert 2008), pp. 79–85. ISSN: 1571-0661. DOI: `10.1016/j.entcs.2009.09.008`

[12] Z. Micskei, I. Majzik, and F. Tam. “Comparing Robustness of AIS-Based Middleware Implementations”. In: *International Service Availability Symposium (ISAS 2007)*. Ed. by M. Malek, M. Reitenspieß, and A. van Moorsel. Vol. 4526. LNCS. Springer, May 2007, pp. 20–30. DOI: `10.1007/978-3-540-72736-1_3`

[13] H. Waeselynck, Z. Micskei, M. D. Nguyen, and N. Rivière. “Mobile Systems from a Validation Perspective: a Case Study”. In: *Sixth International Symposium on Parallel and Distributed Computing (ISPDC ’07)*. Ed. by D. Kranzlmüller, W. Schreiner, and J. Volkert. Hagenberg, Austria: IEEE Computer Society, July 2007, pp. 85–92. ISBN: 0-7695-2936-4. DOI: `10.1109/ISPDC.2007.37`

[14] I. Majzik, Z. Micskei, and G. Pintér. “Development of Model Based Tools to Support the Design of Railway Control Applications”. In: *Computer Safety, Reliability, and Security*. Ed. by F. Saglietti and N. Oster. Vol. 4680. LNCS. Springer Berlin / Heidelberg, Sept. 2007, pp. 430–435. ISBN: 978-3-540-75100-7. DOI: `10.1007/978-3-540-75101-4_41`

[15] G. Pintér, Z. Micskei, and I. Majzik. “Supporting Design and Development of Safety Critical Applications by Model Based Tools”. In: *10th Symposium on Programming Languages and Software Tools (SPLST 2007)*. Ed. by Z. Horváth, L. Kozma, and V. Zsók. Dobogókő, Hungary: Eotvos University Press, June 2007, pp. 61–75

[16] Z. Micskei, I. Majzik, and F. Tam. “Robustness Testing Techniques for High Availability Middleware Solutions”. In: *International Workshop on Engineering of Fault Tolerant Systems (EFTS2006)*. Luxembourg, Luxembourg: University of Luxembourg, June 2006, pp. 55–66

[17] Z. Micskei and I. Majzik. “Model-based Automatic Test Generation for Event-Driven Embedded Systems using Model Checkers”. In: *Dependability of Computer Systems (DepCoS-RELCOMEX 2006)*. IEEE Computer Society, May 2006, pp. 191–198. ISBN: 0-7695-2565-2. DOI: `10.1109/DEPCOS-RELCOMEX.2006.37`

Local conferences

[18] Z. Micskei. “Specifying Tests for Ad-Hoc Mobile Systems”. In: *15th PhD Mini-Symposium*. Budapest University of Technology and Economics. Budapest, Hungary: Department of Measurement and Information Systems, Feb. 2008, pp. 32–35. ISBN: 978-963-420-938-6

[19] Z. Micskei. “Robustness Comparison of High Availability Middleware Systems”. In: *14th PhD Mini-Symposium*. Budapest University of Technology and Economics. Budapest, Hungary: Department of Measurement and Information Systems, Feb. 2007, pp. 70–73. ISBN: 978-963-420-895-2

[20] Z. Micskei. “Robustness Testing of High Availability Middleware Solutions”. In: *13th PhD Mini-Symposium.* Budapest University of Technology and Economics. Budapest, Hungary: Department of Measurement and Information Systems, Feb. 2006, pp. 36–37. ISBN: 963-420-853-3

[21] Z. Micskei. “Nagy Rendelkezésre Állást Biztosító Köztes Rétegek Robosztusság Tesztelése”. In: *Proceedings of Tavaszi Szél 2006.* Kaposvár, Hungary: Doktoranduszok Országos Szövetsége, May 2006, pp. 295–298. ISBN: 963-229-773-3

[22] Z. Micskei. “Automatikus tesztgenerálás modell ellenőrzővel”. In: *X. Fiatal Műszakiak Tudományos Ülésszaka (FMTU).* ed. by E. Bitay. Erdélyi Múzeum Egyesület. Kolozsvár, Románia, Mar. 2005, pp. 47–50. ISBN: 973–8231–44–2

Technical reports

[23] Z. Micskei and H. Waeselynck. *A survey of UML 2.0 sequence diagrams' semantics.* Tech. rep. 08389. Laboratoire d'Analyse et d'Architecture des Systemes (LAAS), Aug. 2008, pp. 1–37

[24] Z. Micskei, H. Waeselynck, M. D. Nguyen, and N. Rivière. *Analysis of a group membership protocol for Ad-hoc networks.* Tech. rep. 06797. Laboratoire d'Analyse et d'Architecture des Systemes (LAAS), Nov. 2006, pp. 1–42

References

[ABH12] S. Ali, L. Briand, and H. Hemmati. “Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems”. In: *Software and Systems Modeling* 11.4 (2012), pp. 633–670. DOI: 10.1007/s10270-011-0206-z.

