TALLINN UNIVERSITY OF TECHNOLOGY
DOCTORAL THESIS
64/2018

Aspect-Oriented Model-Based Testing

KÜLLI SARNA

TTÜ PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

This dissertation was accepted for the defence of the degree of Philosophy in Computer Science on September 28, 2018.

Defence of the thesis: November 15, 2018, Tallinn

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been submitted for any academic degree.

Külli Sarna

European Union
European Social Fund

Investing in your future

ISSN 2585-6898 (publication)
ISBN 978-9949-83-343-6 (publication)
ISSN 2585-6901 (PDF)
ISBN 978-9949-83-344-3 (PDF)

TALLINNA TEHNIKAÜLIKOOL
DOKTORITÖÖ
64/2018

Aspekt-orienteeritud mudeli-põhine testimine

KÜLLI SARNA

TTÜ KIRJASTUS

Table of Contents

FOREWORD

Since starting my work career in the telecommunications industry more than 20 years ago, I have been looking for ways to apply tool supported development techniques. At a time when the need for professional software testing was gathering recognition and testers were often not participating in development teams yet, I was working with databases and correcting software errors in database systems. This experience guided my understanding and realization of how important systematic software development is in preventing errors. This triggered my interest towards deeper understanding of why software errors occur. I have experienced various roles in the software development process including that of test engineer, programmer, analyst, and QA manager; and have discovered how these roles contribute to the quality of a software product. My journey has been driven by the curiosity of what improvements new software development techniques can provide to the end product quality.

Since the early days of software testing, its role in development has been increasing and progressively studied by researchers. Nowadays, no one questions the fact that software testing is a vital part of the development process and must be integrated with it from the very beginning. Unfortunately practical needs often forestall the theory. The faster the software development process is, the less time there is left for testing. There is a common belief that test automation using model-based testing should provide the lever to cut time without compromising on software quality, but an important prerequisite for efficient testing is an elaborated design of the system under test, called *design for testability*. Specifically, the design environment must address also the test requirements in addition to regular design requirements.

During my PhD studies, I realized that choosing the correct software development methodology before beginning the process is essential. That is the methodology that supports both requirements engineering as well as testing, especially for the initial stages of software development. Ignoring it is the same as taking a road trip without planning the route.

My four years in the medical software industry culminated in realizing the importance of a rigorous testing discipline. Regardless of the growing popularity of the new service and micro service oriented design paradigms, the need for test automation and methodologies has not diminished. On the contrary, a hard design problem is more effectively solved when decomposed into a set of smaller concerns meaning a single monolithic design representation is divided into smaller ones by following a “divide-and-conquer” approach. An aspect-oriented (AO) approach to development and testing, advocated in this thesis, is one such structuring principle besides the agent-, object-, and component-oriented approaches. The thesis intend to demonstrate that AO testing realized via Uppaal Timed Automata formalism is practical and improves the quality of testing, especially in the domain of cyber-physical systems where the heterogeneous design units combine various design concerns.

LIST OF PUBLICATIONS

This thesis includes results in four original publications written between 2011 and 2014. These publications are listed as follows:

A Külli Sarna, Jüri Vain. Exploiting aspects in model-based testing. In *FOAL'12: Proceedings of the Eleventh Workshop on Foundations of Aspect-Oriented Languages, March 26, 2012, Potsdam, Germany:* New York: ACM, 45 - 47.

B Alar Kuusik, Enar Reilent, Külli Sarna, Marko Parve. Home telecare and rehabilitation system with aspect-oriented functional integration. In *Biomedical Engineering/ Biomedizinische Technik: The 46th annual conference of the German Society for Biomedical Engineering, Jena, Germany, September 17-19, 2012. (Edit.) Dössel, O. De Gruyter*, 1004 - 1007.

C Külli Sarna, Alar Kuusik, Enar Reilent. Home Rehabilitation System Supported by the Safety Model. Studies in health technology and informatics, 2013, 189, 145 - 151.

D Külli Sarna, Jüri Vain. Aspect-oriented testing of a rehabilitation system. In *VALID 2014: The Sixth International Conference on Advances in System Testing and Validation Lifecycle, October 12 - 16, 2014, Nice, France: (Edit.) Kanstrén, Teemu; Helle, Philipp.* Venice: IARIA, 73 - 78.

AUTHOR’S CONTRIBUTION TO THE PUBLICATIONS

Contribution to the publications in this thesis are:

In all papers except Publication B the author was the main contributor – a paper writer. My supervisor was also involved with the initial idea throughout my PhD work. Without his guidance the writing of articles would not have been possible.

A The author’s contribution was the implementation of the original idea that came from my supervisor to use aspect-oriented concepts in model-based testing. This refinement-based aspect-oriented modelling approach was fundamental for the subsequent research and the following publications.

B In Publication B the author’s contribution was the initial idea of using aspect-oriented requirements engineering principles in home telecare system development. The author identified the need for knowledge engineering. Also ideas of how to divide a system into aspects and how to use system analyses prior to implementation. The paper was written by a system implementer as he needed to know the requirements which must be satisfied. I took part in the discussions and implementation based on my aspect-oriented analysis.

C For publication C the author was the main contributor as she designed the modelling and verification techniques being used in home rehabilitation system testing. My idea involved aspect-oriented model-based technique which allows adding incrementally new design aspects along improving the tele-rehabilitation system quality.

D In publication D the author presents a model-based testing approach to support automated test generation with aspect-oriented concepts. This includes test model building using aspect-oriented constructions that make it possible to generate a set of test cases according to an aspect related coverage criteria.

Abbreviations

AC	Aspect Coverage
ALM	Application Lifecycle Management
AO	Aspect-Oriented
AOM	Aspect-Oriented Modelling
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
AOT	Aspect-Oriented Testing
APC	Aspect Path Coverage
BPM	Business Process Modelling
BT	Bench Testing
CE	Coverage Expression
CPS	Cyber-Physical System
ETSI	European Telecommunication Standards Institute
HRS	“Home rehabilitation System” - Telecare system
HSA	Heterogeneous System Architecture
IEEE	Institute of Electrical and Electronics Engineers
IOCO	Input-Output Conformance
JPC	Join Point Coverage
LTL	Linear Temporal Logic
LTS	Labelled Transition System
MB	Model-Based
MBT	Model-Based Testing
MEC	Model Element Coverage
PCD	Provably correct development
RMT	Requirements Management Tool
SAC	Strong Aspect Coverage
SDL	Specification and Description Language
SJPC	Strong Joint Point Coverage
SAPC	Strong Aspect Path Coverage

SMEC	Strong Model Element Coverage
SpO2	The percentage of haemoglobin in the blood that is saturated with oxygen
SUT	System Under Test
SysML	Systems Modelling Language
TCTL	Timed Computation Tree Logic
UML	Unified Modelling Language
Uppaal TA	Uppaal Timed Automata
V&V	Verification and Validation
WAC	Weak Aspect Coverage
WAPC	Weak Aspect Path Coverage
WJPC	Weak Join Point Coverage
Z	The Z notation – a formal specification language for describing and computing systems

Terms

Bench testing	In the context of software or firmware or hardware engineering, a test bench is an environment in which the product under development is tested with the aid of software and hardware tools.
Cyber-Physical Systems	CPS are integrations of computation, networking, and physical processes. Embedded computers and networks monitor and control the physical processes, with feedback loops where physical processes affect computations and vice versa.
Heterogeneous (Hardware) System Architecture	HSA is a cross-vendor set of components that allows for the integration of central processing units on the same bus, with shared memory and tasks.
Provably correct development	PCD has a precise (mathematical) specification for software that provably (in machine check-able way) fulfils the software requirements.
System Under Test	SUT is a system being tested. SUT can be a full system of software-controlled hardware, a system of hardware alone, or even a system of SUTs. A SUT can also be a single software unit, or a collection of software units.

List of Figures

List of Tables

1 INTRODUCTION

1.1 Chapter overview

This chapter introduces the general context of the thesis. That is the development of aspect-oriented models for model-based testing of cyber-physical systems. It summarizes the methodology issues, the motivation, the research context and contribution of the thesis, and also describes how thesis is organized.

1.2 The Role of Software testing

Software development processes regardless of the specifics of the underlying process model – Waterfall [6], V-model [7], Spiral [8], Agile [9], etc. – involve activities such as requirements analysis resulting in a software requirements specification, software design, implementation, verification and validation, integration, deployment (or installation), and maintenance.

According to the standard IEEE Std 1012-2012 testing is considered to be part of the software verification and validation (V&V) processes. While *verification* focuses on evaluating whether the software matches its specification, the *validation* focuses on assessing that the specification matches the customer's requirements. As stated in [10] software testing constitutes up to 50 percent (or even more in mission critical applications) of the total development costs of software. Authors of [11] report that the root reasons of 56 percent of all defects identified in software projects are introduced in the requirements phase. They profess that low software quality is mainly due to the problematical test coverage and incorrect requirements. In addition, 50 percent of incorrect requirements are caused by incomplete specification and another 50 percent by unclear and ambiguous requirements. From the above it follows that any increase in productivity of testing processes has a strong impact on the productivity of the whole development process.

Another factor that increases the importance of V&V in the development processes is the change of nature of software to be developed. While modern software becomes more complex it requires new sophisticated testing technologies. In the process of developing complex networked systems such as Cyber-Physical Systems (CPS) or banking systems the problems of inherent concurrency over the wide spectrum of services and heterogeneous architectures needs to be addressed. The heterogeneous components introduce functional, timing, safety, performance, and security features on multiple scales. In particular, in multi-critical (e.g. safety-, security-, time-critical) applications the networking of feature rich components needs to be paired with the predictability of the system's emerging behaviour to guarantee the required quality of service. This is almost impossible to achieve without design validation methods that are relevant and scalable enough to capture the product's usability features in their entirety.

While the features of functionality have gained major attention in traditional software development approaches, achieving the predictable timing of critical services in the presence of heterogeneous and evolving distributed architectures still remains a challenge [12]. Therefore, the traditional validation methods like bench testing or encasing alone, although helpful and widely used, have become inadequate for CPS systems. As stated in [13] software quality and software process productivity issues can be mitigated with model-based techniques and tools that operate on a higher level of abstraction than typical engineering approaches.

MBT, as one group of model-based techniques, provides the opportunities for test automation and thus reducing software testing effort [14]. MBT suggests the use of abstract models for specifying the expected behaviour of the SUT and automatically generating tests from these models. According to the testing taxonomy depicted in Figure 1.1 [15] MBT captures typically the full *Level-Accessibility* plane and extends through all categories along *Aspect* dimension. MBT advantages expose most clearly in Integration and System level testing.

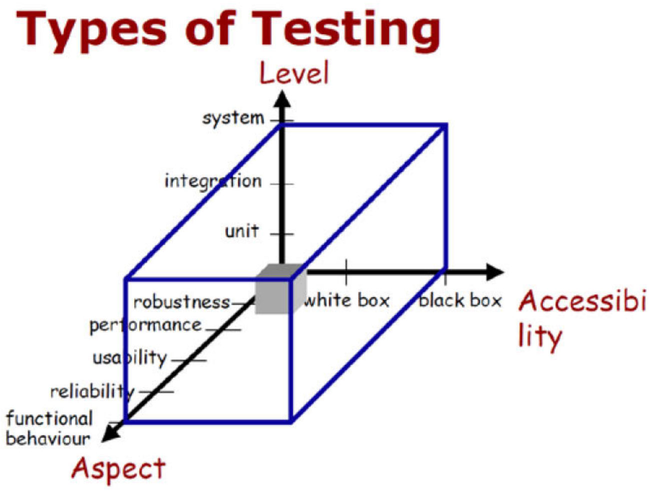


Figure 1.1 Taxonomy of testing types [15]

From the test generation-execution point of view, MBT makes use of the models to generate tests either in offline or online mode. The online testing methods differ in how the test purpose is defined and how the test stimuli are selected on-the-fly. Also online test execution requires more run-time resources for interpreting the model. In offline testing, it is required to explore the whole state space of the model of SUT prior to the tests being generated. In online testing, the decisions about the next test actions are made by observing the current output of the SUT.

Regardless the capability of abstraction the key issues with model based methods is still their *scalability*. The test models of real-world problems and systems rapidly grow to such an extent that managing the complexity without using relevant modularization techniques becomes impractical. To introduce modularity the test models can be structured using different criteria: architectural entities, e.g. UML objects and classes; functionality guided use cases, e.g. UML methods; specification refinement [16]; aspect-oriented mechanisms; design viewpoints [12] and other techniques. In all cases the modularization attempts have been driven by the need to improve the comprehension of models and to reduce the complexity of test generation and getting the test cases to a manageable size, both time-wise and computationally.

Referring to [17], subsystems should have a maximum cohesion and a minimum coupling wherever possible. *Cohesion* measures the dependence among classes, e.g. high cohesion means the classes in the subsystem perform similar tasks and are related to each other (via associations), low cohesion means lots of miscellaneous and auxiliary classes, and no associations. *Coupling* measures dependencies between subsystems. In the case of high coupling, changes to one subsystem will have a high impact on the other subsystem (change of model, massive recompilation, etc.); in the case of low coupling, a change in one subsystem does not affect any other subsystem.

In this thesis we focus on the model-based *conformance testing* where the SUT is considered as a “black-box”, i.e. only its inputs and outputs are assumed to be externally controllable and observable respectively. The internal behaviour of the system is abstracted away. The aim of black-box conformance testing, according to [14], is to check whether the behaviour observable on the system interfaces conforms to that given in the system requirements specification. During MBT a tester executes selected test cases (extracted from the system requirements model) by running SUT in the test harness and emits a test verdict (*pass*, *fail, inconclusive*). The verdict shows test results in the sense of a conformance relation between SUT and the requirements model. A “standard” conformance relation used most often in MBT is Input-Output Conformance (IOCO) introduced by Tretmans [18]. The behaviour of an IOCO-correct implementation should respect, after some observations, the following restrictions:

- the outputs produced by SUT should be the same as allowed in the requirements model;
- if a quiescent state (a situation where the system cannot evolve without an input from the environment) is reached in SUT, this should also be the case in the requirements model; and
- any time an input is possible in the requirements model, this should also be the case in SUT.

Following the general goal of MBT, the aim of the thesis is to develop a method for deriving tests from formal specifications so that the tests are well-targeted towards achieving the test purpose, i.e. providing measurable coverage in terms of test items. This allows defining the relative completeness of the tests, e.g. covered states of the test model. Also, the derived tests should be cost/time efficient, i.e. performing with possibly a low need of resources. The derived tests should be *correct*, which means that they should not detect errors in correct implementations. The derived tests should be *meaningful*, erroneous implementations should be detected with a high certainty [17]. To address the problems of complexity and traceability in testing the thesis extends the model-based conformance testing with the concepts of *Aspect-Oriented Modelling* (AOM) and elaborates the test coverage criteria that together form the theoretical basis – for Aspect-Oriented Testing (AOT).

More specific goal of the thesis is to introduce the principles of AOM in terms of Uppaal Timed Automata (Uppaal TA) and to define a method for constructive development of well-structured models and test purpose specifications by referring to the attributes of aspect models symbolically. The Uppaal TA that are proposed for AO modelling, and test development, support the specification of not only functional but also timing features of SUT. The theoretical results of the thesis are validated using a practical case study "Home rehabilitation System" (HRS). The quantitative evidence of the advantages provided by the method are exposed using practical measurements of work put into the test development as well as analytical reasoning on the complexity of modelling and test generation processes.

The motivation of the dissertation is based on the following research questions:

- How the existing aspect-oriented requirements engineering methodology can be applied for aspect-oriented test model construction?
- What model transformations are needed for constructing AO test models?
- How to specify the test purpose and test coverage criteria on AO test models?
- How to express the AO test purposes symbolically?
- Given an AO SUT model and AO test purpose how to verify the correctness and feasibility of the testing tasks?
- How AO MBT improves the productivity of the overall testing process?

1.3 The role of modelling in model-based testing

In model-based software engineering the development processes are based on system models. These models describe a system from different viewpoints and on different levels of detail. Due to the multitude of viewpoint models their composition is inevitable in the system integration phase where the models need to be checked for *consistency* and *integrity* of crucial design aspects.

Model-based testing is a formal, systematic verification method to validate systems design by generating *test cases* from the system models. A collection of test cases used to show that SUT has a specified set of behaviours is called a *test suite*. The creation of a test suite is directed by a pre-defined set of *coverage criteria*. Typically the system development process includes modifications and updates of design requirements. Therefore, the initially created model and test harness need to be modified incrementally to match these implementation updates. Keeping the specification and implementation increments synchronised is the main goal of continuous integration methodologies [5]. Such an incremental software development allows one to initiate the testing of implemented features as soon as these updates are inserted into the earlier version.

Early discovery and correction of design faults is one of the main factors of reducing the development costs: catching errors in models is significantly cheaper than finding them in the final system or even in a prototype implementation. Successful designs rely on the separation of concerns based on time scales, interface protocols, imposition of constraints, and other mechanisms to facilitate a decomposition of the design problem into manageable and tractable sub problems [23].

As stated in [24] MBT is relevant in the field of cyber-physical systems where the complexity of interactions cannot be addressed properly without test suites generated and executed automatically.

Test models. MBT relies on formal models. The models are built from the requirements of the system in order to describe the expected behaviours. The model can be presented, for example, in terms of the input sequences accepted by the system, the actions, and the outputs performed by the system. Since the model is a description of the application behaviour, the model should be understandable by testing people who decide on test goals. Moreover, the model should be precise, clear, and should be presented in a formal way for consistency and feasibility checks.

Another main purpose of using models in MBT is that the model of the SUT is used to retrieve a test suite consisting of a set of test cases. The test cases are selected by means of a test case specification. The standard ETSI ES 202 951 v1.1.1 (2011-07) “Requirements for Modelling Notations” is used to define characteristics of MBT [37]. These characteristics concern main phases of the MBT process: SUT and its environment modelling, test purpose specification that defines test coverage criteria, test generation and test execution. An example of MBT process is depicted in Figure 1.2.

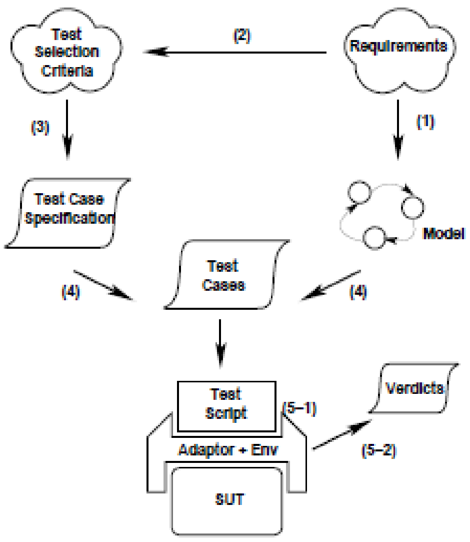


Figure 1.2 Model-based testing process [14]

Modelling phase. In the modelling phase the requirements to SUT can be represented under three different perspectives:

- modelling the *data input* to the SUT (data model);
- modelling the SUT based on the *interactions with a potential user* (tester model); and
- modelling the *dynamic behaviour* of the system (design) itself.

A wide range of modelling languages, such as UML, SDL, Z, various state machines and logics, each with their own notations, semantics, and pragmatics, have established a niche for themselves [25, 26, 27, 28, 29, 30].

By the rigour of formal semantics the models used in testing can be classified into formal, semi-formal and informal models. The models with strict formal semantics provide certainty that if the models represent systems correctly all properties verified really hold. However, formal models have some inherent limitations for MBT, in particular, their usage for test generation does not scale to large systems. Due to the high effort required their usage is typically limited to critical software domains such as automotive, medical, military, and critical infrastructure systems. The general purpose software industry typically uses semi-formal modelling languages such as Unified Modelling Language (UML), Systems Modelling Language (SysML) [32], and others which are very expressive and intuitive to designers. Regardless of the lack of complete formal semantics they are preferred for their elaborate graphical representations and tool support. Informal models are used to communicate the main ideas but they lack clear semantics and are not suitable for the development of critical systems.

MBT techniques are oriented mostly towards black-box testing [31]. Therefore, the models used in testing have to be relevant for describing interactions between SUT and its environment. These interactions are represented as sequences of test inputs to SUT, and expected SUT outputs which are reactions to these inputs. The *de facto* standard modelling language is UML. Regardless of the wide usage of UML, a considerable amount of testing theory related research has been conducted on formal models, in particular on different classes of state machines. For an exhaustive survey we refer to [33].

Test coverage criteria. An important feature that the test modelling formalisms are selected by is their relevance for representing test coverage criteria. The modelling notations suggest the kinds of *structural coverage* criteria. For example, with pre-post condition notations, cause-effect coverage or disjunctive normal form coverage of the post-condition are the common coverage criteria, while for algebraic model notations, the coverage of the axioms is an obvious coverage criteria [14]. Examples of coverage criteria commonly used in state modes are *all states*, *all transitions*, *all transition-pairs*, and *all cycles*. More complex criteria such as *all paths*, *branching condition coverage* etc. are aggregates of simpler ones. In general, a structural coverage criterion refers to a set of structural items, which are called *coverage items*.

Although the structural coverage items refer to the structural elements of the SUT (either program or its models) they can be used at the same time also for measuring the behavioural coverage since the structural coverage items specify the behaviours that ensure their coverage. In that sense strict separation of structural and behavioural coverage criteria is not possible.

The problem of generating a test case for a coverage criterion can be treated as a reachability problem. These criteria are based on the specification of control flows represented by the model in which the bugs may be exposed. Authors of [34] emphasize that extracting test selection criteria from models has been inspired by the well-researched field of code coverage. In white-box testing the code coverage criteria are used for measuring the sufficiency of a test suite and deciding when to stop testing. In black-box testing these two ideas are applied to models of externally observable SUT behaviour. The model coverage criteria are applied to measure the adequacy of the test suite.

While the structural elements of the formal model used in MBT constitute the lowest layer of coverage items, superimposing some structuring principle (modules, aspects views, etc.) upon such models generates 2nd order coverage items in the model. The pragmatic viewpoints inspired by such model structures introduce the possibility of defining coverage items which may be aggregates of the elementary items. The model modularization related aggregates of coverage items are often closer to user domain related notions than ground level model elements and provide better traceability back to the terms understandable to domain engineers (this idea will be elaborated further in Chapter 4).

Test case generation. Given a model of the SUT and the test case specification as an environment model, usually with some additional constraints, the test cases can be generated by using graph algorithms, model checking, property checking, symbolic execution, or deductive theorem proving. Random generation of tests is done by sampling the input space of a system. In the case of reactive systems, finite traces can be selected randomly by sampling the input space and applying these inputs to the model of the SUT in order to infer the expected output from model. A random walk through the model may result in test suites with different characteristics. Random walks can also be performed on usage models, and obviously, this results in certain transition probabilities for the SUT [38]. In this dissertation we apply either random walk or (coverage) property satisfying test sequences. As for our contribution we extract those coverage criteria from AO Uppaal TA models and generate test sequences by Timed Computation Tree Logic (TCTL) model checking.

Offline – Online test generation. MBT can be applied for both off-line and online generation of test cases. In offline testing the test suites are generated before running the tests and execution is a separate step. It is possible to create a tool chain: modeller, test generator and test executor. Offline test generation typically presumes extensive state space exploration and computations to generate test input data to achieve required coverage.

In the case of online testing, the test generation procedure derives only one test input at a time from the model and feeds it immediately to the SUT as opposed to deriving a complete test case in advance as in off-line testing. In online testing, it is not required to explore the whole state space of the model of the SUT every time the test stimulus is generated. Instead, the decisions about the next actions are made by observing the current output of the SUT. However, online test execution requires more run-time resources for interpreting the model and choosing proper test stimuli to reach the test goal in non-deterministic models. Thus, the online testing methods differ in how the test purpose is defined, how the test stimuli are selected on-the-fly, and what the planning effort is behind each choice [24].

Automated test execution. A number of model-based test execution tools has been reported in different categories: commercial, proprietary, and academic ones [32, 35, 40, 41, 42]. In the following we examine briefly only few tools which have had pioneering role in MBT.

AGEDIS – an acronym of Automated Generation and Execution of Test Suites for Distributed Component-based Software - was a research project coordinated by IBM Research. AGEDIS includes an integrated environment for modelling, test generation, test execution, and other test related activities. It also provides the framework and tools that support MBT methodology and test automation. In addition to test preparation and execution, AGEDIS also includes a feedback loop that integrates coverage and defect analysis tools. Three types of information are used to describe the system under test: the behavioural model of the system, the test execution directives which describe the testing architecture of the SUT, and the test generation directives which describe the strategies to be used in testing. The SUT behaviour and testing architecture are specified using a UML modelling tool equipped with the AGEDIS UML profile (e.g. Objecteering UML Modeller), whereas test generation primitives are input via an XML editor (e.g. XML Spy).

Conformiq Creator is a commercial type of MBT tool. Creator uses a custom modelling language which is based on activity diagrams and a graphical domain specific action language. Models can be created via an import from existing assets (e.g., flowcharts, BPM and manual tests), requirements can be downloaded from the Requirements Management Tool (RMT), and generated tests can be exported to Application Lifecycle Management (ALM) tools, Excel, various scripting languages, or test execution with Conformiq Transformer. In Conformiq Designer the models can be created as UML State Machines and in Qtronic Modelling Language (QML). Tests can be exported to TTCN-3 language format and organized by Conformiq's proprietary test management tool. Conformiq Test Generator allows creating test cases from UML state charts, which represent a high-level graphical test script. It has to be stressed that the state charts do not represent the actual SUT but only the test script, which means that the tool is more a test script editor.

The TorX tool is a prototype testing tool for conformance testing of reactive software. JtorX is a reimplementation of TorX in Java with additional features. It can be used for model-driven test derivation and execution. The Labelled Transition System (LTS) specification of SUT can be given in multiple formats, and it can interact on-the-fly with SUT. There are four main components: Explorer, Prinmer, Driver, and Adaptor. Explorer provides access to specification (either the Model being checked or a Test Purpose). Explorer components are specific to the formalism used in the specification. A Primer provides access to a formalism-independent version of the model, in which states that are not of interest (e.g. if they have been analysed in a previous test run) are ignored. The Driver controls a test run, as directed by the user, and records the test results. An Adaptor provides the connection between the Driver and SUT. Adaptor components are specific to the type of SUT and are model-dependent. The TorX environment currently allows automatic test derivation and execution for the LOTOS, PROMELA, and SDL languages.

The Uppaal tool family [52], supports modelling, validation and verification of real-time systems. The modelling formalism Uppaal Timed Automata is appropriate for systems that can be abstracted as a collection of non-deterministic processes with a finite control structure and real-valued clocks, communicating through channels and (or) shared data structures. Typical application areas include real-time controllers, communication protocols, and other systems in which timing aspects are critical. For online conformance testing the test execution tool Uppaal Tron [52] and its extension for distributed testing DTron [77] are exploited.

When comparing the MBT tools referred above, then AGEDIS, Conformiq and TorX rely on formal models that either do not have an explicit notion of metric time (Agedis, Conformiq) or have the notion of clocks in the limited form of safety timed automata (TorX). Since Uppaal timed automata extend the original timed automata to capture the notion of time intervals combined with data types and functions Uppaal modelling formalism and tools are chosen in this work for testing systems with parallel processes and non-trivial timing and data constraints.

1.4 The correctness of model-based test development

Though MBT workflow relies inherently on the techniques of model engineering, the verification of the test development process and its intermediate results is not generally a compulsory part of MBT. The *provably correct development* (PCD) approaches, in contract, tie the development process with obligatory verification and validation steps or rely on *correct by construction* approaches [36]. Like in software development, applying PCD processes in test development is motivated by the need to improve the trustworthiness of the process products. In MBT it means showing formal correctness of test increments at each of their development phase. In this thesis we focus on the model-based testing of systems with timing constraints, and in particular, incorporating timing aspects into AO test models. Verification of such test models is important not only from the point of view of expressing adequately the properties of SUT but also to assure that their testing results can be trusted and traced back to the root causes, either in the requirements or in the implementation.

Model-based analysis of test models can reveal the design errors already before any testing. Also, it if the validation of test models reveals the consistency and/or relative completeness errors in them.

Considering the pragmatic aspects of modelling, it would be a great advantage from a modelling effort point of view to extract test models directly from those used for design specification. Unfortunately it is not always possible because the test models needs to represent only information of a given test case. Since the design models carry a multitude of implementation details extracting that which is only needed in testing may prove to be impractical. Another reason preventing the use of the same models for design and testing is the need for keeping design and test activities independent. Otherwise, faults in requirements modelling would propagate to both design and testing. Therefore, to avoid making same faults the test models are developed and verified independently from the design models.

The model validation techniques can be divided into simulation based, algorithmic state space exploration and deductive methods. The first hand rough validation method of models is visual inspection and simulation which are intuitive but not exhaustive. By simulating the test scenarios on models we can detect the inconsistences with the behaviours expected by end user.

The second group of analysis techniques relies on model state space exploration [20]. For example, model checking can explore the state space of test models and discover unintended behaviours, such as states and transitions that the test case never reaches. The exploration can also answer concretely formulated questions specified as model checking queries. So model checking can perform safety analysis that identifies unsafe states and a liveness analysis that identifies dead states. Model checking queries are typically temporal logic expressions interpreted on separate executions (in the case of linear temporal logics) or on reachability trees (in the case of computation tree logics) of models. The liveness properties are expressed as reachability constraints of legal model states and safety properties as non-reachability of illegal or unintended states. Typically deadlocks and live locks indicate violation of liveness properties. Such a model-based analysis can reveal the design errors of tests before their execution.

In addition to standard safety and liveness properties verifiable in test models AOM introduces additional model correctness conditions. For instance, when studying the correctness of AO models we have to be sure that representing the system aspect-wise provides the same testing results as that of doing it with monolithic non-aspect models. This group of model correctness properties is called *aspects non-interference properties*. Both the AO model consistency and non-interference analysis are addressed in the thesis. This topic will be detailed in Sections 3.2.2 and 3.2.4.

1.5 Main hypothesis and problem statement

1.5.1 Hypothesis of the thesis

As claimed in [39] aspect-orientation as one of the model structuring principles allows improving the efficiency of model-based testing. This thesis explores this generic hypothesis in the context of Uppaal Timed Automata by substantiating it from following perspectives:

- Test model construction and update effort decreases compared to non-AO models along with improving the model comprehension due to reduction in the number and in the severity of modelling errors and the need for their corrections;
- Defining the formal semantics of AO models and model transformations used in AO model construction allows applying compositional test generation and execution;
- Aspect-oriented test cases are more compact and allow saving test execution time (improved performance); and

- Defining the test cases and their coverage criteria relative to aspects provides better traceability of the causes of bugs and locating them in the AO requirements specification.

1.5.2 Research questions

To validate the hypothesis stated in Subsection 1.5.1 the following research problems have to be answered in the context of Uppaal TA formalism:

- How AO MBT improves the productivity of the overall testing process?
- How the generic aspect-oriented model engineering methodology can be instantiated for aspect-oriented Uppaal TA models?
- How to specify the test purpose and test coverage criteria in AO test models?
- How to express the AO test purpose symbolically?
- Given an AO SUT model and AO test purpose how to verify the correctness and feasibility of the tests?

1.5.3 Problem statement

To answer the research questions following tasks have to be solved:

- providing experimental evidence based on a real life case study, that AO testing improves the efficiency of MBT compared to the methods that are based on non-aspect oriented (monolithic) models;
- developing the AO test model construction method for Uppaal TA that includes the definition of join points, pointcut expressions and weaving mechanism to compose the base and advice models of weakly invasive [53] aspects;
- formulating the correctness conditions of AO models and the conditions that allow applying AO models for compositional testing; and
- defining the AO test coverage criteria for Uppaal TA expressible in TCTL.

1.6 Methodology

The research methodology applied in this thesis relies on the techniques of formal modelling and model-based testing. Specifically, the methodology covers formal techniques of the following subdomains:

- mapping AO programming constructs to AO modelling constructs;
- representing the MBT coverage criteria in terms of AO model structural elements;
- interpreting abstract AO tests in terms of timed traces of AO models;

- expressing test coverage criteria symbolically using elements of AO models as terms of test coverage expressions; and
- executing the abstract conformance tests in the online test execution tool Uppaal Tron and its extension for distributed testing Dtron.

1.7 Contribution of the thesis

The main contribution of this thesis is four-fold:

- An original aspect-oriented model engineering methodology is introduced in MBT. This methodology is based on an aspect-oriented requirements engineering paradigm that results in three advantages: testability of SUT aspect-related quality attributes, a simple rule for composition, and better comprehension of test models. An example is presented as an exploration of the practical utility of this methodology.
- A set of aspect-oriented test coverage criteria is defined. That gives meaningful automatic test design options based on SUT models which are defined by quality attributes related to aspects. It is shown that coverage criteria can be formalized in temporal logic TCTL.
- Weaving of aspects is implemented as a set of model superposition refinement operators. AO tests can be generated automatically by running TCTL model checking queries on woven models and applying resulting witness traces as test sequences of AO test cases.
- A developed MBT method which defines the AO test coverage criteria and provides an AO test generation algorithm is validated on a realistic case-study.

1.8 Structure of the thesis

The thesis main results are published in four research articles attached in Appendices A, B, C, and D.

The structure of the thesis, after introducing the research context in this chapter, is as follows:

Chapter 2 presents theoretical preliminaries of aspect-oriented testing by highlighting the principles of aspect-oriented modelling, the syntax and semantics of Uppaal TA, defining the notions of conformance relations and clarifying the meaning of aspect-oriented conformance testing.

Chapter 3 elaborates the aspect-oriented model construction technique for Uppaal TA and compares two aspect weaving approaches proposed for Uppaal TA. The correctness conditions of weaving are defined in timed temporal logic TCTL. Finally, the aspect-oriented test coverage criteria are defined and it is shown how test sequences are generated from them using model checking.

Chapter 4 exemplifies how the theoretical results are applied in a practical case-study - Home Rehabilitation System.

Chapter 5 provides the methodology of demonstrating the advantages of aspect-oriented testing compared to non-aspect oriented approaches. The method allows verifying the bisimilarity of non-aspect oriented and aspect-oriented Uppaal TA models relative to test interface behaviour. This is necessary for comparison of models from different perspectives such as model update effort, test purpose specification effort, test generation effort and test execution effort. This analysis is done analytically and the results confirm the experimental evaluation results presented in Chapter 4.

The conclusion summarises the contribution of the thesis and outlines the open issues and suggestions for future research.

2 PRELIMINARIES

2.1 Chapter overview

This chapter provides technical preliminaries and definitions which are used to extend the model-based testing with AO concepts. Aspect-Oriented Modelling, in Section 2.2, and Model-Based Testing, in Section 2.3, are two methods which tie together concepts used in Uppaal TA for testing. Section 2.4 elaborates the principles of MBT in the context of aspect-oriented testing. Section 2.5 provides the definition of semantics of Uppaal TA and Section 2.6 introduces the conformance relation and notions of conformance testing with Uppaal TA. Section 2.7 discusses the related work on AOM and AOT. The underlying concepts of each chapter are introduced in publication A.

2.2 Aspect-Oriented Modelling

2.2.1 Basics of AOM

One way to manage with system complexity is the separation of concerns in its description. A concern is a part of the problem that is treated as a single conceptual unit. The essence of AO is articulated the best in Aspect-Oriented Requirements Engineering (AORE) methodology. It is the methodology that can help to improve requirements *completeness, maintainability,* and *reduce the cost* of software development. AORE is suitable for distributed system development processes lacking a single "holistic view" to the system and for integration of independent, goal-oriented tasks. AORE focuses on resolving issues with the *scattering* and *tangling* of requirements to improve the *modularization*, *maintainability,* and *completeness* of the models of requirements. The model is put together using different stakeholders' viewpoints and AORE analysis techniques. It is important in the analysis phase to define the application decomposition and identify the inventory of concerns that lay the ground for the modularization and the structure to reach a harmonized requirements model [43].

Aspects are usually defined as "units of system decomposition that can be either functional or non-functional". An aspect in the requirements is a concern that crosscuts requirements artefacts. Early identification and managing aspects helps to improve modularity in the requirements and in architectural design and to detect conflicting concerns that need resolving by finding feasible trade-offs. In addition, identifying aspects at one stage provides benefits downstream. Knowing the requirement-level aspects helps the architect to design a better system, whereas, knowing the architecture-level aspects helps producing a more robust implementation [44].

While AO originally has emerged in programming [2, 19], it now stretches also over all other phases of model based development. Like in requirements engineering, also in all other phases of software development (AOSD) the aspect-orientation provides improved separation of concerns, ease of maintenance, evolution and customization, and greater flexibility in development [3]. In a survey of industrial projects [45] it is outlined that the main benefits of AOSD software development are substantial reduction in model size and improved *design stability*.

AOSD aims at addressing crosscutting concerns by providing means for their systematic *identification, separation, representation,* and *composition*. Crosscutting concerns are encapsulated in separate aspects and composition mechanisms are later used to weave them back with other modules. In particular, AOSD focuses on the modularization and composition of crosscutting concerns.

The term crosscutting concern refers to properties of software that cannot be effectively modularized using traditional software development techniques, such as object-oriented methods. Typical examples of crosscutting concerns are non-functional requirements, such as security, safety, fault tolerance, and persistency. However, crosscutting concerns can also be functional requirements. Aspects will allow the modularization of crosscutting concerns that cannot be encapsulated by a single use case or viewpoint, and are typically spread across several ones.

Aspect-oriented modelling (AOM) [21, 46] is a paradigm inspired by AOSD and it also promotes the idea of separation of concerns in order to build more modular and easy to update specifications. In AOM, an aspect describes a particular concern of the system from a particular viewpoint, allowing the developers to focus on individual features of the system in isolation. Regardless, AO concepts are well-known, for almost two decades the main body of AOSD and AOM technologies provide conceptual frameworks rather than define a rigorous interpretation of operations needed in AOM. Thus, the main research challenges concerning AOM and applying AO concepts in testing can be summarized as follows:

- How to unify the semantics of AO notions?
- How to hide the complexity of AOM composition mechanisms?
- How to exploit the AO notions and AOM composition mechanisms in MBT?

It is generally assumed in AOM that introducing aspects starts from some *base model* where aspects are not yet represented explicitly. Aspects can be modelled separately and added to the base model incrementally in the form of *advices*. Composing a base model with advices is called *aspect weaving*. An aspect can be woven with a *base model* in many places and in different ways. Such places in the base model are called *join points*. *Pointcuts* in the base model are the rules which specify join points, i.e. where and under which conditions the aspects can be woven. The composition rules or weaving directions tell how to weave the advice at the join points which satisfy the pointcut specification.

In this thesis we present an AO modelling method for model-based testing in the semantic framework of Uppaal timed automata (Uppaal TA). This choice is motivated by sufficiency of expressive power and relevance of Uppaal TA for specifying behavioral aspects and incorporate timing constraints as explicit dimensions of aspects.

The rationale behind this work is to provide a) a rigorous constructive approach to the weaving of aspects in the context of Uppaal TA and b) well-defined coverage criteria for aspect-oriented testing by means of Uppaal TA models.

2.3 Model-based testing

MBT is typically a black box testing technique where state machine models are used as specifications of observable interactions between SUT and its environment. The goal is to replicate the behaviours of the model in SUT by sending model generated test stimuli to SUT and observing if reactions of SUT conform to those specified in the model.

The development process of model-based tests includes typically five phases: *modelling of SUT, specification of the test purpose, test generation, deployment, and execution.* A waterfall shape test development process model is shown in Figure 2.1.

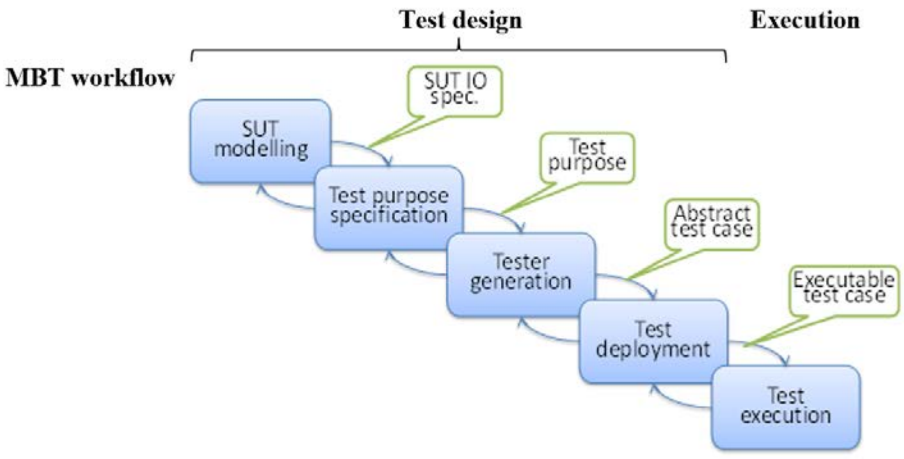


Figure 2.1 MBT workflow

A test model is constructed based on the test requirements and the test plan, at first. The model is usually an abstract, partial presentation of the expected behaviour of a SUT. The test model is used to generate the test cases that together form an abstract test suite. In principle, the test models can represent an infinite set of SUT behaviours. Therefore, test selection criteria, specified as *test purpose* are meant to select a finite and practically executable set of proper test cases. For example, different model coverage criteria, such as all-states, all transitions, selected branching points etc. can be used to derive the corresponding test cases.

The coverage of model structural elements (states and transitions) can be used also as a measure of *thoroughness for a test suite*. Thus, a *test purpose* is a specific objective (or property) that the tester would like to test, and can be seen as a specification of a test case. It may be expressed in terms of a single coverage item, scenarios, duration of the test run etc.

Let us consider a requirement "*Test a state change from state s_A to state s_B*" in the model M^{SUT}. For this purpose a test case should be generated that, when starting from the initial state s_0, covers the specific state transition in M^{SUT}. At first, it requires that the test drives SUT to state s_A, then specified transition is executed and when s_B is reached the test should terminate in some safe state of M^{SUT}.

In case of non-deterministic systems a single precomputed test sequence may never reach the test goal, and instead of a sequence we need an online testing strategy that is capable of reaching the goal even when SUT provides non-deterministic responses to a test stimulus. The issue is addressed in [47] where the reactive planning online tester synthesis method is proposed.