[AMM10] R. Almeida, N. Mendes, and H. Madeira. “Sharing Experimental and Field Data: The AMBER Raw Data Repository Experience”. In: *Distributed Computing Systems Workshops (ICDCSW), 2010 IEEE 30th International Conference on.* 2010, pp. 313–320. DOI: 10.1109/ICDCSW.2010.75.

[AMN12] C. Andrés, M. G. Merayo, and M. Núñez. “Formal passive testing of timed systems: theory and tools”. In: *Software Testing, Verification and Reliability* (2012). DOI: 10.1002/stvr.1464.

[AMS03] S. Acharya, H. Mohanty, and R. Shyamasundar. “MOBICHARTS: a notation to specify mobile computing applications”. In: *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on.* 2003. DOI: 10.1109/HICSS.2003.1174844.

[Avi+04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. “Basic Concepts and Taxonomy of Dependable and Secure Computing”. In: *IEEE Trans. Dependable Secur. Comput.* 1 (2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2.

[AWR13] P. André, H. Waeselynck, and N. Rivière. *A UML-Based Environment for Test Scenarios in Mobile Settings.* Tech. rep. 13064. Laboratoire d'Analyse et d'Architectures des Systèmes (LAAS), 2013.

[Bar+90] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek. “Fault injection experiments using FIAT”. In: *Computers, IEEE Transactions on* 39.4 (1990), pp. 575–582. DOI: 10.1109/12.54853.

[Bau+03] H. Baumeister, N. Koch, P. Kosiuczenko, P. Stevens, and M. Wirsing. “UML for Global Computing”. In: *Global Computing. Programming Environments, Languages, Security, and Analysis of Systems.* Vol. 2874. LNCS. 2003, pp. 1–24. DOI: 10.1007/978-3-540-40042-4_1.

[BC11] M. Broy and M. Cengarle. “UML formal semantics: lessons learned”. In: *Software and Systems Modeling* 10 (4 2011), pp. 441–446. DOI: 10.1007/s10270-011-0207-y.

[BDR07] M. Baldauf, S. Dustdar, and F. Rosenberg. “A survey on context-aware systems”. In: *Int. J. Ad Hoc Ubiquitous Comput.* 2.4 (2007), pp. 263–277. DOI: 10.1504/IJAHUC.2007.014070.

[Bec02] K. Beck. *Test Driven Development: By Example.* Addison-Wesley Professional, 2002. ISBN: 0321146530.

[Bei90] B. Beizer. *Software Testing Techniques.* 2nd Edition. International Thomson Press, 1990. ISBN: 1850328803.

[BL02] L. Briand and Y. Labiche. “A UML-Based Approach to System Testing”. In: *Software and Systems Modeling* 1.1 (2002), pp. 10–42. DOI: 10.1007/s10270-002-0004-8.

[BM04] E. Belloni and C. Marcos. “MAM-UML: An UML Profile for the Modeling of Mobile-Agent Applications”. In: *Chilean Computer Science Society, International Conference of the.* 2004, pp. 3–13. DOI: 10.1109/QEST.2004.14.

[BME07] BME. *Robustness test suite for OpenAIS framework.* 2007. URL: http://mit.bme.hu/~micskeiz/pages/robustness_testing.html#aisrobustness.

[Bow06] J. K. F. Bowles. “Decomposing Interactions”. In: *11th International Conference on Algebraic Methodology and Software Technology (AMAST 2006).* 2006, pp. 189–203. DOI: 10.1007/11784180_16.

[BP94] G. V. Bochmann and A. Petrenko. “Protocol testing: review of methods and relevance for software testing”. In: *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis.* ISSTA ’94. 1994, pp. 109–124. DOI: 10.1145/186258.187153.

[Bro+05] M. Broy, B. Jonsson, J. Katoen, M. Leucker, and A. Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures.* Springer-Verlag, New York, Inc., 2005. ISBN: 3540262784. DOI: 10.1007/b137241.

[Bro+08] M. Broy, M. V. Cengarle, H. Grönniger, and B. Rumpe. *Modular description of a comprehensive semantics model for the UML (Version 2.0).* Tech. rep. 2008-06. Carl-Friedrich-Gaus-Fakultat, Technische Universitat Braunschweig, 2008.

[Bru+04] D. de Bruin, J. Kroon, R. van Klaveren, and M. Nelisse. “Design and test of a cooperative adaptive cruise control system”. In: *Intelligent Vehicles Symposium, 2004 IEEE.* 2004, pp. 392–396. DOI: 10.1109/IVS.2004.1336415.

[Cav+04] A. Cavalli, C. Grepet, S. Maag, and V. Tortajada. “A Validation Model for the DSR Protocol”. In: *24th International Conference on Distributed Computing Systems Workshops.* ICDCSW ’04. 2004, pp. 768–773. DOI: 10.1109/ICDCSW.2004.1284120.

[CB05] D. R. Choffnes and F. E. Bustamante. “An integrated mobility and traffic model for vehicular wireless networks”. In: *Proceedings of the 2nd ACM international workshop on Vehicular ad hoc networks.* VANET ’05. 2005, pp. 69–78. DOI: 10.1145/1080754.1080765.

[Cen07] M. V. Cengarle. “System model for UML – The interactions case”. In: *Methods for Modelling Software Systems (MMOSS).* Dagstuhl Seminar Proceedings 06351. 2007. URL: http://drops.dagstuhl.de/opus/volltexte/2007/857.

[CG00] L. Cardelli and A. D. Gordon. “Mobile ambients”. In: *Theoretical Computer Science* 240.1 (2000), pp. 177–213. DOI: `10.1016/S0304-3975(99)00231-5`.

[CGP03] A. Cavalli, C. Gervy, and S. Prokopenko. “New approaches for passive testing using an Extended Finite State Machine specification”. In: *Information and Software Technology* 45.12 (2003), pp. 837–852. DOI: `10.1016/S0950-5849(03)00063-6`.

[CGW06] M. V. Cengarle, P. Graubmann, and S. Wagner. “Semantics of UML 2.0 Interactions with Variabilities”. In: *Electr. Notes Theor. Comput. Sci.* 160 (2006), pp. 141–155.

[CK04a] A. Cavarra and J. Küster-Filipe. “Formalizing Liveness-Enriched Sequence Diagrams Using ASMs”. In: *Abstract State Machines 2004. Advances in Theory and Practice.* Vol. 3052. LNCS. 2004, pp. 62–77. DOI: `10.1007/978-3-540-24773-9_6`.

[CK04b] M. V. Cengarle and A. Knapp. “UML 2.0 Interactions: Semantics and Refinement”. In: *3rd Int Workshop on Critical Systems Development with UML (CSDUML04, Proceedings), Technical Report TUM-I0415.* 2004, pp. 85–99.

[CK05a] A. Cavarra and J. Küster-Filipe. “Combining Sequence Diagrams and OCL for Liveness”. In: *Electronic Notes in Theoretical Computer Science* 115 (2005), pp. 19–38. DOI: `10.1016/j.entcs.2004.09.025`.

[CK05b] M. V. Cengarle and A. Knapp. *Operational Semantics of UML 2.0 Interactions.* Tech. rep. TUM-I0505. Institut für Informatik, Technische Universitat München, 2005.

[CK08] M. V. Cengarle and A. Knapp. *An Institution for UML 2.0 Interactions.* Tech. rep. TUM-I0808. Technische Universität München, 2008. URL: `http://www4.in.tum.de/~cengarle/papers/TUM-I0808.pdf`.

[Coo12] S. Cook. “Looking back at UML”. In: *Software and Systems Modeling* (2012), pp. 1–10. DOI: `10.1007/s10270-012-0256-x`.

[CS04] C. Csallner and Y. Smaragdakis. “JCrasher: an automatic robustness tester for Java”. In: *Softw. Pract. Exper.* 34 (11 2004), pp. 1025–1050. DOI: `10.1002/spe.602`.

[DeM+88] R. DeMillo, D. Guindi, W. McCracken, A. Offutt, and K. King. “An extended overview of the Mothra software testing environment”. In: *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis.* 1988, pp. 142–151. DOI: `10.1109/WST.1988.5369`.

[DH01] W. Damm and D. Harel. “LSCs: Breathing Life into Message Sequence Charts”. In: *Formal Methods in System Design* 19.1 (2001), pp. 45–80. DOI: `10.1023/A:1011227529550`.