In the third phase, the abstract test suite is generated from the model consisting of SUT and environment component so that the test purpose can be reached by executing the test suite. This is typically done using a transformation tool which translates each abstract test case into an executable test case. The abstract test cases are deployed in the test execution environment by transforming them directly into an executable test scripts or by introducing test adapters which map symbolic model inputs to executable ones and the concrete outputs of SUT back to symbolic form to compare them with ones given in the model. An advantage of the separation between abstract test suite and concrete test suite is the *platform* and *language independence* of the abstract test cases. Thus the same abstract test case can be reused in different test execution environments.

In the fifth phase, the deployed test cases are executed against the SUT. The test execution will result in the report that contains the outcome of the test case execution. After test execution, given results are analyzed and corrective actions are taken in the implementation if needed. Hereby, for each test that reports a failure, the cause of the failure is determined and the program (or model) is corrected.

An example of the symbolic test execution tool for Uppaal TA is Uppaal Tron [52] which conceptual architecture depicted in Figure 2.2.

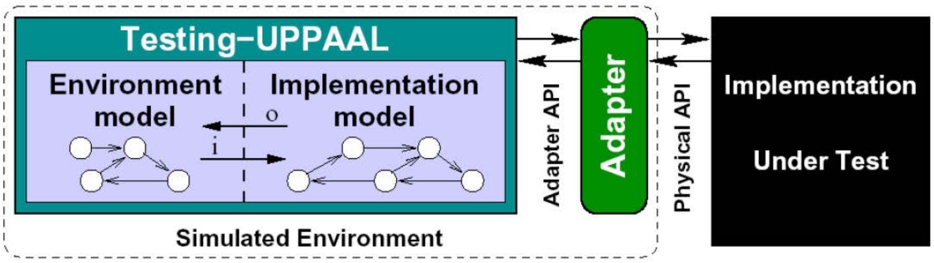


Figure 2.2 Online MBT execution architecture: Uppaal Tron [52]

2.4 Aspect-Oriented Testing

In this section we explain the concepts of AO modelling that are applicable in aspect-oriented MBT. The AOM allows one to organize the models so that they address crosscutting requirements and corresponding test cases. AO testing can be considered as an example of decomposition testing where the integration of components is tested after components have been tested separately. In a MBT context it means that test cases are determined only by the local contexts of advice models and only when conformance of their aggregated interface behaviour needs to be tested.

In an AO setting we address the test purpose in terms of aspects and aspect related model structures. Thus, the test cases for a test purpose should be derived from the aspect model(s) of concern where the rest of SUT specification is abstracted away. Aspects may contain sub-aspects that have their own particular test cases. In this manner the AOM and AOT can be applied recursively.

The efficiency of aspect-oriented verification and testing, depends on whether these activities can be done *compositionally*, i.e., if it is possible to infer the properties and test verdicts of the composition from the verified properties or passed tests of components in separation. In order to enable a compositional approach, we need to construct Uppaal TA specification in a modular way by applying principles of AOM. Secondly, for compositionality the non-interference between the components of aspect models needs to be ensured. In terms of AOM it means non-interference verification between the aspects. In Chapter 3, we detail how aspect non-interference can be introduced and verified in Uppaal TA via assume-guarantee reasoning. The symbolic AO coverage criteria are expressed in timed temporal logic TCTL to specify timing constraints and 1st order logic formulas to specify state properties.

2.5 Uppaal timed automata

We start with the formal definition of Timed Automata as in [48]:

Definition 2.1 (**Timed Automaton**)

Assume Σ denotes a finite alphabet of actions a, b, ... and C a finite set of real-valued variables x, y, z, standing for clocks. A guard is a conjunctive formula of atomic constraints of the form $x \sim n$ for $c \in C$, where $\sim \in \{\geq, \leq, =, >, <\}$ and $n \in \mathbf{N}^{+.}$ We use $G(C)$ to denote the set of guard conditions on clocks of C.

A timed automaton A is a tuple $\langle L, l_0, E, I\rangle$ where

– L is a finite set of locations (or nodes),

– $l_0 \in N$ is the initial location,

– $E \in L \times G(C) \times \Sigma \times 2^C \times L$ is the set of edges and

– I: $L \rightarrow G(C)$ assigns invariants to locations (here we restrict to constraints in the form: $x \leq n$ or $x < n$, where $n \in \mathbf{N}^+$. For shorthand we write $l \rightarrow_{g,a,r} l'$ to denote edges.

We use a function known as clock assignment (or clock reset) that maps C to non-negative naturals $\mathbf{N}^+$.

To model concurrent systems we extend the Definition 2.1 with synchronous parallel composition. A *network of timed automata* is the parallel composition $A_1 \| \ldots \| A_n$ of timed automata $A_1, \ldots, A_n$ called processes and combined into a single system by the CCS parallel composition operator with all external actions hidden (this composition principle applies so called *closed world assumption*). Synchronous communication between the processes is by *hand-shake synchronization* using input and output actions (note that asynchronous communication can modelled by using shared variables, this will be explained at Uppaal TA below). To model hand-shake synchronization, the action alphabet Σ is assumed to consist of symbols for input actions denoted a? and output actions denoted a!. The internal actions of automata are denoted by ε [48].

To adjust the modelling power and keep the analysis traceable for test synthesis we limit the class of timed automata to rectangular automata where guard conditions are in conjunctive form with conjuncts including besides clock constraints also constraints on integer and Boolean variables and their arrays. Similarly to clock conditions the propositions on integer variables, e.g. k are of the form $k \sim n$ for $n \in \mathbf{N}$, and $\sim \in \{\geq, \leq, =, >, <\}$. This extension to Timed Automata is called *Uppaal Timed Automata* (*Uppaal TA*). The advantage of this extension is that the model has rich enough modelling power to represent real-time and resource constraints and at the same time to be decidable for reachability analysis.

Definition 2.2 (**Operational Semantics**)

The semantics of timed automata is defined by means of transition systems where the *configuration* consists of the vector of concurrent locations (one for each automaton in the network), valuation of state variables and the current values of clocks. There are two types of transitions between states: the automata running in parallel may either delay for some time (*delay transition*), or follow an enabled edge (*action transition*).

To keep track of the changes of clock values, we use functions known as clock assignments mapping C to the non-negative reals $\mathbf{R}^+$. Let u, v denote such functions, and $u \in g$ means that clock values denoted by u satisfy the guard g. For delay $d \in \mathbf{R}^+$ let $u + d$ denote the clock assignment that maps all $x \in C$ *to* $u(x) + d$ and for $r \subseteq C$ let $[r \mapsto 0]$ denote the clock assignment mapping all clocks to 0 and agree with for the other clocks in $C\backslash r$.

The operational semantics of timed automata is represented using timed transition system where states are pairs $\langle l, u\rangle$ and transitions are defined by the rules:

– $\langle l, u\rangle \rightarrow_d \langle l, u + d\rangle$ if $u\in I(l)$ and $(u + d) \in I(l)$ for a non-negative real $d \in \mathbf{R}^+$

– $\langle l, u\rangle \rightarrow_a \langle l', u'\rangle$ if $l \rightarrow_{g,a,r} l'$, $u \in g$, $u' = [r \mapsto 0]u$ and $u' \in I(l')$.

The graphical representation of a timed automaton is considered as a directed graph, where locations are represented by the vertices and they are connected by edges (see Figure 2.3). Locations are labelled with invariants. Invariants are conjunctive Boolean expressions where the literals consist clock variables and bound conditions of clock variables, e.g. $x \leq n$.

Edges are annotated with guards, synchronisations and updates. An edge is enabled by a guard in a state if and only if the guard evaluates to true. Processes (parameterized instances of Uppaal TA templates) can synchronize over channels. Edges labelled with same channel symbol synchronise, e.g. in Figure 2.3, the edge '*WaitingCard*$\rightarrow$*Idle*' of Customer automaton and the edge '*printReceipt* $\rightarrow$*Idle*' of ATM automaton synchronize over channel '*card*'. Updates express the change of the state of the system when the edge is executed, e.g., update '*Clock1 = 0*' resets the value of model clock '*Clock1*'.

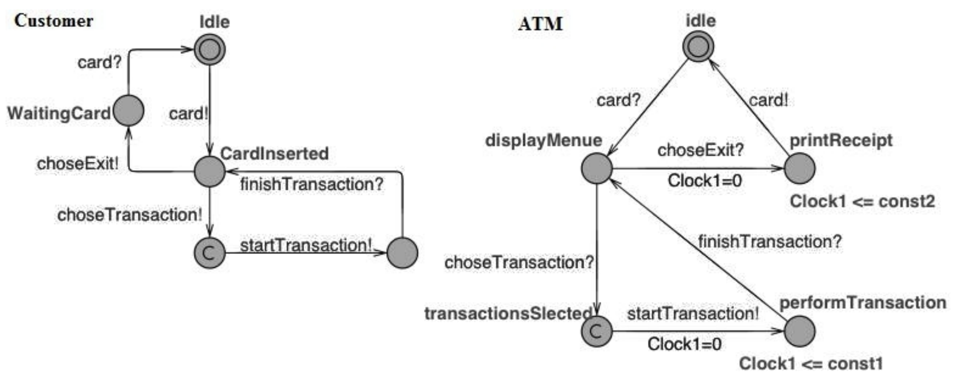


Figure 2.3 The Uppaal TA model of ATM

2.6 Conformance testing with Uppaal TA

During a test session, MBT tool Uppaal Tron [52] uses the Uppaal verification engine to generate symbolic timed traces of the Uppaal TA model. For each symbolic state, the next reachable symbolic states to visit are calculated, and the actual next state is chosen randomly from those reachable via enabled transitions. A test session ends when the model reaches a final state, the test duration expires, or a violation of conformance between implementation and specification is encountered.

A symbolic timed trace *TTrS* of an Uppaal TA model is a (possibly infinite) sequence of symbolic states, each state being defined as a tuple ($\bar{l}$, D, $\bar{v}$), where $\bar{l}$ is a locations vector, D is the set of clock constraints (zone) [49] and $\bar{v}$ a vector of non-clock variable values [50]. As shown in operational semantics, the transition from a symbolic state to another can be either an action (a_i) or a delay (δ_i).

$$(\bar{l}_i, D_i, \bar{v}_i) \xrightarrow{ai\ /\delta i} (\bar{l}_j,, \bar{v}_j) \qquad (1)$$

In Uppaal TA, an action may be composed of an I/O event *e* and assignments to variables of *V*. As a consequence, when the system state changes, we can observe either an event *e*, updates of *V*, or both. An example is shown in Figure 2.4 [48]. The symbolic state *a* is visited after evaluating the guard *g*, performing the variable update (in case of clocks, reset operation *r*), and observing an action *A*. Similarly, states *b*, *c*, … , *d*, *e* are visited after evaluating their respective guards *g*, reset *r* and action event *A*. The state *f* represents an error state where Uppaal Tron assigns a verdict *failed f* (failed) to the test run.

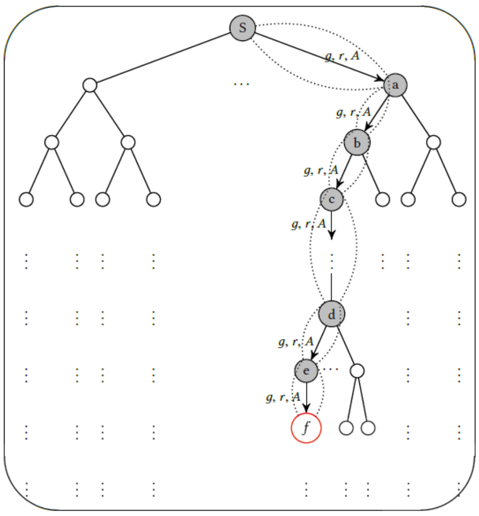


Figure 2.4 Traversal of the symbolic state space [48]

The decision on which state transitions are enabled in a given state is done based on the interaction between Uppaal Tron [52] and the SUT by evaluating received SUT output, available inputs or delays.

In order to identify the observable behaviour between the tester and the SUT, Tron partitions the Uppaal TA model into two parallel partitions $\mathcal{S}$ and $\mathcal{E}$, which model respectively the SUT and its environment. The interaction between $\mathcal{S}$ and $\mathcal{E}$ is implemented via observable (at test interface) actions, further divided into input (A_I) and output (A_O) actions. The former are used as stimuli to the SUT during testing whereas the latter are used for deciding on conformance. Additionally, $\mathcal{S}$ and $\mathcal{E}$ have internal actions ε confined to each partition, evolving the partition to the next state where the next observable action can be taken.

During test run, the observable actions A_I and A_O are triggered based on a testing event e, following an observable delay $\Delta \in \mathbf{R} \geq 0$ which abstracts the internal events. A vector of externally visible variables $\bar{v}$ (in Uppaal TA they are defined as global variables) which contains the value of data variable at the time of the event is also observable. The events and variables are partitioned into three disjoint sets of input events/variables $\mathcal{E}v_{in}/V_{in}$, output events/variables $\mathcal{E}v_{\mathrm{out}}/V_{\mathrm{out}}$, and internal events/variables $\mathcal{E}v_{int}/V_{int}$ [50].

Thus, after dividing the model into the environment and SUT partitions, a symbolic trace can be rewritten as a timed I/O trace. The latter is a (possibly infinite) sequence of observations starting from a given state, where each observation is a tuple $(e, D, \bar{v})$ consisting of an event $e \in \mathcal{E}v_{int/out}$ a clock zone D in which event occurs, and a vector $\bar{v} \in V_{(in/out\,)}$ containing the values of data variables that are externally visible as inputs/outputs at the time of event e.

$$ttr_{i/o} = (e_0, D_0, \bar{v_0}), (e_1,D_1,\bar{v_1}), \ldots (e_i ,D_i ,\bar{v_i}), \ldots \qquad (2)$$

Uppaal Tron is using the externally visible (observable) events to interact with the SUT, while abstracting away the internal actions ε and the internal delays d as observable delays Δ. Thus, the result of a test session will be a finite sequence of events T_{seq} of the form:

$$T_{seq} = (e_0, (\tau_0 + d_0), \bar{v_0}), (e_1, (\tau_1 + d_1), \bar{v_1}), \ldots ,$$
$$(e_n, (\tau_n + d_n),\bar{v_n}), (e_{n+1}, (\tau_{n+1} + d_{n+1}),\bar{v_{n+1}}), \qquad (3)$$

which can be written in terms of observable delays and actions as:

$$T_{seq} =(e_0, \Delta_0,\bar{v_0}), (e_1, \Delta_1,\bar{v_1}), \ldots ,$$
$$(e_n, \Delta_n,\bar{v_n}), (e_{n+1}, \Delta_{n+1},\bar{v_{n+1}}), \ldots \qquad (4)$$

This allows one to check the timed conformance of the SUT against the specification via the *rtioco* relation, by allowing the SUT to refine the timing behaviour of the specification [51].

Definition 2.3 Relativized timed input/output conformance (*rtioco*) [48]

An implementation $\mathcal{I}$ conforms to its specification $\mathcal{S}$ under the environmental constraints if for all timed input traces $\sigma \in TTr_i(\mathcal{E})$ the set of timed output traces of $\mathcal{I}$ is a refinement of the set of timed output traces of S for the same input trace.

$\mathcal{I}$ *rtioco* $\mathcal{S}$ iff $\forall\sigma \in TTr_i(\mathcal{E})$:

$$TTr_o((\mathcal{I}, \mathcal{E}), \sigma) \sqsubseteq TTr_o((\mathcal{S}, \mathcal{E}), \sigma) \qquad (5)$$

The resulting test sequence is provided by Uppaal Tron as a sequence of test events. For each test event, symbolic state in which the event occurred is specified in terms of clock constraints, variable valuation, list of next available states, and a list of input/output actions. A new test event occurs at a specific time, the clock constraints are updated, a transition to a new symbolic state occurs and the list of the next available states is updated [51].

2.7 Related work on AOM and AOT

In an early work, Jacobson [21] describes the development of design aspects based on use cases, which are then composed to create different views of the system. The work provides the conceptual background of AO but does not explicitly give details about the transformation of models, rules of composition, and structural relations.

From a modelling perspective, UML [55] has been the *de facto* modelling language in AOM and several profiles have been proposed for modelling aspects (e.g., [56, 57]). In addition, studies [58, 59] provide surveys and assessments of aspect-oriented modelling techniques.

While introducing the AOM constructs, we target semantic unambiguity and mature tool support. These prerequisites are satisfied by Uppaal TA, in contrast to UML that does not have commonly understood formal semantics. Although Uppaal TA is less expressive than UML, it is better suited for timed model checking and test generation. The earliest attempt of implementing AOM concepts in Uppaal TA has been proposed in our publication [54]. This work suggests handling the aspect models as refinements of locations and edges in the base model.

The work [60] also uses Uppaal TA but the focus is on extending the functionality of the system with new features by defining a set of different weaving operators. In addition, the non-interference of aspects via assume-guarantee assertions has been suggested as a prerequisite for compositional verification and test case generation.

Aspect interference is a well-known issue in AOSD. The interference occurs when weaving conflicting aspects with the same base model. This issue has been discussed in [58, 61], while a detailed analysis has been presented in [61]. In order to address the interference problems the thesis relies on the work presented in [53], which suggests non-interference criteria for weakly-invasive aspects. Weakly invasive aspects are aspects that may change the control flow and the values of non-local variables, as long as the state after returning from advice to the base model is reachable in the original base model.

While the combination of propositional and linear temporal logic (LTL) has been used in [53] for expressing non-interference conditions, we presume that the aspect specifications are expressed in TCTL [62]. This allows one to express also the non-interference of explicit timing properties. We use the Uppaal model checker like in [60] to verify whether the aspects are interference-free, and to decide on the suitability of weaving them in the joint test model.

Test generation from abstract models targeted at aspect-oriented programs has been suggested in [63]. The tests are also built from the requirements of the system using AOM techniques. For instance, D. Xu generated tests from protocol state machines [64] and use case diagrams [65] so that aspectual use cases were used in generating test requirements. They transform the use case diagrams into aspect-oriented Petri nets [66], and then, extract the corresponding use case sequences using transition, state, and use case coverage. The approach of [65] is similar to ours but is limited to use case models without considering time. It also does not use tool support, neither for the weaving nor for the test generation.

In [67] the authors suggested an AO extension for UML models (class diagram and sequence diagram) in order to generate tests for AO programs. Similarly, in [68] a UML profile has been suggested for modelling behaviour via aspect state machines. This approach uses model transformations and tool support for test generation. Our approach however differs from it by separating aspects to aim at test coverage criteria that are aspect specific and can be tested aspect-wise. We are also targeting timed specifications using the Uppaal TA formalism and TCTL-based verification instead of UML.

2.8 Conclusion

This chapter presented the theoretical foundations of AO MBT. The basics of aspect-oriented modelling were introduced and related to model-based testing concepts. Formal definition of Uppaal TA provided a sematic ground for mapping the AOM constructs onto Uppaal TA. We have elaborated the principles of MBT in the context of aspect-oriented testing and defined the conformance relation RTIOCO and the notions of conformance testing with Uppaal TA. The chapter concludes with related work on AOM and AOT.

3 ASPECT-ORIENTED MODEL ENGINEERING

3.1 Chapter overview

This chapter presents the process of AO model construction, specifically how AO modelling notations are interpreted in Uppaal TA. Two alternative approaches are studied, one inspired by weaving constructs of AO programming, and the other, more abstract, that hides the weaving details and implements weaving as model superposition refinement operators. The correctness conditions of weaving are defined in a timed computation tree logic TCTL. The aspect-oriented test coverage criteria are defined then and it is shown how test sequences are generated from them using model checking.

3.2 Creating AO test models in Uppaal TA

Before presenting the Uppaal TA based AOT approach, we introduce the generic AO notions and then give their interpretation in the context of Uppaal TA:

- An *aspect model* is an Uppaal TA process or a set of parallel processes that implements a crosscutting concern;
- A *base model* is a set of Uppaal TA processes that model the core functionality of the system;
- An *advice model* introduces features and behaviours specific to given aspect;
- *Join points* are model fragments in the base model to which an aspect can be woven;
- A *pointcut* is the set of join points and conditions under which an advice can be woven. A *pointcut expression* is a logic condition which uniquely defines the model fragments (join points) where the weaving is applied;
- *Weaving* is the process of composing a base model with the *advice model* that represents the action taken by an aspect at a particular join point; and
- A *woven model*, sometimes referred to as *augmented model*, is an Uppaal TA in which the base model is woven with intended aspects.

In the rest of this section two weaving approaches are introduced in detail. The first approach, highlighted in the next subsection, was originally introduced in [69]. This work was inspired by AOP where weaving operators such as *around*, *before, after* are defined. We will outline this work as a base case to position our approach introduced later in Subsection 3.2.2.

3.2.1 Approach 1: AO model construction using weaving adapters

In the adapter-based weaving approach the *adapters* are model fragments that allow the execution of an advice model at the designated join points. In [69] the approach is limited to join points which are Uppaal TA edges with synchronization. A weaving adapter encodes the pointcut expression, the advice type and the join point. The approach is based on the following assumptions:

- The individual instances of an advice model (defined by an Uppaal TA template) are woven at each join point of a base model;
- The execution of an advice is *atomic* w.r.t. its join point. This means that once a join point is reached, the control flow of the base model process containing the join point will be passed to the aspect model, and the base model process will wait for the aspect to complete and return to the same join point. However, this does not restrict several join points located in different processes of the base model to be enabled at the same time and their corresponding instances of advice models to be executed simultaneously;
- An advice model has one entry point and one or several exit points which return to the same join point;
- The base model and advice model can be woven using Uppaal TA specific communication and synchronization constructs, e.g. synchronizing the entry and exit of the advice model with *wait* in the base model, sharing or refining data between base and advice model, etc. and
- Join point definitions cannot refer to the elements of weaving adapters in order to ensure that the weaving does not introduce or remove join points for another adapter. However, new join points can be introduced in the advice models.

In [69], four types of weaving adapters are defined. They provide support for weaving an advice *before, after,* and *around* a join point, similar to the homonym advice types in AspectJ. The fourth adapter type, *conditional*, has been suggested based on practical considerations.

In this approach, a join point is restricted to Uppaal TA model fragment, namely, an edge that is labelled with a *guard expression, channel,* and *update* as depicted in Figure 3.1. The channel labels denoted by *channel*? and *channel!* represent synchronization of actions and can be interpreted either as an input or output action of the process the edge belongs to. Additionally, the edge may be labelled with guard expression and an update expression.

The weaving adapters allow a systematic weaving of advice models at designated join points in the base model. A weaving adapter is a merge of base model side and advice model side, specifying the model fragment to be included in the base model and, respectively, to the advice model, during weaving.

The *after adapter* Figure 3.2 (top) implements the execution of an advice after a join point edge(in this case channel synchronization). Its introduction substitutes the *End* location (in Figure 3.1) with two new locations *AspectStart* and *Call* (shown in Figure 3.2), and introduces two new channels *enterAdvice!* and *exitAdvice?*. Whenever the *pointcut_expression* is true, the advice is executed, otherwise the advice is skipped.

The advice model side adapter partition is shown in Figure 3.2 (bottom). The execution of the advice model is triggered from the base model via the join point by receiving the *enterAdvice?* synchronization and, after executing the advice model, it returns the control via the *exitAdvice!* synchronization.

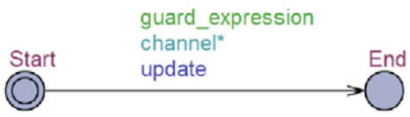


Figure 3.1 Model fragment with channel synchronization

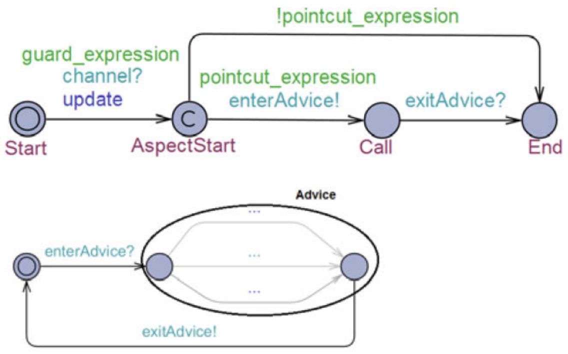


Figure 3.2 Generic adapter (top) and generic advice (bottom)

Weaving Process. In the AO approach it is assumed that aspects can be designed independently from specifications and the aspects are woven incrementally. That is, for a given base model and a set of advices, we weave one advice at a time to all of its designated join points. We regard the weaving process as a model transformation that takes a base model, advice model and a selected weaving adapter as inputs. The pointcut expression is used as a model pattern which identifies join points. The transformation inserts the adapter at the join point and instantiates the template of the advice for each join point.

It is also assume that the weavings are applied to the class of *weakly-invasive* aspects and that the weaving is a conservative transformation with respect to the class of Uppaal TA. Weakly-invasive aspects may change the control flow and the values of non-local variables, as long as the state after returning the execution to the base model is reachable in the base model without the aspect woven [53].

Verification of aspect non-interference is a prerequisite allowing taking advantage of compositional verification and testing of the aspect-oriented models. That means inferring the properties and test verdicts of the composition from verified properties or passed tests of components in separation. The detailed guidelines for enabling compositional verification and testing of aspect-oriented Uppaal models are presented in [70].

3.2.2 Correctness of weaving adapter-based AO models

The correctness criteria of aspect models are specified in the form of *assume-guarantee assertions*. Assuming a system S comprises a set of aspects $A_{1, \ldots,} A_m$, the underlying environment models are assumed to satisfy the aspects' assumption and the augmented system with the aspect model woven satisfies the guarantee assertion. The specification of an aspect A_i is then a pair (P_{A_i}, R_{A_i}), where P_{A_i} represents the assumption on the underlying system and R_{ai} expresses the guarantee of the augmented system after the aspect A_i is woven. Thus, for an aspect A_i, R_{A_i} is the conjunction of TCTL formulas of the form:

$$\mathsf{A}\Box\ (pointcut_{A_i}) \Rightarrow \varphi),$$

stating that every time the *pointcut* of A_i is matched, φ should hold. Note also that φ is a temporal logic formula expressing what A_i 's execution guarantees. The guarantees of the form:

$$\varphi = \mathsf{A}\lozenge\ \psi_{ret}$$

are expressing what is expected eventually of each execution of the aspect A_i (the TCTL operator $\lozenge$ denotes the existential quantification over the set of states of an execution path). Since TCTL model checker of Uppaal does not allow nesting of temporal operators, it is practical to transform the temporal sub-formulas of R_{ai}, where $R_{ai} \equiv \bigwedge_i \mathrm{A}\Box\ (pointcut_{A_i} \Rightarrow \mathrm{A}\lozenge\ \psi_{ret})$ to equivalent form by means of bounded leads-to operator " $\theta \sim>_d \psi_{ret}$" where d is the deadline (with respect to the time instant where θ becomes *true*) of reaching the state where ψ_{ret} holds. Here expression ψ_{ret} denotes the propositional state formula that includes terms such as location names $l \in L(M^{base})$, state variables V and clocks $c \in C$.

Verification of aspects non-interference. In [69] the work on non-interference of weakly-invasive aspects of [53] has been instantiated for Uppaal TA, as follows.

Let $\oplus$ denote the sequential-weaving operator, $\boldsymbol{A} = \{A_1, \ldots, A_n\}$ be a set of aspects, S a system, and (P_{Ai}, R_{Ai}) be the specification of an aspect A_i.

Definition 2.4 The set $\boldsymbol{A}$ of aspects is said to be interference free (*denoted IF*) if and only if the following holds:

$$IF\ (\boldsymbol{A}) \equiv S \models \wedge_{i=1}^{n} P_{Ai} \Rightarrow S \oplus (A_1, \ldots, A_n) \models \wedge_{i=1}^{n} R_{Ai}$$

The verification conditions that aspects must satisfy in order to guarantee non-interference can be summarized as follows:

1. The aspect A_i is correct by itself:

$$IF^0(A_i)\colon S \models P_{Ai} \Rightarrow S \oplus A_i \models R_{Ai}$$

This rule guarantees aspect correctness with respect to its specification (P_{Ai}, R_{Ai}). Given that the assumption holds, the system obtained from weaving the aspect and all possibly inserted aspects, must satisfy the guarantees.

2. Let A_i be the aspect currently being verified and A_j any other aspect. The rules to detect interferences are:

$$IF^P(A_i, A_j)\colon S \models P_{Ai} \wedge P_{Aj} \Rightarrow S \oplus A_i^{PAi} \models R_{Ai}$$

This rule expresses that when weaving A_i to a system, where the assumption of another aspect A_j holds, its assumption should be preserved:

$$IF^R(A_i, A_j)\colon S \models R_{Ai} \wedge P_{Aj} \Rightarrow S \oplus A_j^{PAj} \models R_{Ai}$$

This rule expresses that when an aspect Ai has already been woven, weaving another aspect A_j preserves the guarantee of A_i.

In order to guarantee non-interference, the rules above must be satisfied for every pair of aspects. Symmetrically, corresponding IF-rules for $(A_j; A_i)$ need to be satisfied. When constructing a model of n aspects, for compositional testing of that model n times $IF^0(A_i)$ verification tasks and $n(n\text{-}1)$ times $IF(A_i, A_j) \equiv IF^P(A_i, A_j) \wedge IF^R(A_i, A_j)$ verification tasks must be solved.

3.2.3 Approach 2: refinement-based AO model construction

As shown in [71] the usage of adapters for AO model weaving may cause structural overhead in the augmented model and the increase of model checking and test generation complexity. In [54], we have suggested superposition refinement instead of using adapters as a weaving operator, i.e. refining the join point carrier element (either an edge or location which satisfies the pointcut expression), with the advice model. Although the lack of weaving operators *before, after, around* does not allow defining the shift with respect to join point explicitly, the structural overhead caused by adapters is eliminated and the method results in better scalability. From the modelling point of view the refinement-based weaving requires just a “place holder” element in the base model which defines the join point exactly where the substitution is performed. If the shift operators *before* and *after* with respect to the join point are still needed for some reason, then either the join point carrier element to be refined should be selected from those immediately preceding or following the element that satisfies the pointcut expression. Alternatively the pointcut expression can be modified to take into account the shifts. Second advantage of the refinement approach is that an inference test can be avoided because the refinement correctness conditions guarantee that the augmented model will be correct-by-construction.

3.2.4 Correctness of the refinement-based AO model construction

In [54] the weaving of aspect models is implemented as superposition refinement of locations and edges that represent join points in the base model. We call these refinement operators *location refinement* (denoted by $\sqsubseteq_l$) and *edge refinement* (denoted by $\sqsubseteq_e$) respectively. To keep the base and advice models still structurally distinguishable after weaving (for better comprehension) we implement the superposition refinement not by direct substitutions of model elements but by semantically equivalent construct. Namely, by applying a synchronous parallel composition between the base model and advice model. Here the semantic equivalence between direct substitution and parallel composition is granted by composition correctness conditions introduced in the following.

Let the advice model M^{el} be woven to the base model M at join point carrier element $el \in L(M) \cup E(M)$ by synchronous parallel composition $\|_{sync}$, so that, $M \sqsubseteq M \|_{sync} M^{el}$ where $\sqsubseteq \in \{\sqsubseteq_e, \sqsubseteq_l\}$. Synchronous composition of M and M^{el} should preserve the semantics of M also after superposition (like non-interference conditions of Approach 1). Technically, this composition means that entry and exit points of the advice M^{el} have to be synchronized (via auxiliary channel) with a join point carrier edge e in case of edge refinement or before and after edges in case of location refinement of M. For further elaboration we define the location and edge refinement relations in separate.

Definition 2.5 (*Location refinement)*

We say that a synchronous parallel composition of automata M and M^{li} is a *location refinement* of location l_i of M, ($M \sqsubseteq_l M \|_{\text{sync}} M^{li}$) iff $l_i \in L_M$, and there exists M^{li} s.t. $P_1 \wedge P_2 \wedge P_3$, where properties P_1, P_2, P_3 are defined as follows:

- P_1 (*interference free new updates*): no variable of M is updated in M^{li}, i.e. no variable of M occurs in the left-hand side of any update expression in M^{li};

- P_2 (*preservation of non-blocking*): $[(M \parallel M^{li}), (l_0, l'_0) \vDash \mathrm{E}\Diamond\ \mathit{deadlock}] \Rightarrow [M, l_0 \vDash \mathrm{E}\Diamond\ \mathit{deadlock}]$;

- P_3 (*non-divergence*): $inv(l_i) \equiv x \leq n$ for all clocks $x \in C_M$, $n < \infty \Rightarrow \exists\ d \leq n$: $[M^{li}, l'_0 \vDash l'_0 \leadsto_d l'_\mathrm{F}]$, where" $\leadsto_d$" denotes bounded reachability operator with time bound d; locations l'_0 and l'_F denote respectively initial- and final-nodes of the M^{li}.

Properties P_2 and P_3 are specified in TCTL. The predicate symbol '*deadlock*' is a standard predicate in Uppaal query language that denotes the existence of a deadlock state in the model. P_3 requires that the invariant of l_i is not violated due to accumulated delays of M^{li} runs.

Definition 2.6 (*Edge refinement*)

Let l'_0, l'_F denote the entry and exit locations of the advice model M^{ei}, respectively. A synchronous parallel composition of automata M and M^{li} is an *edge t_i refinement* (denoted $M \sqsubseteq_e M \parallel M^{ei}$, where $t_i \in E(M)$) if the conditions P'_1, P_3, P_4, P_5 are satisfied:

- P'_1 (*interference free new updates*): no variable of M is updated in M^{ei}, i.e. no variable of M occurs in the left-hand side of any update in M^{ei};

- P_3 (*weakest precondition of paths*): let $\langle l'_0, l'_\mathrm{F} \rangle$ denote a set of all feasible paths from the initial location l'_0 to final location (exit point of an advice) l'_F in M^{ei} and $\langle l'_0, l'_\mathrm{F} \rangle_k \in \langle l'_0, l'_\mathrm{F} \rangle$ be k-th path in that set, then $\forall k \in [1, |\langle l'_0, l'_\mathrm{F}\rangle|]$: $\bigwedge_{j\in[1,Length(k)]} wp(\langle l'_0, l'_\mathrm{F}\rangle, l'_\mathrm{F}) \Rightarrow grd(t_i)$, i.e., the weakest pre-condition wp of any path in $\langle l'_0, l'_\mathrm{F}\rangle$ cannot be inconsistent with the guard of the join point carrier edge t_i.

- P_4 (0-*duration unwinding*): $\forall l'_i \in (N_{Mei} \setminus l'_0)$: $Type(l'_i) = \mathit{committed}$, i.e., since the execution of any path in the refinement M^{ei} must be atomic and instantaneous (by Uppaal TA definition), all locations must be *committed* (committed is a location type of Uppaal TA which satisfies the condition $c = 0$ for all clocks c occurring in the invariant of the location).

- P_5 (*non-divergency*): $grd(t_i) \Rightarrow M^e, l'_0 \vDash \mathrm{A}\Diamond\ l'_\mathrm{F}$, i.e. validity of $grd(t_i)$ implies the existence of a feasible path in M^{ei}.

Similarly to location refinement we implement the edge refinement of Uppaal TA by means of synchronous parallel composition $\parallel_{\mathrm{sync}}$ and by defining locations l'_0 and l'_F in M^{ei}, and edges from l'_0 and to l'_F which model entry and exit points to/from the advice model (Figure 3.3).

Except the property P0 (P0’) which can be verified by syntax check, all other properties can be verified by TCTL model checking locally with respect to M^{el} only. This guarantees the compositionality and better scalability of the approach 2.

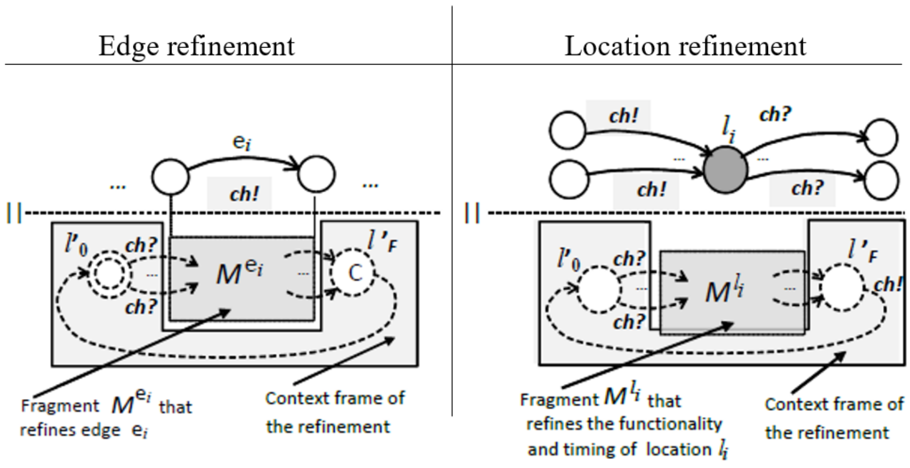

Edge refinement
Location refinement
e_i
...
ch!
...
||
l'_0
ch?
ch?
M^{e_i}
C
l'_F
Fragment M^{e_i} that refines edge e_i
Context frame of the refinement
ch!
l_i
ch?
ch!
ch?
||
l'_0
ch?
ch?
M^{l_i}
l'_F
ch!
Fragment M^{l_i} that refines the functionality and timing of location l_i
Context frame of the refinement

Figure 3.3 Uppaal TA model patterns for edge and location refinement

A refinement-based weaving example is depicted in Figure 3.4 where two aspects *Transaction* and *BalanceCheck* are introduced respectively by location and edge superposition refinements (shown with dashed arrows).

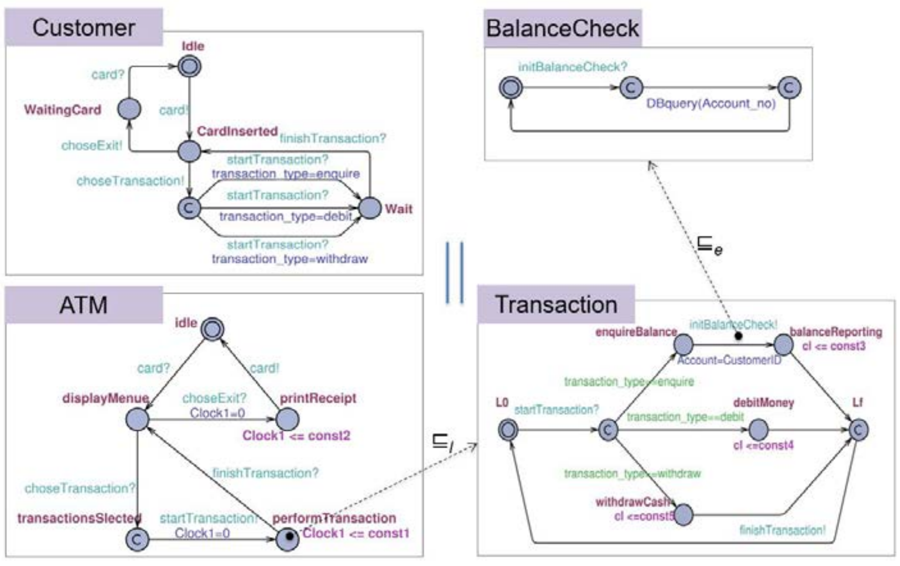


Figure 3.4 AO model of the ATM

3.3 AO test coverage criteria

The augmented test models composed according to the Approach 2 (see Subsections 3.2.3 and 3.2.4 for details) include the structural elements that allow specifying various aspect related structural coverage criteria for AO testing. The semantics and scoping of AO coverage constraints can be defined by following the hierarchy and sub-types of AO model elements.

For specifying the coverage criteria we use the expressions of TCTL with 1st order terms on the alphabet of AO model elements and call these formulas to *coverage expression* (*CE*). Thus, *CE* has hierarchical structure where the AO coverage sub-expressions are concatenated in the following order: *AC*, *JPC*, *APC*, *MEC* where

- *AC* stands for aspect coverage,
- *JPC* – join point coverage,
- *APC* – aspect path coverage,
- *MEC* – advice model element coverage.

Having this ordering of coverage item types, each of the coverage sub-expressions defines the prefix and scope within which the next one has to be interpreted. For instance, if for *JPC* the *AC*-prefix specifies aspects A_1 and A_2 then join points in *JPC* are implicitly assumed to be only those of A_1 and A_2.

Aspect Coverage (*AC*) requires executing all or some aspects in the augmented model at least once. In *Strong Aspect Coverage* (*SAC*), given an aspect model *M*, all advices of the aspects specified in the *CE* must be covered by the tests.

To implement the *SAC* we use the parameterized Uppaal TA templates where the template parameter p_i ranges over indexes [1, *n*] that identify the aspect. Let *P*(*i*) be the predicate symbol assigned value *true* only when the *i*-th aspect advice model is executed. Then the traces of *M* (p_i) that test *SAC* should satisfy the query:

```
E◊ forall (i: int [1,n]) P(i).
```

Recall that given query is valid only for paths that traverse all aspects' advice models. In general, the model *M* may not be connected and a single path including all aspects may not exist. Therefore, we introduce an auxiliary *reset*- transition into *M* that guarantees that if *n* advice models are reachable in *M* then at most with *n* traversals all of them can be visited. The *reset*-transition connects the final location of *M* to its initial location. Due to this construct the Uppaal model checker is able to generate a trace that traverses all advice models.

In case the strong coverage trace appears to be unreasonably long, a test suite with shorter test cases can be achieved by "chopping" that trace at *reset*-transitions to several shorter sub traces.

Weak Aspect Coverage (*WAC*) refers to the case where at least one advice model of some aspect is traversed by the test path. The query

```
E◊ exists (i:int [1,n]) P(i)
```

differs from the *SAC* constraint by existential quantification of advices, therefore only one advice of each aspect is sufficient to be covered and consequently it provides shorter traces as a rule.

Join Point Coverage **(***JPC***)** requires executing all or some join points of the aspects specified in *AC*-prefix of *CE*. *Strong Join Point Coverage* (*SJPC*) may presume similarly to *SAC* the introduction of an auxiliary *reset*- transition into *M*. Regardless the prefix (*SAC* or *WAC*) of the query the *SJPC* defines in the *CE* a conjunct of form

```
... forall (j: int [1,m]) P(i) && R(j),
```

where `j` ranges over the join point indexes of the aspects referred in the *AC*-prefix and `R(j)` is a Boolean variable at each join point updated to *true* whenever this join point is visited. For instance, in the model of Figure 3.4, we can add an assignment `R[j] = true` to join point edge *EnquireBalance*→ *BalanceReporting* that registers the entry into advice "BalanceCheck".

Weak Join Point Coverage (*WJPC*) is satisfied if there is at least one trace for given formula prefix satisfying

```
...exists (j: int [1,m]) P(i) && R(j).
```

Aspect Path Coverage (*APC*) requires executing all or some paths between the entry and exit of the advice join point specified in the *JPC*-prefix. Assume the entry and exit transitions of each advice model are decorated with *entry*(*i, j,k*) and *exit*(*i, j,l*) predicates where *i, j, k, l* range over the set of aspects, join points, and their advice entry and exit points respectively. Whenever the transition is executed, these predicates evaluate to *true*. Then, the *Strong Aspect Path Coverage* (*SAPC*) is specified by the sub formula prefixed with aspect and join point constraints as follows:

... `forall (k: int [1, K]) forall (l: int [1,L]) P(i) && R(j) && [(`$\vee_{k=1,\ K}$ `entry(k))` $\wedge$`(`$\vee_{l=1,L}$ `exit(l))`.

SAPC, like earlier strong coverage criteria, may presume the *reset*-transitions related construct. *Weak Aspect Path Coverage* (WAPC) comparing to *SAPC* replaces universal quantifiers with existential ones for variables `k` and `l`, the coverage constraint becoming

... `exists(k: int[1,K]) exists(l: int[1,L]) P(i) && R(j) && [(`$\vee_{k=1,K}$ `entry(k))` $\wedge$ `(`$\vee_{l=1,L}$ `exit(l))`.

Advice Model Element Coverage (*MEC*) criteria imposes constraints on the types of Uppaal TA elements to be covered in the advice model, e.g. *Strong* (resp. *Weak*) *MEC* can be specified with the Uppaal TA element type, e.g. Transition and universally (resp. existentially) quantified over given type. More specific coverage constraints can be constructed using type discriminating predicates on the data variables of an advice model. For instance, a test that is checking successful completion of Balance Check (example of Figure 3.4) is specified using query

```
E<> exists (i: UserID) i > 510030.
```

The possible combinations of AO coverage subexpressions are shown in the Table 3.1.

Table 3.1 Summary of AO Test coverage criteria

Coverage constraint Type of coverage entity	Strong (universal) coverage $\forall$	Weak (existential) coverage $\exists$	Discriminating predicate
Aspect A	All aspects of the model $\forall A \in \mathbf{A}.$ …	Some aspects of the model $\exists A \in \mathbf{A}.$ …	Predicate on aspect constants /variables
i-th join point $jp(A, i)$	All join points of aspect A $\forall jp(A, i) \in JP(A)$. …	Some join points of aspect A $\exists jp(A, i) \in JP(A)$. …	Pointcut condition
Entry-exit path λ of an advice model $M^{A'}$ $\lambda \in Paths(M^{A'})$	All paths initiated at i-th join point $\forall \lambda \in Paths(M^{A'})$	Some paths initiated at i-th join point $\exists \lambda \in Paths(M^{A'})$	Path predicate, e.g. constraint on path length
Model element of type $\boldsymbol{T}$ (location, transition, function, data, etc) included in the path $\lambda \in Paths(M^{A'})$	All elements of type $\boldsymbol{T}$ in $M^{A'}$	Some elements of type $\boldsymbol{T}$ in $M^{A'}$	Predicate on the attributes of type $\boldsymbol{T}$

3.4 Conclusion

We have shown that the AO model constructs can be introduced in Uppaal TA, using model transformations that are conservative with respect to given model class. Two alternative approaches were studied from the expressiveness point of view of weaving operators. The first, finer grain method was inspired by weaving constructs of AO programming, and second, more abstract, that implements weaving as model superposition refinement operators was introduced by the author. The correctness conditions of weaving were defined in timed computation tree logic TCTL. Regardless of the concrete weaving method and differences in their interpretation, the aspect-oriented test coverage criteria suggested were applicable on AO test models constructed using both ways. It has been shown how the test sequences that satisfy introduced AO coverage criteria are generated from AO models using an Uppaal TCTL model checker.

4 CASE-STUDY: HOME REHABILITATION SYSTEM

4.1 Chapter overview

This chapter presents the case-study that has been explored to evaluate and validate the developed AOM and AO MBT methods.

4.2 Home rehabilitation system general description

According to [75] about 70% of software projects in the medical domain are delayed because of development and testing problems occurring in the application and/or middleware tiers. The main reasons are due to safety critical nature and a non-trivial combination of functional, performance and security features of the medical systems. The Home Rehabilitation System (HRS) developed in [76] has been selected as an example of a medical system where the application of AO MBT can improve the quality of the system design and speed up the development process and/or reduce the need for resources required for testing.

HRS is a personal health monitoring system which collects a patient's health condition information online using sensors attached to the patient's body. HRS drives the sensor devices, analyses the gathered data, interacts with the patient and submits relevant patient information to the hospital through the internet. Figure 4.1 illustrates the placement of an experimental sensor system to measure

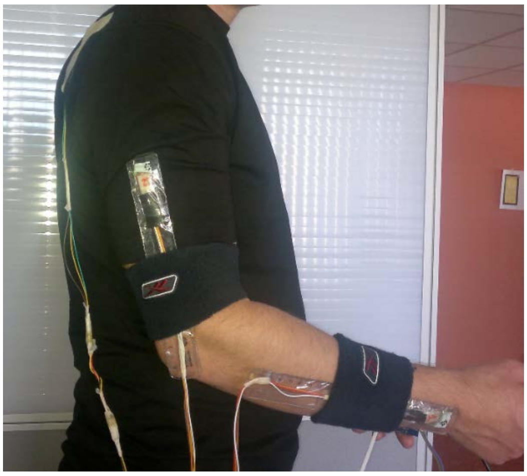


Figure 4.1 The placement of body area sensors of HRS

movement data, SpO2, temperature, heartbeats, blood pressure, blood sugar, and other parameters needed for patient’s online monitoring. HRS software contains the following sub-components:

- a dedicated health hub that operates as a communication gateway;
- a vital signals' sensory system for patient measurements;
- a movement tracking sensor system for fall down detection;
- physical activity and exercise monitors.

HRS can operate in the following modes:

- setting up the treatment plan;
- home exercising;
- passive monitoring of pulse and blood pressure;
- reporting on how well the exercising plan is followed.

In the following, we focus on the “home exercising” use case.

4.3 HRS home exercising use case

Main use scenario “home exercising” of HRS system contains two steps: system preparation in the hospital and home use to monitor exercising sessions. There are three actors involved: Patient, HRS and Doctor, who all are interacting in this use case.

4.3.1 Initial configuring of HRS system

The physiotherapist instructs the Patient and also configures the HRS equipment before the Patient leaves hospital surveillance. During the hospital training sessions, the HH records MEMS based motion monitoring sensor signal patterns of different exercises. Using these the physiotherapist judges whether the signals are correct or incorrect for calibration. If correct, this information is used to train a neural network that provides reference values for further home exercising monitoring. Additionally, the physiotherapist can set up safety limits for heart rate and blood pressure, and activate a safety checking procedure. The safety limits are set from the point of view of motor rehabilitation, and may be overruled by primary care physician requirements during the treatment.

While creating the rehabilitation plan, the physiotherapist first chooses and adjusts exercises for the Patient. The plan is the basis for monitoring the training at home, giving reminding signals and recommendations as well as the basis for the evaluation of the Patient's independent training quality. To specify the treatment plan, the doctor has to compose the list of exercising sessions and set additional constraints. There can be any number of exercising sessions per day, either identical or different. The doctor sets the number of sessions and a starting time for each session, or just a suitable time interval between sessions. To compose a session, the doctor chooses exercises and orders them. The minimum and maximum number of repetitions can also be specified as well as the duration in seconds for every exercise.

Some important details, however, do not fit into the exercising schedule, therefore, must be added as separate rules specifying certain assisting actions and their triggering conditions. For example, the Patient is required to measure blood pressure and heart rate at given times or in relation to exercising sessions. Similarly to measurements, the Patient can be asked to answer questions about pain, stress level, feeling, etc. The exercising plan can be adjusted by these results as shown in Figure 4.2. For example, if the blood pressure is too high before the exercising, then the exercising session will be postponed or conducted with a reduced number of repetitions. The treatment plan can also be adjusted upwards during the home rehabilitation after a certain period of time.

4.3.2 Home use scenario

After the HRS system is configured in the hospital by providing the treatment plan and recorded samples of movements, the HRS is ready to be used at home. The system keeps track of the treatment plan fulfilment. Primary care doctors can insert additional safety rules into the system, e.g. reduce allowed blood pressure or pulse rate.

All Patients' movements are collected while the sensors are turned on and worn. Based on the reference data the sensor raw input stream is divided into segments that correspond to the known movements which are compared to the plan. Within one session, the system monitors if the Patient is doing the exercises as many times as required. Also, the quality indicators with the reference are calculated for individual movements and displayed to the Patient.

For the Patient's safety HRS monitors also if the Patient has fallen down. This is implemented using the central body activity sensor. It reminds user to keep these measurements running.

The event log and statistics that are gathered during the home rehabilitation period, are recorded in the database called Whiteboard and can be uploaded to the hospital information system using telemedicine services accessible via internet.

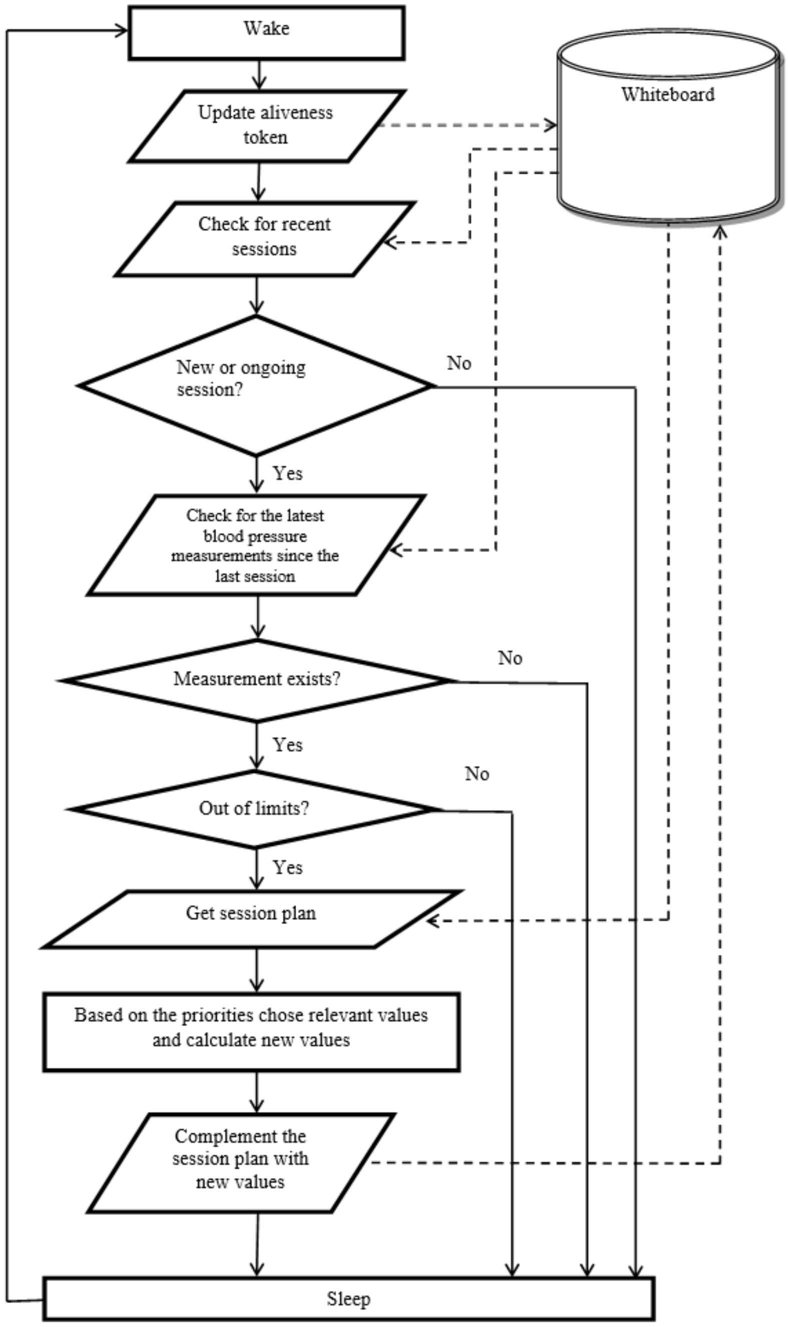

Wake
Update aliveness token
Whiteboard
Check for recent sessions
New or ongoing session?
No
Yes
Check for the latest blood pressure measurements since the last session
Measurement exists?
No
Yes
No
Out of limits?
Yes
Get session plan
Based on the priorities chose relevant values and calculate new values
Complement the session plan with new values
Sleep

Figure 4.2 The treatment plan adjusting software agent

4.4 Aspect-oriented modelling of HRS

We demonstrate the AO modelling of HRS home use scenario on two levels of abstraction: at first, on the requirements level in terms of agents such as Patient, physiotherapist, sensor system, exercise monitoring; and second, on the level of software agents that implement the HRS. The requirements level aspects are directly related to the application concerns such as *Patient's safety, monitoring,* the *Exercising quality* and *Exercising performance.* Software agents level modelling extracts the aspects from implementation entities such as object classes, their attributes and relations. From testing point of view, both of these modelling levels serve their purpose – the first is meant for *system level testing* and the second for software *integration testing,* respectively.

4.4.1 Requirements level model of HRS

We start the modelling with the base model that incorporates requirements level actors and their interactions. Thereafter, the aspects *Patient's safety*, *Exercising quality* and *Exercising performance* are added incrementally by superposition refinement weaving. After weaving the correctness of weaving is verified using model checking of correctness conditions specified in *Definition* 2.5 and *Definition* 2.6. The base model, aspects and corresponding to them Uppaal TA model templates are listed in Table 4.1.

Table 4.1 Model templates by use case actors and aspects

Use case actor / Aspect	Physiotherapist	Patient	HRS
Base model	*Doctor*	*_physical_ condition, _exercising*	*_sampler*
Aspect 1: Patient's safety	*Emergency_ consulting*	*_exercising* (refined)	*_posture_sensor, _emergency_monitor*
Aspect 2: Exercising quality	*Adjusting reference values*	*_physical_condition1 _exercising1*	*_quality_ monitor*
Aspect 3: Exercising performance	*Adjusting performance settings*	*_exercising1* (refined)	*_performance_monitor*

4.4.1.1 Base model (base functionality)

The base model represents the interaction of the three main entities of the application: the Physiotherapist, Patient and HRS on a high level of abstraction. The Uppaal TA templates *Doctor*, *Patient*, and *HRS* model them respectively. The *Doctor* triggers the training session by initiating the output action *Exercise* which represents acknowledging the training plan and enabling the session to start. The *Patient* starts *exercising* by the plan according to which the exercising session can last from *session_Tlb* to *session_Tub* time units provided the Patient's physical condition is *Normal*.

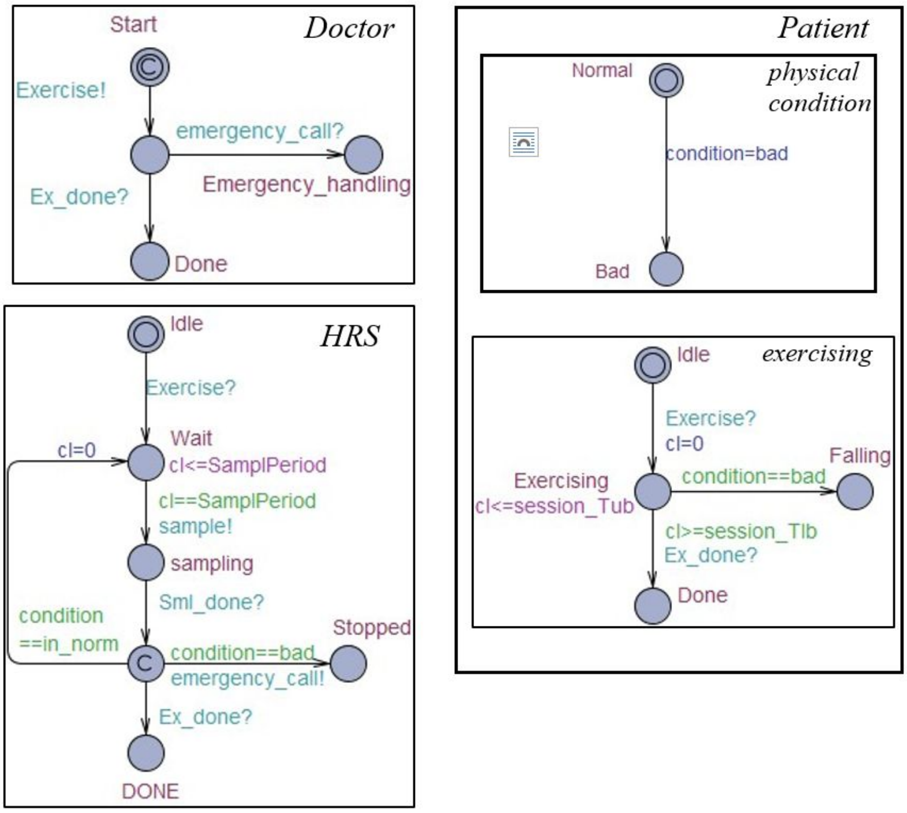


Figure 4.3 Base model of the HRS home use scenario

The occurrence of abnormal activity, e.g. falling down, is represented with edge *Normal*→*Bad* in the template *Patient physical condition* and communicated to HRS via global variable *condition* update *condition* = *bad*. The execution of edge updates the variable *condition* to bad which is also the guard condition of edge *Exercising*→*Falling* of template *Patient_exercising*.

The *HRS* activation is also synchronised with *exercising* via channel *Exercise* to initiate the patient's physiological data sampling at the same time when the exercising starts. The patient's measurement data is sampled periodically with period *SamplPeriod* until the signal *Ex_done* is received or patient's health condition turns to *Bad*. The first is synchronized with the patient's exercising completion by executing edge *Exercising*→*Done*.

4.4.1.2 Aspect 1: Patient's safety

The advice, needed for transforming the base model introduced in 4.4.1.1 to the Patient's safety aspect model, is represented by templates *HRS_posture_sensor*, *HRS_emergency monitor* (Figure 4.4)

The *HRS_posture_sensor* is woven with base model using location refinement of the location *sampling* in *HRS* and the advice *HRS_emergency monitor* is woven by edge refinement of the edge *sampling*→ - in the template *HRS_sampler*. The template *HRS* of the base model is renamed to *HRS_sampler* here. The full safety aspect model is exposed in Figure 4.4. Notice that due to the refinement weaving, the guard condition *condition ==bad* of the transition _→*Stopped* in the template *HRS_sampler* is refined with predicate *emergency* which represents the emergency signalling condition.

The advice model *HRS_posture_sensor* represents the behaviour of the sensor that detects falling of the Patient. Body posture measurement takes constant time - one Tick and if the Patient falls, modelled with state constraint *condition ==bad* then *HRS_posture_sensor* assigns to its output variable *posture* new value *down*.

HRS_emergency_monitor samples the value of variable *posture* and when the value *down* is read, its output variable *emergency* is updated to *true.* This, in turn, triggers the emergency call by *HRS_ sampler* which is responded by Doctor's reaction modelled as transition to location *Emergency_handling*.

Advice *Emergency_consulting* (referred in Table 4.1) which refines the Doctor's activities after receiving the *emergency_call* is omitted from this example since it does not concern the functionality of HRS directly and can be abstracted away from current use case.

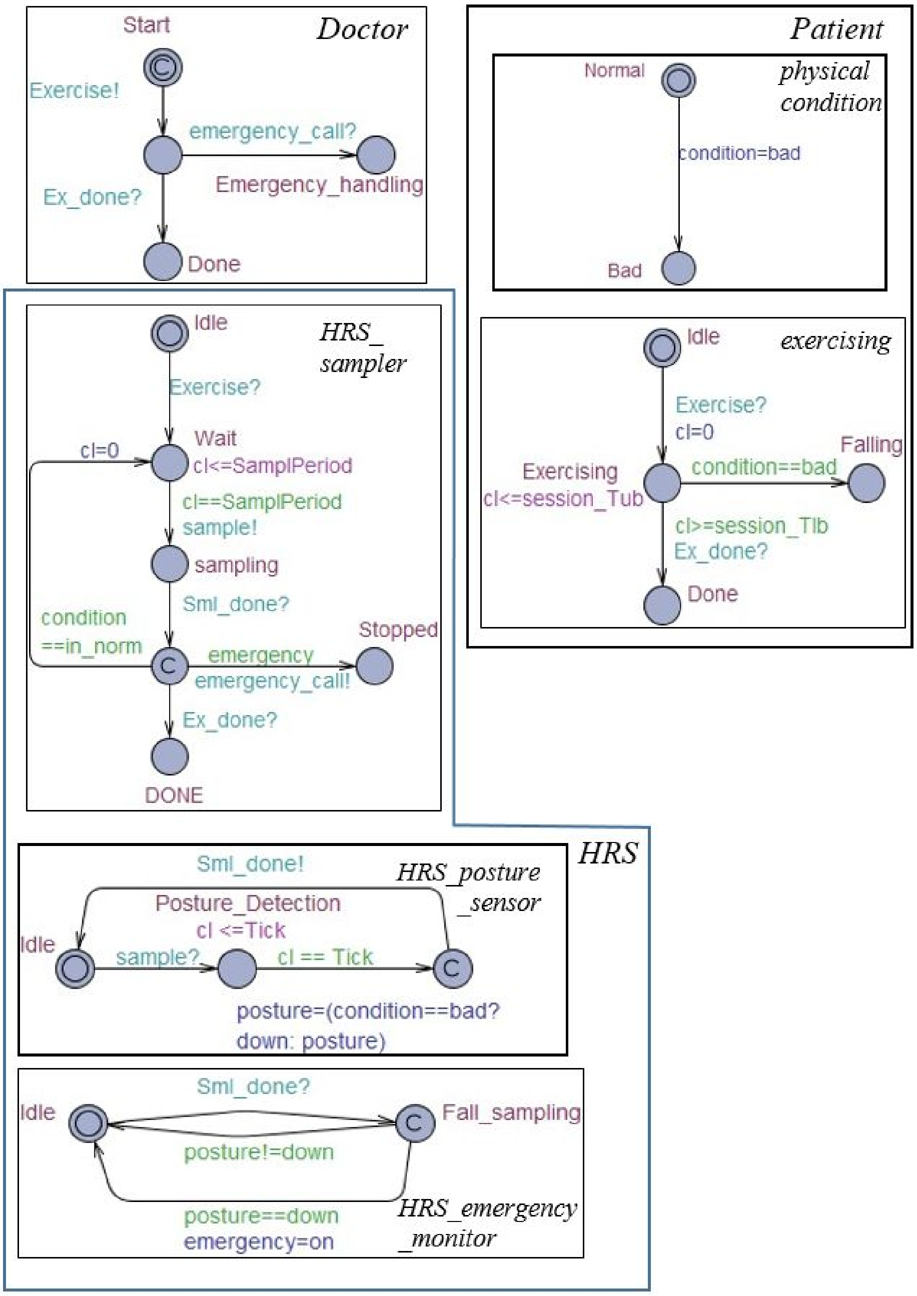


Figure 4.4 Safety aspect model

4.4.1.3 Aspect 2: Exercising quality

The *exercising quality* aspect specifies how HRS must react to the changes of the Patient's biometric characteristics during exercising and how the deviations from nominal values are signalled back to the Patient.

Patent's biometric characteristics' deviations are modelled by introducing three new (sub) states *normal2*, *better* and *worse* which refine the base model's template *Patient_physical _condition* coarser state *normal* (Figure 4.5). The aspect advice introduces the template *Patient_physical _condition1* which is the refinement of the location *Normal* in the template *Patient_physical _condition*. This refinement is necessary because HRS has to react also to the deviations that are not critical (i.e., they are not sub-states of the *Bad* state) but still need special handling to assure the quality of exercising.

To get the quantitative modelling of Patient's physiological condition *M* numeric values in the ranges *val_N*, *val_B*, *val_W* are generated periodically once in *Tick* period. The value regions *val_N*, *val_B*, *val_W* of body characteristics correspond to the states *Normal2*, *Better*, and *Worse* respectively. The values of body characteristics are generated in the model dynamically by self-loops attached to theses states in the template *Patient_physical _condition1*. One can implement these self-loops also in separate advice template and weave them via join point locations *Normal2*, *Better*, *Worse*. For compactness reasons to avoid too many weaving steps we introduce them at once in the template *Patient_physical _condition1*. Generating numeric values in the state *Bad2* is omitted since the body characteristic values that correspond to an emergency situation do not concern the training use case.

Patient_exercising template is also refined in the advice model using location *Exercising* refinement with template *Patient_exercising1*. This is due to the need to introduce the exercising quality indicators such as *exercising_counter* and the potential duration of performing an exercise – time interval [*Ex_lb*, *Ex_ub*].

HRS_sampler template is refined with advice template *HRS_quality_monitor* and it models reactions of the HRS when Patient sampling data deviate from the nominal values specified by a physiotherapist as an interval [*L_bound*, *U_bound*]. These boundaries are defined for *M* different body characteristics which are sampled. The training online guidance is performed by indicating qualitative values *LB_warning*, *normal*, and *UB_warning* which are exposed on the HRS user interface screen (in the model global variable *Screen* is updated with these values).

Advice *Emergency_consulting* (referred in Table 4.1) which refines the Doctor's activities after receiving the *emergency_call* is omitted from the thesis since it does not concern the functionality of HRS and can be ignored from AOT point of view.

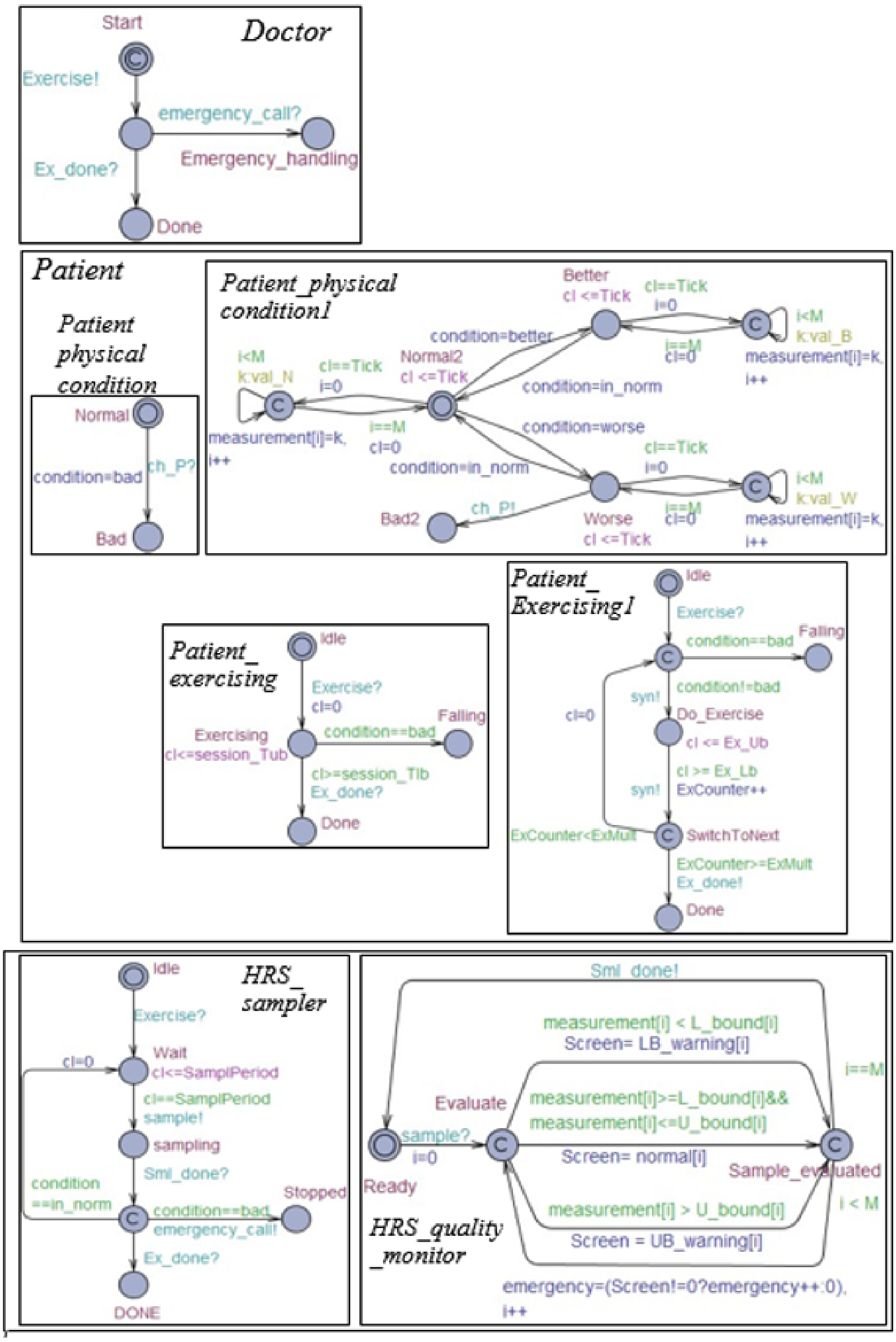


Figure 4.5 Exercising quality aspect model

4.4.1.4 Aspect 3: Exercising performance

Last aspect of the exercising use case is *exercising performance*, i.e. the speed of performing exercises (Figure 4.6). As for other body characteristics there are also prescribed limits for the duration of how long performing of each exercise should take.

Performance_monitor is started by *HRS_sampler* and it measures the speed of performing an exercise (variable *Tmax* shows how many times in one sampling cycle. *Performance_monitor* measures the speed of performing exercises by synchronizing its local clock *stopwatch* via channel *syn* with the beginning and ending events of an exercise. The exercise performing is modelled with location *Do_Exercise* in *Patient_exercising1*. Depending on the measurement result the duration measurement is sent to user interface using HRS output variable *SpeedScreen*. If the speed is too slow then symbolic value *Too_slow* is shown on screen, if too fast the value *Too_fast* is indicated and if the duration is within norm value *Nominal* is exposed. Template *Doctor_adjusting_performance_settings* is omitted in this advice model since it is tested under HRS different use case.

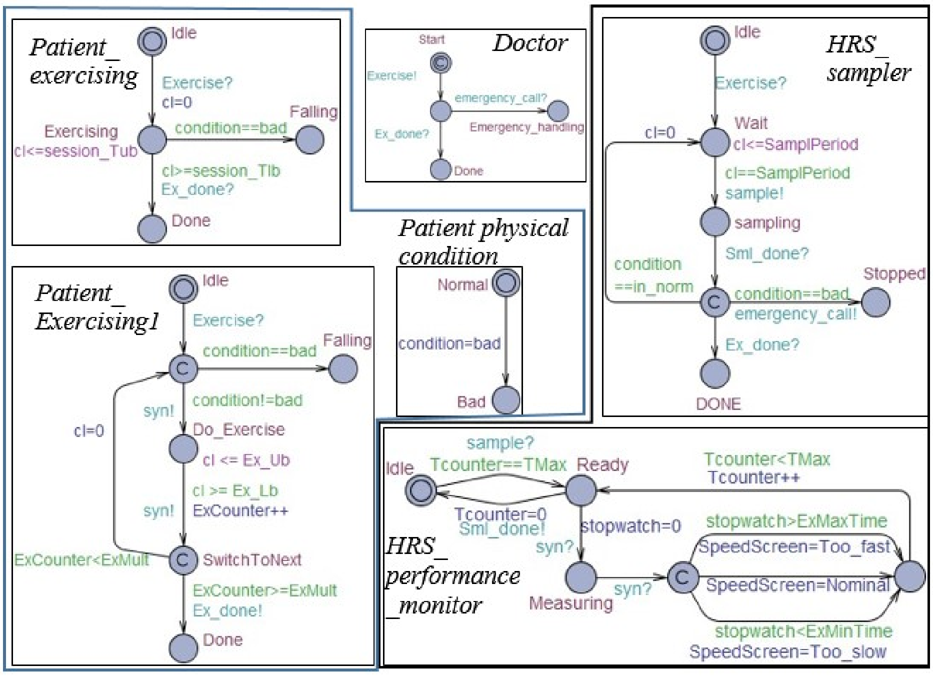


Figure 4.6 Exercising performance aspect model

4.4.1.5 The correctness of AO models and generated tests

The system under test in this case study is HRS and it has test interfaces with two actors *Doctor* and *Patient* who constitute the environment of HRS. The environment, behaviour scenarios define the test cases. In the following, we prove by using the Uppaal model checker that:

(1) The base model is correct and guarantees the reachability of all control locations;

(2) The aspect models are constructed correctly from the base model;

(3) The AO coverage criteria introduced by aspects are *feasible*, i.e. there exists traces satisfying the coverage criteria, and the traces (mapped to test sequences) are optimal either in terms of sequence length or in terms of their execution time;

(4) The AO test sequences generated are shorter than non-AO ones and their generation complexity is lower in average.

For that we verify the correctness of test models and introduce the test coverage criteria by aspects specified above.

(1) *Verification of test models*

Base model. The base model is verified against three properties that yield the reachability correctness stated in (1) above.

Property *BM1*: If the *Patient_physical _condition* is in state *Normal* then the process *Patient_exercising* terminates successfully in the location *Patient_exercising.Done* and automaton *HRS_sampler* terminates in correct final state modelled with location *HRS_sampler.Done*. The TCTL query is:

```
Doctor.Start && Patient_physical_condition.Normal &&
Patient_exercising.Idle&&HRS_sampler.Idle --> Q_clock >=
Patient_exercising.Ex_Ub * ExMult && Patient_exercising.Done
&& HRS_sampler.Done.
```

In this query an auxiliary formula clock `Q_clock` is used to be compared with the time bound `Patient_exercising.Ex_Ub * ExMult` when the exercising has to be finished at latest.

The model checking report in Appendix F.1 BM1 confirms that the property is satisfied.

Property BM2: If Patient's physical condition turns bad (*Patient_physical_condition* reaches location *Patient_physical_condition.Bad*) then Patient's exercising stops in the location *Patient_exercising.Falling* after *Ex_Ub* time units latest and *HRS_sampler* makes an *emergency_call* during *SamplePeriod* time units after that event and moves to the state *HRS_sampler.Alert*. These requirements are model checked using queries BM2.1 and BM2.2 while the initial location of *Patient_physical_condition* is set to *Bad*:

Property BM2.1:

```
A<> Patient_physical_condition.Bad && Patient_exercising. Falling
&& Q_clock <= Patient_exercising.Ex_Ub
```

Property is satisfied (see Appendix F.2).

Property BM2.2:

```
A<> Patient_physical_condition.Bad && HRS_sampler.Alert && Q_clock
<= HRS_sampler.SamplPeriod
```

Property is satisfied (see Appendix F.3).

Note, that for referring to the time interval between executing *Normal→Bad* in the template *Patient_physical_condition* and reaching locations *Patient_exercising.Falling and HRS_sampler.Alert* when the *emergency_call* by *HRS_sampler* is done, we apply again the global property clock *Q_clock* which is compared with the given upper time bounds.

Property BM3:

To prove that HRS completes a sufficient number of samplings specified with parameter *SmplMult* during an exercising session and provided the session is not interrupted due to the Patient's emergency condition, we verify that when ending the sampling in the location *HRS_sampler.Done* then condition ExCounter >= *SmplMult* is satisfied.

```
A[] HRS_sampler.Done && Patient_exercising.Done imply
                S_counter >= SmplMult
```

Property is satisfied (see Appendix F.4).

(2) ***Verification of the aspect models' weaving correctness***

In the following we prove properties *P*1-*P*3 (given by Definition 2.5) in the base model join points where location refinement is used for advice weaving and properties *P*1' and *P*3-*P*5 (given by Definition 2.6) in the base model join points where edge refinement is used.

Weaving correctness of Aspect 1: Patient's safety

HRS_posture_sensor is woven using location refinement and properties P1-P3 have to be verified.

Property P1: *interference free new updates*. There is an update of only one variable *posture* in the template *HRS_posture_sensor*. Since this variable does not occur in the base model (it is called *fresh variable*) this yields interference freedom with the base model.

Property P2: *preservation of non-blocking*. Once started via channel *sample* the *HRS_posture_sensor* always returns to the location *Idle* after exactly *Tick* time units (verified by simple visual inspection).

Property P3: *non-divergence*. The property holds since *HRS_posture_sensor* always returns control to the base model after *Tick* time units while the join point carrier location *Sampling* has invariant *cl* <= *SamplPeriod* which yields that for non-divergence the condition *Tick* ≤ *SamplPeriod* must be satisfied.

HRS_emergency _monitor is woven using edge refinement and it is activated via broadcast channel *sml_done* simultaneously with *HRS_posture_sensor* returning the control to the *HRS_sampler* (verified by simple visual inspection).

Property P1': *interference free new updates*. There is an update of only fresh variable *emergency* in the template *HRS_emergency _monitor*.

Property P2: *weakest precondition of paths*. The property is satisfied since two alternative paths exist in the template and since the guard conditions of the paths are mutually exclusive, exactly one of them is enabled any time the location *Fall_sampling* is reached and thus, internal blocking within the template never occurs, except in the location *Idle* of weaving context frame.

Property P3: 0-*duration unwinding*. *HRS_emergency_monitor* has exactly one location (except for the context frame location *Idle*) which is of type committed.

Property P4: *non-divergence*. The internal guard conditions *posture*!=*down* and *posture*==*down* are not contradicting the guard condition *cl*==*SamplPeriod* of the refined edge *Sampling*→ -.

Weaving correctness of Aspect 2: Exercising quality

The base model template *Patient_physical_condition* is woven with advice model template *Patient_physical_condition2* using location refinement where join point carrier is location *Normal* and another base model template *HRS_sampler* is woven with advice model template *HRS_quality_monitor* via join point carrier location *Sampling*.

Advice ***Patient_physical_condition1*** generates concrete parameter values which can occur in the Patient's *Normal* state and which are monitored by HRS. The edge *Worse*→*Bad2* exiting advice is synchronized with the edge *Normal*→*Bad* in the base model template *Patient_physical_condition.* The activation edge typical to the location refinement context frame pattern is substituted with committed initial location and outgoing from it three edges to locations *Worse*, *Normal*, *Better* in the advice.

Property P1: *interference free new updates*. All variables except *condition* in *Patient_physical_condition1* are fresh variables and thus satisfy the correctness property P1. Variable *condition* is updated with new values which are data refinements of value *normal*. The symbolic value *normal* is the default value of variable *condition* in the base model initial location *Normal*. Since the valuations of *condition* in the *Patient_physical_condition1* do not interfere with the update *condition=bad* in the base model *Patient_physical_condition* the property P1 is satisfied.

<u>Property P2</u>: *preservation of non-blocking*. The only deadlocking location in the template *Patient_physical_condition1* is the final state *Bad2*. Model checking query below verifies that both deadlocks in *Patient_physical_condition1* and in *Patient_physical_condition* are reachable in the same global state. Satisfaction of this property is granted by construction, i.e. the synchronization *ch_P* between edges *Normal*→*Bad* and Worse→*Bad2*.

```
A[]   Patient_physical_condition_1.Bad2   imply   Patient_
physical_condition.Bad
```

Property is satisfied (see Appendix F.5).

<u>Property P3</u>: *non-divergence*. Since the exit from location *Normal* in the template *Patient_physical_condition* is not obligatory by the semantics of Uppaal TA there is no obligation for *Patient_physical_condition1* to have *Bad2* reachable in all traces. Suffices only to prove the existence of such a finite trace. This is done by query in Appendix F.6.

Note that since location *Normal* in *Patient_physical_condition* does not have upper bound in time invariant there is no obligation to have it also in the reachability condition of *Patient_physical_condition1.Bad2*.

```
E<> Patient_physical_condition_1.Bad2
```

Property is satisfied (see Appendix F.6).

HRS_quality_monitor

Advice template *HRS_quality_monitor* is depicted in Figure 4.5.

<u>Property P1</u>: *interference free new updates*. The only updated variables *Screen* and *emergency* in *HRS_quality_monitor* are fresh variables which satisfy the correctness property P1.

<u>Property P2</u>: *preservation of non-blocking*. Deadlock freeness of *HRS_quality_monitor* is proved by showing that the initial state *Ready* is always reachable after reaching the refinement carrier location *HRS_sampler.Sampling* and sampling the sensor values by *HRS_quality_monitor*. To distinguish two consecutive visits of location *HRS_quality_monitor.Ready*, an additional condition $i > 0$ is conjoined with the location predicate to specify the visit after sampling.

```
HRS_sampler.Sampling --> HRS_quality_monitor.Ready &&
HRS_quality_monitor.i > 0
```

Property is satisfied (see Appendix F.7).

Property P3: *non-divergence.* By proving P2 it is shown that exit point of the advice is always reachable and since all locations (except the location *Ready* of the context frame) of the advice template *HRS_quality_monitor* are of type *committed* the reachability is without delays. This satisfies an invariant *true* of the refinement carrier location *HRS_Sampling*, meaning that P2 yields also the validity of P3.

Weaving correctness of Aspect 3: Exercising performance

HRS_performance_monitor

The advice template *HRS_performance_monitor* is woven with the base model template *HRS_sampler* via refining location *Sampling* (Figure 4.5). The weaving is correct if properties P1-P3 of Definition 2.5 are satisfied.

Property P1: *interference free new updates*. Both variables *SpeedScreen* and *Tcounter* updated in the template *HRS_performance_monitor* are fresh variables. This guarantees that the correctness property P1 is satisfied.

Property P2: *preservation of non-blocking*. We show that *HRS_performance_monitor* does not introduce deadlocks, i.e. *Tmax* measurement cycles are completed if the Patient's physical condition is in state *Normal*. The query is depicted in Appendix F.8.