[DHC07] H. Dan, R. M. Hierons, and S. Counsell. “A Thread-tag Based Semantics for Sequence Diagrams”. In: *Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods.* 2007, pp. 173–182. DOI: `10.1109/SEFM.2007.3`.

[DM02] J. Duraes and H. Madeira. “Characterization of operating systems behavior in the presence of faulty drivers through software fault emulation”. In: *Pacific Rim International Symposium on Dependable Computing.* 2002, pp. 201–209. DOI: `10.1109/PRDC.2002.1185639`.

[Eic+05] C. Eichner, H. Fleischhack, R. Meyer, U. Schrimpf, and C. Stehno. “Compositional Semantics for UML 2.0 Sequence Diagrams Using Petri Nets”. In: *SDL 2005: Model Driven Systems Design.* 2005, pp. 133–148. DOI: `10.1007/11506843_9`.

[EL12] M. Elaasar and Y. Labiche. “Model Interchange Testing: A Process and a Case Study”. In: *Modelling Foundations and Applications.* Vol. 7349. LNCS. 2012, pp. 49–61. DOI: 10.1007/978-3-642-31491-9_6.

[Fer+07] J. M. Fernandes, S. Tjell, J. B. Jorgensen, and O. Ribeiro. “Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net”. In: *Proceedings of the Sixth International Workshop on Scenarios and State Machines.* 2007, p. 2. DOI: 10.1109/SCESM.2007.1.

[FMP05] J.-C. Fernandez, L. Mounier, and C. Pachon. “A model-based approach for robustness testing”. In: *Proceedings of the 17th IFIP TC6/WG 6.1 international conference on Testing of Communicating Systems.* TestCom’05. 2005, pp. 333–348. DOI: 10.1007/11430230_23.

[FRT10] H. Fouchal, A. Rollet, and A. Tarhini. “Robustness testing of composed real-time systems”. In: *J. Comp. Methods in Sci. and Eng.* 10 (1-2S2 2010), pp. 135–148. DOI: 10.3233/JCM-2010-0274.

[Fu+05] C. Fu, A. Milanova, B. G. Ryder, and D. G. Wonnacott. “Robustness Testing of Java Server Applications”. In: *IEEE Trans. Softw. Eng.* 31 (4 2005), pp. 292–311. DOI: 10.1109/TSE.2005.51.

[GHN93] J. Grabowski, D. Hogrefe, and R. Nahm. “Test Case Generation with Test Purpose Specification by MSCs”. In: *In: SDL’93 - Using Objects (Editors: O. Faergemand, A. Sarma), North-Holland, October 1993.* 1993.

[GMS04] V. Grassi, R. Mirandola, and A. Sabetta. “A UML Profile to Model Mobile Systems”. In: *UML 2004 - The Unified Modeling Language. Modelling Languages and Applications.* Vol. 3273. LNCS. 2004, pp. 128–142. DOI: 10.1007/978-3-540-30187-5_10.

[GS05] R. Grosu and S. A. Smolka. “Safety-Liveness Semantics for UML 2.0 Sequence Diagrams”. In: *Proceedings of the Fifth International Conference on Application of Concurrency to System Design.* 2005, pp. 6–14. DOI: 10.1109/ACSD.2005.31.

[Hal+06] H. H. Hallal, S. Boroday, A. Petrenko, and A. Ulrich. “A formal approach to property testing in causally consistent distributed traces”. In: *Form. Asp. Comput.* 18 (1 2006), pp. 63–83. DOI: 10.1007/s00165-005-0082-9.

[Ham06] Y. Hammal. “Branching Time Semantics for UML 2.0 Sequence Diagrams”. In: *Formal Techniques for Networked and Distributed Systems - FORTE 2006.* Vol. 4229. LNCS. 2006, pp. 259–274. DOI: 10.1007/11888116_20.

[Ham10] Á. Hamvas. “Using UML Sequence Diagrams for the Requirement Analysis of Mobile Distributed Systems”. MSc thesis. Budapest University of Technology and Economics (BME), 2010.

[Hau+05] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. “Why Timed Sequence Diagrams Require Three-Event Semantics”. In: *Scenarios: Models, Transformations and Tools.* 2005, pp. 1–25. DOI: 10.1007/11495628_1.

[Hes+08] A. Hessel, K. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. “Testing Real-Time Systems Using UPPAAL”. In: *Formal Methods and Testing.* Vol. 4949. LNCS. 2008, pp. 77–117. DOI: 10.1007/978-3-540-78917-8_3.