HRS_performance_monitor.Idle && Patient_physical_condition.Normal --> HRS_performance_monitor.Tcounter == HRS_performance_monitor.Tmax

Property P3: *non-divergence.* To prove the non-diverging execution of *HRS_performance_monitor* we verify that while *Patient* is exercising then the *HRS_performance_monitor* if started from location *Measuring*, then it always terminates in the location *Done* within *Ex_Ub* + *PrepT* time units (proof statistics are depicted in Appendix F.9). Here *Ex_Ub* and *PrepT* denote the duration upper bound of performing an exercise and the time interval between two consecutive exercises respectively:

Patient_exercising.Exercising && HRS_performance_monitor.Measuring --> HRS_performance_monitor.Done && Q_clock <= Ex_Ub + PrepT

Summary of AO model verification effort. The statistics of weaving correctness verification effort shown in Table 4.2 confirm that in the case of practical applications of HRS size not extensive computational resources are needed for AOM and its verification. Due to the compositionality of superposition refinement weaving operators the correctness of augmented AO model is a direct consequence of single weaving correctness conditions. For showing weaving correctness we proved by model checking that the advices do not violate the constraints of weaving join points in the base model.

Table 4.2 Aspect models correctness verification resources

Model	Property/ Reference in Appendix F	Verification time (msec)	Elapsed Time (msec)	Resident memory (KB)	Virtual memory (KB)
Base model	BM1/ F.1	141	131	7536	27096
	BM2.1/ F.2	0	0	9008	46780
	BM2.2/ F.3	0	15	8996	46768
	BM3/ F.4	15	15	7632	27356
Aspect 1: safety					
- *HRS_posture_ Sensor*	P1/*in text*	-	-	-	-
	P2/ *in text*	-	-	-	-
	P3/ *in text*	-	-	-	-
- *HRS_emergency _monitor*	P1'/ *in text*	-	-	-	-
	P3/ *in text*	-	-	-	-
	P4/ *in text*	-	-	-	-
	P5/ *in text*	-	-	-	-
Aspect 2: quality					
- *Patient_physical_condition2*	P1/ *in text*				
	P2/ F.5	15	20	8160	28308
	P3/ F.6	16	16	8152	28428
- *HRS_quality_ monitor*	P1/ *in text*				
	P2/ F.7	3984	4126	33056	77508
	P3/ *in text*				
Aspect 3: performance					
- *HRS_performance_monitor*	P1/ *in text*				
	P2/ F.8	0	16	7564	27336
	P3/ F.9	16	15	7452	26852

4.5 Fully augmented model of the HRS

To compare the processor time and memory consumption required for test generation in case of AO and in case of non-AO models we introduce in addition to aspect models presented in sub-sections 4.4.1.1 - 4.4.1.4 the fully augmented model with all aspects involved (see Figure 4.7). Since this model is bisimilar to the monolithic non-aspect model and does not carry the overhead typical of the AO modelling Approach 1, we can use it in the role of monolithic non-aspect model of the HRS (as reference case) for evaluation of AOM feasibility and efficiency.

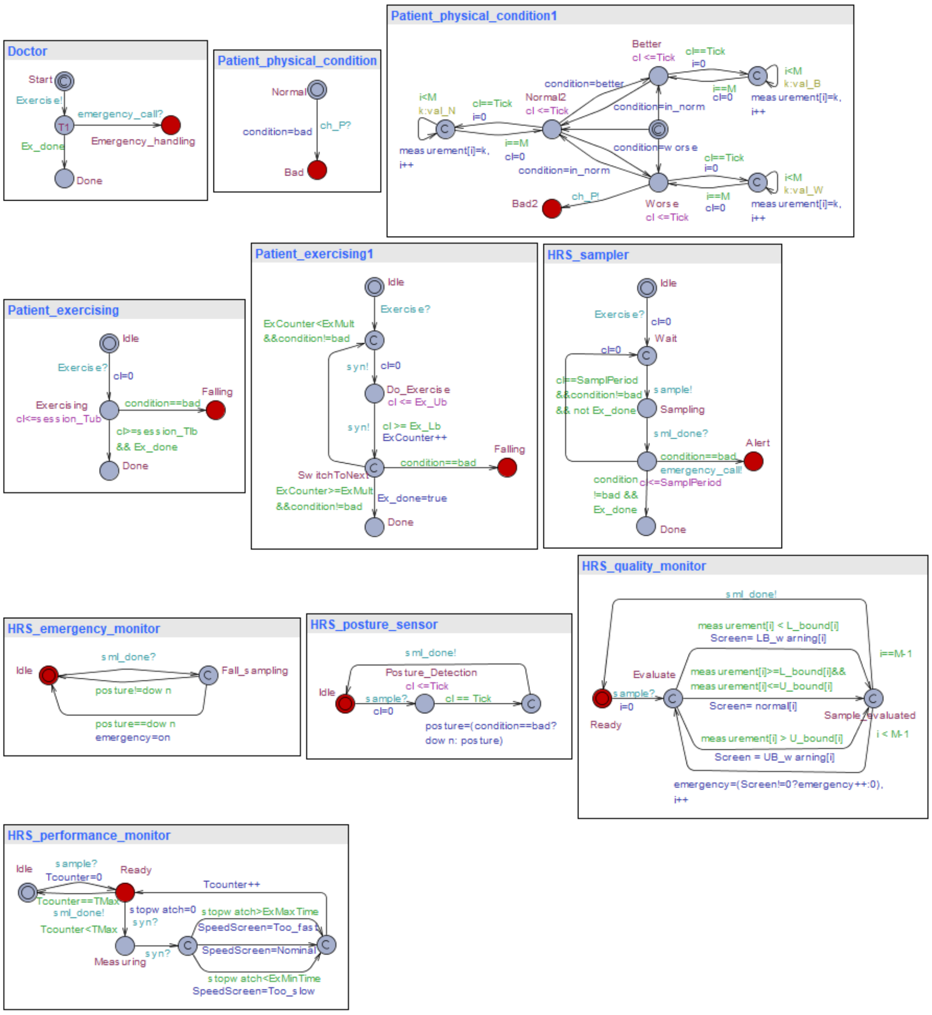


Figure 4.7 Full monolithic model of HRS

4.6 AO test generation

In this subsection we demonstrate how the tests that satisfy AO coverage criteria summarized in Table 3.1 are specified and generated for HRS. To validate the usability of the AOM and AOT methods proposed in thesis we focus on the finest strong coverage criteria, namely *Strong Model Element Coverage - SMEC* which presumes (*i*) specifying the aspect specific model elements such as locations, edges and their attributes in aspect advice models and (*ii*) require a most resource demanding search by model checker.

4.6.1 AO Tests of Aspect 1: Patient safety

In the Patient's safety aspect model, there are two advices represented by templates *HRS_posture_sensor* and *HRS_emergency monitor* woven to the base model (Figure 4.4). We define the strong coverage of both advice template attributes as in Figure 4.8 where *Trap1*, *Trap2* and *Trap3* are auxiliary Boolean variables that allow referring to the coverage items of the edge *HRS_posture_sensor*.-⟶ *HRS_posture_sensor.Idle* and to two alternative edges: *Trap2* labels *HRS_emergency_monitor.Fall_sampling* → *HRS_emergency_monitor.Idle* satisfying guard *posture==down* and *Trap3* labels an alternative edge *HRS_emergency_monitor.Fall_sampling* → *HRS_emergency_monitor.Idle* executed when the guard *posture != down* is true. These trap variables updated to *true* (encoded with numeric value 1) are added to the edge assignments (Appendix F.10).

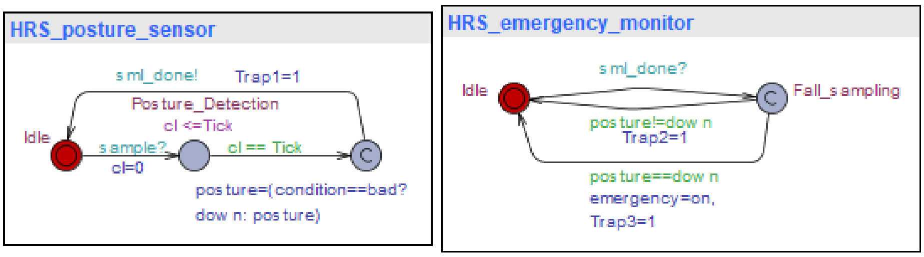


Figure 4.8 Test coverage items labelled with Boolean assignments Trap1÷Trap3

The shortest simulation trace that satisfies this coverage criteria on the aspect model is depicted in Appendix F.11 and the shortest trace of non-aspect model is depicted in Appendix F.12. The traces are presented in the format the Uppaal simulator visualizes them. The validity of query

```
E<> Trap1 && Trap2 && Trap3
```

is shown for AO model and for non-AO model in Appendix F.10 a) and b) respectively.

4.6.2 AO Tests of Aspect 2: Patient exercising quality

In the aspect model *Patient exercising quality*, the advice templates *Patient_physical_condition1*, *Patient_exercising1* and *HRS_quality_monitor* are woven to the base model shown in Figure 4.5. We define the strong coverage for elements of advice template *Patient_exercising1* to capture HRS reactions to Patient's monitored physiological states. For that we label all the edges of *Patient_physical_condition1* which depart from sub-states of *Normal* and the edge that enters the state *Bad*. The traps used for this are *Trap4* ÷ *Trap9* (Figure 4.9).

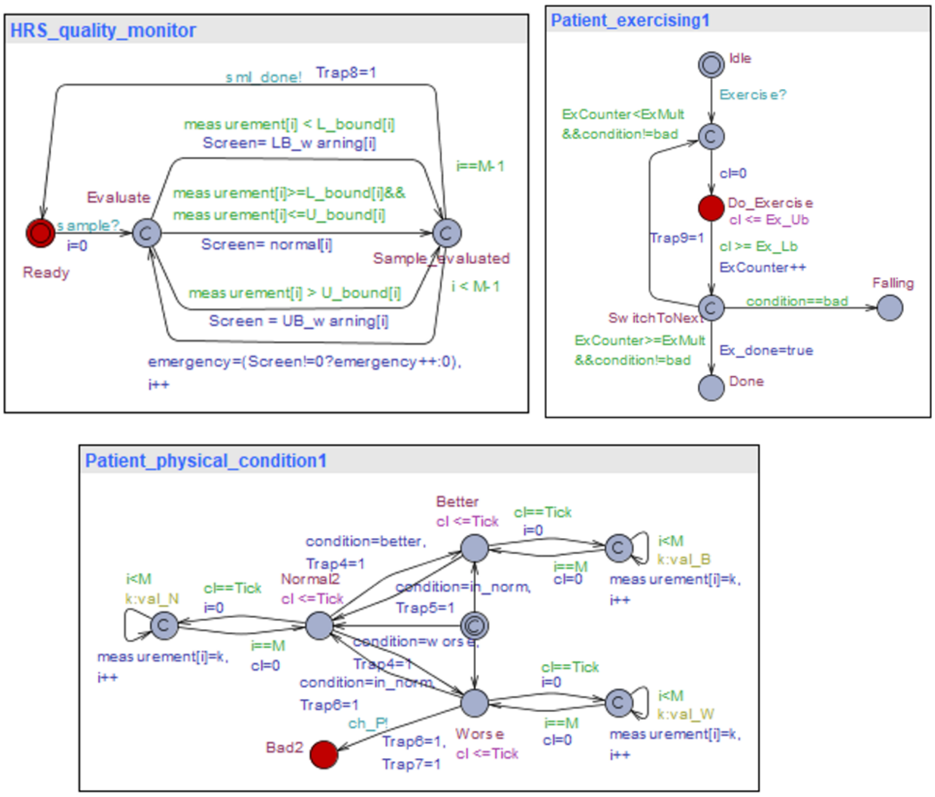


Figure 4.9 Labelling of exercising quality advice templates with traps

The query *E<> Trap4 && Trap5 && Trap6 && Trap7 && Trap8 && Trap9* is executed with statistics shown in Appendix F.13 and the simulation traces of AO model and non-AO model respectively in Appendix F.14 and Appendix F.15.

4.6.3 AO Tests of Aspect 3: Patient exercising performance

The advice template *Patient_exercising1* and *HRS_performance_monitor* woven with base model depicted in Figure 4.6 constitute the patient's exercising performance aspect model. To cover all alternative reaction of HRS in case of exercise duration deviations we label the edges of *HRS_performance_monitor* as shown in Figure 4.10 and prove that all of them are reachable in one test sequence that satisfies query *E<> Trap10 && Trap11 && Trap12 && Trap 13* (see Appendix F.16).

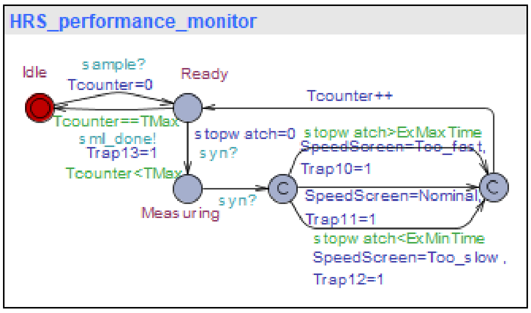


Figure 4.10 Labelling of the exercising performance advice template with traps

Summary of AO and non-AO test generation statistics

Regardless of the relatively small number of requirements level tests it can be seen from the Table 4.3 that the time needed for generating tests on HRS AO model is considerably (on average 364 msec) shorter than that of generating tests with the same coverage on non-AO model. The difference is even more apparent in memory usage, the difference is respectively 4715 KB of resident memory and 6321 KB of virtual memory in average. Though, the length of shortest test sequences differs relatively little – the traces of non-AO model are 1-2 steps longer than AO models ones, this means that there is relatively little overhead in both models. The major disproportion is in terms of time and memory consumption needed for model checking and this is because of the overhead in non-aspect models which is not needed in testing the aspect coverage criteria and therefore can be discarded in the aspect models.

Table 4.3 Summary of test generation effort on AO model compared with non-AO model

Aspect	Verification time (msec)		Elapsed Time (msec)		Resident memory (KB)		Virtual memory (KB)		Test length (no of transitions)	
Test\model type	AOM	NOM	AOM	NOM	AOM	NOM	AOM	NOM	AOM	NOM
Aspect 1: Safety *E<> Trap1 && Trap2 && Trap3*	1	0	0	2	27420	28776	7576	7300	14	15
Aspect 2: Quality *E<> Trap4 && … && Trap9*	0	47	16	47	8148	7956	28396	27724	16	18
Aspect 3: Performance E<>Trap10 &&…&& Trap13	0	1047	0	1272	9436	22416	30996	50908	28	29
Mean signed difference (MSD)	-364,33		-435		-4714,66		-6321,33		-1,33	

Note: In the table 4.3 the following notations are used:

AOM – aspect-oriented model; NOM – non-aspect oriented model;

Value 0 in the table means duration that is less than 1 ms;

Mean signed difference is calculated by formula MSD = $\sum_i^n \frac{p_i^{AO} - p_i^{NO}}{n}$, where i ranges over indexes of aspects, n is total number of aspects for which the comparison is performed, and p_i denotes the characteristic compared in AO and non-AO models.

4.7 Software agents level modelling of HRS aspects

One of the main HRS software design principles is keeping the agents of HRS independent as much as possible so that they can be activated and suspended independently from each other. Otherwise, the dependencies would disable certain functionalities and invalidate running workflows easily. Another key issue of the HRS software system is its scalability when new functionalities and agents need to be added. The core component of the HRS software implementation is a whiteboard memory database (WMDB). The number of agents that are executed in parallel without noticeable interference depends on the performance of the WMDB.

A proposed aspect-oriented design solution is implemented and tested on an off-the-self Linux running handheld (HH) device with a sub-gigahertz ARM processor. On the given platform, writing 1500 rows of bulk data into the whiteboard takes 100 ms and writing 100 rows of data takes 15ms. A long term average reading time of one row is around 100 ms. Taking into account the practical real-time requirements of the HRS – response time below 1 second, makes it possible to execute 100-200 software agents in parallel. Here, we assume that agents are performing computationally relatively simple tasks and each are using only a small number of rows in the WMBD.

4.7.1 Model of the HRS software

The full monolithic model of HRS software includes altogether 98 process of 26 templates. Due to the space limit of thesis the description of model templates is presented in the Appendix E. Verifying and generating tests from this large model by model checking is clearly out of the human comprehension and capabilities of explicit state model checkers. We demonstrate that by aspect-oriented approach the verification and test generation become a practical task and can be solved even using a standard laptop (Intel(R) Core(TM) i7-4600U CPU 2,1 GHz, RAM 8 GB and 64-bit OS) in reasonable time.

The templates are grouped according to system architecture into three groups:

Templates that model *Agents* are following: *data_controller*, *pressure_checker*, *heartbeat_checker, telefon_agent, get_pressure, get_heartbeat, database_cleaner*.

The templates that model *Whiteboard* are *wb_insert_triples, wb_get_buffered_triples, wb_get_selected_triples, wb_delete_selected_triples, wb_sequence_for_ids*, *wb_sequence_for_keys*.

The templates that model *Database*: *wg_start_read, wg_start_write, wg_end_read, wg_end_write, wg_create_record, wg_delete_record, wg_get_first_record, wg_get_next_record, wg_get_field, wg_set_field, wg_make_query, wg_fetch.*

We focus on generating tests for 3 aspects:

- data completeness of DB read-write protocol;
- cleaning the DB by cleaner agent; and
- ensuring the uniqueness of data keys.

4.7.2 Aspect 1: DB data completeness in read-write protocol

Two agents *Pressure Checker* and *Data Controller* are running in parallel where one is writing and the other is reading from the database. The purpose of tests of this aspect is to check if the saved or read data are complete and not corrupted in this process.

The model of Aspect 1 is a composition of the following templates (templates are given in Appendix E):

pressure_checker, data_controller, get_pressure, wb_insert_triples wb_sequence_for_keys, wb_insert_triples, wb_get_buffered_triples, wg_start_write, wg_end_write, wg_set_field, wg_create_record.

4.7.2.1 Test cases

In the following we present the queries that specify the coverage criteria of tests and the results of tests that are derived from these model checking queries.

Test 1

Goal: check if there is any state where data in the DB is incomplete.

Query: *E<> system_data_controller.ERROR*

Test result: *Failed*

Test 2

Goal: check if there is any deadlock stated except the final state.

Query*: E<> deadlock && !Test_1.EndState*

Test result*: Failed*

Test 3

Goal: check if there exists a deadlock then it occurs only in the EndState.

Query: *A[] deadlock implies Test_1.EndState*

Test result: *Passed*

Test 4

Goal: Do all computations reach the deadlock eventually?

Query: *A<> deadlock*

Test result: *Passed*

4.7.3 Aspect 2: Cleaning the DB by cleaner agent

The purpose of testing Aspect 2 is to make sure that DB cleaning functionality is implemented correctly, i.e. that only those data that are meant to be deleted will be actually deleted. To run the tests it is assumed that DB may include two types of data – those that have to be deleted and the others that have to be kept untouched. The data items are referred in DB by the *sequence_for_ids* and by the *sequence_for_keys*. In the testing process the agent *Database Cleaner* is executed two times: at first, when it has to delete all references to the data, and second, when the data itself are deleted.

The model of Aspect 2 is a composition (weaving results) of following templates (templates are given in Appendix E):

database_cleaner, wb_sequence_for_ids, wb_sequence_for_keys, wg_fetch, wg_start_read, wg_end_read, wg_create_record, wg_set_field, wg_get_field, wg_start_write, wg_end_write, wg_make_query.

4.7.3.1 Test cases

Test 1:

Goal: Check if there is any state where data in the DB is incomplete.

Query: *E<> system_data_controller.ERROR*

Test result: *Failed*

Test 2:

Goal: Check if there is a state where data marked as deleted are actually not deleted or the data are deleted when they are not supposed to be deleted.

Query: *E<> Test_2.EndState && DB_index_stack_used!=2*

Test result: *Failed*

Test 3:

Goal: check if only the data marked for deleting are actually deleted. This is negated goal of *Test 1*.

Query: *A[] Test_2.EndState imply DB_index_stack_used==2*

Test result: *Passed*

Test 4:

Goal: check if the cleaner agent terminates in the final state. This is extension to *Test 2.*

Query: *E<> Test_2.EndState*

Test result: *Passed*

Test 5:

Goal: check if there is any other deadlock state except the final state (*EndState*).

Query: *E<> deadlock && !Test_2.EndState*

Test result: *Failed*

4.7.4 Aspect 3: Uniqueness of data keys

The data stored in DB by different agents and at different time instances should have different keys. The test goal is to detect if there are such data with the same key in the DB. For that the test runs simultaneously the *Pressure Checker* and *Heartbeat Checker* agent which both write data to DB.

The model of Aspect 3 is a composition of the following templates (templates are given in Appendix E):

pressure_checker, heartbeat_checker, get_pressure, wb_insert_triples, wb_sequence_for_ids, wb_sequence_for_keys, wg_start_write, wg_get_field, wg_create_record, wg_delete_record, wg_set_field, wg_get_first_record, wg_start_write, wg_end_write.

4.7.4.1 Test cases

Test 1:

Goal: check if there is any state where the number of different keys is less than the number of generated keys (current test generates 1028 keys).

Query: *E<> Test_3.EndState && DB[1][3] < 1028*

Test result: *Failed*

Test 2:

Goal: check if all data items have unique keys. It is negation of *Test* 1.

Query: A[] Test_3.EndState imply DB[1][3] (DB[1][3] == 1028)

Test result: *Passed*

Test 3:

Goal: check if the test terminates in the final state?

Query: *E<> Test_3.EndState* (Addition to *Test* 2)

Test result: *Passed*

Test 4:

Goal: check if the test deadlocks in some other state than its final state?

Query: *E<> deadlock && !Test_3.EndState*

Test result: *Failed*

Test 5:

Goal: check if the occurrence of a deadlock implies, it happens in the *EndState*. It is negation of *Test 4*.

Query: *A[] deadlock implies Test_3.EndState*

Test result: *Passed*

Test 6:

Goal: check if there is any deadlock?

Query: *E<> deadlock*

Test result: *Passed* (in *EndState*)

4.8 Conclusion

Aspect orientation introduces an alternative modularization principle to multi-agent software design and testing. The medical monitoring and control systems involve different stakeholders and are difficult to develop, maintain, and use because of interplay of multiple viewpoints. The Home Telecare system with different monitoring and assisting functionalities, is an example of such multi-agent and multi-aspect system. The home monitoring and motor rehabilitation system (HRS) studied in this chapter involves requirements related to physiotherapist, patient and implementation of HRS. We demonstrated that AO-requirement engineering improves the comprehension of system functionality descriptions and allows modularization of models. Separation of concerns in AO models provides also reduction in terms of test purpose specification and test generation effort because the AO coverage criteria presume the reachability analysis with related aspect models without the need to explore the large monolithic model in one piece.

The model checking statistics of the HRS show that abstraction and AO decomposition are two approaches to reduce the complexity of practical verification and test generation. The experiments with a requirements level abstract AO model show that the verification and test generation task can be solved within seconds and with less than 1 MB of memory on a standard laptop. In the second part of the chapter we exposed the AO model of HRS on the level of software agents.

The non-AO model includes 98 process instantiations of 26 templates. Although the verification and test generation based on a monolithic model of that size is infeasible, the verification and test generation by aspects provided a computationally acceptable solution and helped system developers better address the test results in terms of HRS design aspects.

5 ANALYSIS AND VALIDATION OF AOT METHOD

5.1 Chapter overview

In this chapter, the aspect–oriented modelling and testing concepts and methods introduced in Chapter 3 and illustrated with the HRS case study in Chapter 4 are studied analytically in order to provide the quantitative and qualitative evidence of their advantages compared with non-aspect oriented methods.

5.2 Proving equivalence of non-aspect and aspect models

To compare the performance and usability characteristics of aspect-oriented and non-aspect-oriented modelling/verification/testing methods we have to show at first that the comparable models represent the same behaviour observable at the test interface. For that reason we demonstrate how to check the bisimulation relation (relative to test I/O actions) between the models.

The models to be compared are non-AO test model M and its AO counterpart model M^{ao}. We use M^{ao} that is derived from M by separating the aspects A_1, …, A_n and the base functionality of M, at first, and weaving the aspect advice models M^{A1},…, M^{An} back to the base model M^B, thereafter. After the aspect model M^{ao} is constructed we compose it with non-aspect model M by synchronous parallel compositions so that one model has the role of word generator on the test interface I/O alphabet $\Sigma_{test}^{i/o} = \Sigma_{test}^{i} \cup \Sigma_{test}^{o}$ and the other model has the role of word acceptor. If the timed I/O sequence acceptance is established in one direction, then the roles of the models are changed opposite and the same check repeated.

Definition 2.7
We say that M and M^{ao} are *observationally bisimilar* (denoted $M \sim_{i/o} M^{ao}$) with respect to alphabet $\Sigma_{test}^{i/o}$, if

- both M and M^{ao} have same test interface I/O alphabet $\Sigma_{test}^{i/o}$;
- when M is generating and M^{ao} is accepting Uppaal TA on alphabet $\Sigma_{test}^{i/o}$ then all timed words $TW(M) \in (\Sigma_{test}^{i/o})^*$ generated by M are recognizable by M^{ao},

and

- when M^{ao} is generating and M is accepting Uppaal TA on alphabet $\Sigma_{test}^{i/o}$ then all timed words $TW(M^{ao}) \in (\Sigma_{test}^{i/o})^*$ generated by M^{ao} are recognizable by M.

In other words, two test models cannot be distinguished by an external observer by interactions between the tester and SUT. Bisimulation is a symmetrical relation. Bisimulation for timed automata has been originally introduced in [48] and, as shown in [72] it is decidable for parallel timed processes.

5.2.1 Bisimulation verification

In order to show the relative bisimilarity relation $M \sim_{i/o} M^{ao}$ between a non-aspect oriented model M and aspect-oriented model M^{ao} of the same SUT, where $M^{ao} = M^{B}||_{i=1}^{n} M^{A_i}$, we can decompose the bisimulation verification task by individual aspect models M^{A_i} and due to the compositionality of M^{ao} limit ourselves by observing subsets of the test interface I/O alphabet that include only symbols of one M^{A_i} at a time. Given a M^{A_i} it is needed to compare then only the timed words projections $TW(M^{A_i})_{|\,\Sigma^{i/o}_{test_i}}$ onto sub-alphabet $\Sigma^{i/o}_{test_i} \in \Sigma^{i/o}_{test}$ with timed words projection $TW(M)_{|\,\Sigma^{i/o}_{test_i}}$ of the non-AO model M.

The bisimilarity check is performed in two steps as follows:

Step 1: To keep all I/O actions executions of models M and M^{ao} in lockstep the synchrony of selecting these I/O actions needs to be ensured so that if one of the models executes an I/O action or selects non-deterministically any action the other model should execute the same I/O action or make the same non-deterministic choice. Otherwise the traces of non-deterministic models may diverge and are not compatible. For this reason both – state as well as time step wise non-deterministic transitions with same labelling in comparable models need to be synchronized. To construct such a synchronous parallel composition of models M and M^{ao} all pairs of edges (e, e'), where $e \in E(M)$ and $e' \in E(M^{ao})$ need to be found such that

- *e* and *e'* have the same labelling (they model the same actions),
- *e* is nondeterministic either stae- or time-wise if and only if the *e'* is.

We denote the set of such edges with $E^{\#}$. If the edges e, $e' \in E^{\#}$ are already labelled with an I/O action label $a \in \Sigma^{i/o}_{test}$ then we split both e and e' into two edges e^0 and e^{00} connected via an auxiliary committed location, so that e^0 copies the labelling of e, and e^{00} is labelled with a unique side-effect free auxiliary channel $ch^{aux} \notin Channels(M) \cup Channels(M^{ao})$ for inter model synchronization. Adding a new edge e^{00} is necessary due to Uppaal TA syntax constraint that allows at most one channel label per edge. Such a model transformation example is depicted in Figure 5.1.

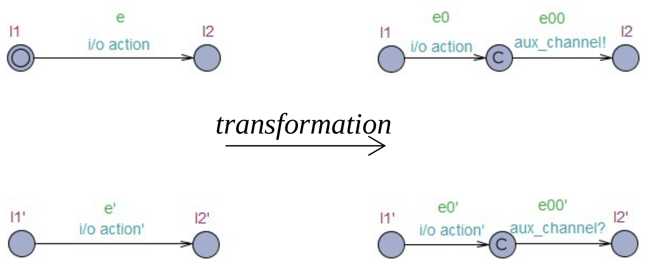


Figure 5.1 Model transformation for bisimulation verification

Step 2: Sufficient condition of models' bisimilarity after performing Step1 is following: if the models M and M^{ao} separately do not deadlock in their locations l_i and l'_i which are the departure locations of edges in $E^{\#}$ then the synchronous composition $M \parallel^{testIO}_{syn} M^{ao}$ of compared models does not deadlock in the global configuration (l_i, l'_i, .), and *vice versa*. It is stated formally as TCTL model checking query satisfiability condition:

$$\forall l_i \in L(M)\text{: } M, l_i \models \text{A[] } \textit{not deadlock} \wedge M.l_i \qquad (5.1)$$
$$\wedge$$
$$M^{ao}, l'_i \models \text{A[] } \textit{not deadlock} \wedge M.l_i \wedge M^{ao}.l'_i$$
$$\Leftrightarrow$$
$$M, l_i \parallel^{testIO}_{syn} M^{ao}, l'_i \models \text{A[] } \textit{not deadlock} \wedge M.l_i \wedge M^{ao}.l'_i$$

Thus, before evaluating the AO modelling and testing approaches with respect to the verification and test generation effort, we have to prove the validity of formula (5.1) on these models by Uppaal model checker.

5.2.2 Bisimulation verification example

To ensure bisimilarity between AO and non-AO models of HRS the model in Figure 4.7 is compared with the one depicted in Figure 5.2 by following the steps described in Section 5.2.1.

Let the HRS model-based test interface input alphabet be $\Sigma^{i}_{test} = \{Exercise,$ $sync0,\ sync\}$ and output alphabet $\Sigma^{o}_{test} = \{emergency\_call,\ refresh\_scr\}$. To synchronize the AO and non-AO models I/O actions and nondeterministic actions we introduce auxiliary channels as required in Step 1 (Section 5.2.1). At first, the channels for edges labelled with input actions are defined as follows: channel $i1$ for synchronizing input action *Exercise*, channel $i2$ for *sync*0, and channel $i3$ to synchronize the input action *sync*. Similarly, auxiliary channels are introduced to synchronize output actions denoted by symbols in the alphabet Σ^{o}_{test} : channel $o1$ to synchronize the output action *emergency_call* and channel $o2$ to synchronize the output action *refresh_scr*.

Secondly, we define auxiliary channels to synchronize the non-deterministic selecting of AO and non-AO models' internal actions. The internal auxiliary channels are introduced for automata templates as shown in Table 5.1.

Table 5.1 Auxiliary channels for checking AO and non-AO models bisimilarity

Template: *Patient_physical _condition*					
Edge	Aux. Chan.	Edge	Aux. Chan.	Edge	Aux. Chan.
In_N→*In_N*	*e*1	*Initial*→*Normal*	*e*4	*Normal*→*Better*	*e*8
In_B→ *In_B*	*e*2	*Initial*→Better	*e*5	*Better*→*Normal*	e9
In_W→*In_W*	*e*3	*Initial*→*Worse*	*e*6	*Normal*→*Worse*	*e*10
		Worse→*Normal*	*e*7	*Worse*→*Bad*	*e*11

Template: *Patient_ exercising*	
Edge	Auxiliary channel
Do_Exercise → *SwitchToNext*	*e*12

Note that when executing the edges *In_N*→*In_N*, *In_B*→*In_B*, and *In_W*→*In_W* a random value is generated to *k* which is used for updating variable *measurement*. To avoid diverting the timed traces in AO and non-AO models same random values generated in one must be duplicated also for the other model at the same time. For that both updates need to done either in one or in the other model and then duplicated. In our case we implement the non-deterministic assignment in non-AO model and duplicate it for AO model variable *k_prim*. The models with added auxiliary channels and random value duplications are depicted in Figure 5. 3.

As described in Sub-section 5.2.1 Step 2 establishes that for all locations where the non-deterministic selection of the next action is possible, if there is no deadlock in the location of one model then there should not be a deadlock possible also in the corresponding location of the other model. To verify this we model check for all locations of non-deterministic choice the condition set by formula 5.1.

The non-deterministic locations of the template *Patient_physical_condition* are: *In_N*, *In_B*, *In_W*, *Initial*, *Worse*, *Normal*, *Better*; and in the template *Patient_exercising* there is only one such location *Do_exercise*.

By unifying the location names in the formula 5.1 we get the necessary model checking queries, e.g. for location *Patient_physical _condition. In_N* of AO model the query is A[] *not deadlock* && M^{ao}. *Patient_physical_condition.In_N.*

Similarly, such queries have to be checked for all the above locations listed in the non-AO model and thereafter in the synchronous parallel composition of both models.

After the synchrony condition of non-deterministic choices has been verified, it remains to show simultaneous non-blocking execution of I/O actions. To prove this we can use again the non-existence condition of deadlocks at source locations of edges that are labelled with some I/O symbol. But since the condition needs to be checked for all I/O edges we apply an alternative way. Namely, we introduce an auxiliary Boolean vector T^{io} in the model *M* and its counterpart vector T^{io}' in the model M^{ao}. The size of vectors | T^{io} | = | T^{io}'| = | $\Sigma^{i/o}_{test_i}$|, so that there is one-to-one correspondence between the elements of T^{io} (T^{io}') and the elements of $\Sigma^{i/o}_{test_i}$ as follows (updates of T^{io} and T^{io}' are shown also in Figure 5.3):

Exercise, Exercise' → T^{io}[0], T^{io}'[0],

sync0, sync0' → T^{io}[1], T^{io}'[1],

sync, sync' → T^{io}[2], T^{io}'[2],

emergency_call, emergency_call' → T^{io}[3], T^{io}'[3],

refresh_scr, refresh_scr' → T^{io}[4], T^{io}'[4].

Each time an I/O action with a symbol from $\Sigma^{i/o}_{test_i}$ is executed, its corresponding element in T^{io} is updated to *true*. If the I/O action is executed repeatedly its corresponding elements in T^{io} and in T^{io}' have to be reset back to *false* in the next transitions, following immediately the edges with update *true*. This makes the repetitive execution of I/O actions observable to model checker. By running now the Uppaal model checking query – A [] forall (*i*: [1, | T^{io} |]) T^{io}[*i*] == T^{io}'[*i*] with a positive result, it confirms the bisimilarity of AO and non-AO models with respect to the test I/O alphabet $\Sigma^{i/o}_{test_i}$.

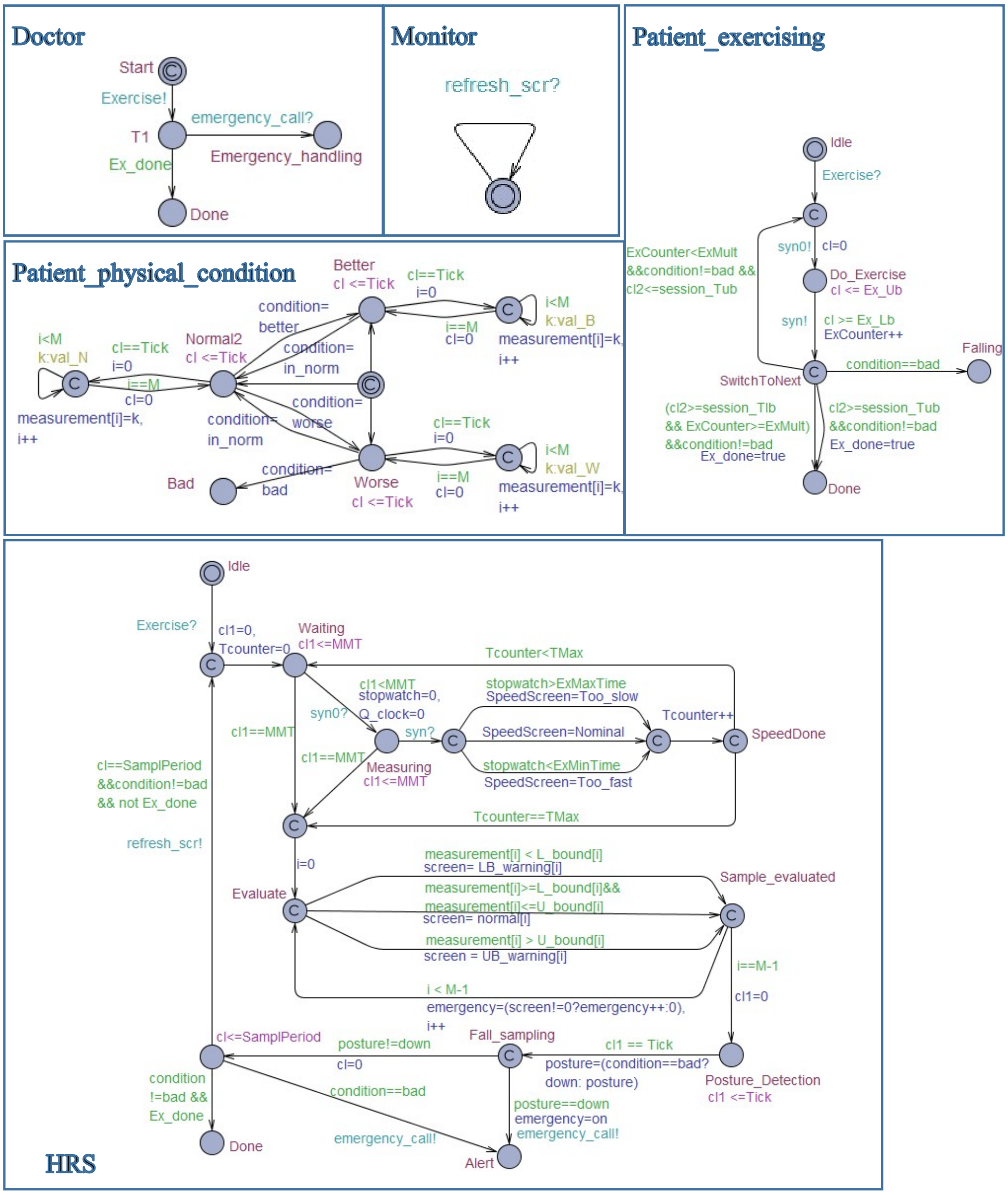


Figure 5.2 Non-aspect-oriented model of the HRS

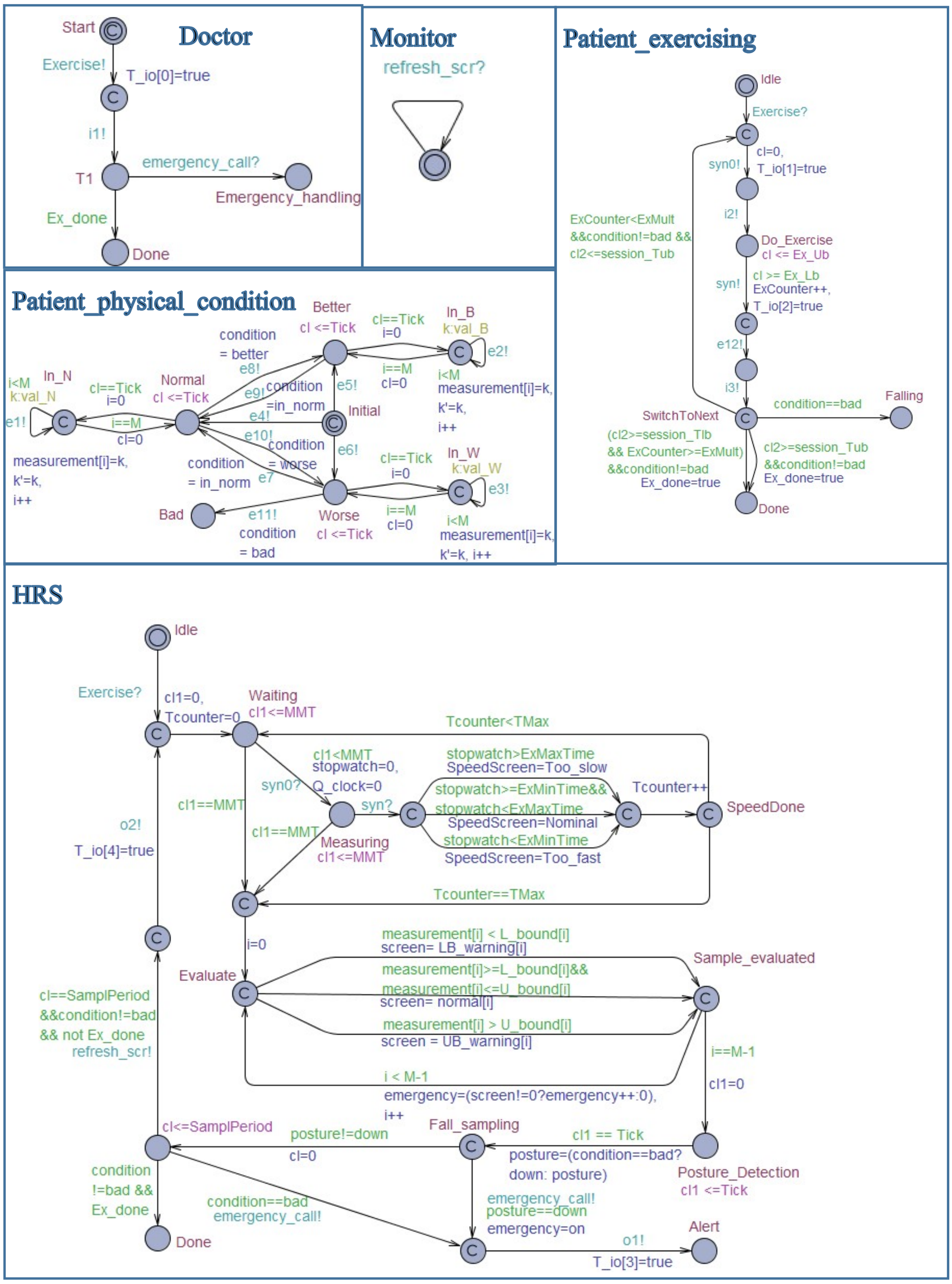


Figure 5.3 Non-AO model of the HRS enhanced for bisimulation checking

5.3 Comparison of model update effort

To make reliable conclusions on modelling effort a large volume of statistical data is needed. In order to evaluate the modelling and update effort two independent teams with equal skills and performance should be involved, one constructing the AO version and the other non-AO version of the model, so that the models would be bisimilar as described in Section 5.2. Since collecting sufficient statistical evidence on that matter was not possible under the current thesis research we refer to conclusions drawn in [71] instead.