[HID09] HIDENETS. *Highly dependable ip-based networks and services.* EU FP6 Specific Targeted Research Project (STREP), IST 026979. 2009. URL: http://www.hidenets.aau.dk.

[HJR04] Q. Huang, C. Julien, and G. Roman. “Relying on safe distance to achieve strong partitionable group membership in ad hoc networks”. In: *IEEE Transactions on Mobile Computing* 3.2 (2004), pp. 192–205. DOI: 10.1109/TMC.2004.14.

[HKM07] D. Harel, A. Kleinbort, and S. Maoz. “S2A: A Compiler for Multi-modal UML Sequence Diagrams”. In: *10th International Conference on Fundamental Approaches to Software Engineering (FASE 2007)*. 2007, pp. 121–124. DOI: 10.1007/978-3-540-71289-3_11.

[HM03] D. Harel and R. Marelly. *Come, Let’s Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[HM08] D. Harel and S. Maoz. “Assert and negate revisited: Modal semantics for UML sequence diagrams”. In: *Software and Systems Modeling* 7.2 (2008), pp. 237–252. DOI: 10.1007/s10270-007-0054-z.

[Hon+01] H. S. Hong, I. Lee, O. Sokolsky, and S. D. Cha. “Automatic Test Generation from Statecharts Using Model Checking”. In: *In Proceedings of FATES’01, Workshop on Formal Approaches to Testing of Software, volume NS-01-4 of BRICS Notes Series*. 2001, pp. 15–30.

[HR04] D. Harel and B. Rumpe. “Meaningful modeling: what’s the semantics of "semantics"?” In: *Computer* 37.10 (2004), pp. 64–72. DOI: 10.1109/MC.2004.172.

[HS03] Ø. Haugen and K. Stølen. “STAIRS - Steps To Analyze Interactions with Refinement Semantics”. In: *The Unified Modeling Language. Modeling Languages and Applications*. Vol. 2863. LNCS. 2003, pp. 388–402. DOI: 10.1007/978-3-540-45221-8_33.

[HW08] G. Huszerl and H. Waeselynck, eds. *Refined design and testing framework, methodology and application results*. HIDENETS project deliverable D5.3. 2008, pp. 1–118. URL: http://www.hidenets.aau.dk/.

[IEE10] Institute of Electrical and Electronics Engineers. *Systems and software engineering – Vocabulary*. Standard 24765:2010. 2010, pp. 1–418. DOI: 10.1109/IEEESTD.2010.5733835.

[IEE11] Institute of Electrical and Electronics Engineers. *IEEE Standard for Automatic Test Markup Language (ATML) for Exchanging Automatic Test Equipment and Test Information via XML*. Standard 1671-2010. 2011, pp. 1–388. DOI: 10.1109/IEEESTD.2011.5706290.

[IST10] International Software Testing Qualifications Board. *Standard glossary of terms used in Software Testing*. Version 2.1. 2010. URL: http://istqb.org/display/ISTQB/Downloads.

[ITU07] International Telecommunication Union. *Testing and Test Control Notation version 3: TTCN-3 core language*. Recommendation Z.161. 2007. URL: http://www.itu.int/rec/T-REC-Z.161.

[ITU11] International Telecommunication Union. *Message Sequence Chart (MSC)*. Recommendation Z.120. 2011. URL: http://www.itu.int/rec/T-REC-Z.120.

[JJ05] C. Jard and T. Jéron. “TGV:, theory, principles and algorithms”. In: *International Journal on Software Tools for Technology Transfer (STTT)* 7.4 (2005), pp. 297–315. DOI: 10.1007/s10009-004-0153-x.

[Kan+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-021. Carnegie-Mellon University Software Engineering Institute, 1990.

[KD00] P. Koopman and J. DeVale. "The exception handling effectiveness of POSIX operating systems". In: *Software Engineering, IEEE Transactions on* 26.9 (2000), pp. 837–848. DOI: 10.1109/32.877845.

[KDD08] P. Koopman, K. Devale, and J. Devale. "Interface Robustness Testing: Experience and Lessons Learned from the Ballista Project". In: *Dependability Benchmarking for Computer Systems.* Ed. by K. Kanoun and L. Spainhower. John Wiley & Sons, Inc., 2008, pp. 201–226. ISBN: 9780470370506. DOI: 10.1002/9780470370506.ch11.

[Klo03] J. Klose. "Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior". PhD thesis. Carl von Ossietzky Universitat Oldenburg, 2003.

[Koc+98] B. Koch, J. Grabowski, D. Hogrefe, and M. Schmitt. "Autolink-a tool for automatic test generation from SDL, specifications". In: *Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE, Workshop on.* 1998, pp. 114–125. DOI: 10 . 1109 / WIFT.1998.766305.

[KS08] K. Kanoun and L. Spainhower, eds. *Dependability Benchmarking for Computer Systems.* Wiley-IEEE Computer Society Press, 2008. ISBN: 9780470230558. DOI: 10 . 1002 / 9780470370506.

[KSH07] H. Kugler, M. J. Stern, and E. J. A. Hubbard. "Testing scenario-based models". In: *Proceedings of the 10th international conference on Fundamental approaches to software engineering.* (Braga, Portugal). FASE'07. 2007, pp. 306–320. DOI: 10.1007/978-3-540-71289-3_24.

[Küs06] J. Küster-Filipe. "Modelling Concurrent Interactions". In: *Theoretical Computer Science* 351.2 (2006), pp. 203–220. DOI: 10.1016/j.tcs.2005.09.068.

[KW07] A. Knapp and J. Wuttke. "Model Checking of UML 2.0 Interactions". In: *Models in Software Engineering.* Vol. 4364. LNCS. 2007, pp. 42–51. DOI: 10.1007/978-3-540-69489-2_6.

[Lei+10] B. Lei, X. Li, Z. Liu, C. Morisset, and V. Stolz. "Robustness testing for software components". In: *Science of Computer Programming* 75.10 (2010), pp. 879–897. DOI: 10.1016/j.scico.2010.02.005.

[LNY04] K. R. P. H. Leung, J. K.-Y. Ng, and W. L. Yeung. "Embedded Program Testing in Untestable Mobile Environment: Embedded Program Testing in Untestable Mobile Environment". In: *Proceedings of the 11th Asia-Pacific Software Engineering Conference.* APSEC '04. 2004, pp. 430–437. DOI: 10.1109/APSEC.2004.48.

[LRS11] M. Lund, A. Refsdal, and K. Stølen. "Semantics of UML Models for Dynamic Behavior". In: *Model-Based Engineering of Embedded Real-Time Systems.* Vol. 6100. LNCS. 2011, pp. 77–103. DOI: 10.1007/978-3-642-16277-0_4.

[LS06] M. Lund and K. Stølen. "A Fully General Operational Semantics for UML 2.0 Sequence Diagrams with Potential and Mandatory Choice". In: *FM 2006: Formal Methods.* 2006, pp. 380–395. DOI: 10.1007/11813040_26.

[Lun08] M. S. Lund. "Operational analysis of sequence diagram specifications". PhD thesis. University of Oslo, 2008.

[Man+06] R. Mangharam, D. Weller, R. Rajkumar, P. Mudalige, and F. Bai. “GrooveNet: A Hybrid Simulator for Vehicle-to-Vehicle Networks”. In: *Mobile and Ubiquitous Systems - Workshops, 2006. 3rd Annual International Conference on.* 2006, pp. 1–8. DOI: 10.1109/MOBIQW.2006.361773.

[Mat+09] F. Mattiello-Francisco, E. Martins, A. Corsetti, A. Cavalli, and E. Yano. “Extended interoperability models for timed system robustness testing”. In: *Communications, 2009. LATINCOM ’09. IEEE Latin-American Conference on.* 2009, pp. 1–6. DOI: 10.1109/LATINCOM.2009.5304903.

[MCM07] B. P. Miller, G. Cooksey, and F. Moore. “An empirical study of the robustness of MacOS applications using random testing”. In: *SIGOPS Oper. Syst. Rev.* 41.1 (2007), pp. 78–86. DOI: 10.1145/1228291.1228308.

[MD04] R. Morla and N. Davies. “Evaluating a location-based application: a hybrid test and simulation environment”. In: *Pervasive Computing, IEEE* 3.3 (2004), pp. 48–56. DOI: 10.1109/MPRV.2004.1321028.

[MFS90] B. P. Miller, L. Fredriksen, and B. So. “An empirical study of the reliability of UNIX utilities”. In: *Commun. ACM* 33.12 (1990), pp. 32–44. DOI: 10.1145/96267.96279.

[MGR05] A. J. Mooij, N. Goga, and J. M. Romijn. “Non-local Choice and Beyond: Intricacies of MSC Choice Nodes”. In: *Fundamental Approaches to Software Engineering.* 2005, pp. 273–288. DOI: 10.1007/978-3-540-31984-9_21.