In [71] two different versions of the application model were developed in parallel and the effort was measured. At first, a simple authentication procedure was modelled for a Crises Management System (CMS). In this case study a resource used at crises management missions provides its credentials (authentication token) and the CMS checks if they correspond to a list of known credentials. The CMS specification was developed both as non-AO and AO models and the two models were checked for equivalence via bisimulation. Next, the model was upgraded to a more advanced authentication version, where the authentication was based on the Needham-Schroeder Public-Key protocol [73].

When upgrading, the number of changes (statement additions, updates, removals) was measured in both, in the non-AO and AO model. The results show that modifying the non-AO models required editing approximately 40% of the model elements, versus 20% of editing in the case of the AO model. The effort of updating the models was much smaller in the case of aspect models also in terms of time spent on modifications. It was easier to identify which elements had to be changed, and these elements had in general a local scope, without affecting the specification of the other aspects. This is an expected result according to different studies [3, 5, 22], which confirms that AOM reduces the scattering and tangling of requirements.

5.4 Comparison of test purpose specification effort

We estimate the test purpose specification effort by the time of finding and labelling edges with Boolean variables (so called *traps*) to define edges as test coverage items. The traps are updated to *true* whenever the trap-labelled edge is executed during a test run. Thus, the AO coverage can be expressed with traps that label the edges and are syntactic elements used in AO coverage expression.

For quantitative characterization of test purpose specification effort we derive an empirical formula that correlates with factors of human capability of finding the edge to be labelled amongst the set of all edges of the model. As the base case, we estimate the specification time on the non-aspect-oriented model, at first, and thereafter, compare it with a case of aspect models sharing the same set of traps between aspects.

Non-aspect oriented case. Assume the total number of edges in the model M is $n = |E(M)|$ and the number of edges to be labelled with traps is k. We assume that once an edge has been labelled it is memorized and does not need to be inspected when searching an edge for the next trap. Thus, in the worst case when searching for an edge to label it with i-th trap $n - i$ options should be inspected. Let the duration needed for inspection of one edge be constant d. Then the upper time bound T^{tr} of labelling k edges with traps can be calculated by formula

$$T^{tr}(M) = \sum_{i=0}^{k-1}(n-i)d \qquad (5.2)$$

Rewriting (5.2) in the form of a square function, we get the formula

$$T^{tr}(M) = \sum_{i=0}^{k-1}(n-i)\, d = \frac{n^2-(n-(k-1))^2}{2}d \qquad (5.3)$$

Aspect-oriented case. For comparison of test purpose specification effort we assume that the non-aspect model M and its AO counterpart M^{ao} are observationally (with respect to test interface) bisimilar. Also, the edges in M labelled with traps should occur in the augmented AO model M^{ao}. (Recall that in the full augmented model all aspect models $M^{ao}{}_1$, …, $M^{ao}{}_m$ and the initial base model $M^{ao}{}_0$ are woven together). Thus, the set of trap labelled edges in M is equal to the union of trap labelled edges in the individual aspect models: $E^{tr}(M) = \cup_i E^{tr'}(M^{ao}{}_i)$. In other words, we assume that the original trap labelled set of edges is partitioned so that: k_1 traps specify coverage in $M^{ao}{}_1$, k_2 traps specify the coverage in $M^{ao}{}_2$, and k_m traps specify the coverage in $M^{ao}{}_m$.

Having all m aspects the condition $\sum_{i=1}^{m} k_i = k$ is assumed to be satisfied. By second assumption the set of edges of any aspect model is a strict subset of edges of the non-aspect model M, i.e.

$$\forall i{:}\, E(M_i^{ao}) \subset E(M) \Rightarrow |E(M_i^{ao})| < |E(M)|\ .$$

When denoting the number of edges $|E(M_i^{ao})|$ of aspect model $M^{ao}{}_i$ with n_i and applying formula 5.2 we get the labelling effort upper bound for each aspect model $M^{ao}{}_i$

$$T_i^{tr}(M_i^{ao}) = \frac{n_i^2-(n_i-(k_i-1))^2}{2}d,$$

and total test purpose specification time

$$T^{tr}(M^{ao}) = \sum_{i=1}^{m}(T_i^{tr}(M_i^{ao})) = \sum_{i=1}^{m}(\frac{n_i^2-\left(n_i-(k_i-1)\right)^2}{2}d).$$

In the following we demonstrate under which conditions AO has advantage over non-AO test models, i.e.

$$T^{tr}(M) > T^{tr}(M^{ao}). \qquad (5.4)$$

By substituting $T^{tr}(M)$ and $T^{tr}(M^{ao})$ in (5.3) we get

$$\frac{n^2-(n-(k-1))^2}{2}d > \sum_{i=1}^{m}(\frac{n_i^2-\left(n_i-(k_i-1)\right)^2}{2}d) \qquad |\ ()d/2$$

$$\frac{n^2-(n-(k-1))^2}{2}d > (\sum_{i=1}^{m}(\frac{n_i^2-\left(n_i-(k_i-1)\right)^2}{2}))d \qquad |\ {:}d/2$$

$$n^2-(n-(k-1))^2 > \sum_{i=1}^{m}(n_i^2-\left(n_i-(k_i-1)\right)^2) \quad |\ \Sigma(a+b)=\Sigma a+\Sigma b$$

$$n^2-(n-(k-1))^2 > \sum_{i=1}^{m}n_i^2-\sum_{i=1}^{m}\left(n_i-(k_i-1)\right)^2$$

Since for any n, $m \geq 2$ and for any positive a, if $a = \sum_{i=1}^{m} a_i$ then $a^n > \sum_{i=1}^{m} {a_i}^n$, we can conclude that $T^{tr}(M) > T^{tr}(M^{ao})$.

□

The plot of $T^{tr}(M)$ and $T^{tr}(M^{ao})$ dependency on parameters *m* and *k*$_i$, where *i* = 1,…, *m*, is depicted in Figure 5.4. We can assume without loss of generality that the traps are distributed equally over the aspects and just for illustration the following parameter values are selected: the number of edges *n* =100, the number *k* of traps varies from 10 to 100 with step 5, and the number *m* of aspects varies from 1 to 10 with step = 1. To compare non-AO and AO cases it is assumed, like in the case of traps, the sum of edges in the aspect models equals to the sum of edges in the bisimilar non-AO model. Under given assumptions it is easy to see in Figure 5.4 that due to the nonlinearity in the number of aspects the sum of labelling efforts of AO models is less than that in the non-AO model provided the total number of traps and edges in both cases is the same.

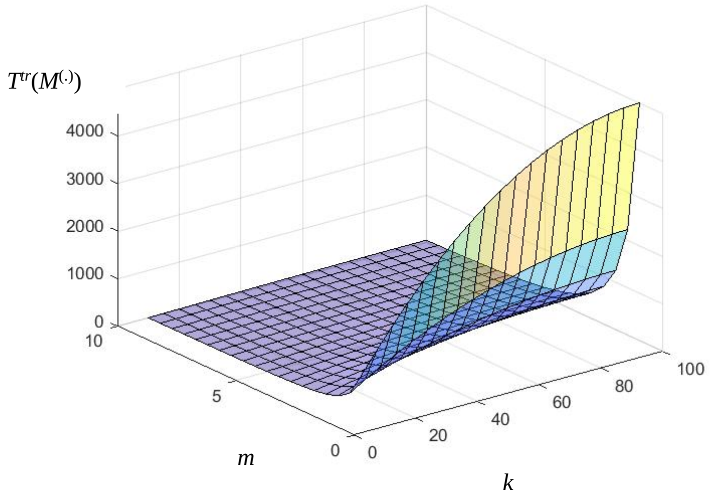


Figure 5.4 The dependency of test purpose specification effort on the number of traps k and number of aspects m

5.5 Comparison of test generation effort

The generation of test sequences in this approach is based on using timed witness traces produced by an Uppaal model checker as the result of checking queries that encode the test purposes (coverage reachability) symbolically. Thus, the test generation effort is measurable in terms of time complexity of model checking such TCTL formulas that express AO coverage criteria. In Section 3.3, it was shown how to express the AO coverage criteria in TCTL.

The worst-case time complexity $\mathcal{O}$ of model checking TCTL formula φ over timed automaton M, with the clock constraints of φ and of M in ψ is, according to [74]

$$\mathcal{O}\,(|\varphi| \times (n! \times 2^n \times \textstyle\prod_{x\in\psi} c_x \times |L|^2)); \qquad (5.5)$$

where n is the number of clock regions, ψ - set of clock constraints, c_x -maximum constant the clock x is compared with; $L : Loc \rightarrow 2^{AP}$ is a labelling function for symbolic states of M. L denotes the product of data constraints over all locations and edges defined in the Uppaal TA model and AP is the set of atomic propositions used in guard conditions and invariants.

From (5.5) it can be seen that the time complexity of model checking TCTL is:

(i) Linear in the length of the formula φ;
(ii) Exponential in the number of clocks; and
(iii) Exponential in the maximal constants c_x with which each clock x is compared to the model M and in φ.

However, using state space reduction techniques the worst case time complexity can be reduced to being quadratic in the number of symbolic states on data variables in the model [74].

Regarding *space complexity*, the lower bound for the complexity of model checking TCTL for a Timed Automata model is known to be PSPACE-hard [62].

In practice, time and space complexity of model-checking TCTL on Uppaal TA, boils down to the size of the symbolic state space and more specifically, to the number of symbolic states (including clock zones) to be explored, and respectively stored during the verification. Since by definition the number of locations $|L(M^{ao})|$ and edges $|E(M^{ao})|$, as well as the number of variables $|V(M^{ao})|$ of an aspect model M^{ao} is not greater than that of a behavioural equivalent non-aspect model M, we can conclude from the complexity formula (5.5) that every small reduction in the number of model elements, and related to them, the number of symbolic states provides an exponential decrease in the number of steps of the model exploration. Naturally, this applies also to checking the aspect related properties that need to be satisfied on the AO test models. Due to the superposition refinement based weaving (synchronization built into the weaving constructs), the weaving Approach 2 does not introduce additional interleaving between base model and advices model transitions (see Section 3.2.4 for details).

Relying on the arguments above we can state the following claims on symbolic states to be processed and stored for model checking based test generation. Let us consider the non-aspect-oriented model M and, respectively, the aspect-oriented model M^{ao} that specifies the same behaviour of a system. Recall that M^{ao} consists of a base model M^B with which a set of models of non-interfering aspects A_i, $i = 1\ldots n$ are woven. Then we yield the following:

Claim 5.1 (*Verification effort*): For any reachability property φ_i of any A_j, $i = 1\ldots n$ decidable on $M^B \oplus M^{A_j}$ the model checking effort $\boldsymbol{E}$ (in terms of time or space) is equal or less than the effort of model checking the property φ_i on the non-aspect oriented model M, where the semantics of $M^B \oplus M^{A_j}$ is a subset of semantics of M, i.e.

$$M^B \oplus M^{A_j} \models \varphi_i \wedge [|M^B \oplus M^{A_j}|] \subseteq [|M|] \Rightarrow M \models \varphi_i \wedge \boldsymbol{E}(M^B \oplus M^{A_j} \models \varphi_i) \leq \boldsymbol{E}(M \models \varphi_i) \tag{5.6}$$

Assuming the tests are generated using model checking traces the formula (5.6) yields the Claim 5.2.

Claim 5.2 (*testing effort)*: Under the assumptions of Claim 5.1, the effort $\boldsymbol{E}$ (in terms of time or space) of generating the test case T^{φ_i} that is interpretation of aspect A_j property φ_i and is bounded with aspect-oriented model $M^B \oplus M^j$, is less than or equal to the effort of generating the test T^{φ_i} from non-aspect oriented model M:

$$M^B \oplus M^{A_j} \models \varphi_i \wedge [|M^B \oplus M^{A_j}|] \subseteq [|M|] \Rightarrow \boldsymbol{E}(M^B \oplus M^{A_j}, \mathrm{T}^{\varphi_i}) \leq \boldsymbol{E}(M, \mathrm{T}^{\varphi_i}). \tag{5.7}$$

Here notation $[|M|]$ denotes the operational semantics (a set of behaviours) of model M. The validity of formulas (5.6) and (5.7) stems from the fact that aspect-oriented models represent subsets of the behaviour of the non-aspect model of the same system. The performance gain due to the compositionality via aspect-oriented modelling is demonstrated with the numerical example in the Figure 5.5.

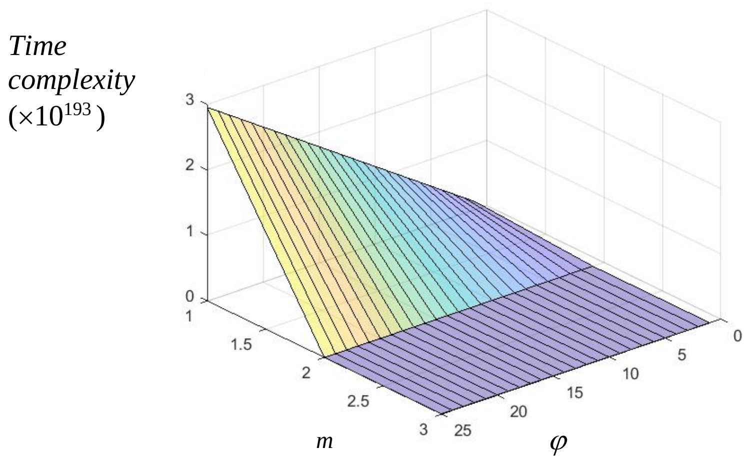


Figure 5.5 The dependency of test generation effort on the number of logic connectives in the test coverage formula φ and the number of aspects m

The plot in Figure 5.5 is generated using the approximating function $f(m, |\varphi|) = m\ (|\frac{\varphi}{m}| \times (\left(\frac{n}{m}\right)! \times 2^{\frac{n}{m}} \times C \times |\frac{L}{m}|^2))$ which is derived from 5.5 by applying assumptions that the set of locations and the set of literals of coverage formula is partitioned between m aspect models equally and $\prod_{x \in \psi} c_x$ is some constant C.

It can be seen from Figure 5.5 that the test generation effort measured in terms of the model checking complexity of the test coverage formula φ is, according to formula 5.5, inverse super-exponential in the number of aspects m and exponential in the number of literals in φ. It means that when partitioning the model checking task to m smaller ones verifiable on an aspect model instead of one monolithic non-aspect model it provides in total exponentially smaller test generation effort.

5.6 Comparison of generated tests (length of test sequences)

As concluded in Section 5.5 generating the witness traces by model checking a TCTL formula φ on a non-AO model has disadvantages compared to generating the traces of φ sub-formulas (when conjunction is equivalent to φ) on AO models. Since the AO and non-AO models to be compared must be bisimilar by assumption their witness traces generated are equivalent in the sense of coverage. Thus, regarding test execution effort we can conclude that the test sequences derived from AO models and non-AO models are coverage equivalent although their traces can differ slightly due to the interleaving introduced by the other structural elements of full monolithic model. Just note that the Uppaal model checker enables generating witness traces that are optionally either shortest or fastest in the set of all witness traces satisfying some AO coverage expression φ.

5.7 Conclusion

In Chapter 5, the AOM and testing concepts and methods introduced in Chapters 3-5 have been analysed in order to provide the quantitative and qualitative evidence on their advantages compared with non-aspect oriented methods. It was shown analytically and with reference to complexity results of TCTL model checking that the effort of modifying AO models, as well as the effort of specifying AO coverage criteria and the test generation from AO models is less than that from non-aspect oriented models. Note also that when the tests are generated offline the test execution effort does not depend on the method of test generation, so, only the length of test sequences matters.

6 CONCLUSION

6.1 Chapter overview

Last chapter summarizes the results obtained from this thesis.

6.2 Main results

The background study of the thesis domain that is *aspect orientation and testing* brought up some principal questions on the model-based development and particularly on the model-based testing:

- How the existing aspect-oriented requirements engineering methodology can be applied to aspect-oriented test model construction?
- What model transformations are needed for constructing AO test models?
- How to specify the test purpose and test coverage criteria on AO test models?
- How to express the AO test purposes symbolically?
- Given an AO SUT model and the AO test purpose how to verify the correctness and feasibility of the testing tasks?
- How AO MBT improves the productivity of the overall testing process?

Based on the state-of-the-art results scattered over a large volume of related publications, the following research hypotheses were formulated in this thesis:

1. Test model construction and update effort decreases along with improving the model comprehension due to a reduction in the number and severity of modelling errors and the need for their corrections;
2. Aspect orientation can be introduced compositionally in test generation and execution that reduces the test generation effort;
3. Aspect-oriented test cases are more compact and allow the saving of test execution time (improved performance); and
4. Defining the test cases and their coverage criteria relative to aspects provide better traceability of bugs' causes and locating them in the requirements specification.

To validate the hypotheses the generic problems, stated above, were instantiated and studied in the context of Uppaal TA formalism. This led to the main results of the thesis being summarised as follows:

- A new aspect-oriented model engineering methodology for Uppaal TA was introduced in MBT. This methodology is based on an aspect-oriented requirements engineering paradigm that results in three advantages: testability of SUT quality attributes, a simple rule for composition, and better comprehension of test models.

- Aspects weaving is implemented as a set of model superposition refinement operators. AO tests can be generated automatically by running TCTL model checking queries on woven models and applying the resulting witness traces thereafter as test sequences of AO test cases.

- A new set of aspect-oriented test coverage criteria is defined. That gives meaningful automatic test design options based on SUT models which are defined by quality attributes related to aspects. It is shown that coverage criteria can be formalized in temporal logic TCTL.

- The usability of the AO MBT method is demonstrated on the Home Rehabilitation System testing case-study. This provides an experimental evidence that AO testing improves the efficiency of MBT compared to the methods that are based on non-aspect oriented models.

- Last but not least, the quantitative and qualitative evidence of the advantages of AOT was provided with reference to complexity results of TCTL model checking. It is shown also analytically that the effort of modifying AO models, as well as the effort of specifying AO coverage criteria and the test generation from AO models is less than that of non-aspect oriented models.

6.3 Future work

Aspect-oriented testing provides a discipline that supports achieving better structure and comprehension of test models. This has implications, in turn, to test coverage criteria and the reduction of total conformance testing effort. In future work, we plan to merge the aspect-oriented conformance testing with mutation testing to address the bugs which are caused by too permissive software implementations. While the conformance testing is capable of detecting bugs where implementation does not conform to the specification, it is not revealing bugs where the implementation under test has behaviours that are not in the requirements specification. We also plan to conduct a set of larger case studies to evaluate the scalability of the approach as well as its advantages from the point of view of incremental test suite updates.

REFERENCES

[1] R.V. Binder, “Model-based testing, user survey: Results and analysis,” http://robertvbinder.com/wp-content/uploads/rvb-pdf/arts/MBT-User-Survey.pdf (Jan 2012), online, last accessed 11.5.2015, p.26.

[2] R.E. Filman, et al., “Aspect-Oriented Software Development.” Addison-Wesley, Boston, 2005.

[3] S. J. Sutton, “Aspect-Oriented Software Development and Software Process,” In: *M. Li, B. Boehm, L. Osterweil, (eds.) Unifying the Software Process Spectrum, Lecture Notes in Computer Science*, vol. 3840, pp. 177-191. Springer, Berlin, Heidelberg, 2006.

[4] R. France and B. Rumpe, „Model-driven development of complex software: A research roadmap,“ In *Proceedings of the 29th International Conference on Software Engineering*, IEEE Computer Society, 2007, pages 37-54.

[5] S. Ali, T. Yue, L. Briand, “Assessing quality and effort of applying aspect state machines for robustness testing: A controlled experiment,” In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference*, March 2013, pp. 212-221.

[6] W. W. Royce, „Managing the Development of Large Software Systems: Concepts and Techniques,“ *Proc. WESCON, IEEE Computer Society Press*, Los Alamitos, CA, 1970. Reprinted at the ICSE'87, Monterey, California, USA. March 30 - April 2, 1987.

[7] M. Eigner, T. Dickopf, H. Apostolov, P. Schaefer, K. G. Faißt, et al. System Lifecycle Management: Initial Approach for a Sustainable Product Development Process Based on Methods of Model Based Systems Engineering. Shuichi Fukuda; Alain Bernard; Balan Gurumoorthy; Abdelaziz Bouras. *11th IFIP International Conference on Product Lifecycle Management (PLM), Jul 2014, Yokohama, Japan. Springer*, IFIP Advances in Information and Communication Technology, AICT-442, pp. 287-300.

[8] B. Boehm (1996), "A Spiral Model of Software Development and Enhancement," In: *ACM SIGSOFT Software Engineering Notes*, ACM, 11(4): 14-24, August 1986.

[9] A. Cockburn, „Agile Software Development,“ Boston, Addison-Wesley, 2002.

[10] B. Hunt, G. T. Abolfotouh, J. Carpenter, R. Gioia, “Software test costs and return on investment (ROI) issues,” http://www.iceaaonline.com/ready/wp-content/uploads/2014/03/Software-Test-Cost-and-ROI-Galorath-Feb-14-Hunt.pdf, online, last accessed 15.12.2016.

[11] P. Skokovic, „Requirements-Based Testing Process in Practice,“ IJIEM, Vol.1 No 4, 2010, pp. 155-161.

[12] EU SPEEDS project. “Inria research report n° 8147,“ november 2012, (https://hal-univ-tlse2.archives-ouvertes.fr/hal-01178467/document) retrieved 11.5.2015.

[13] E. A. Lee, “Cyber-physical systems-are computing foundations adequate,” in *Position Paper for NSF Workshop On Cyber-Physical Systems: Re-search Motivation, Techniques and Roadmap*, vol. 2, 2006.

[14] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297-312, 2012, online, last accessed: 18.5.2015. Available: http://dx.doi.org/10.1002/stvr.456.

[15] J. Tretmans. “Model Based Testing: Property checking for real,” Keynote address at the International Workshop for Construction and Analysis of Safe Secure, and Interoperable Smart Devices, 2004. Last accessed: 27.5.2018. Available http://www-sop.inria.fr/everest/events/cassis04.

[16] J.-R. Abrial, “The B-Book: Assigning Programs to Meanings,” JCambridge University Press, 1996. ISBN 0-521-49619-5.

[17] G. J. Tretmans, “A formal approach to conformance testing,” PhD dissertation, University of Twente, 1992.

[18] J. Tretmans, “Test Generation with Inputs, Outputs, and Quiescence,” In *TACAS* , volume 1055 of LNCS, Springer, 1996, pages 127-146.

[19] G.Kiczales and others, “Aspect-Oriented Programming,” in *ECOOP ’97 - Object-Oriented Programming*, 1997, vol. 1241, pp. 140–149.

[20] M. Badri, L. Badri, and M. Bourque-Fortin, “Generating unit test sequences for aspect-oriented programs: towards a formal approach using UML state diagrams,” in *Information and Communications Technology, 2005. Enabling Technologies for the New Knowledge Society: ITI 3rd International Conference on*, 2005, pp. 237-253.

[21] S. Clarke and E. Baniassad, “Aspect-Oriented Analysis and Design,” The Theme Approach. Addison-Wesley, 2005.

[22] S. Ali, T. Yue, L. C. Briand, “Does Aspect-Oriented Modelling Help Improve the Readability of UML State Machines?,” *Software & Systems Modelling*, vol. 13, no. 3, pp. 1189-1221, 2014.

[23] A. Bhave, B. H. Krogh, D. Garlan, B. Schmerl, “View Consistency in Architectures for Cyber-Physical Systems,” *2011 IEEE/ACM Second International Conference on Cyber-Physical Systems*, Chicago, IL, 2011, pp. 151-160. doi: 10.1109/ICCPS.2011.17

[24] J. Vain, M. Kääramees, M. Markvardt, “Online Testing of Nondeterministic Systems with the Reactive Planning Tester,“ 2012.

[25] S. Ghosh, R. France, C. Braganza, N. Kawane, A. Andrews and O. Pilskalns, "Test adequacy assessment for UML design model testing," *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, 2003, pp. 332-343.

[26] J. Ellsberger, D. Hogrefe, A. Sarma, „SDL Formal Object-Oriented Language for Communication Systems,“ Prentice Hall 2007.

[27] J. M. Spivey, „The Z Notation: A Reference Manual,“ Second ed., Published 1998 by J. M. Spivey, Oriel College, Oxford, England.

[28] E. Bernard, F. Bouquet, A. Charbonnier, B. Legeard, F. Peureux, M. Utting, and E. Torreborre, "Model-based Testing from UML Models," In *Procs. of the Int. Workshop on Model-based Testing (MBT'2006)*, volume P-94 of *Lecture Notes in Informatics*, Dresden, Germany, pages 223-230, October 2006.

[29] A. Cavarra, „Data Flow Analysis and Testing of Abstract State Machines,“ In *Proceedings of the 1st international conference on Abstract State Machines, B and Z* (ABZ '08), E. Börger, M. Butler, J. P. Bowen, and P. Boca (Eds.). Springer-Verlag, Berlin, Heidelberg, 85-97.

[30] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes, „Generating Finite State Machines from Abstract State Machines,“ ACM SIGSOFT Software Engineering Notes 27 (4), 2002, 112-122.

[31] C. Pfaller, “Requirements-based test case specification by using information from model construction,” In *Proceedings of the 3rd international workshop on Automation of software test*, 2002, ACM, New York, NY, USA 7-16.

[32] M. Sarma, P. V. R. Murthy, S. Jell, and A. Ulrich, “Model-Based Testing in Industry – A Case Study with Two MBT Tools,” in *AST '10 Proceedings of the 5th Workshop on Automation of Software Test,* ACM, New York, NY, USA 87-90.

[33] A. C. D. Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, „A survey on model-based testing approaches: a systematic review,“ In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007* (WEASELTech '07). ACM, New York, NY, USA, 31-36. DOI=http://dx.doi.org/10.1145/1353673.1353681

[34] M. Utting, B. Legeard, “Practical model-based testing: A Tools Approach,“ 2007, Elsevier Inc.

[35] M. Bernardino, E. M. Rodrigues, A. F. Zorzo and L. Marchezan, "Systematic mapping study on MBT: tools and models," in *IET Software*, vol. 11, no. 4, pp. 141-155, 8 2017. doi: 10.1049/iet-sen.2015.0154

[36] R. Jeords, C. Heitmeyer, M. Archer, and E. Leonard, “A formal method for developing provably correct fault-tolerant systems using partial refinement and composition, in FM,“ 2009, Formal Methods, pp. 173-189, Springer.

[37] ETSI ES 202 951, “Methods for Testing and Specification (MTS); Model-Based Testing (MBT); Requirements for Modelling Notations,” July 2011.

[38] S.J. Prowell, “JUMBL: A tool for model-based statistical testing,” in *Proc. 36th Annual Hawaii International Conference on System Sciences, 2003*, pp. 9 pp.-. doi: 10.1109/HICSS.2003.1174916

[39] C. Sant’Anna, et al., "On the reuse and maintenance of aspect-oriented software: An assessment framework," *Proceedings of Brazilian symposium on software engineering*. 2003.

[40] A. Huima, (2007) "Implementing Conformiq Qtronic," Testing of Software and Communicating Systems, Springer, pp. 1-12. DOI: 10.1007/978-3-540-73066-8_1

[41] A. Belinfante, (2012) "JTorX: a Tool for On-Line Model-Driven Test Derivation and Execution". In *Proc. of TACAS'10*, Springer, pp. 266-270. DOI: 10.1007/978-3-642-12002-2_21

[42] A. Hartman and K. Nagin (2004), "The AGEDIS tools for model based testing," *SIGSOFT Softw. Eng. Notes*, 29:4, pp. 129-132. DOI: 10.1145/1013886.1007529

[43] A. Schauerhuber, et al., “Survey on Aspect-Oriented Modelling Approaches,” 2002.

[44] G. Geri, et al., “An Aspect-Oriented Methodology for Designing Secure Applications,” 2009.

[45] A. Rashid, et al., “Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe,” Computer 43(2), 19-26 (2010).

[46] R.B. France, et al., “An aspect-oriented approach to design modelling,” *IEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design* 151(4) (aug 2004)

[47] J. Vain, et al., 2011, “Online testing of nondeterministic systems with reactive planning tester,” *Dependability and Computer Engineering: Concepts for Software-Intensive Systems,* (113-150), Hershey, PA: IGI Global.

[48] J. Bengtsson, W. Yi, 2004, “Timed automata: Semantics, algorithms and tools,” *Lecture Notes on Concurrency and Petri Nets, Lecture Notes in Computer Science* vol. 3098.

[49] J. Bengtsson, “Clocks, DBMs and States in Timed Systems,” PhD dissertation, Dept. of Information Technology, Uppsala University, 2002.

[50] P. Herber, 2010, “A Framework for Automated HW/SW Co-Verification of SystemC Designs using Timed Automata,” Logos Verlag Berlin GmbH.

[51] J. Iqbal, D. Truscan, J. Vain, I. Porres, “Reconstructing timed symbolic traces from rtioco-based timed test sequences using backward-induction,” In: V.*Vranić, R. Ondřej, Ondřej (Ed.). Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems, ECBC 2017* : 31 August - 01 September 2017, Larnaca, Cyprus (1-10).

[52] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, 2008, “Testing Real-Time Systems Using UPPAAL,” In *Formal Methods and Testing*, RobertM. Hierons, JonathanP. Bowen, and Mark Harman (Eds.). LNCS, Vol. 4949. Springer Berlin Heidelberg, 77-117.

[53] E. Katz and S. Katz, “Incremental analysis of interference among aspects,” In *Proceedings of the 7th Workshop on Foundations of Aspect-oriented Languages, FOAL '08*, pages 29-38, New York, NY, USA, 2008. ACM.

[54] K. Sarna and J. Vain, “Exploiting aspects in model-based testing,” In *Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages, FOAL'12*, pages 45-48, New York, NY, USA, 2012. ACM.

[55] OMG, Unified Modelling Language Infrastructure Specification, version 2.1.2, Oct. 2007. Document formal/2007-11-04, available at http://www.omg.org/.

[56] S. Ali, T. Yue, and L. C. Briand, “Does Aspect-oriented Modelling Help Improve the Readability of UML State Machines?,” Software & Systems Modelling, 13(3):1189-1221, July 2014.

[57] S. Op de Beeck, et al., “A study of aspect-oriented design approaches,” Technical Report CW 435, Department of Computer Science, K.U.Leuven, Leuven, Belgium, Feb 2006.

[58] R. Pawlak, L. Duchien, and L. Seinturier, “CompAr: Ensuring Safe Around Advice Composition,” In *Proceedings of 7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS),* volume 3880 of LNCS, pages 75-105, Athens, Greece, June 2006. Springer-Verlag.

[59] A. Mehmood and D. Jawawi, “A quantitative assessment of aspect design notations with respect to reusability and maintainability of models,” In *Software Engineering Conference (MySEC)*, 2014 8th Malaysian, pages 136-141. IEEE, Sept 2014.

[60] D. Truscan, J. Vain, M. Koskinen, “Combining aspect-orientation and UPPAAL timed automata,” *ICSOFT-PT : Proceedings of the 9th International Conference on Software Paradigm Trends, Vienna, Austria, 29-31 August, 2014.* Ed. Holzinger, Andreas; Cardoso, Jorge; Cordeiro, José; van Sinderen, Marten; Mellor, Stephen. SciTePress, 159-164.

[61] F. Sanen, R. Chitchyan, L. Bergmans, J. Fabry, M. Sudholt, and K. Mehner, “Aspects, dependencies and interactions: Report on the workshop adi at ecoop 2007,” in *Proceedings of the 2007 Conference on Object-oriented Technology, ser. ECOOP’07*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 75-90, online, last accessed 18.05.2015. Available: http://dl.acm.org/citation.cfm?id=1787553.1787563

[62] R. Alur, C. Courcoubetis, and D. Dill, “Model-checking for real-time systems,” in *Logic in Computer Science, 1990. LICS’90, Proceedings, Fifth Annual IEEE Symposium on e. IEEE*, 1990, pp. 414-425.

[63] M. Badri, L. Badri, and M. Bourque-Fortin, “Generating unit test sequences for aspect-oriented programs: towards a formal approach using UML state diagrams,” in *Information and Communications Technology, 2005. Enabling Technologies for the New Knowledge Society: ITI 3rd International Conference on. IEEE*, 2005, pp. 237-253.

[64] D. Xu et al., “Automated Test Code Generation from UML Protocol State Machines,” in *Proc. of the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE’07)*, 2007.

[65] D. Xu and X. He, “Generation of Test Requirements from Aspectual Use Cases,” in *Proc. 3rd workshop on Testing aspect-oriented programs (WTAOP’07)*. ACM, 2007, pp. 17-22, online; last accessed: 18.05.2015.

[66] D. Xu and K. E. Nygard, “Threat-driven modelling and verification of secure software using aspect-oriented petri nets,” IEEE Transactions on Software Engineering, vol. 32, no. 4, pp. 265-278, 2006, online; last accessed: 18.05.2015.

[67] D. Xu et al., “Testing Aspect-oriented Programs With UML Design Models,” Intl. *Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, vol. 18, no. 3, pp. 413-437, 2008.

[68] S. Ali, L. Briand, and H. Hemmati, “Modelling Robustness Behavior Using Aspect-Oriented Modelling to Support Robustness Testing of Industrial Systems,” Simula Research Laboratory, Tech. Rep. 2010-03, 2010.

[69] D. Truscan, J. Vain, M. Koskinen, “A Tool-supported Approach for Introducing Aspects in UPPAAL Timed Automata,” in *Software Technologies - The 9th International Conference, ICSOFT 2014, Vienna, Austria, 2014, Revised Selected Papers*, 2014.

[70] J. Iqbal, L. Tsiopoulos, D. Truscan, J. Vain, and I. Porres, “The Crisis Management System – A Case Study in Aspect-Oriented Modelling Using UPPAAL,” Turku Centre for Computer Science, 2016.

[71] J. Vain, D. Truscan, J. Iqbal, L. Tsiopoulos, “On the Benefits of Using Aspect-Orientation in UPPAAL Timed Automata,” *Conference Proceedings. 2017 International Conference on Infocom Technologies and Unmanned Systems*

(ICTUS) (Trends and Future Directions). Ed. S. K. Khatri, R.K. Kapur, A. Rana S. Singh, P.K Kapur, New Delhi: Excellent Publishing House, 81-88.

[72] K. Čerāns, “Decidability of bisimulation equivalences for parallel timer processes,” in *Computer Aided Verification: Fourth International Workshop, CAV ’92 Montreal, Canada, June 29 -- July 1, 1992 Proceedings*, G. von Bochmann and D. K. Probst, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 302-315.

[73] R. Needham, M. Schroeder, (December 1978) "Using encryption for authentication in large networks of computers," Communications of the ACM. 21 (12): 993-999. doi:10.1145/359657.359659

[74] J. Katoen, “Concepts, Algorithms, and Tools for Model Checking,” ser. Arbeitsberichte des Instituts für Mathematische Maschinen und Datenverarbeitung. IMMD.

[75] A.Yakout, A. Mohamed, Nov 2010, „Case Study: Aspect-Oriented Requirements Engineering,“ last accessed: 18.05.2015. [Online] Available: http://www.ccsenet.org/journal/index.php/cis.

[76] A. Kuusik, E. Reilent, K. Sarna, and M. Parve, “Home telecare and rehabilitation system with aspect-oriented functional integration,” Biomedical Engineering. DOI: 10.1515/bmt-2012-4194

[77] A. Anier, J. Vain, L. Tsiopoulos, (2017) “DTRON: A tool for distributed model-based testing of time critical applications,” Proceedings of the Estonian Academy of Sciences, 66 (1), 75-88.10.3176/proc.2017.1.08.

ACKNOWLEDGEMENTS

At the time I was writing this thesis, I received emotional support from many people – and without the support and the people, I would not have written it with such a joy. I am truly grateful to everyone who contributed to the mental support. I will thank them personally.

Yet, I must single out certain people for special thanks. Firstly, I am grateful to my former colleagues Oliver Väärtnõu and Alar Kuusik in Eliko competence centre. Along with the support on understanding the practical design issues highlighted in thesis use a practical case study example. I would like to thank Enar Reilent for encouraging me to write the articles on that case study. Discussions with him and others in the development team, provided me with insight and tips on how to stay on the right track with practical testing problems.

I appreciate the time I spent at the Technical University of Graz in winter 2013/2014 and I am grateful to the people I met there. I recall with gratitude the inspiring time in this wonderful environment.

A special word of thanks goes to my supervisor Professor Jüri Vain as without him and his guidance, and support, this thesis would never have been completed.

Lastly, big thanks to my husband and parents, who took care of my children and me during the period of writing the thesis. I did intensive writing at different periods during my doctoral studies. I am deeply thankful for your patience and love.

Funding

This work has been partially supported by the Competence Centre programme of Enterprise Estonia.

The Doctoral School has contributed in the participation at conferences and summer schools. The last conference visit was supported by the European Social Fund’s Doctoral Studies and Internationalisation Programme (DoRa).

I received financial support from the Tiigriülikool during my studies, which is very much appreciated. I am truly grateful for the support from all sources.

ABSTRACT

You know you have achieved perfection in design, not when you have nothing more to add, but when you have nothing more to take away.
Antoine de Saint-Exupery

Model-based testing (MBT) is an umbrella term specified by ETSI standard ES 202 951 v1.1.1 (2011-07) that captures various approaches to generating tests from models of systems under test (SUT). A system model is a computer-readable behavioural model that describes the intended external operational characteristics of the SUT.

The standard of MBT captures basic concepts and notations necessary for modelling and testing the SUT. Due to its generality the standard is independent of a specific modelling language. It mandates only modelling concepts that support tool usage and facilitate the generation of tests.

Compared to traditional testing methods, in MBT, the testing effort is shifted from mere test purpose specifications to modelling the requirements which the SUT should conform to. However, a recent survey by Binder [1] showed that two of the main challenges of MBT are in updating the models and in handling their complexity. In addition, the models used in MBT are not always intuitive and usually only part of the system behaviour is modelled to reduce the test generation effort. Therefore, modularization and abstraction techniques applicable in creating test models have become key factors that determine the usability of MBT in applications of a practical scale.

Aspect-orientation offers a new modularization concept for improving the modularity of crosscutting concerns in software development. Aspect-Oriented (AO) Software Development is a paradigm, based on aspect-oriented programming (AOP) [2, 19] that addresses the effects of crosscutting concerns on software artefacts: *scattering* (specifications related to one concern are distributed over several units), and *tangling* (a given unit contains specifications related to several concerns). The main principle of Aspect-Oriented Software Development (AOSD) is to develop multiple concerns of a software system in isolation (via aspects) and later on to combine (*weave*) them into a complete working system. The perceived benefits of AOSD are [3, 20]:

- improved separation of concerns;
- ease of maintenance, evolution and customization; and
- greater flexibility in development.

Aspect-Oriented Modelling (AOM) [21] combines the ideas behind AOSD with those of model-based (MB) software development, where the main focus is placed on how different concerns of the system can be modelled independently

and combined later on via composition mechanisms [4]. Experiments show that using AOM techniques provides models of better quality and improved readability [22]. Results from [5] confirm that aspect state machines used as test models are significantly more complete and correct, although their construction takes significantly more time than the standard approach with state machines that directly model the entire system behaviour, including crosscutting concerns.

This thesis presents a novel approach to aspect-oriented modelling and testing to address the needs of MBT. In particular, the approach aims at providing assistance for incremental test model creation as well as for abstract test purpose specification by referring to attributes of aspects using symbolic expressions. The proposed AO modelling principles and test development steps are implemented based on Uppaal Timed Automata [48] which allow specifying not only functional but also timing features, data dependencies and synchronization conditions of the SUT. The thesis validates the AO test development approach on a practical case study, namely - "Home Rehabilitation System" (HRS) and the evidence of the advantages is provided by all steps of AO test development.

KOKKUVÕTE

Teate, et Te olete saavutanud disainis täiuslikkuse mitte siis, kui teil pole midagi lisada vaid siis, kui teil pole enam midagi ära võtta.
Antoine de Saint-Exupery

Mudelipõhine testimise määratlus on antud ETSI standardis ES 202 951 v1.1.1 (2011-07), kuhu on kogutud nõuded erinevate testimisviiside ühtseks käsitluseks. Standardi eesmärgiks on anda ühtne raamistik testitava süsteemi mudelitest testide genereerimise meetoditele. Süsteemi mudeli all mõeldakse arvutile loetavat mudelit, mis kirjeldab testitava süsteemi käitumist ja selle parameetreid. Lisaks kirjeldab standard testitava süsteemi modelleerimiseks ja testimiseks vajalikke põhikontseptsioone ja notatsioone. Oma üldisusastme tõttu on standard sõltumatu konkreetsest modelleerimiskeelest..

Võrreldes traditsiooniliste testimismeetoditega on mudelipõhises testimises raskuspunkt liikunud testieesmärgi spetsifitseerimiselt nõuete modelleerimisele, millele testitav süsteem peab vastama. Hiljutine Binderi uuring [1] näitab, et mudelipõhise testimise peamine raskus on mudelite konstrueerimine ja ajakohastamine ning mudeli keerukusest tulenevate probleemide lahendamine. Lisaks sellele ei ole mudelid mitte alati intuitiivsed ja tavaliselt modelleeritakse süsteemi käitumist osade kaupa, et vähendada testide genereerimise töömahtu. Seetõttu on testimudelite loomiseks kasutatavad modulariseerimise ja abstraktsioonitehnikad peamiseks faktoriks, millega on määratud mudelipõhise testimise kasutatavus praktilistes rakendustes.