[MP05] A. Muscholl and D. Peled. “Deciding Properties of Message Sequence Charts”. In: *Scenarios: Models, Transformations and Tools.* 2005, pp. 43–65. DOI: 10.1007/11495628_3.

[MS04] G. J. Myers and C. Sandler. *The Art of Software Testing.* John Wiley & Sons, 2004. ISBN: 0471469122.

[Net+07] A. C. D. Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. “A survey on model-based testing approaches: a systematic review”. In: *1st International Workshop on Empirical assessment of software engineering languages and technologies.* 2007, pp. 31–36. DOI: 10.1145/1353673.1353681.

[Ngu09] M. D. Nguyen. “Méthodologie de test de systèmes mobiles : Une approche basée sur les sénarios”. PhD thesis. Université Paul Sabatier - Toulouse III, 2009.

[NV05] F. Ngani Noudem and C. Viho. “Modeling, verifying and testing the mobility management in the mobile IPv6 protocol”. In: *8th Int. Conf. on Telecommunications (ConTEL 2005).* 2005, pp. 619–626. DOI: 10.1109/CONTEL.2005.185970.

[NWR10] M. D. Nguyen, H. Waeselynck, and N. Riviere. “GraphSeq: A Graph Matching Tool for the Extraction of Mobility Patterns”. In: *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on.* 2010, pp. 195–204. DOI: 10.1109/ICST.2010.53.

[OM09] J. Olah and I. Majzik. “A Model Based Framework for Specifying and Executing Fault Injection Experiments”. In: *Proceedings of the Fourth International Conference on Dependability of Computer Systems.* DEPCOS-RELCOMEX ’09. 2009, pp. 107–114. DOI: 10.1109/DepCoS-RELCOMEX.2009.41.

[OMG05] Object Management Group. *UML Testing Profile v1.0 (U2TP).* 2005. URL: http://www.omg.org/technology/documents/formal/test_profile.htm.

[OMG11a] Object Management Group. *Unified Modeling Language (UML) 2.4.1 Infrastructure Specification.* formal/2011-08-05. 2011.

[OMG11b] Object Management Group. *Unified Modeling Language (UML) 2.4.1 Superstructure Specification.* formal/2011-08-06. 2011.

[Pic03] S. Pickin. "Test des composants logiciels pour les télécommunications". PhD thesis. Université de Rennes, 2003.

[Pin+05] G. Pintér, H. Madeira, M. Vieira, I. Majzik, and A. Pataricza. "A data mining approach to identify key factors in dependability experiments". In: *Proceedings of the 5th European conference on Dependable Computing.* EDCC'05. 2005, pp. 263–280. DOI: 10.1007/11408901_20.

[Pió+08] M. Piórkowski, M. Raya, A. L. Lugo, P. Papadimitratos, M. Grossglauser, and J.-P. Hubaux. "TraNS: realistic joint traffic and network simulator for VANETs". In: *SIGMOBILE Mob. Comput. Commun. Rev.* 12.1 (2008), pp. 31–33. DOI: 10.1145/1374512.1374522.

[PJ04] S. Pickin and J.-M. Jézéquel. "Using UML Sequence Diagrams as the Basis for a Formal Test Description Language". In: *Integrated Formal Methods.* Vol. 2999. LNCS. 2004, pp. 481–500. DOI: 10.1007/978-3-540-24756-2_26.

[PK07] M. Popovic and J. Kovacevic. "A Statistical Approach to Model-Based Robustness Testing". In: *International Conference and Workshops on the Engineering of Computer-Based Systems.* ECBS '07. 2007, pp. 485–494. DOI: 10.1109/ECBS.2007.13.

[R3C11] R3-COP. *Resilient Reasoning Robotic Co-operating Systems.* ARTEMIS research project nr. 100233. 2011. URL: http://www.r3-cop.eu/.

[ReS09] ReSIST. *Resilience for survivability in IST.* EU FP6 Network of Excellence, IST 026764. 2009. URL: http://www.resist-noe.org/.

[RH09] P. Runeson and M. Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Softw. Engg.* 14.2 (2009), pp. 131–164. DOI: 10.1007/s10664-008-9102-8.

[RHS05a] R. K. Runde, Ø. Haugen, and K. Stølen. "How to transform UML neg into a useful construct". In: *Norsk Informatikkonferanse (NIK'05).* 2005, pp. 55–66.