Aspekt-orienteeritus annab uudse lähenemine modulaarsusele võimaldades paremini adresseerida tarkvaraarenduses probleeme, millele on seni suhteliselt vähe tähelepanu pööratud. Aspekt-orienteeritud tarkvara arenduse ideed põhinevad aspekt-orienteeritud programmeerimisel [2, 19] mis käsitleb arenduse tülikamaid probleeme nagu nõuete hajutamine (*scattering*) erinevatel komponentile ja kokkusegamine (*tangling*), kus komponendi spetsifikatsioonis on elemente erinevat liiki nõuetest. Aspekt-orienteeritud tarkvaraarenduse võtmeidee on arendada esmalt tarkvara aspekte omavahel sõltumatult ning hiljem põimida (*weave*) need terviksüsteemiks. Aspekt–orienteeritud tarkvaraarenduse peamisteks eelisteks on probleemide lahuskäideldavus, tarkvara hõlpsam hooldamine, haldamine, arendamine ja kohandamine ning arendusprotsesside suurem paindlikkus [3, 20].

Aspekt–orienteeritud modelleerimine [21] ühendab endas aspekt-orienteeritud tarkvaraarenduse ja mudelipõhise tarkvaratestimise ideed keskendudes sellele kuidas esmalt modelleerida süsteemi sõltumatult tema osade kaupa ning hiljem need osad omavahel siduda [4]. Katsed kinnitavad, et aspekt-orienteeritud modelleerimistehnika abil on võimalik parandada mudelite arusaadavust ning vältida modelleerimise vigu [22]. Publikatsiooni [5] tulemused näitavad, et kuigi testimudelitena kasutatavate aspekt-orienteeritud olekumasinate konstrueerimiseks kulub oluliselt rohkem aega kui standardsete olekumasinate konstrueerimiseks, on aspekt-orienteeritud mudelid oluliselt korrektsemad ja täielikumad.

Käesolev väitekiri pakub uudse lähenemise aspekt–orienteeritud modelleerimisele ja selle rakendamisele mudelipõhises testimises. Lisaks modelleerimistehnikale on väitekirja eesmärgiks näidata kuidas testieesmärke abstraktselt spetsifitseerida kasutades selleks aspekt-orienteeritud mudeli atribuute. Modelleerimise formalismist tulenevad AO modelleerimispõhimõtted ja testide arendamise sammud on konkretiseeritud Uppaali ajaga automaatide formalismil [48] ning seda toetava tarkvara abil, millega saab testida lisaks funktsionaalsetele omadustele ka ajastusomadusi, andmete sõltuvusi ning paralleelsete protsesside sünkroniseerimistingimusi. Väitekiri valideerib aspekt-orienteeritud lähenemise otstarbekust testide arendamisel praktilise juhtumianalüüsi “Kodune taastusravi süsteem” näitel ja esitab kvantitatiivsed tõendid aspekt-orienteerituse eelistest.

Appendix A

PAPER A:

Külli Sarna, Jüri Vain. **Exploiting aspects in model-based testing**. In: *FOAL'12: Proceedings of the Eleventh Workshop on Foundations of Aspect-Oriented Languages, March 26, 2012, Potsdam, Germany:* New York: ACM, 45 - 47.

Exploiting Aspects in Model-Based Testing

Külli Sarna
ELIKO Competence Centre in Electronics-, Info- and Communication Technologies
Tallinn, Estonia
kylli.sarna@eliko.ee

Jüri Vain
Department of Computer Science
Tallinn University of Technology
Tallinn, Estonia
vain@ioc.ee

Abstract

We introduce an approach to exploiting aspects in model-based testing and describe how an aspect-oriented model for testing purposes can be constructed. At first, we introduce the aspects to be addressed in testing safety and time critical systems and describe how the aspects enhance in defining test cases. We present a way how behavioural aspect models are defined formally as refinements of extended timed automata models, and how the aspect models are used for generating abstract online testers. Applying these techniques aspect-wise allows one to structure the model-based testing process in terms of well-defined model transformation steps. The approach is illustrated with an ATM case study.

Categories and Subject Descriptors:

D.2.5 [Testing and Debugging, Testing tools]: model construction for testing.

General Terms: Design, Aspects, Theory, Verification.

Keywords: Aspect Oriented Modelling; Model Refinement; Model-Based Testing; Test Generation.

1. Introduction

Model-based testing (MBT) and automated test generation have high potential for reducing the costs of testing activities in software development. Especially, it applies in the field of complex distributed and embedded systems where thousands of tests are carried through repeatedly, and the testing processes need to be well coordinated. As shown in [3] the model-based test case generation has demonstrated its practical applicability in test automation. The ETSI standard ES 202 951 v1.1.1 (2011-07) defines key notions and characteristics of MBT: a model of the System Under Test (SUT) is used to retrieve a set of test cases; the test cases are selected by means of a test case specification.

One of the practical obstacles in MBT is the complexity of models that specify industrial applications. Those models suffer often from the lack of clarity and/or integrity even though the semantics of underlying formalism is well-defined. Current modelling approaches provide good support for modularizing design models supporting component–based and/or hierarchical state models. On the other hand, they provide poor support for isolating crosscutting features, that is, functionality that is spread across the modules of the software and tangled with other functionality [5].

In this paper we make use of Aspect Oriented Modelling (AOM) for systematic definition of test cases and for grouping the test cases into consistent test suites when system level and integration tests are designed. In software testing, the test cases usually address the requirements items and the test purpose of each test case can be considered as specification of that requirement item. In the complex distributed systems some crosscutting concerns, e.g. security strongly affects the implementation architecture. AOM enhances modularization of concerns that cannot be modularized using other techniques.

2. Preliminaries

2.1 Aspect oriented modelling

AO concepts are currently exploited to model a system from the beginning of development to its implementation and testing. AOM deals with requirements that cut across the primary modularization of a system, e.g., logging, tracing, security, persistence (non-functional aspects). AO models propose the separation of the crosscutting concerns of software systems into separate entities. This separation avoids the tangled concerns of software and allows the reuse of the same aspect in different entities of the software system (components, modules, etc.). The separation of concerns also improves the isolated maintenance of the different concerns of the software systems. In AOM approach we distinguish a *primary model* and several *aspect models*. Crosscutting features are treated as patterns described by aspect models, and other features are described by a primary model. The result of composing aspect and primary models into an integrated model is called the *composed model*. An aspect model can be integrated with the primary model in many places and in different ways. AOM techniques use the term *advice* for the action an aspect will take and *join points* for where these actions will be inserted in the primary model. *Point cuts* are used to specify the rules of where to apply an aspect. Advice, joint points, and point cuts are specified as one entity, called an *aspect* [5-9].

2.2 Model-based testing

By model-based testing we mean a black box technique where state machine models are used as specifications of observable interactions between SUT and its environment. The model is examined to generate test suites. The coverage of model structural elements (states and transitions) can be used as a measure of thoroughness for a test suite. A test purpose is a specific objective (or property) that the tester would like to test, and can be seen as a specification of a test case. It may be expressed in terms of coverage items, scenarios, duration of the test run etc. In AO setting we address the test purpose in terms of aspects and aspect related properties. Thus, the test cases for a test purpose should be derived from the aspect model(s) of concern abstracting from the rest of SUT specification. As an example of a test purpose, we consider an informal requirement "test of a state change from state A to state B" in an aspect model M_a. For this purpose a test case should be generated that covers the specific state change in M_a. At first, it requires that the test drives SUT in state A, then specified transition is executed and when B is reached the test should terminate in some safe state of M_a. For non-deterministic systems a single test sequence may never reach the test goal and instead of a sequence we need an online testing strategy that is capable of reaching the goal even when SUT provides non-deterministic responses to test stimulus. The issue is addressed in [1] where the reactive planning online tester synthesis method is introduced. Although the [1] relies on EFSM specifications it can be extended easily to models with constant time bounds. In the rest of the paper we use Uppaal Timed Automata (UPTA) [4] to specify such aspect models of SUT.

2.3 Uppaal timed automata

UPTA are appropriate for systems that can be modelled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables [4]. Typical application areas include real-time controllers and communication protocols in particular, those where timing aspects are critical.
The graphical representation of a timed automaton is considered as a directed graph, where *locations* are represented by the vertices of this graph that are connected by *edges* (see Figure 1). Locations are labelled with *invariants*. Invariants are conjunctive expressions of boolean expressions on model variables and simple bound conditions on clock variables, e.g. 'Clock1<= const1'.

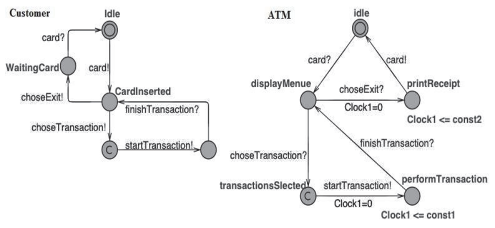


Figure 1. The primary model of ATM.

Edges are annotated with *guards*, *synchronisations* and *updates*. An edge is enabled by a guard in a state if and only if the guard evaluates to true. Processes (parameterized instances of Uppaal automata) can synchronize over channels. Edges labelled with a common channel synchronise, e.g. edge 'WaitingCard->Idle' of Customer automaton and edge 'printReceipt->Idle' of ATM automaton synchronize over channel 'card'. Updates express the change of the state of the system when the edge is executed, e.g., update 'Clock1 = 0' resets the value of model clock 'Clock1'.

3. Construction of Aspect Models

In UPTA we consider aspect models as the refinements of model syntactic units (locations and edges) that represent join points in the primary model. We call them *location refinement* ($\sqsubseteq_l$) and *edge refinement* ($\sqsubseteq_e$) respectively.
Let the refinement of an element *el* in a primary model M be represented by automaton M^{el} that is composed with the primary model M by synchronized parallel composition $\|_{sync}$, i.e., $M \sqsubseteq M \|_{\mathrm{sync}} M^{el}$. Synchronization of M and M^{el} is needed to preserve the contract of element *el* with its context after refinement. Technically, it means decorating the primary automaton M with auxiliary channel labels *ch* to synchronize the entry and leave points to/from the element *el* of M. For further elaboration we define the location and edge refinement relations separately.
We say that a synchronous parallel composition of automata M and M^{li} is a *location refinement* for location l_i of M, ($M \sqsubseteq_l M \|_{\mathrm{sync}} M^{li}$) iff $\mathrm{l}_i \in N_M$, and $\exists\, M^{li}$ s.t. $P_1 \wedge P_2 \wedge P_3$:
P_1 (*interference free new updates*): no variable of M is updated in M^{li}, i.e. no variable of M occurs in the left-hand side of any update in M^{li};
P_2 (*preservation of non-blocking*): $[(M \| M^{li}), (\mathrm{l}_0, \mathrm{l'}_0) \vDash \mathrm{E}\Diamond$ deadlock$] \Rightarrow [M, \mathrm{l}_0 \vDash \mathrm{E}\Diamond$ deadlock$]$;
P_3 (*non-divergency*): $inv(\mathrm{l}_i) \equiv x \leq n$ for a clock $x \in C_M$, $n<\infty$ $\Rightarrow [M^{li}, \mathrm{l'}_0 \vDash \mathrm{l'}_0 \leadsto_n \mathrm{l'}_F]$, where" $\leadsto_n$" denotes bounded reachability operator with time bound n; locations $\mathrm{l'}_0$ and $\mathrm{l'}_F$ denote respectively auxiliary pre- and post-nodes in the context frame of the refinement.
P_2 and P_3 are specified as Uppaal model checking queries expressed in TCTL. 'deadlock' denotes a standard predicate in Uppaal about the existence of deadlocks in the model. P_3 requires that the invariant of l_i is not violated due to accumulated delays of M^{li} runs.
Location refinement can be applied when the aspect model specifies behaviour that in the primary model is represented as non instantaneous and time bounded location. A synchronous parallel composition of automata M and M^{li} is an

edge refinement for edge t_i of M, ($M \sqsubseteq_e M \| M^{ei}$) if conditions $P'_1 \wedge P_3 \wedge P_4 \wedge P_5$ are hold:
P'_1 (*interference free new updates*): no variable of M is updated in M^{ei}, i.e. no variable of M occurs in the left-hand side of any update in M^{ei};
P_3 (*guard sequentialization*): let $\langle l'_0, l'_F \rangle$ denote a set of all feasible paths from the initial location l'_0 to final location l'_F in M^{ei} and $\langle l'_0, l'_F \rangle_k \in \langle l'_0, l'_F \rangle$ be k-th path in that set, then $\forall k \in [1, |\langle l'_0, l'_F\rangle|]. \wedge_{j\in[1,Length(k)]} grd(t'_j) \Rightarrow grd(t_i)$, i.e., the conjunction of edge guards of any path in $\langle l'_0, l'_F\rangle$ is not weaker than the guard of the edge t_i refined.
P_4 (0-*duration unwinding*): $\forall l'_i \in (N_{Mei} \setminus l'_0)$. $Type(l'_i)$ = committed, i.e., all edges in the refinement M^{ei} must be atomic and all locations instantaneous.
P_5 (*non-divergency*): $grd(t_i) \Rightarrow M^e, l'_0 \models A\Diamond\, l'_F$, i.e. validity of $grd(t_i)$ implies the existence of a feasible path in M^{ei}.
Similarly to location refinement we implement the edge refinement by means of $\|_{sync}$ and *context frame* that includes auxiliary locations l'_0 and l'_F, and an edge between them.

4. Example ATM

We demonstrate the use of refinement transformations of Section 3 for composition of aspect models. The primary model of ATM depicted in Figure 1 includes interacting automata Customer and ATM. Refinements in Figure 2 specify aspects of interest: (*i*) aspect model Transaction is defined as location refinement of both ATM and Customer automata; (*ii*) further edge refinement of ATM.Transaction introduces EnquireBalance aspect. Since the refinement (*ii*) introduces new interaction between ATM and a new actor Server (not shown in the model) the edge introduced is labelled with channel 'balanceCheck!'.

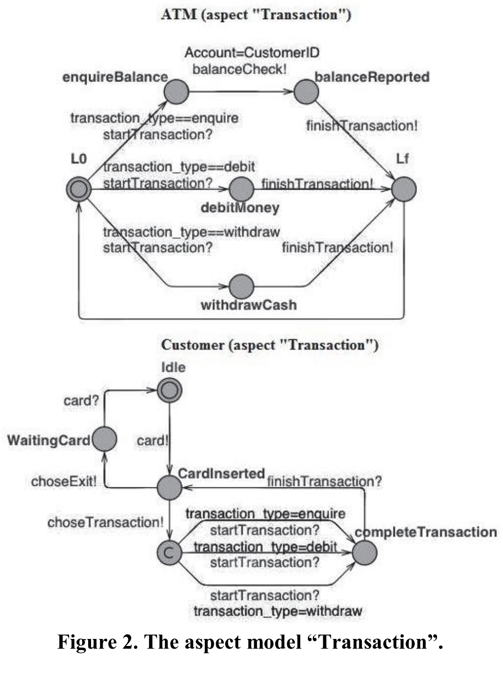


Figure 2. The aspect model "Transaction".

When the aspect related tests have to be generated from the composed model of SUT that includes automata of Figures 1 and 2, we can ignore all the transactions the aspects of interest do not depend on. For instance, when testing the transaction balanceCheck! between ATM and Server the tester model is extracted from the composition of automata Customer and Customer.Transaction by algorithm of [1] so that the test sequence <*card*!, *choseTransaction*!, *transaction_type*:=*enquire, startTransaction*!, *wait,* [*finishTransaction*? | *Timeout*>=*const*1, *TESTFAIL*], *choseExit*!, *card*?, *TESTPASS*> can be executed.

5. Conclusion and Future Work

The refinement-based AOM approach described in this paper has been applied for construction of SUT models of a patient health remote monitoring system [2]. Switching the irrelevant aspects off at test generation allowed saving the SUT model structural complexity in test generation up to 80%. Currently, the bottleneck is automated support for aspect model engineering that needs manual aspect model construction and formulating model-checking tasks. Implementation of the context frame generation for aspect models derived by refinement work is in progress.

References

[1] Vain, J., et al. 2011. Online testing of nondeterministic systems with reactive planning tester. Dependability and Computer Engineering: Concepts for Software-Intensive Systems (113-150). Hershey, PA: IGI Global.

[2] Kuusik, A., et al. 2010. Software architecture for modern telehome care systems. In Proceedings of the 6th International Conference on Networked Computing. IEEE Computer Society Press (326-331).

[3] Pfaller, C. 2008. Requirements-based test case specification by using information from model construction. In Proceedings of the 3rd international workshop on Automation of software test. ACM, New York, NY, USA 7-16.

[4] J. Bengtsson, W. Yi. 2004. Timed automata: Semantics, algorithms and tools. Lecture Notes on Concurrency and Petri Nets, Lecture Notes in Computer Science vol. 3098.

[5] Yedduladoddi, R. 2009. Aspect Oriented Software development: An Approach to Composing UML Design Models. VDM Verlag Dr. Müller.

[6] Schauerhuber, A., et al. 2006. A Survey on aspect-oriented modeling approaches.

[7] Kienzle, J., et al. 2010. Aspect-Oriented Design with Reusable Aspect Models. In Transactions on aspect-oriented software development VII. Springer-Verlag, Berlin, Heidelberg 272-320.

[8] Rashid, A. 2008. Aspect-Oriented Requirements Engineering: An Introduction. In Proceedings of 16th IEEE.

[9] Disenfeld, C and Katz, S. 2011. Compositional verification of events and observers. In Proceeding of 10th FOAL. ACM, New York, NY, USA, 1-5.

Appendix B

PAPER B:

Alar Kuusik, Enar Reilent, Külli Sarna, Marko Parve. **Home telecare and rehabilitation system with aspect-oriented functional integration.** In: *Biomedical Engineering/ Biomedizinische Technik: The 46th annual conference of the German Society for Biomedical Engineering, Jena, Germany, September 17-19, 2012. (Edit.) Dössel, O.* De Gruyter, 1004 - 1007.

Biomed Tech 2012; 57 (Suppl. 1) © 2012 by Walter de Gruyter · Berlin · Boston. DOI 10.1515/bmt-2012-4194

Home telecare and rehabilitation system with aspect oriented functional integration

Alar Kuusik[1], Enar Reilent[1], Külli Sarna[1], Marko Parve[2]
[1]ELIKO Technology Competence Centre, Tallinn, Estonia; Firstname.Lastname@eliko.ee
[2]East-Tallinn Central Hospital, Tallinn, Estonia; Marko.Parve@itk.ee

Abstract

Modern home telecare is shifting from emergency conditions detection to the wellness monitoring, treatment plan observation and rehabilitation. Because of such shift the telecare attracts more beneficiaries besides of traditional elderly and chronically ill people. From the other side, more complex systems for simultaneous treatment and safety monitoring involve more stakeholders – physiotherapists, primary doctors, carers and patients. Possibly contradicting requirements and perspectives of different partners are rising significant system usability and planning problems. In the paper we describe an extendable and requirement conflict tolerant software framework for home health hubs (HHH) for simultaneous motor rehabilitation (post-stroke and -arthroscopy) and patient safety observation. Aspect oriented software design (AOSD) and -requirement engineering (AORE) are used to simplify and automate integration of software subprograms representing interest or viewpoints of different participators.

1 Introduction

Due the scalability problems of traditional healthcare service models there is growing tendency to decentralization, shifting care from the hospital to the community. Home-centred health care has become an important health management issue [1, 2]. In near further home telecare shall become an efficient extension of hospital treatment procedures [3, 4]. In the paper we describe how to extend classical (passive) home vital signs monitoring solutions with user friendly (active) motor rehabilitation functionalities. The development focuses on a flexible agent based software framework minimizing the home patient safety risks, simplifying treatment and training plan setup and monitoring. The framework is universal and applicable for different homecare scenarios of patient monitoring and training in home rehabilitation assistant (HRA) system. Our current software implementation for home health hub (HHH) device is mainly targeting post-surgery and post-stroke training.

Focused exercise training is beneficial for stroke recovery [5] and essential therapy component after surgeries. Training quality on such cases rely on the availability of skilled therapists or training robots. Unfortunately assistive machines are intended for hospital use only and have limited accessibility due the price constraints.

As stressed by Sabatini already in 2005 there is a need for ambulatory monitoring systems providing objective assessment of human functional abilities outside of laboratory settings [6]. One reason of lack of integrated systems supporting both home safety monitoring and training support is the complexity of handling the requirements of different parties: patient in comfort zone, primary care doctor focusing on patient's general safety, physiotherapist targeting recovery of joint movements. Today's home training procedures composed by physiotherapists are intentionally conservative to avoid any potential risk to patient's safety. Additionally, it would be too complex for patient or direct carers to (manually) adapt individual treatment plan according to changes in health condition or gained progress. Such adaptive treatment would be possible, at least in some extent, with decision support systems capable of handling interests of different stakeholders.

A practical agent based software framework is described in the paper using aspect oriented software design (AOSD) and -requirement engineering (AORE) methodology for designing the telecare system for formally safe home rehabilitation and patient monitoring. Developed prototype software is able to instruct user during exercising in real time and takes into account of personal defeasible safety rules.

2 HRA instrumentation

Home rehabilitation assistant (HRA) system targeting machine supervised patient exercising contains subcomponents: HHH as data acquisition gateway and user interface device; generic vital signs sensor system (VSSS); micro-electro-mechanical sensor system (MEMSS) for patient physical activity and exercise quality monitoring.

The HHH (Image 1) acquires and analyses sensor data, interacts with the patient and communicates with the hospital EHR repository over the Internet. For realizing HRA functionality on HHH we reuse our existing telemedicine home gateway solution [7] (Image 1). The sensor data, patient parameters, configurations, and system states are kept in the in-memory database organized as whiteboard data exchange solution.

Besides the extended VSSS subsystem (containing BP and SpO2 meters, weight scale, sensorized box of pills), HHH serves as a MEMSS data processor in real time. Computerized analysis of exercising data is essential for presenting intelligible numbers to the physiotherapists.

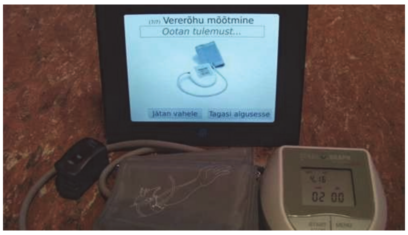


Image 1 Home Health Hub with sensors

Several different practical methods of human motion and gait analysis are concluded in a review [8]. Several off-the-self MEMSS based (patient) physical activity monitoring products exist from different vendors - activPAL, PAM-Sys, ActiGraph and others. However, such devices typically act as standalone activity loggers or can just recognize basic patterns of walking, sitting, lying. Extensive review of experimented activity classification methods for human activity recognition by MEMSS is presented by Preece [9]. Tests with an original MEMSS system [10] (Image 2) showed that limb training quality can be sufficiently correctly monitored using neural networks (NN) and k-nearest neighbor (kNN) classification methods and 3 gyroscope sensor nodes.

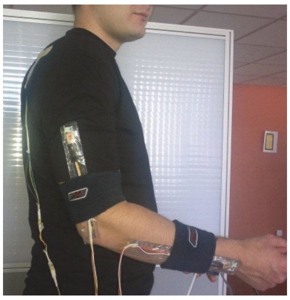


Image 2 Prototype MEMSS for hand training

3 HRA use scenario

We see the purpose of HRA as a low cost assistive tool guaranteeing correct exercising without immediate physiotherapist supervision. A physiotherapist initially instructs the patient and also configures the HRA equipment at hospital. During the calibration session the patient does exercises that the physiotherapists evaluates correct or incorrect. The HHH records signal patterns of MEMS sensors used later as reference data for exercising assessment. For each exercise reference data for training a neural network has to be recorded.
For creating the rehabilitation plan the physiotherapist first choses and adjusts exercises for the patient. The plan is the basis for the system to monitor patient's training at home, give reminder alarms and recommendations as well as the basis for the evaluation of the patient's independent training. Some important details, however, does not fit into the exercising schedule, these must be added as separate rules. For example, the patient is required to measure BP and HR at given times or in relation with exercising sessions. The exercising plan could be affected by the measurement results, for example if the BP is too high before the exercising the session might be delayed.
As an essential enhancement the proposed solution supports simultaneous training and safety monitoring. The preconfigured HRA system tracks the treatment plan fulfillment at home. Primary care doctors or carers can insert additional safety rules to the HHH. All patient movements are collected while the sensors are operational. During an exercising session the system monitors if the patient is doing correct number of exercises in the required order. Also the similarity measure with the reference is calculated for individual movements and displayed to the patient with a multimedia guiding option.
All of the event log and statistics that are gathered during the home rehabilitation period is recorded for physicians and can be uploaded to the hospital information system using previous telemedicine framework [11].

4 Aspect oriented approach for realizing HRA software

Due to the functional complexity of proposed HRA solution there is a need for provably correct approach for solving contradictory requirements. A fundamental principle in addressing system's complexity is separation of concerns. It is important in the analysis phase to define the application decomposition and identify the inventory of concerns that lay ground for modularization and structure of a future requirements model. Aspect oriented requirement engineering (AORE) is a methodology that can help to improve requirements completeness, maintainability, and reduce cost of software development [12, 13]. AORE is suitable for distributed systems applications lacking system's "global picture" [14] or integration of independent, goal-oriented tasks. The model is put together using all viewpoints of stakeholders and AORE analysis techniques.
Aspect oriented software development (AOSD) aims at addressing crosscutting concerns by providing means for their systematic identification, separation, representation and composition. Crosscutting concerns (CC) are encapsulated in separate aspects and composition mechanisms are later used to weave them with other modules.

4.1 Implementation of HRA system

Now we describe actual HHH software which is realized as the optimal set of aspects gained from AORE methodology. Each aspect is following its own goal independently from the targets of others. All aspects are implemented as one or more software agents dealing with specific type of data items on the HHH whiteboard corresponding to certain activities and situations. Thanks to the AORE meth-

odology new agents can be added to the HRA system in runtime which significantly improves the system usability. Particular agents to be present in certain cases are defined by the treatment plan and its constraints. All components of the system are intended to run independently and complement the overall system behavior without need to explicitly know about the presence of other agents. Cross-cutting and general interoperability is achieved by the utilization of whiteboard where every agent could interact with the entire knowledge existing in the system for intervening ongoing workflows. An agents does not overwrite values given by other agents, instead of that it adds its own version and marks itself as the source. All the aspects and agents are ordered by priority. Depending on the situation the agent picks the value with the highest source priority or the value inserted by someone with lower priority compared to itself.
One of the key issues of the system is its scalability which means introductions of new aspects and agents in the future. The HRA solution is currently implemented on an off-the-self Linux running HHH device with a sub-gigahertz ARM processor. On the given platform writing 1500 rows of bulk data to the HHH whiteboard takes 100ms. A long term average reading time of one row is around 100us. Assuming that the agents are performing simple tasks and using small number of rows on the whiteboard, it is possible to run the HRA effectively with around 100 independent software agents.

4.1.1 Treatment plan aspect

Treatment plan aspect is responsible for steering the patient to follow the treatment plan. The aspect is split to several smaller agents. "*Treatment plan follower*" agent checks current time and the original treatment plan inserted by the doctor. When it is time to start exercising again, the agent prepares data structures for upcoming exercising session. Original treatment plan is left unmodified; all adjustments are done to the copy.
"*Exercising plan feeder*" agent checks for and interprets the actual session plan whenever it is present and generates guidance messages for the patient. As the agent reads from the whiteboard what the patient is supposed to do and what exercises are already done it can determine what the patient should do next. The output is not directly rendered to the screen but saved back to the whiteboard to be picked up by corresponding agents.
A slightly different task is given to the "*sequence checking*" agent. The patient could do whatever movements regardless of the exercising plan and system's messages. As the order of exercises, which includes relaxation in certain points, is relevant then the patient should be notified if he or she does something wrong in respect to the plan.
The treatment plan "*adjusting agents*" adjust the session plan accordingly to the constraints and rules. For example, if the patient has completed all previous sessions during the last five days without deviation from the plan the next session could increase the number of repetitions for some exercises.

4.1.2 Correctness aspect

Correctness of movements is considered as a separate issue and handled by "*Correctness evaluator*" agent who analyses incoming movements by comparing them to the examples recorded at the hospital and calculate the similarity measure SM. These evaluations can be presented to the patient during the exercising session for indicating if the patient is doing exercises correctly or not. At the end of the exercising session when all movements have been captured and evaluated the "*guide agent*" could present the media guide if some exercises have high failing ratio.

4.1.3 Rehabilitation progress aspect

The aspect of progress is basically concerned with the results of the rehabilitation and monitoring its progress. "Statistical analyzer" agent prepares reports for the doctor. The agent runs only once per day and looks back at the exercising session plans and committed exercises. Similarly it can look back at the exercises of previous days and make conclusions of how much the patient is exercising, whether the patient can tolerate more load than before, is the trend of the correctness of movements satisfactory, has the movement range of the joint increased etc. Quick reports are useful for getting overview of important circumstances of the rehabilitation when revisiting the hospital.

4.1.4 Daily living aspect

"*Inter-session activities counter*" agent pays attention to movements committed outside the exercising sessions. Provided that the sensor is online and registers patient activity when there is no exercising session going on, this information should be taken into account. If wireless sensors are used and some movements of daily living (like opening doors, picking up objects) are similar enough to the prescribed exercises and are recognized by the system the agent can adjust upcoming exercising session. When a new session plan is created the agent could decrease the number of recommended repetitions in the case where it can identify considerable load to the damaged joint to avoid overburden.

4.1.5 Safety aspect

Safety requirements have to be continuously fulfilled in runtime therefore a set of agents deal with safety monitoring and reporting of dangerous situations to various stakeholders. For example, when the "*emergency agent*" detects fall down and no further activity within the next moments, it initiates an emergency call to summon help. Also other critical situations are monitored depending on the patient's conditions and available sensor readings.
The "*doctor alert generator*" agent concentrates on the events that cannot be classified to emergencies but still need intervention. While minor problems of following the treatment plan do not need to be reported in real time as the doctor checks the data at the patient's next visit to hospital, then occasions of major violations could be checked earlier. For example, if there are no traces of any exercis-

ing activity in the past three days it sends a notification messages to the doctor.
The "*patient alert generator*" agent outputs local warning messages intended for the patient. The patient should be informed about the actions and situations which are unfavorable. Whenever some exercising fairly exceeds limits recommended by the session plan or the patient performs movements too vigorously the agent creates outputs warnings.

4.1.6 Input-output aspect

The agents responsible for direct input and output actions form the input-output aspect. Each sensor device (MEMS system, vital signal sensor) is handled by corresponding input "*adapter agent*". Adapter agents communicate to the sensors and receive raw data, restructure it and store on the whiteboard.
"*Output agents*" are adapters to output devices like the screen or speakers and also the agent for network communication. The screen adapter searches the whiteboard for information about the contents of different virtual windows (warnings, correctness measures, feedback, etc) that it renders to the physical screen areas.
The "*output manager*" performs as a mediator between the system and the output adapters. Incoming exercises are displayed on one virtual window, warnings are played by speakers, etc. If there are several possible messages suitable for one channel then they are sorted by the source priority and the information with the greatest impact is chosen. When the situation on the whiteboard is updated the agent also reappraises its output.

4.1.7 System aliveness aspect

Self-monitoring mechanism is two-layered. Every agent registers itself on the whiteboard when started and updates its personal aliveness token. The "*self-monitoring*" agent constantly monitors all registered tokens and restarts the malfunctioning agent. The self-check agent guards itself with the hardware level watchdog that reboots HHH immediately if contact is lost. Such multi-layer watchdog solution reduces the number of unnecessary restarts.

5 Conclusions

Complex monitoring and control systems involving different stakeholders are difficult to develop, maintain and use because of different viewpoints. Based on an example of patient monitoring and motor rehabilitation involving requirements of physiotherapists, patients and clinicians we demonstrated that aspect oriented requirement engineering is a practical and approach for such systems. At certain extent, essential functionalities of home based rehabilitation were presented as aspects which demonstrate applicability of AOSD techniques for user adaptive telecare. HRA software agents were described and implemented driven by AORE analysis results and tested within a whiteboard based agent software framework. The presented aspects form an optimal set for the considered HRA task.

Acknowledgement: This work was supported by Competence Centre program of Enterprise of Estonia.

6 References

[1] World Health Organization: The World Health Report 2008: Primary Health Care: Now More Than Ever. Geneva, 2008

[2] Doarn C.R., Merrell R.C.: A roadmap for telemedicine: barriers yet to overcome. Telemed J E Health 2008, No 14(9): pp 861-862

[3] Jolly K., Taylor R.S., et al.; Home-based cardiac rehabilitation compared with centre-based rehabilitation and usual care: a systematic review and meta-analysis. Int J Cardiol 2006, 111: pp 343-351

[4] Vieira D.S.R., Maltais F., Bourbeau J.: Home-based pulmonary rehabilitation in chronic obstructive pulmonary disease patients. Curr Opin in Pulm Med. No 10, 2010, pp 134–143

[5] N.F. Gordon, M. Gulanick, et al.: Physical activity and exercise recommendations for stroke survivors, Circulation, vol. 109, pp. 2031–2041, 2004

[6] Sabatini, A.M.: Inertial sensing in biomechanics: a survey of computational techniques bridging motion analysis and personal navigation. In Comp. Int. for Movement Sciences: Neural Networks and Other Emerging Techniques; Idea Group Pubilishing: Hershey, PA, USA, pp. 70–100, 2006

[7] A. Kuusik, E. Reilent, I. Lõõbas, M. Parve: Software Architecture for Modern Telehealth Care Systems, Journal of Advances on Information Sciences and Service Sciences; Vol 3, no 2, pp. 141-151,2011

[8] Zhou, H. and Hu, H.: A Survey - Human Movement Tracking and Stroke Rehabilitation, TECHNICAL REPORT: CSM-420, University of Essex, 2004

[9]S. J. Preece, J. Y. Goulermas, L. P. J. Kenney, D. Howard, K. Meijer and R. Crompton: Activity identification using body-mounted sensors—a review of classification techniques, Physiological Measurements, Vol 30 No4, 2009

[10]S. Ovsjanski: Real time wireless wearable motion monitoring system for motor rehabilitation, Master Thesis, Tallinn University of Technology, 2012

[11]A. Kuusik, E. Reilent, I. Lõõbas, M. Parve: Semantic Formal Reasoning Solution For Personalized Home Telecare, Proc. of 2010 Int. Conf. on Mechanical and Electrical Technology, pp. 72 – 76

[12]R. France, I. Ray, G. Georg, S. Ghosh: An Aspect-Oriented Approach to Early Design Modeling, in Software, IEEE, no 4, 2004, pp 173-185

[13]Gray, J., Lin, Y., Zhang, J.: Automating change evolution in model-driven engineering. Computer 39(2), pp 51–58, 2006

[14]E. P. Freitas, M. A. Wehrmeister, et al.: Using Aspect-Oriented Concepts in the Requirements Analysis of Distributed Real-Time Embedded Systems, IFIP Int. Fed. for Information Processing, vol. 231, pp 221-230, 2007

Appendix C

PAPER C:

Alar Kuusik, Külli Sarna, Enar Reilent. **Home Rehabilitation System Supported by the Safety Model.** Studies in health technology and informatics, 189, 145 - 151 (Article of scientific journal).

pHealth 2013
B. Blobel et al. (Eds.)
IOS Press, 2013
© 2013 The authors and IOS Press. All rights reserved.
doi:10.3233/978-1-61499-268-4-145

Home Rehabilitation System Supported by the Safety Model

Alar KUUSIK[1], Külli SARNA and Enar REILENT
[a] *Eliko Competence Centre*

Abstract. The paper describes a tele-rehabilitation system for simultaneous motor rehabilitation (post-stroke and post-arthroscopy) and continuous patient's condition assessment with the focus on patient safety observations. Micro-electro-mechanical accelerometer and gyroscope sensors attached to the patient gather information about the performed therapeutic exercises. The measurement data processing of the vital sign sensors and accelerometers is done in the health hub device in real time with the patient feedback. Model verification is used for providing that the specified requirements have been actually fulfilled. By the safety model validation, we supplement clinical evaluation, which means the efficacy of the system is proven by the rules given by the physician for the particular patient.

Keywords. Motor rehabilitation, tele-rehabilitation, telemedicine, model-based verification, multi agent system.

Introduction

The care is shifting from the hospital to the community due to the scalability problems in the traditional healthcare [1]. Due to the insufficient availability and remarkable service fees of neurologists and physiotherapists, the systematic evaluation of the condition changes of the patients suffering from neurodegenerative diseases and stroke survivors is complicated in sparsely populated and rural areas. In this respect, one of the major goals is the technologically assisted home therapy and continuous monitoring systems in the form of cost efficient short time rentals.

We describe a practical motor rehabilitation system dedicated to individual patients, and a safety model that ensures the system working according to the physician's rules. The developed solution is suitable for objective patient motor condition assessment. It supports training outside of clinical environments and should automatically monitor the patient's exercising process and give real time feedback in the form of instructions and warnings. Because there are several circumstances to be taken into consideration with this kind of tele-rehabilitation system the solution makes use of the model verification techniques. It has been tested before implementation by patient models and concern models.

The rest of the paper is structured as follows: The section number one elaborates the idea of using MEMS (Micro-Electro-Mechanical Systems) for motion tracking. The section number two gives the background details of the development of the system for

[1] Corresponding Author: Alar Kuusik; Eliko Competence Centre, Mäealuse2/1, 12618, Tallinn, Estonia; Email: alar.kuusik@eliko.ee

rehabilitation assistance at home. This is followed by the description of the solution of how the system ensures the required properties in the section number three and the conclusion at the end.

1. MEMS Technologies in Movement Detection

The main aim of our activities is the development of a low cost portable motion recording and analysis solution which employs lightweight low power wireless sensors and is capable for movement pattern classification for home or ambulatory training purposes. We used functional reference for our solution as the reference of Shimmer wearable sensor platform [2]. To date the Shimmer is technologically relatively obsolete – its modules are too big and heavy for convenient use.

Another option is video bridging between the clinician and the patient but video monitoring based solutions do not allow discovering small property changes (without special expensive 3D motion tracking equipment) like slightly worse flexibility. It is not well suitable for fatigue analysis with duration of some hours perhaps and requires full intention from the therapist evaluating of series of videos.

The importance of evaluation of fatigue dynamics of muscles as one essential disability component is stressed in today's clinical research. Unfortunately such long term gait monitoring is not possible with wired (ambulatory) solutions and a small wireless sensory system is required for fatigue analysis.

Meijer proposed already in 1991 [3] the use of MEMS for patient movement analysis. Ambulatory use of MEMS devices was also proposed by Foerster already in 1999 [4]. Acceleration sensors in healthcare are used mainly for recognition of patient's basic activities e.g. walking, sitting, laying [5], [6] and prevention/detection of fall down [7]. From the technological point of view MEMS are much more capable, e.g. technology allows measuring angles with accuracy of one degree, collecting simultaneous 3D acceleration data from several joints, etc. From the clinical perspective high-precision motor performance assessment is extremely beneficial because it allows much more detailed presentation of patient's condition or condition changes compared today's e.g. Rankin or EDSS scores for stroke and Sclerosis Multiplex patients respectively. However, due the price constraints of such high-end devices [8], precise movement tracking technology is still used only in the experimental level [9].

2. Health Hub System for Motor Rehabilitation Monitoring

Our Health Hub (HH) software supports motion recognition and analysis with safety reasoning. The patient motion tracking solution is built into a Linux running HH described in [10]. Currently the HH device supports up to 4 MEMS body sensors. The low cost sensor device is based on Texas Instruments' Zigbee chip CC2530 which assembles 6DOF movement data from Invensense' IMU device. The sample rate of the sensors is set to 50Hz as a tradeoff between the signal quality and the amount of data to be transmitted. Signal preprocessing is not necessary because the signal is high quality and if there are missing samples then the previous sample is repeated. With current 1500 mAh batteries the continuous time of the system is ca 50 hours.

The algorithm for recognizing and classifying movements of therapeutic exercises is clearly presented in [11] and is based on ANARX neural networks. Modeling is performed in the following way. For each therapeutic exercise one of the sensors outputs is chosen to be modeled by the outputs of the other sensors observed over certain period of time. The neural network is trained to predict the value of one output on the bases of remaining outputs. Therefore, the modeled output plays the role of system output and remaining outputs are system inputs. Such model corresponds to the neural networks with restricted connectivity, which leads less computation power required. Corresponding neural network were trained with Levenberg-Marquardt algorithm for 1000 epochs. The actual HH software is realized as a set of independent agents responsible for all kind of different functional and non-functional aspects [12]. All agents complement the overall system behavior without the need to explicitly know about the presence of other agents as the whiteboard-based communication is used.

In the setup phase of the rehabilitation system the physiotherapist designs the individual treatment plan for every patient consisting of proper selection of exercises, constraints and additional requirements. The patient is instructed on the exercise program and the usage of the HH device and the MEMS sensor nodes. Sensors are attached onto the frame that guarantees a stable position during the exercises (Figure 1.). The patient must ensure that the frame is always installed in the same place – the height, direction, and etc. One way to ensure this is to use a skin patch to labeling, and place the frame onto the patch every time. If the sensors are mispositioned with small error it does not affect the system's ability to classify movements. During the initial exercising session the patient performs exercises while wearing the wireless MEMS sensors and the physiotherapists evaluates the movements being correct or incorrect. This provides the reference data for training the neural network for automatic exercise recognition and assessment.

The prescribed exercises with elaborating parameters (such as the number and starting times of the exercising sessions, order of exercises within a session, number of repetitions of certain exercises, etc.) form the rehabilitation plan in the context of the HH-based training system. This gives the basis for the system to monitor the patient's independent training at home, analyze the gathered data and give reminder alarms, recommendations and warnings. There can be other matters as well in addition to the exercising schedule which correlate with the vital signs' measurements (blood pressure, heart rate, SpO_2, body temperature, body weight). The execution of the exercising plan could be affected by the measurement results, for example if the blood pressure is too high it can delay the next exercising session or reduce the number of some movements.

In the rehabilitation phase in the patients' home environment, the configured HH system keeps track of the treatment plan fulfillment and tries to ensure the patient's safety which means the continuous checking of the rules established by the physician. To keep the patient within the safety zone determined by these rules a set of software agents is called into existence to keep track of whether the patient's actions comply with the targets of the rehabilitation plan and not overload her in any way.