[RHS05b] R. K. Runde, Ø. Haugen, and K. Stølen. "Refining UML interactions with underspecification and nondeterminism". In: *Nordic J. of Computing* 12.2 (2005), pp. 157–188.

[Run07] R. Runde. "STAIRS – understanding and developing specifications expressed as UML interaction diagrams". PhD thesis. University of Oslo, 2007.

[SAF06] SAF Test. *An open source repository for SAF-conformance test suites.* 2006. URL: http://saftest.sourceforge.net/.

[SAF07] Service Availability Forum (SA Forum). *Application Interface Specification (AIS).* 2007. URL: http://www.saforum.org.

[Sch+05] C. Schroth, F. Dötzer, T. Kosch, B. Ostermaier, and M. Strassberger. "Simulating the traffic effects of vehicle-to-vehicle messaging systems". In: *5th International Conference on ITS Telecommunications (ITS-T).* (Brest, France). 2005.

[SDM12] Software Development Laboratory. *srcML: A document-oriented XML representation of source code.* 2012. URL: http://www.sdml.info/projects/srcml/.

[SE04] K. Saleh and C. El-Morr. "M-UML: an extension to UML for the modeling of mobile agent-based software systems". In: *Information and Software Technology* 46.4 (2004), pp. 219–227. DOI: 10.1016/j.infsof.2003.07.004.

[Sel04] B. Selic. “On the Semantic Foundations of Standard UML 2.0”. In: *Formal Methods for the Design of Real-Time Systems.* Vol. 3185. LNCS. 2004, pp. 75–76. DOI: `10.1007/978-3-540-30080-9_6`.

[SRC07] F. Saad-Khorchef, A. Rollet, and R. Castanet. “A framework and a tool for robustness testing of communicating software”. In: *Proceedings of the 2007 ACM symposium on Applied computing.* SAC ’07. 2007, pp. 1461–1466. DOI: `10.1145/1244002.1244315`.

[SS07] I. Solheim and K. Stølen. *Technology research explained.* Tech. rep. SINTEF A313. SINTEF ICT, 2007.

[Stö03a] H. Störrle. “Assert, Negate and Refinement in UML-2 Interactions”. In: *Workshop on Critical Systems Development with UML (CSDUML’03), Technical Report TUM-I0317.* 2003, pp. 79–94.

[Stö03b] H. Störrle. “Semantics of Interactions in UML 2.0”. In: *IEEE Symposium on Human Centric Computing Languages and Environments.* 2003, pp. 129–136. DOI: `10.1109/HCC.2003.1260216`.

[Stö04] H. Störrle. *Trace Semantics of Interactions in UML 2.0.* Tech. rep. Institut für Informatik, Ludwig-Maximilians-Universität München, 2004.

[SVN08a] H. Shen, A. Virani, and J. Niu. “Formalize UML 2 Sequence Diagrams”. In: *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE.* 2008, pp. 437–440. DOI: `10.1109/HASE.2008.51`.

[SVN08b] H. Shen, A. Virani, and J. Niu. *Formalize UML 2 Sequence Diagrams.* Tech. rep. CS-TR-2008-13. University of Texas at San Antonio, 2008.

[Tam09] F. Tam. “Service Availability Standards for Carrier-Grade Platforms: Creation and Deployment in Mobile Networks”. Tampere University of Technology, 2009. ISBN: 978-952-15-2134-8. URL: `http://URN.fi/URN:NBN:fi:tty-200904281053`.

[TDM08] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for software security testing and quality assurance.* Artech House, 2008. ISBN: 1596932147.

[TIJ96] T. Tsai, R. Iyer, and D. Jewitt. “An approach towards benchmarking of fault-tolerant commercial systems”. In: *Proceedings of Annual Symposium on Fault Tolerant Computing.* 1996, pp. 314–323. DOI: `10.1109/FTCS.1996.534616`.

[TT12] M. Toeroe and F. Tam, eds. *Service Availability: Principles and Practice.* John Wiley & Sons, 2012. ISBN: 978-1-1199-5408-8.

[UL06] M. Utting and B. Legeard. *Practical Model-Based, Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., 2006. ISBN: 0123725011.

[Woh+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell. *Experimentation in Software Engineering.* Springer, 2012. ISBN: 978-3-642-29043-5. DOI: `10.1007/978-3-642-29044-2`.

[WS08] S. Weißleder and B.-H. Schlingloff. “Models in Software Engineering”. In: 2008. Chap. Deriving Input Partitions from UML Models for Automatic Test Generation, pp. 151–163. DOI: `10.1007/978-3-540-69073-3_17`.