All the patient's movements are collected while the sensors are turned on and worn which also means the system knows when the patient is refraining from therapy. Within one exercising session the system monitors if the patient is doing correct number of exercises in the required order. The similarity measure to the reference recording of the MEMS signals is calculated for individual movements and the feedback is given to the patient in real time about the correctness of exercising to avoid the situations of doing useless training or further damaging disabled parts of the body.

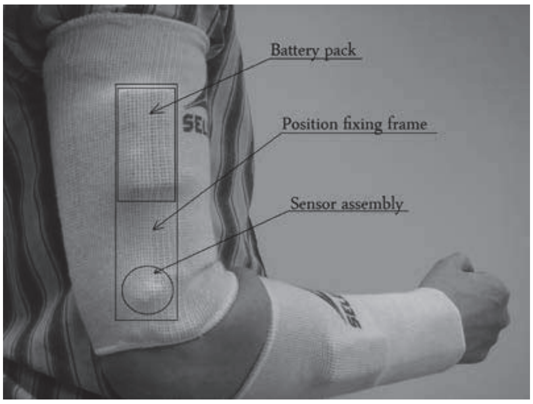


Figure 1. Wearing wireless MEMS sensors

During the home rehabilitation period all log and statistics that are gathered is recorded for physicians and can be uploaded to the hospital information system using previous telemedicine framework [13].

To illustrate the case study, we describe a use case: Rehabilitation process at home.

Actors: Patient, HH System. Pre-condition: Patient has a treatment plan (what includes already MEMS technology. In Figure 2. it is located in *motion* model under the function of *MotionRecognition(Treatment_plan)*). Patient status is “at home”. Sensor frames are placed in position. Post-condition: Patient exercises are recorded. The message is displayed on the screen in the following exercises start time.

Normal course of events: Patient has the time to start doing exercises. The system copies the treatment plan to session plan to keep the original. *TreatmentPlan* agent keeps the current session. The patient makes the first exercise. Sensors send the information and *Motion* agent analyses exercise recognition. *TreatmentPlan* agent manages the ongoing session. Screen displays of what to do next. If something is wrong then display warning on the screen immediately. *PersonSafety* agent creates local warning messages. *Environment* agent watches aliveness and resets certain components if necessary.

3. The Safety Model of the HH System

We have constructed the abstract models according to the behaviors of the independent agents, the HH-based system and the patient’s exercises. Our system is described by the environment model, patient’s exercising, and the safety model (Figure 2). The safety model secures that the exercising guidelines presented at any given time by the HH are meant personally to the particular patient and are satisfied by physician’s rules.

Model checking has been widely used in system design and verification. We use model checking to derive a reliable system. It is essential to verify the system design and also to model check its implementations. Model checking is a method for formally verifying finite-state concurrent systems such as communication protocols and real-time systems. It is a technique for formally and automatically checking whether a particular model or a set of models meets a given specification.

We give a simplified example of safety model and the related environment. In the current case Uppaal tool [14] is used for validation and verification of models. There are 4 models which belong to safety zone: patient exercising model, motion recognition model, treatment plan model and person safety model. In addition there is the environment model containing an *invariant x<=2* which refers to a system aliveness watchdog. The system can be in this state only 2 time units and then it has to leave the state using one of the three transitions. Models communicate by global variables and synchronization channels.

The model of *patient* represents behavior of the patient exercising at home (Figure 2, in top left). Activity is started by the *treatmentPlan* aspect by calling the *exercise!*. There is a time in the plan to start exercising and function *x_exe=x_tp* does a copy from the original treatment plan and all adjustments are done to the copy. The model of *treatmentPlan* represents the behavior of the agent that is responsible for steering the patient to follow the treatment plan. It presents the next exercise during the actual session lasts. Model of *environment* provides by concurrently call *safety!* against the model of *safety* that the safety requirements comply with the rules established by the physician. During the session after every exercise the call *initAccuracyCheck!* goes to analyze incoming movements in model of *motion*. There under the *MotionRecognition* location is the algorithm which solves the movements' comparison. The end of the exercising is informed by the *treatmentPlan.*

Model verification environment ensures the correct model by verification that the model is correct in the particular software environment. One of the model verification techniques – safety analysis checks whether anything bad can happen and another – liveness analysis checks if something good will happen. It searches for dead states from which goals cannot be reached. Model composition is performed automatically by the tool. All models work as concurrent processes. By safety model we show that the patent is in safety condition while using the HH system.

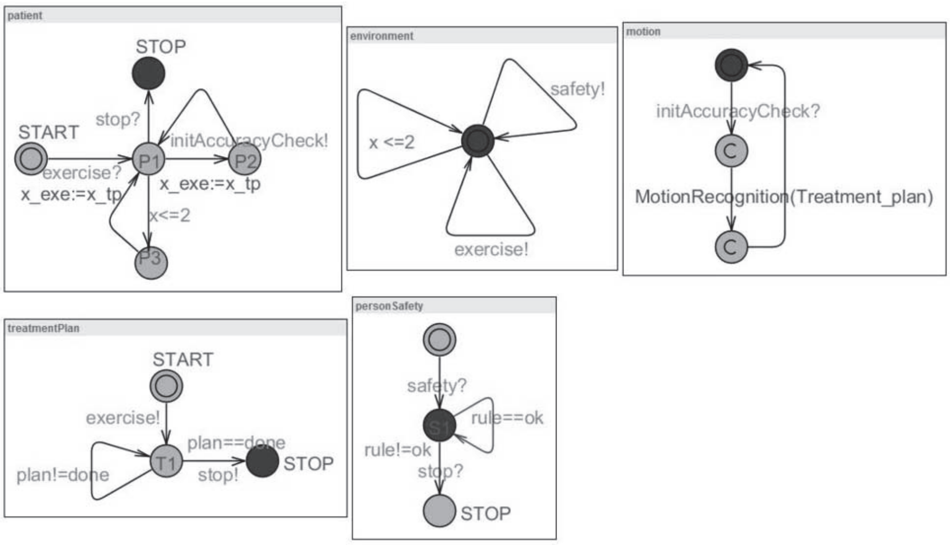


Figure 2. Model checking environment

4. Conclusions

For patients under post-surgery rehabilitation and patients suffering neurodegenerative diseases it is convenient to receive motor condition assistance and assessment outside of hospital environments in the form of a tele-rehabilitation equipment that can gather the information about the patient's movements, interpret it, and provide real-time guidelines. On the other hand, it saves labor cost of neurologist and physiotherapists.

Focus on tele-rehabilitation systems research is currently on a combination of virtual-reality and game based methodologies as indicated by the SWORD system [15] which enables three-dimensional space. Our idea involves model based techniques which can add as much dimensions as needed for improving the tele-rehabilitation system. Including a model checker we improve software quality and also ensure that agents we have deployed are verified. Model checking in the medical domain is obviously needed activity because of the safety critical nature of the systems. This formal approach adds important attributes like safety and security for these systems. By using models early during the developing process, quality is added as early as possible. As a result of this work, we are able to confirm that the system has verified and reliable.

The future work focuses on conducting the clinical trials to assess the effect of the training equipment on the effectiveness of the motor rehabilitation at home.

Acknowledgments

This research has been supported by European Union through European Regional Development Fund.

References

[1] Baum P, Abadie F. Market developments – Remote Patient Monitoring and Treatment. Telecare, Fitness/Wellness & mHealth, (SIMPHS 2), JRC IPTS Technical Report, European Communities, 2012.

[2] http://www.shimmer-research.com/r-d/hardware/extension-modules accessed in Sept 2012.

[3] Meijer G, Westerterp KR, Verhoeven FM, Koper HB, and ten Hoor F. Methods to assess physical activity with special reference to motion sensors and accelerometers. IEEE Transactions on Biomedical Engineering 1991; 38(3):221–229.

[4] Foerster F. Detection of posture and motion by accelerometry: a validation study in ambulatory monitoring. Computers in Human Behavior 1999; 15(5):571–583.

[5] Ravi N, Dandekar N, Mysore P, Littman ML. Activity Recognition from Accelerometer Data. In: Proc of the 17th Conference on IAAI, AAAI Press, 2005, 1541-1546.

[6] Lorincz K, Chen B, Welsh M, et al. Mercury: a wearable sensor network platform for high-fidelity motion analysis. Proc of the 7th ACM Conference on Embedded Networked Sensor Systems. SenSys2009, Berkley (2009), 183-196. DOI= http://dx.doi.org/10.1145/1644038.1644057

[7] Hirata Y, Komatsuda S, and Kosuge K. Fall prevention control of passive intelligent walker based on human model. In: Proc of the Int. Conf. on Intelligent Robots and Systems, 2008, 1222–1228.

[8] www.xsens.com, accessed in Sept 2012.

[9] Cloete T, Scheffer C. Repeatability of an off-the-shelf, full body inertial motion capture system during clinical gait analysis. Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE. Nov 2010, 5125 – 5128, DOI= 10.1109/IEMBS.2010.5626196.

[10] Kuusik A, Reilent E, Lõõbas I, Parve M. Software Architecture for Modern Telehealth Care Systems. Journal of Advances on Information Sciences and Service Sciences 2011; 3(2):141-151.

[11] Kuusik A, Nõmm S, Osjanski S, Orunurm L, and Reilent E. Wearable system for patient motor condition assessment and training monitoring. IEEE PHT: Bangalore, India, 16-18 Jan 2013.

[12] Kuusik A, Reilent E, Sarna K, Parve M. Home telecare and rehabilitation system with aspect oriented functional integration. Biomed Tech (Berl) 2012; DOI: 10.1515/bmt-2012-4194.

[13] A. Kuusik, E. Reilent, I. Lõõbas, M. Parve: Semantic Formal Reasoning Solution For Personalized Home Telecare, Proc. of 2010 Int. Conf. on Mechanical and Electrical Technology, pp. 72 – 76
[14] http://www.uppaal.com
[15] Virgilio F. Bento, Vitor T. Cruz, David D. Ribeiro, Marcio M. Colunas and Joao P. S. Cunha. The SWORD Tele-Rehabilitation System. pHealth 2012. Doi: 10.3233/978-1-61499-069-7-76.

Appendix D

PAPER D:

Külli Sarna, Jüri Vain. **Aspect-oriented testing of a rehabilitation system.** In: *VALID 2014: The Sixth International Conference on Advances in System Testing and Validation Lifecycle, October 12 - 16, 2014, Nice, France: (Edit.) Kanstrén, Teemu; Helle, Philipp.* Venice: IARIA, 73 - 78.

Aspect-Oriented Testing of a Rehabilitation System

Külli Sarna and Jüri Vain
Department of Computer Science
Tallinn University of Technology
Tallinn, Estonia
Kylli.Sarna@eliko.ee; Juri.Vain@ttu.ee

***Abstract*— The paper focuses on modularizing test models by adapting aspect-oriented modelling techniques. Model-based testing is an unavoidable part of contemporary model-driven software processes. The essence of model-based testing is to provide methods and tools to validate software systems by generating test cases systematically from models. From the practical usage point of view, it is critical to construct models that capture the essential aspects of the system under test. The proposed test design approach allows systematic separation of testing concerns, that, in turn, helps to overcome the complexity issues. Also, verification conditions are proposed to ensure the correctness of derived aspect test models and their compatibility with base test models. We demonstrate the technique of test model construction using timed automata models and illustrate it with a home rehabilitation system case study.**

Keywords-aspect-oriented testing; model-based testing; test model design; test generation.

I. INTRODUCTION

In the current practice of software testing, including Model-Based Testing (MBT), the test cases are frequently insufficiently structured and specified. The test designers use component-based or hierarchical state models. However, these modelling approaches provide poor support for isolating crosscutting features, specifically, functions that are spread across the software modules and tangled with other functions. We use the principles of Aspect-Oriented Modelling (AOM) to modularize such crosscutting functions into aspects. The AOM approach has evolved from aspect-oriented programming [2] to produce well-structured and well-encapsulated software. We enhance MBT design methodology with aspect handling capabilities taken from AOM [3]. Using the principles of AOM we can encapsulate typical cases like specifying requirements (use cases) that do not specify one property (scattering) or different functionalities (tangling). In this paper, we will explain how to conceptualize concerns into aspects and how to extract test cases from these aspect test models.

In MBT, the tests are generated from formal models of the System Under Test (SUT). The AOM technique introduced by Sarna and Vain [9] models SUT using timed automata and defines aspect models as refinements of the base model. The structural test coverage criteria considered are the same as those commonly used in state models, i.e., state, and transition coverage. As a novelty, in this paper we demonstrate how a test suite can be generated according to structural units that are specific to AOM. This gives us new test coverage criteria that address implemented features – aspect, advice, join-points coverage, etc. - and provide more intuitive reference to the parts of SUT to be tested for those features.

Another advantage of Aspect-Oriented (AO) MBT is the possibility of easy modification of the test suite. When new requirements arise, new advice models can be woven into the test suite without redesigning the existing base model.

Applying the principles of AOM does not provide compositional testing techniques *per se*. Compositionality of proposed AO testing is achieved by imposing extra constraints on how the advice models are constructed and model weaving operations defined. We define these rules in the semantic framework of Uppaal timed automata [6] and formulate the proof obligations to be model-checked. Our approach is illustrated with a home rehabilitation system testing framework.

The rest of the paper is structured as follows. We introduce the technical background in Section 2. Section 3 describes AO MBT. In Section 4, the home rehabilitation system is introduced. Finally, Section 5 concludes the paper.

II. BACKGROUND

A. Aspect-oriented modelling

AOM is a way of modularizing *crosscutting concerns* much like object−oriented programming is a way of modularizing *common concerns*. *Crosscutting concerns* generally refer to non-functional properties of software, such as security, synchronization, mobility, resilience, etc. In addition, every system may contain its own application specific crosscutting concerns [5].

Cottenier et al. [4] and Rashid [8] have admitted that AOM technologies have the potential to simplify software deployment, and the ability to improve the categorization of crosscutting concerns. Also, AOM aids in modular extension of object systems, where the treatment of crosscutting concerns is encapsulated in separate modules called *aspects*. We use concepts taken from AOM, such as *Aspect, Advice*, *Join-points*, *Pointcut*, and *Weaving.*

An aspect consists of two parts: the code/model associated with treatment of the concern (called *advice*), and a predicate defining when the advice should be applied during system executions (called a *pointcut*). The points in the code/model that are identified by a pointcut are called *join-points*.

A *pointcut* selects a subset of join-points based on defined criteria. The criteria can be explicit function names, or function names specified by wildcards. *Pointcuts* can be composed using logical operators. Customized *pointcuts* can be defined, and *pointcuts* can identify *join-points* from different aspects. The process of adding aspects to a base system is called *weaving*; and the result is referred to as the *woven system* [5]. AOM techniques use the term *advice* for the action an aspect will take and *join-points* for where these actions will be inserted in the base system model. *Pointcuts* are used to specify the rules of where to apply an aspect. *Advice*, *join-points*, and *pointcut* are specified as one entity, called an *aspect* [7].

As in AOM, AO testing uses a *base test model* and several *aspect test models.* An example of a base test model is depicted in Figure 1 (for better understanding of the relationship between the models, we use an Automatic Teller Machine (ATM) as an example of a well-known system). The ATM test model specifies the use case of withdrawing money from an ATM. Crosscutting features are treated as patterns described by aspect advice models, and common features are described in the base model. The result of weaving the base model with advice models is called the *composed aspect model*. An advice model can be woven with the base model in many places and in different ways. The Transaction advice model is defined as location refinement of both ATM and Customer automata. The details of advice model construction in the test design level are presented in [9].

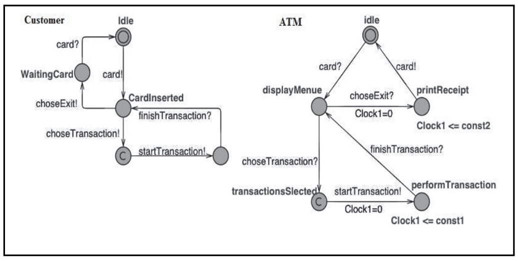


Figure 1. The base test model of ATM.

The base model of an ATM depicted in Figure 1 includes interacting Customer and ATM automata. Refinements in Figure 2 specify aspects of interest: (i) the Transaction advice model is defined as location refinement of both ATM and Customer automata; (ii) edge refinement of ATM. The aspect behaviour is launched from the base model explicitly with the help of channels. We model in Uppaal (www.uppaal.com), a tool box for modelling, simulation and verification of timed automata. In Uppaal [12], the synchronization mechanism is a hand-shaking synchronization: two processes go through a transition at the same time, one will be labelled x !, and the other x ?, where suffixes ?, and ! after the channel name x distinguish sending and receiving synchronization information respectively. A system is composed of concurrent processes, each of them modelled as an automaton. The automaton has a set of locations and edges to specify the control flow. A transition specified by an edge is enabled if its guard and synchronization conditions are satisfied. The transaction automaton in Figure 2 introduces the EnquireBalance aspect

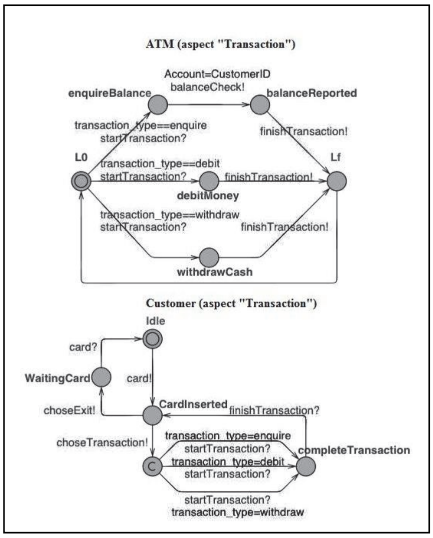


Figure 2. The aspect model “Transaction”.

advice. Since the refinement (ii) introduces a new interaction between ATM and a new actor Server (not shown in the model) the edge introduced is labelled with the ‘balanceCheck!’ channel. When the aspect related tests have to be generated from the composed model of SUT that includes the automata in Figures 1 and 2, we can ignore all the transactions that the aspects of interest do not depend on. For instance, when testing the balanceCheck! transaction between ATM and Server the tester model is extracted from the composition Customer || Customer(Transaction) by algorithm of [1] so that the test sequence <card!, choseTransaction!, transaction_type := enquire, start-Transaction!, wait,[finishTransaction?| timeout >= const1, TESTFAIL], choseExit!, card?, TESTPASS > can be executed.

B. *Model-based testing*

MBT uses abstract behavioural models for specifying the expected behaviour of the SUT and for automatically generating tests to check if the behaviour of SUT conforms to the model. The SUT is an executable implementation which is considered as a black-box during the testing process, i.e., only inputs and outputs of the system are visible externally. The SUT is tested incrementally by applying test cases. A test case in MBT is defined as a sequence of test stimuli paired with expected SUT outputs. A specified set of test cases constitutes a test suite.

C. *Uppaal timed automata*

Assume a finite alphabet *Σ* ranged over by *a*, *b*,... stands for actions and a finite set *C* of real-valued variables ranging over by *x*, *y*, *z*, standing for clocks.

A guard is a conjunctive formula of atomic constraints of the form $x \sim n$ for $c \in C$, $\sim \in \{\geq, \leq, =, >, <\}$ and $n \in$ N. We use $G(C)$ to denote the set of guards, ranged over by g.

Definition 1 (**Timed Automaton**) [6]
*A timed automaton **A** is a tuple* $\langle N, l_0, E, I\rangle$ where
– N is a finite set of locations (or nodes),
– $l_0 \in N$ is the initial location,
– $E \in N \times G(C) \times \Sigma \times 2^C \times N$ is the set of edges and
– I: $N \rightarrow G(C)$ assigns invariants to locations (here we restrict to constraints in the form: $x \leq n$ or $x < n$, $n \in$N. For shorthand we write $l \rightarrow_{g,a,r} l'$ to denote $\langle l, g, a, r, l'\rangle \in E$.
To model concurrent systems, timed automata are extended with parallel composition. In the UPPAAL modelling language, the CCS parallel composition operator is used, which allows interleaving of actions as well as hand-shake synchronization. The parallel composition of a set of automata is the product of the automata.
The semantics of timed automata is defined as a transition system where configuration consists of the current location, valuation of state variables and the current values of clocks. There are two types of transitions between states: the automata may either delay for some time (delay transition), or follow an enabled edge (action transition).

To keep track of the changes of clock values, we use functions known as clock assignments mapping C to the non-negative reals R$_+$. Let u, v denote such functions, and $u \in g$ means that clock values denoted by u satisfy the guard g. For $d \in$ R$_+$ let $u + d$ denote the clock assignment that maps all $x \in C$ to $u(x) + d$ and for $r \subseteq C$ let $[r \mapsto 0]$ denote the clock assignment mapping all clocks to 0 and agree with for the other clocks in $C\backslash r$.

Definition 2 (**Operational Semantics**) [6]
The semantics of a timed automaton is a transition system (also known as a timed transition system) where states are pairs $\langle l, u\rangle$ and transitions are defined by the rules:
– $\langle l, u\rangle \rightarrow_d \langle l, u + d\rangle$ if $u\in I(l)$ and $(u + d) \in I(l)$ for a non-negative real $d \in$ R$_+$
- $\langle l, u\rangle \rightarrow_a \langle l', u'\rangle$ if $l \rightarrow_{g,a,r} l'$, $u \in g$, $u' = [r \mapsto 0]u$ and $u' \in I(l')$.
To increase the modeling power keeping the analysis traceable for planner synthesis we lift the model class to rectangular timed automata where guard conditions are in conjunctive form with conjuncts including besides clock constraints also constraints of integer variables.

Similarly to clock conditions, the integer variable conditions are of the form $k \sim n$ for $k \in$ Z, $\sim \in \{\geq, \leq, =, >, <\}$ and $n \in$ N. The advantage of this extension is that the model has rich enough modelling power to represent real-time and resource constraints being same time efficiently decidable for reachability analysis.

III. Aspect-Oriented Model-Based Testing

In this section, we explain the concepts of AOM applicable in aspect-oriented MBT. The AOM allows the models to be organized so that they address particular requirements (including crosscutting ones) and corresponding test cases. The AO test model includes a base model and aspect-related advice models. Aspects may contain sub-aspects that require sub-advices and their own test cases. Sub-aspect models have to be easily inserted into their parent aspect models. In our examples, we use name prefixes that refer to the parent models so that they are convenient to comprehend and maintain.

AO testing can also be considered as an example of compositional testing where the test results of the composed system can be inferred from the test results of its components. In the MBT context, it means that the test cases are determined only by the context of the aspect advice models and the interface behaviour of their composition. AOM also provides a conceptual basis for defining test coverage criteria in terms of aspect related model elements. The hierarchy of those criteria is depicted in Table I.

TABLE I. AO TEST COVERAGE CRITERIA

Coverage constraint / Type of coverage entity	Strong (universal) coverage $\forall$	Weak (existential) coverage $\exists$	Discriminating predicate
Aspect A	All aspects of the model $\forall A \in \mathbf{A}$. ...	Some aspects of the model $\exists A \in \mathbf{A}$. ...	Predicate on aspect constants /variables
i-th join point $jp(A, i)$	All join points of aspect A $\forall\, jp(A, i) \in JP(A)$	Some join points of aspect A $\exists jp(A, i) \in JP(A)$	Point cut condition
Entry-exit path λ of an advice model $M^{A'}$ $\lambda \in Paths(M^{A'})$	All paths initiated at i-th join point $\forall\, \lambda \in Paths(M^{A'})$	Some paths initiated at i-th join point $\exists \lambda \in Paths(M^{A'})$	Path predicate, e.g. constraint on path length
Model element of type $\boldsymbol{T}$ (location, transition, function, data, etc) included in the path $\lambda \in Paths(M^{A'})$	All elements of type $\boldsymbol{T}$ in $M^{A'}$	Some elements of type $\boldsymbol{T}$ in $M^{A'}$	Predicate on the attributes of type $\boldsymbol{T}$

The criteria shown in Table I can be expressed as closed 1st order logic formula in prenex normal form, where the signature includes variables of particular types of structural elements of Uppaal Timed Automata (UPTA) (template, location, transition, label, function, data, etc.). The prefix of the prenex formula includes bound variables in a fixed order that is determined by the natural hierarchy of modelling entities: aspect, join-points, and path. These entities model the structural elements of UPTA, where the structural elements can be referred to directly by name or indirectly by constraints on their attributes. The matrix part may include discriminating predicates of all the above listed types.

The semantics and scoping of AO coverage constraints is defined by the hierarchy and type structure of AO model elements (left most column in Table I). Thus, the scope of constraints on bound variable in the formula matrix part is defined by the position of the bound variable in prefix. For instance, the scope of a path constraint is defined by the join-point and aspect constraints because these elements precede path variable in the prefix. When not explicitly expressed in coverage constraint the default scoping means existential quantification over all those variables preceding in the prefix of coverage constraint. For characterization of coverage criteria in terms of Uppaal query language, we assume that the aspect model *M* is constructed according to the rules described in [9]. The idea is to use Uppaal model checker queries for selecting traces that constitute the test paths of the given test case. Uppaal query based online test generation methods are described by Vain et al. [1] and Hessel et al. [10].

Aspect Coverage criteria impose to execute all or some aspects in a woven model at least once. In *Strong Aspect Coverage* (SAC), given an aspect model *M*, all possible test paths must be covered by the tests. To implement the Strong Aspect Coverage we use the parameterized UPTA templates where the template parameter p_i ranges over indexes [1, *n*] that identify the aspect. Let `P(i)` be a predicate updated to true whenever the i-th aspect advice model is entered. Then the traces of *M* (p_i) under Strong Aspect Coverage criteria should satisfy the query: `E<> forall (i: int [1,n]) P(i)`. Note that given query is valid only for paths that include traversal of all aspects' advice models. In general, the model *M* may not be fully connected and a single path including all aspects may not exist. Therefore, we introduce an auxiliary *reset-* transition into *M* that guarantees that if *n* advice models are reachable in *M* then at most with *n* traversals all of them are visited. The *reset*-transition connects the final location of *M* with its initial location. Due to this construct the Uppaal model checker is able to generate a trace that includes visits of all advice models. The tests paths for a final test case can achieved simply by "cutting" that trace at *reset-* transitions to many shorter sub traces.
Weak Aspect Coverage (WAC) refers to the case where at least one advice model of some aspect is traversed by the test path. The query `E<> forall (i:int [1,n]) P(i)` differs little from the strong coverage constraint but it does not require including *reset*-transitions in the model *M*.

Join Point Coverage criteria impose to execute all or some join points of each aspect in a woven model at least once. *Strong Join Point Coverage* (SJPC) presumes similarly to strong aspect coverage introduction of an auxiliary *reset*-transition into *M*. Regardless the prefix (SAC or WAC) of the query the SJPC contributes a conjunct of form ...`forall (j: int [1,m]) P(i) && R(j)` where `j` is ranging over join point indexes of the aspects referred in the prefix of that query and `R(j)` is a Boolean variable at each join point updated to `true,` whenever this join point is visited. *Weak Join Point Coverage* (WJPC) is satisfied if there is at least one trace for given formula prefix satisfying ...`exists (j: int [1,m]) P(i) && R(j)`. Here, like in WAC, auxiliary *reset*-transition is not needed.

Aspect Path Coverage criteria impose to execute all or some paths of each aspect in a woven model at least once. Assume the entry and exit transitions of each advice models are decorated with *entry*(*i, j,k*) and *exit*(*i, j,l*) predicates where *i, j, k, l* range over the set of aspects, join points, and their advice entry and exit points respectively. Whenever the transition is executed these predicates evaluate to `true`. Then, the *Strong Aspect Path Coverage* (SAPC) contributes a conjunct to the query prefixed with aspect and join point constraints as follows: ... `forall (k: int [1,K]) forall (l: int [1,L]) P(i) && R(j) && [(`$\vee_{k=1,K}$` entry(k)) `$\wedge$`(`$\vee_{l=1,L}$` exit(l))`. SAPC, like earlier strong coverage criteria, presumes the *reset*-transitions related construct. *Weak Aspect Path Coverage* (WAPC) comparing to SAPC replaces universal quantifiers with existential ones for variables k and l, the coverage constraint becoming to ... `exists(k: int[1,K]) exists(l: int[1,L]) P(i) && R(j) && [(`$\vee_{k=1,K}$` entry(k)) `$\wedge$` (`$\vee_{l=1,L}$` exit(l))`.

The **Model Element Coverage** criteria impose constraints on the types of UPTA elements to be covered in the advice model or set the specific constraints on the attributes of those elements, e.g. *Strong* (resp. *Weak*) *Model Element Coverage* can be parameterized with the element type, e.g. `Transition` and universally (resp. existentially) quantified over given type. More specific coverage constraints can be constructed using type discriminating predicates on, e.g., local data variables of an advice model.

IV. EXAMPLE: TESTING HOME REHABILITATION SYSTEM

The AO MBT approach described in Section 3 has been applied in testing a Home Rehabilitation System (HRS). The model-based testing is needed in the medical domain because of the safety critical nature of the systems and non-trivial combination of functional, performance and security features [11]. The HRS is an application which drives sensor devices, analyses the gathered data, interacts with the patient and submits relevant information to the hospital through the Internet. HRS software contains the following subcomponents: dedicated health hub as communication gateway; vital signals' sensor system for patient measurements; movement tracking sensor system for fall detection, physical activity and exercise monitoring.

There are three actors, namely, Patient, Plan and Sample, interacting in the "home exercising" use case. The composition of automata *Plan* and *Sample* constitute the base model that can be woven with different advice models depending on what body characteristic (pulse, blood pressure, etc.) is monitored. For instance in Figure 3, the use-case *exercising* is refined with two advice models that are instances of the same automaton template. The advice models linked to the base model are location refinements of the unnamed location in the automaton *Sample.* Channel *Sample* ensures that the advice models are executed synchronously with the edge departing from location *Measure* in the automaton *Sample*. A weak join point coverage of completing exercising can be specified now using query `E<>exists(Screen=UB_warning[1])`. The test case ensures that while a patient is exercising, a warning will be shown on a screen when the patient's pulse is greater than the number in U_bound. On the other hand U_bound is the upper value of pulse that the patient may have during exercising and this is specific to each patient. For example if the U_bound is 140 then a warning on a screen goes red and warns "wait until your pulse will be normal". We measure the pulse under "measurement [1]" and an upper bound and a lower bound are indicated. A normal pulse measurement have to be between U_bound and L_bound.

A strong join point coverage of completing exercising can be specified using query `E<>forall (Screen=normal[1])measurement[1]>=L_bound [1]&&measurement[1]<=U_bound[1]`. That means the screen indicates in green that everything is alright and the patient can continue exercising because their pulse is within the allowed range. By this strong join point test coverage, we ensure that our system is able to give the right warnings whenever necessary.

V. Conclusion

In this work, we have introduced an aspect-oriented approach to model-based testing in the context of Uppaal timed automata specifications. We advocate the view that aspect-oriented models help in constructing models of system under test in a systematic and user friendly way, thus helping to defeat the perennial problems of MBT - complexity of construction and maintenance of test models. It has been shown how the aspect related test coverage criteria can be formalized in a systematic way in Uppaal query language Timed Computation Tree Logic (TCTL) and the feasibility of test suites verified on aspect models before real tests are deployed and executed.

Our focus on how a test case can be generated according to structural units that are specific to AOM is novel. This gives new test coverage criteria that address implemented features – aspect, advice, join-points, etc., and provide more intuitive reference to the parts of SUT to be tested for those features.

Another contribution for enhancing MBT by aspects is the possibility of easy update of test case related models. If new requirements arise, new advice models can simply be incorporated by well-defined composition rules. This is especially relevant in regression testing.

Acknowledgment

This research was supported by the European Social Fund's Doctoral Studies and Internationalisation Programme (DoRa), and by the Competence Centre Programme of Enterprise Estonia.

References

[1] J. Vain, M. Kääramees, and M. Markvardt, "Online testing of nondeterministic systems with reactive planning tester," in: L. Petre, K. Sere, and E. Troubtsyna (Eds.). Dependability and Computer Engineering: Concepts for Software-Intensive Systems. Hershey, PA: IGI Global (2012), pp. 113-150.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. - M. Loingtier, and J. Irwin, "Aspect-Oriented Programming", ECOOP'97, June, 1997, pp. 220-242.

[3] J. Kienzle, A. Wisam, F. Fleury, J. Jezequel, and J. Klein, "Aspect-Oriented Design with Reusable Aspect Models." Transactions on Aspect-Oriented Software Development, vol7, 2010, pp. 279-327.

[4] T. Cottenier, A. van den Berg, and T. Elrad, "Stateful Aspects: The Case for Aspect-Oriented Modeling." Proceedings of the 10th AOM Workshop, 2007, pp. 7-14.

[5] E. Katz, and S. Katz, "User queries for specification refinement treating shared aspect join points." Proceedings of the 8th IEEE, 2010, pp. 73-82.

[6] J. Bengtsson, and W. Yi, "Timed automata: Semantics, algorithms and tools." Lecture Notes on Concurrency and Petri Nets, Lecture Notes in Computer Science vol. 3098, 2004, pp. 87-124.

[7] S. Clarke, and E. Baniassad, Aspect-Oriented Analysis and Design: The Theme Approach. Addison-Wesley Professional. 2005.

[8] A. Rashid, "Aspect-Oriented Requirements Engineering: An Introduction". In Proceedings of 16th IEEE, 2008, pp. 173-182.

[9] K. Sarna, and J. Vain, "Exploiting Aspects in Model-Based Testing," in Proceedings of 11th FOAL. ACM, New York, NY, USA, 2012, pp. 45-48.

[10] A. Hessel, K. G. Larsen, P. Pettersson, and A. Skou," Testing Real-Time Systems Using UPPAAL." Lecture Notes in Computer Science vol. 4949. 2008, pp. 77-117.

[11] A. Kuusik, E. Reilent, K. Sarna, and M. Parve, "Home telecare and rehabilitation system with aspect-oriented functional integration." Biomedical Engineering, DOI: 10.1515/bmt-2012-4194 (accessed 01.08.2014).

[12] K. Larsen, P. Pettersson, and W.Yi. UPPAAL in a nutshell. Journal on Software Tools for Technology Transfer, 1997, pp. 134-152.

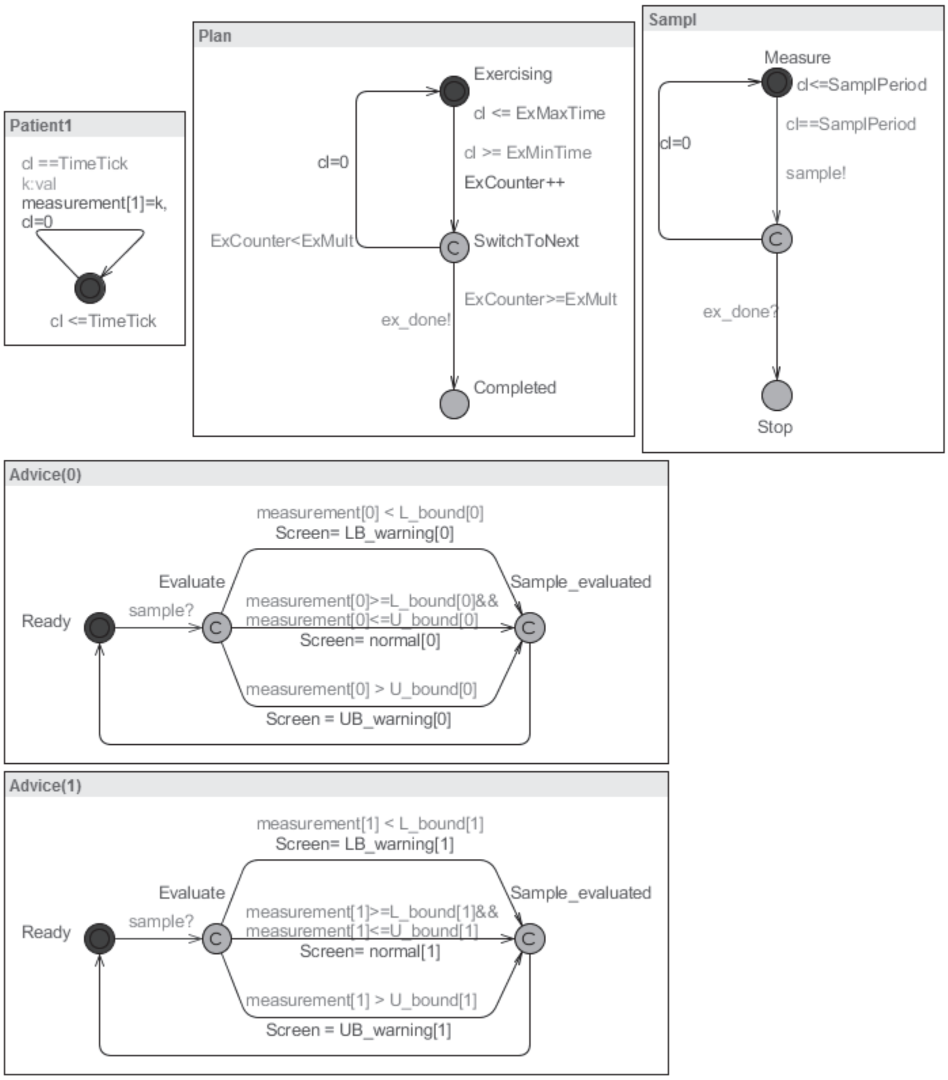


Figure 3. Composing the primary test models and advice model in parallel.

Appendix E

```
//AGENTS
        system_data_controller,
        system_pressure_checker,
        system_heartbeat_checker,
        system_telefon_agent,
        system_database_cleaner,
//USER
        system_get_pressure,
        system_get_heartbeat,
//WB
        system_wb_insert_triples1,
        system_wb_insert_triples2,
        system_wb_insert_triples3,
        system_wb_get_buffered_triples1,
        system_wb_get_buffered_triples4,
        system_wb_get_selected_triples1,
        system_wb_delete_selected_triples0,
        system_wb_sequence_for_ids1,
        system_wb_sequence_for_ids2,
        system_wb_sequence_for_ids3,
        system_wb_sequence_for_keys1,
        system_wb_sequence_for_keys2,
        system_wb_sequence_for_keys3,
//DB
        system_wg_create_record0,
        system_wg_delete_record0,
        system_wg_get_first_record0,
        system_wg_get_next_record0,
system_wg_get_field0,
system_wg_set_field0,
system_wg_make_query0,
system_wg_fetch0,
system_wg_free_query0,
system_wg_start_read0,
system_wg_end_read0,
system_wg_start_write0,
system_wg_end_write0,
system_wg_create_record1,
system_wg_delete_record1,
system_wg_get_first_record1,
system_wg_get_next_record1,
system_wg_get_field1,
system_wg_set_field1,
system_wg_make_query1,
system_wg_fetch1,
system_wg_free_query1,
system_wg_start_read1,
system_wg_end_read1,
system_wg_start_write1,
system_wg_end_write1,
system_wg_create_record2,
system_wg_delete_record2,
system_wg_get_first_record2,
system_wg_get_next_record2,
system_wg_get_field2,
system_wg_set_field2,
system_wg_make_query2,
system_wg_fetch2,
system_wg_free_query2,
```

system_wg_start_read2,

system_wg_end_read2,

system_wg_start_write2,

system_wg_end_write2,

system_wg_create_record3,

system_wg_delete_record3,

system_wg_get_first_record3,

system_wg_get_next_record3,

system_wg_get_field3,

system_wg_set_field3,

system_wg_make_query3,

system_wg_fetch3,

system_wg_free_query3,

system_wg_start_read3,

system_wg_end_read3,

system_wg_start_write3,

system_wg_end_write3,

system_wg_create_record4,

system_wg_delete_record4,

system_wg_get_first_record4,

system_wg_get_next_record4,

system_wg_get_field4,

system_wg_set_field4,

system_wg_make_query4,

system_wg_fetch4,

system_wg_free_query4,

system_wg_start_read4,

system_wg_end_read4,

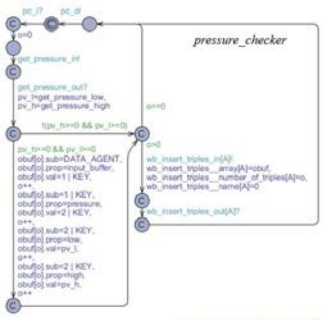
system_wg_start_write4,

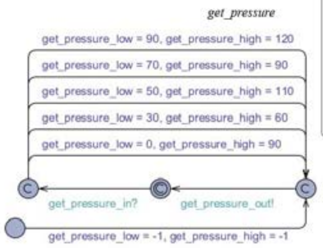
system_wg_end_write4,

system_wg_create_record5,

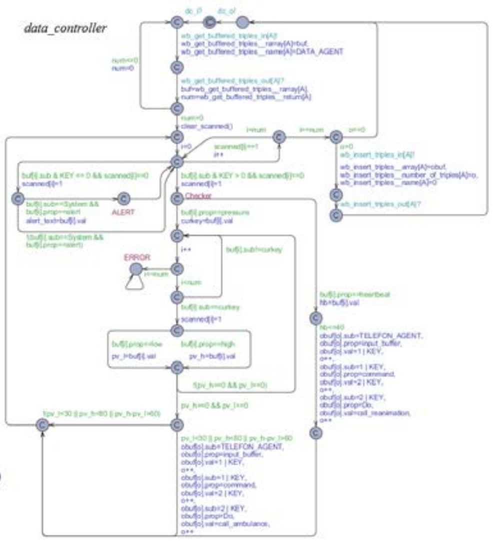
```
system_wg_delete_record5,
system_wg_get_first_record5,
system_wg_get_next_record5,
system_wg_get_field5,
system_wg_set_field5,
system_wg_make_query5,
system_wg_fetch5,
system_wg_free_query5,
system_wg_start_read5,
system_wg_end_read5,
system_wg_start_write5,
system_wg_end_write5,
```

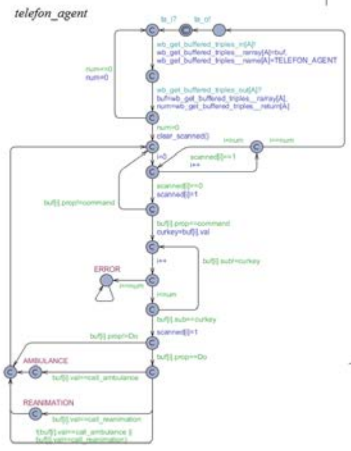
Agents' templates

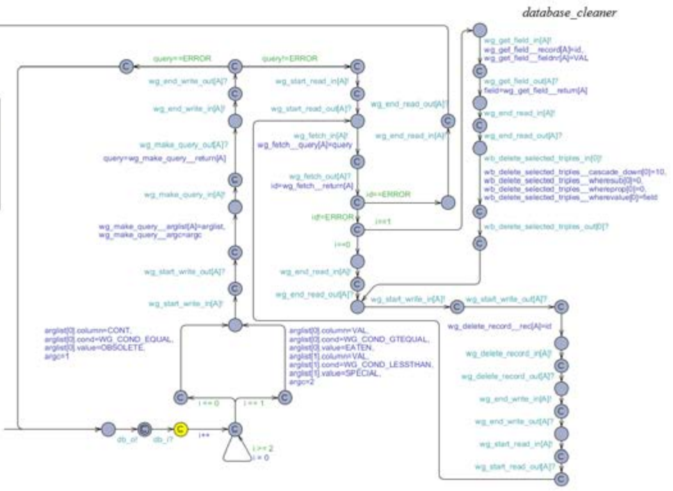


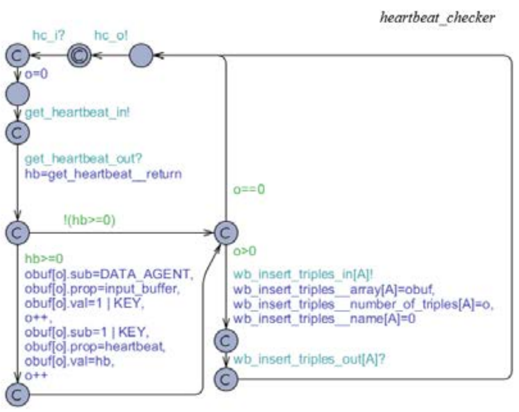


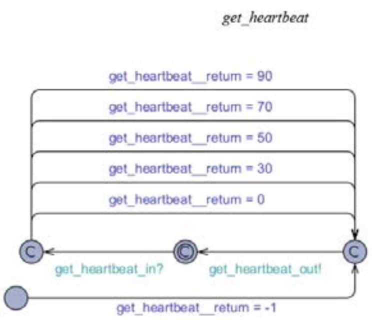
get_pressure
get_pressure_low = 90, get_pressure_high = 120
get_pressure_low = 70, get_pressure_high = 90
get_pressure_low = 50, get_pressure_high = 110
get_pressure_low = 30, get_pressure_high = 60
get_pressure_low = 0, get_pressure_high = 90
get_pressure_in?
get_pressure_out!
get_pressure_low = -1, get_pressure_high = -1



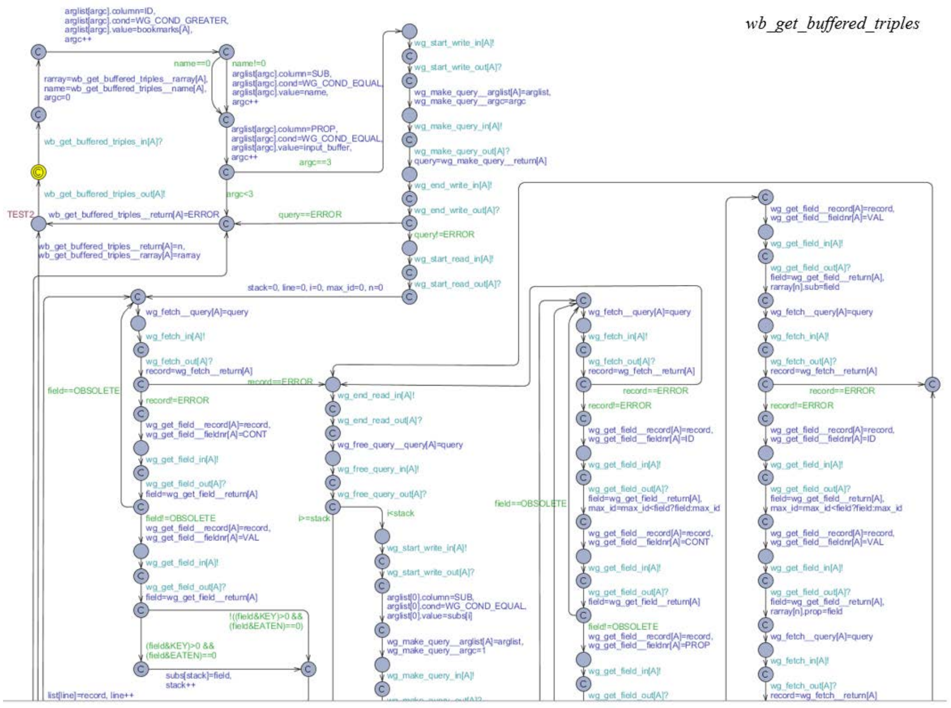


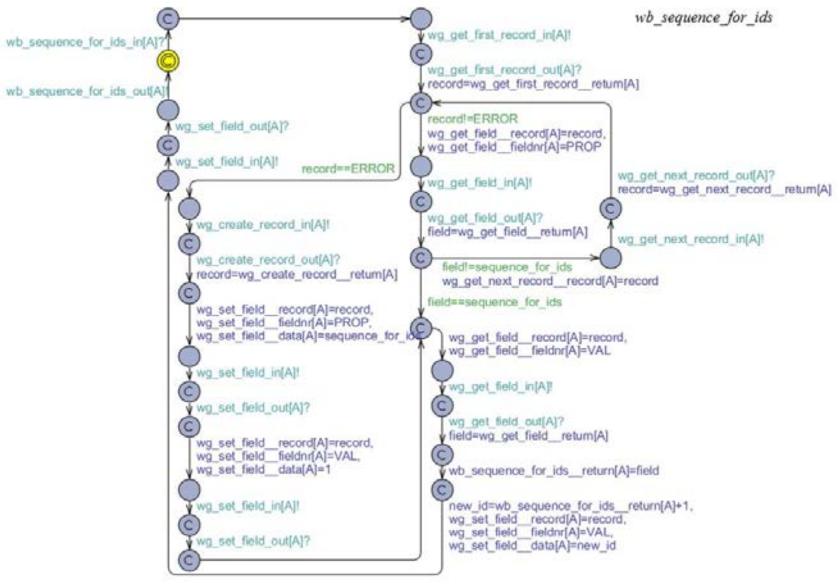


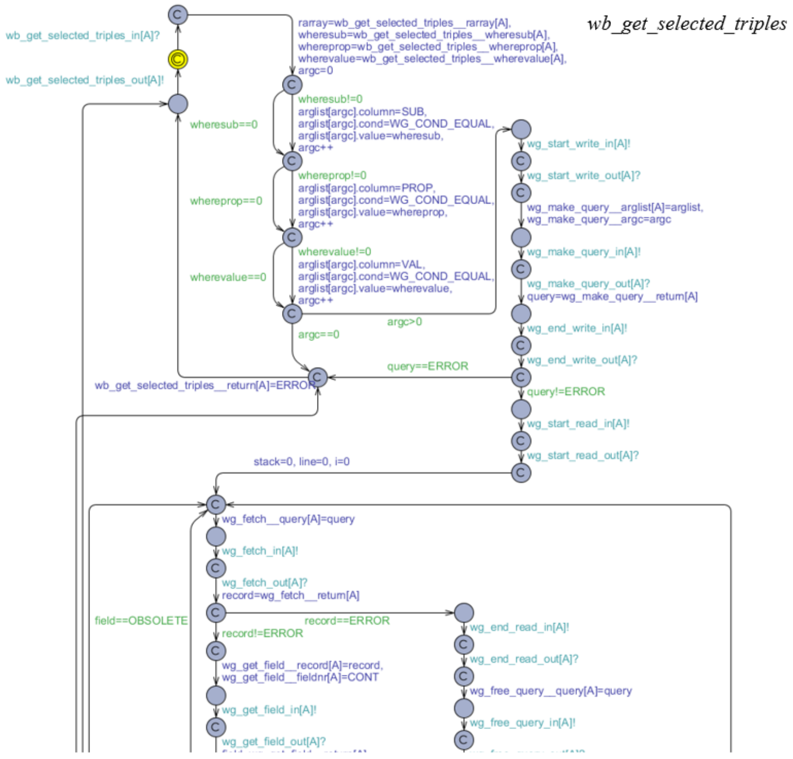


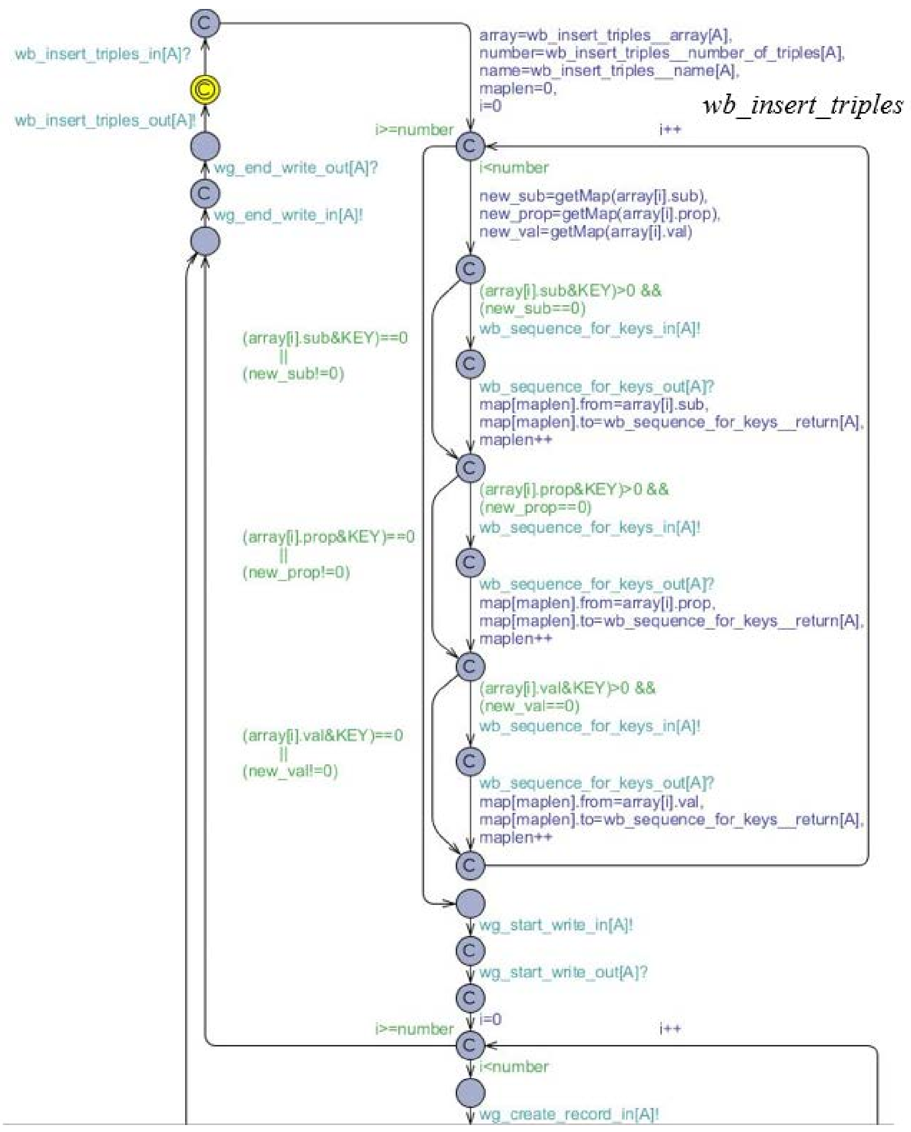

get_heartbeat
get_heartbeat__return = 90
get_heartbeat__return = 70
get_heartbeat__return = 50
get_heartbeat__return = 30
get_heartbeat__return = 0
get_heartbeat_in?
get_heartbeat_out!
get_heartbeat__return = -1

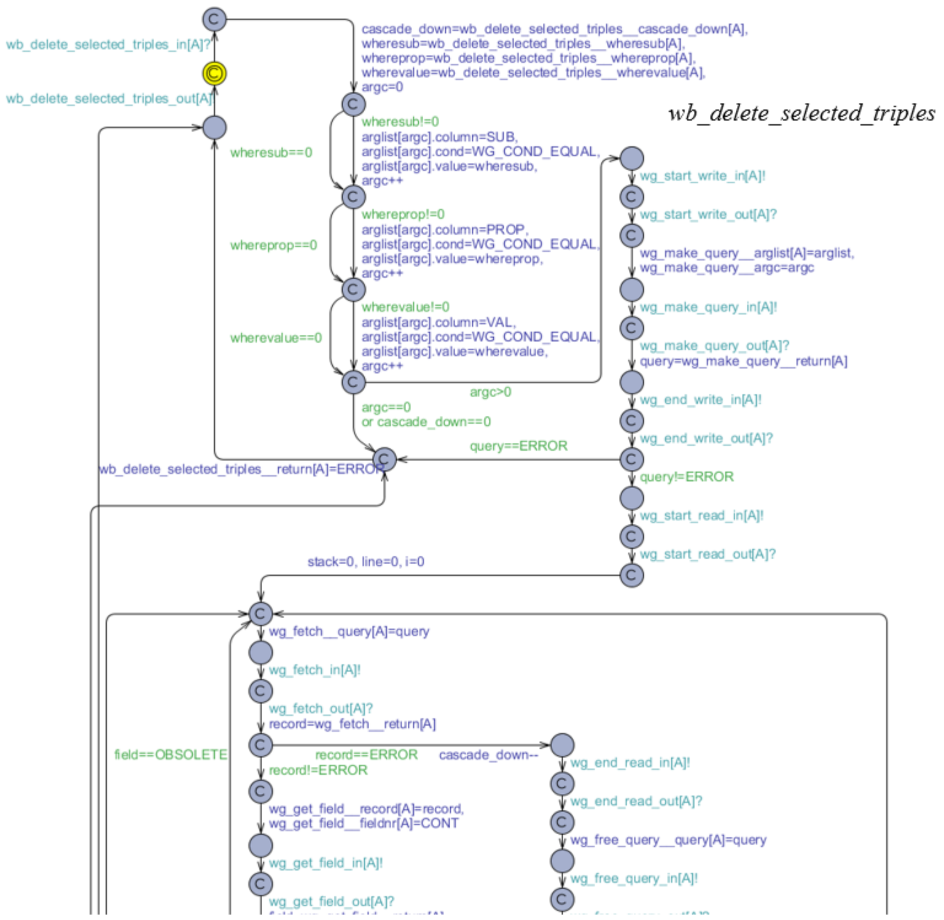
Whiteboard templates


wb_get_buffered_triples
arglist[argc].column=ID,
arglist[argc].cond=WG_COND_GREATER,
arglist[argc].value=bookmarks[A],
argc++
rarray=wb_get_buffered_triples__rarray[A],
name=wb_get_buffered_triples__name[A],
argc=0
wb_get_buffered_triples_in[A]?
wb_get_buffered_triples_out[A]!
TEST2
wb_get_buffered_triples__return[A]=ERROR
wb_get_buffered_triples__return[A]=n,
wb_get_buffered_triples__rarray[A]=rarray
name==0
name!=0
arglist[argc].column=SUB,
arglist[argc].cond=WG_COND_EQUAL,
arglist[argc].value=name,
argc++
arglist[argc].column=PROP,
arglist[argc].cond=WG_COND_EQUAL,
arglist[argc].value=input_buffer,
argc++
argc==3
argc<3
query==ERROR
wg_start_write_in[A]!
wg_start_write_out[A]?
wg_make_query__arglist[A]=arglist,
wg_make_query__argc=argc
wg_make_query_in[A]!
wg_make_query_out[A]?
query=wg_make_query__return[A]
wg_end_write_in[A]!
wg_end_write_out[A]?
query!=ERROR
wg_start_read_in[A]!
wg_start_read_out[A]?
stack=0, line=0, i=0, max_id=0, n=0
wg_fetch__query[A]=query
wg_fetch_in[A]!
wg_fetch_out[A]?
record=wg_fetch__return[A]
record==ERROR
field==OBSOLETE
record!=ERROR
wg_get_field__record[A]=record,
wg_get_field__fieldnr[A]=CONT
wg_get_field_in[A]!
wg_get_field_out[A]?
field=wg_get_field__return[A]
field!=OBSOLETE
wg_get_field__record[A]=record,
wg_get_field__fieldnr[A]=VAL
wg_get_field_in[A]!
wg_get_field_out[A]?
field=wg_get_field__return[A]
!((field&KEY)>0 &&
(field&EATEN)==0)
(field&KEY)>0 &&
(field&EATEN)==0
subs[stack]=field,
stack++
list[line]=record, line++
wg_end_read_in[A]!
wg_end_read_out[A]?
wg_free_query__query[A]=query
wg_free_query_in[A]!
wg_free_query_out[A]?
i>=stack
i<stack
wg_start_write_in[A]!
wg_start_write_out[A]?
arglist[0].column=SUB,
arglist[0].cond=WG_COND_EQUAL,
arglist[0].value=subs[i]
wg_make_query__arglist[A]=arglist,
wg_make_query__argc=1
wg_make_query_in[A]!
field==OBSOLETE
wg_fetch__query[A]=query
wg_fetch_in[A]!
wg_fetch_out[A]?
record=wg_fetch__return[A]
record==ERROR
record!=ERROR
wg_get_field__record[A]=record,
wg_get_field__fieldnr[A]=ID
wg_get_field_in[A]!
wg_get_field_out[A]?
field=wg_get_field__return[A],
max_id=max_id<field?field:max_id
wg_get_field__record[A]=record,
wg_get_field__fieldnr[A]=CONT
wg_get_field_in[A]!
wg_get_field_out[A]?
field=wg_get_field__return[A]
field!=OBSOLETE
wg_get_field__record[A]=record,
wg_get_field__fieldnr[A]=PROP
wg_get_field_in[A]!
wg_get_field_out[A]?
wg_get_field__record[A]=record,
wg_get_field__fieldnr[A]=VAL
wg_get_field_in[A]!
wg_get_field_out[A]?
field=wg_get_field__return[A],
rarray[n].sub=field
wg_fetch__query[A]=query
wg_fetch_in[A]!
wg_fetch_out[A]?
record=wg_fetch__return[A]
record==ERROR
record!=ERROR
wg_get_field__record[A]=record,
wg_get_field__fieldnr[A]=ID
wg_get_field_in[A]!
wg_get_field_out[A]?
field=wg_get_field__return[A],
max_id=max_id<field?field:max_id
wg_get_field__record[A]=record,
wg_get_field__fieldnr[A]=VAL
wg_get_field_in[A]!
wg_get_field_out[A]?
field=wg_get_field__return[A],
rarray[n].prop=field
wg_fetch__query[A]=query
wg_fetch_in[A]!
wg_fetch_out[A]?
record=wg_fetch__return[A]


wb_sequence_for_ids
wb_sequence_for_ids_in[A]?
wb_sequence_for_ids_out[A]!
wg_set_field_out[A]?
wg_set_field_in[A]!
record==ERROR
wg_get_first_record_in[A]!
wg_get_first_record_out[A]?
record=wg_get_first_record__return[A]
record!=ERROR
wg_get_field__record[A]=record,
wg_get_field__fieldnr[A]=PROP
wg_get_field_in[A]!
wg_get_field_out[A]?
field=wg_get_field__return[A]
wg_get_next_record_out[A]?
record=wg_get_next_record__return[A]
wg_get_next_record_in[A]!
field!=sequence_for_ids
wg_get_next_record__record[A]=record
field==sequence_for_ids
wg_create_record_in[A]!
wg_create_record_out[A]?
record=wg_create_record__return[A]
wg_set_field__record[A]=record,
wg_set_field__fieldnr[A]=PROP,
wg_set_field__data[A]=sequence_for_ids
wg_set_field_in[A]!
wg_set_field_out[A]?
wg_set_field__record[A]=record,
wg_set_field__fieldnr[A]=VAL,
wg_set_field__data[A]=1
wg_set_field_in[A]!
wg_set_field_out[A]?
wg_get_field__record[A]=record,
wg_get_field__fieldnr[A]=VAL
wg_get_field_in[A]!
wg_get_field_out[A]?
field=wg_get_field__return[A]
wb_sequence_for_ids__return[A]=field
new_id=wb_sequence_for_ids__return[A]+1,
wg_set_field__record[A]=record,
wg_set_field__fieldnr[A]=VAL,
wg_set_field__data[A]=new_id

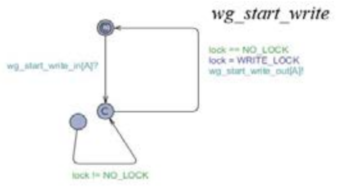

wb_get_selected_triples
wb_get_selected_triples_in[A]?
wb_get_selected_triples_out[A]!
rarray=wb_get_selected_triples__rarray[A],
wheresub=wb_get_selected_triples__wheresub[A],
whereprop=wb_get_selected_triples__whereprop[A],
wherevalue=wb_get_selected_triples__wherevalue[A],
argc=0
wheresub==0
wheresub!=0
arglist[argc].column=SUB,
arglist[argc].cond=WG_COND_EQUAL,
arglist[argc].value=wheresub,
argc++
whereprop==0
whereprop!=0
arglist[argc].column=PROP,
arglist[argc].cond=WG_COND_EQUAL,
arglist[argc].value=whereprop,
argc++
wherevalue==0
wherevalue!=0
arglist[argc].column=VAL,
arglist[argc].cond=WG_COND_EQUAL,
arglist[argc].value=wherevalue,
argc++
argc>0
argc==0
wg_start_write_in[A]!
wg_start_write_out[A]?
wg_make_query__arglist[A]=arglist,
wg_make_query__argc=argc
wg_make_query_in[A]!
wg_make_query_out[A]?
query=wg_make_query__return[A]
wg_end_write_in[A]!
wg_end_write_out[A]?
query==ERROR
wb_get_selected_triples__return[A]=ERROR
query!=ERROR
wg_start_read_in[A]!
wg_start_read_out[A]?
stack=0, line=0, i=0
wg_fetch__query[A]=query
wg_fetch_in[A]!
wg_fetch_out[A]?
record=wg_fetch__return[A]
field==OBSOLETE
record==ERROR
record!=ERROR
wg_get_field__record[A]=record,
wg_get_field__fieldnr[A]=CONT
wg_get_field_in[A]!
wg_get_field_out[A]?
wg_end_read_in[A]!
wg_end_read_out[A]?
wg_free_query__query[A]=query
wg_free_query_in[A]!

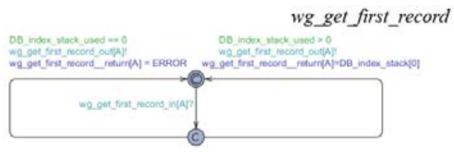

wb_insert_triples
wb_insert_triples_in[A]?
wb_insert_triples_out[A]!
wg_end_write_out[A]?
wg_end_write_in[A]!
array=wb_insert_triples__array[A],
number=wb_insert_triples__number_of_triples[A],
name=wb_insert_triples__name[A],
maplen=0,
i=0
i>=number
i++
i<number
new_sub=getMap(array[i].sub),
new_prop=getMap(array[i].prop),
new_val=getMap(array[i].val)
(array[i].sub&KEY)>0 &&
(new_sub==0)
wb_sequence_for_keys_in[A]!
(array[i].sub&KEY)==0
||
(new_sub!=0)
wb_sequence_for_keys_out[A]?
map[maplen].from=array[i].sub,
map[maplen].to=wb_sequence_for_keys__return[A],
maplen++
(array[i].prop&KEY)>0 &&
(new_prop==0)
wb_sequence_for_keys_in[A]!
(array[i].prop&KEY)==0
||
(new_prop!=0)
wb_sequence_for_keys_out[A]?
map[maplen].from=array[i].prop,
map[maplen].to=wb_sequence_for_keys__return[A],
maplen++
(array[i].val&KEY)>0 &&
(new_val==0)
wb_sequence_for_keys_in[A]!
(array[i].val&KEY)==0
||
(new_val!=0)
wb_sequence_for_keys_out[A]?
map[maplen].from=array[i].val,
map[maplen].to=wb_sequence_for_keys__return[A],
maplen++
wg_start_write_in[A]!
wg_start_write_out[A]?
i=0
i>=number
i++
i<number
wg_create_record_in[A]!

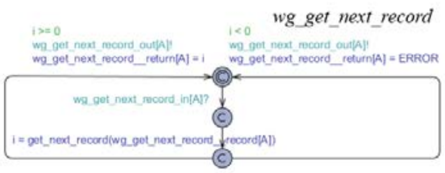


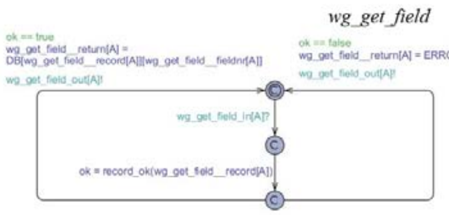
wb_delete_selected_triples
wb_delete_selected_triples_in[A]?
wb_delete_selected_triples_out[A]!
cascade_down=wb_delete_selected_triples__cascade_down[A], wheresub=wb_delete_selected_triples__wheresub[A], whereprop=wb_delete_selected_triples__whereprop[A], wherevalue=wb_delete_selected_triples__wherevalue[A], argc=0
wheresub==0
wheresub!=0 arglist[argc].column=SUB, arglist[argc].cond=WG_COND_EQUAL, arglist[argc].value=wheresub, argc++
whereprop==0
whereprop!=0 arglist[argc].column=PROP, arglist[argc].cond=WG_COND_EQUAL, arglist[argc].value=whereprop, argc++
wherevalue==0
wherevalue!=0 arglist[argc].column=VAL, arglist[argc].cond=WG_COND_EQUAL, arglist[argc].value=wherevalue, argc++
argc>0
argc==0 or cascade_down==0
wg_start_write_in[A]!
wg_start_write_out[A]?
wg_make_query__arglist[A]=arglist, wg_make_query__argc=argc
wg_make_query_in[A]!
wg_make_query_out[A]? query=wg_make_query__return[A]
wg_end_write_in[A]!
wg_end_write_out[A]?
query==ERROR
wb_delete_selected_triples__return[A]=ERROR
query!=ERROR
wg_start_read_in[A]!
wg_start_read_out[A]?
stack=0, line=0, i=0
wg_fetch__query[A]=query
wg_fetch_in[A]!
wg_fetch_out[A]? record=wg_fetch__return[A]
field==OBSOLETE
record==ERROR
cascade_down--
record!=ERROR
wg_end_read_in[A]!
wg_end_read_out[A]?
wg_get_field__record[A]=record, wg_get_field__fieldnr[A]=CONT
wg_free_query__query[A]=query
wg_get_field_in[A]!
wg_free_query_in[A]!
wg_get_field_out[A]?

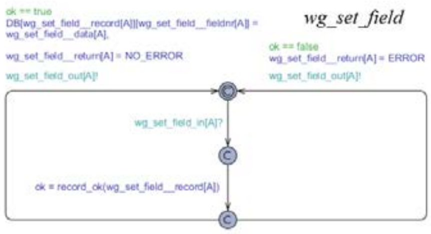
Database templates

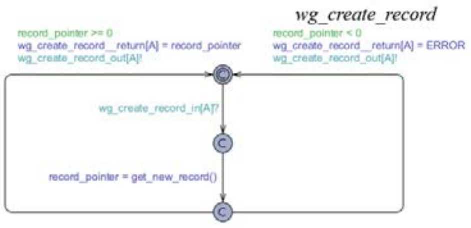


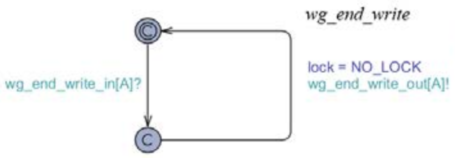

wg_get_first_record
DB_index_stack_used == 0
wg_get_first_record_out[A]!
wg_get_first_record__return[A] = ERROR
DB_index_stack_used > 0
wg_get_first_record_out[A]!
wg_get_first_record__return[A]=DB_index_stack[0]
wg_get_first_record_in[A]?

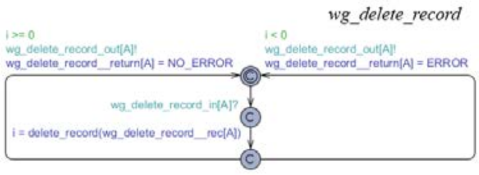

wg_get_next_record
i >= 0
wg_get_next_record_out[A]!
wg_get_next_record__return[A] = i
i < 0
wg_get_next_record_out[A]!
wg_get_next_record__return[A] = ERROR
wg_get_next_record_in[A]?
i = get_next_record(wg_get_next_record__record[A])

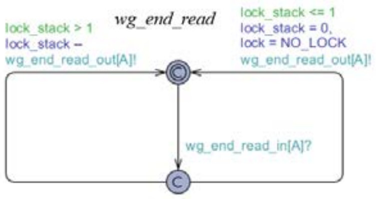

wg_get_field
ok == true
wg_get_field__return[A] =
DB[wg_get_field__record[A]][wg_get_field__fieldnr[A]]
wg_get_field_out[A]!
ok == false
wg_get_field__return[A] = ERROR
wg_get_field_out[A]!
wg_get_field_in[A]?
ok = record_ok(wg_get_field__record[A])

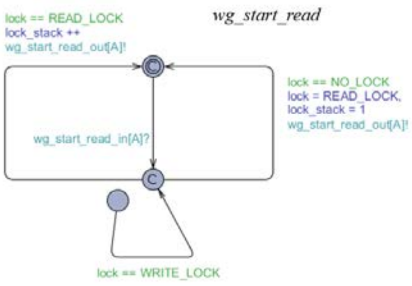

wg_set_field
ok == true
DB[wg_set_field__record[A]][wg_set_field__fieldnr[A]] =
wg_set_field__data[A],
wg_set_field__return[A] = NO_ERROR
wg_set_field_out[A]!
ok == false
wg_set_field__return[A] = ERROR
wg_set_field_out[A]!
wg_set_field_in[A]?
ok = record_ok(wg_set_field__record[A])

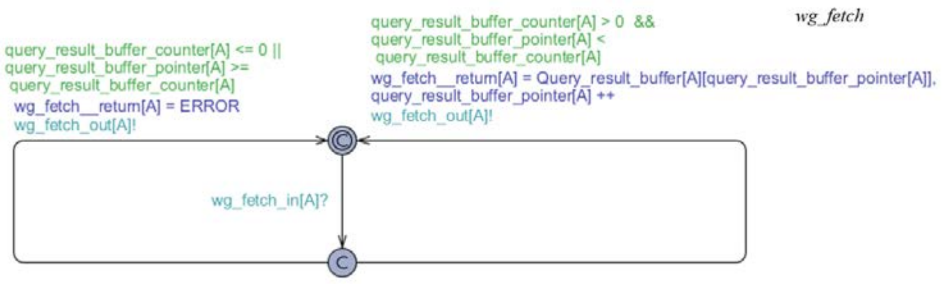


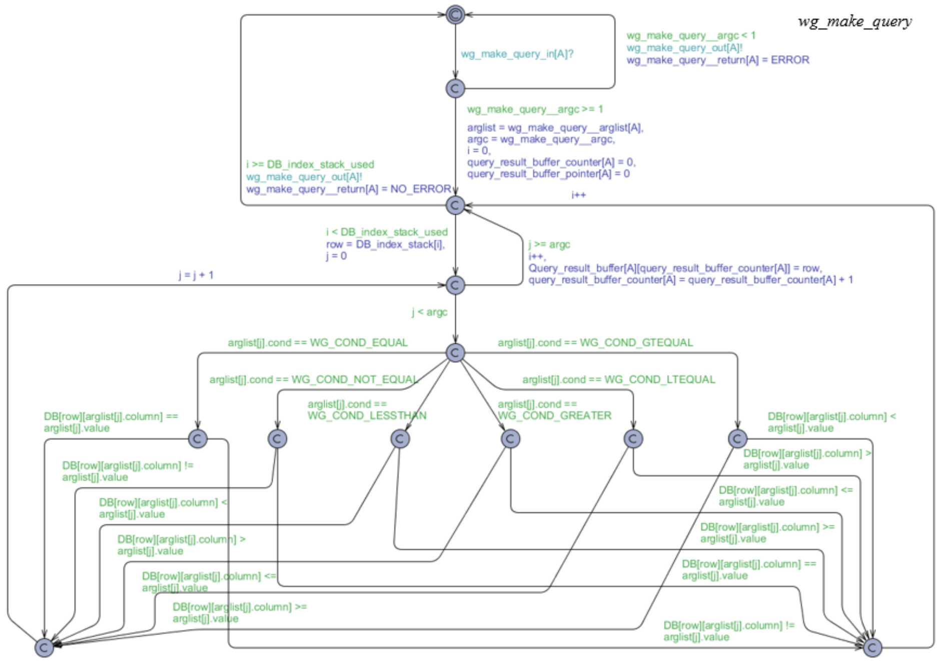



wg_delete_record
i >= 0
wg_delete_record_out[A]!
wg_delete_record__return[A] = NO_ERROR
i < 0
wg_delete_record_out[A]!
wg_delete_record__return[A] = ERROR
wg_delete_record_in[A]?
i = delete_record(wg_delete_record__rec[A])



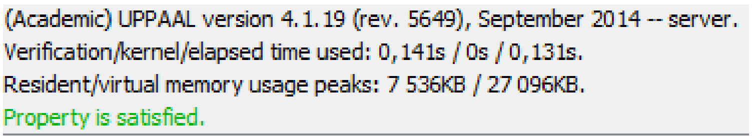



wg_fetch
query_result_buffer_counter[A] <= 0 ||
query_result_buffer_pointer[A] >=
query_result_buffer_counter[A]
wg_fetch__return[A] = ERROR
wg_fetch_out[A]!
query_result_buffer_counter[A] > 0 &&
query_result_buffer_pointer[A] <
query_result_buffer_counter[A]
wg_fetch__return[A] = Query_result_buffer[A][query_result_buffer_pointer[A]],
query_result_buffer_pointer[A] ++
wg_fetch_out[A]!
wg_fetch_in[A]?


wg_make_query
wg_make_query_in[A]?
wg_make_query__argc < 1
wg_make_query_out[A]!
wg_make_query__return[A] = ERROR
wg_make_query__argc >= 1
arglist = wg_make_query__arglist[A],
argc = wg_make_query__argc,
i = 0,
query_result_buffer_counter[A] = 0,
query_result_buffer_pointer[A] = 0
i >= DB_index_stack_used
wg_make_query_out[A]!
wg_make_query__return[A] = NO_ERROR
i++
i < DB_index_stack_used
row = DB_index_stack[i],
j = 0
j >= argc
i++,
Query_result_buffer[A][query_result_buffer_counter[A]] = row,
query_result_buffer_counter[A] = query_result_buffer_counter[A] + 1
j = j + 1
j < argc
arglist[j].cond == WG_COND_EQUAL
arglist[j].cond == WG_COND_GTEQUAL
arglist[j].cond == WG_COND_NOT_EQUAL
arglist[j].cond == WG_COND_LTEQUAL
arglist[j].cond == WG_COND_LESSTHAN
arglist[j].cond == WG_COND_GREATER
DB[row][arglist[j].column] == arglist[j].value
DB[row][arglist[j].column] < arglist[j].value
DB[row][arglist[j].column] != arglist[j].value
DB[row][arglist[j].column] > arglist[j].value
DB[row][arglist[j].column] < arglist[j].value
DB[row][arglist[j].column] <= arglist[j].value
DB[row][arglist[j].column] > arglist[j].value
DB[row][arglist[j].column] >= arglist[j].value
DB[row][arglist[j].column] <= arglist[j].value
DB[row][arglist[j].column] == arglist[j].value
DB[row][arglist[j].column] >= arglist[j].value
DB[row][arglist[j].column] != arglist[j].value

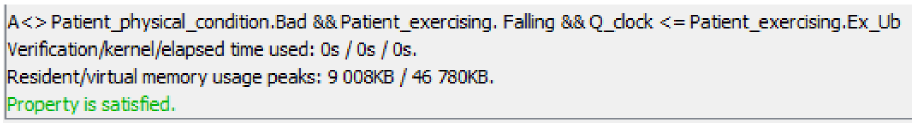
Appendix F

F.1 BM1 Reachibility of base model terminal states



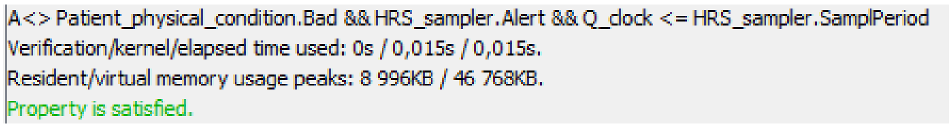
```
(Academic) UPPAAL version 4.1.19 (rev. 5649), September 2014 -- server.
Verification/kernel/elapsed time used: 0,141s / 0s / 0,131s.
Resident/virtual memory usage peaks: 7 536KB / 27 096KB.
Property is satisfied.
```

F.2 BM2.1 Time bounded reachability of Patient model location .Falling



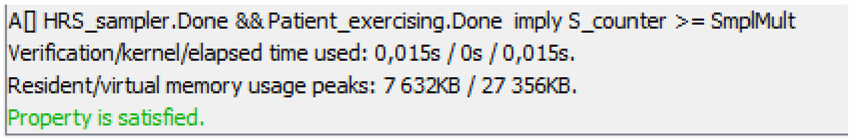
```
A<> Patient_physical_condition.Bad && Patient_exercising. Falling && Q_clock <= Patient_exercising.Ex_Ub
Verification/kernel/elapsed time used: 0s / 0s / 0s.
Resident/virtual memory usage peaks: 9 008KB / 46 780KB.
Property is satisfied.
```

F.3 BM2.2 Time bounded reachability of HRS location Alert



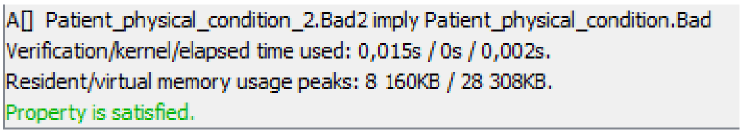
```
A<> Patient_physical_condition.Bad && HRS_sampler.Alert && Q_clock <= HRS_sampler.SamplPeriod
Verification/kernel/elapsed time used: 0s / 0,015s / 0,015s.
Resident/virtual memory usage peaks: 8 996KB / 46 768KB.
Property is satisfied.
```

F.4 BM3 Correct termination of Patient's exercising



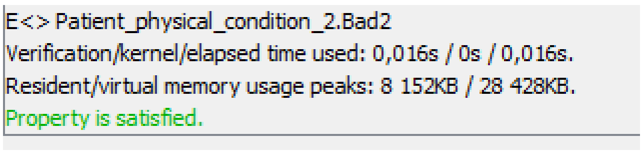
```
A[] HRS_sampler.Done && Patient_exercising.Done  imply S_counter >= SmplMult
Verification/kernel/elapsed time used: 0,015s / 0s / 0,015s.
Resident/virtual memory usage peaks: 7 632KB / 27 356KB.
Property is satisfied.
```

F.5 Non-blocking of template Patient_physical condition



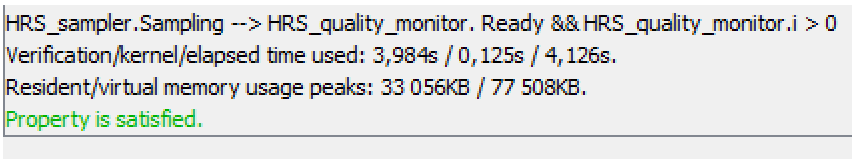
```
A[]  Patient_physical_condition_2.Bad2 imply Patient_physical_condition.Bad
Verification/kernel/elapsed time used: 0,015s / 0s / 0,002s.
Resident/virtual memory usage peaks: 8 160KB / 28 308KB.
Property is satisfied.
```

F.6 Reachability of location Patient_physical_condition_1.Bad2



```
E<> Patient_physical_condition_2.Bad2
Verification/kernel/elapsed time used: 0,016s / 0s / 0,016s.
Resident/virtual memory usage peaks: 8 152KB / 28 428KB.
Property is satisfied.
```

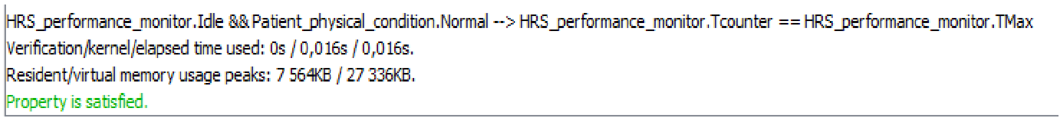
F.7 Non-blocking of advice template HRS_quality_monitor



```
HRS_sampler.Sampling --> HRS_quality_monitor. Ready && HRS_quality_monitor.i > 0
Verification/kernel/elapsed time used: 3,984s / 0,125s / 4,126s.
Resident/virtual memory usage peaks: 33 056KB / 77 508KB.
Property is satisfied.
```

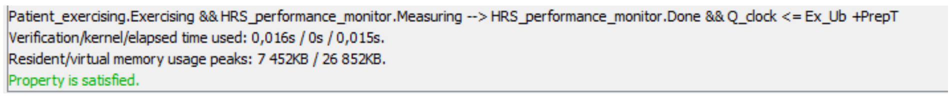
F.8 Non-blocking of the template HRS_performance_monitor

.



```
HRS_performance_monitor.Idle && Patient_physical_condition.Normal --> HRS_performance_monitor.Tcounter == HRS_performance_monitor.TMax
Verification/kernel/elapsed time used: 0s / 0,016s / 0,016s.
Resident/virtual memory usage peaks: 7 564KB / 27 336KB.
Property is satisfied.
```

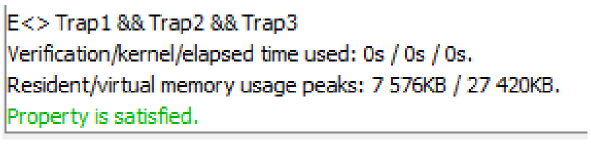
F.9 Non-divergence of HRS_performance_monitor



```
Patient_exercising.Exercising && HRS_performance_monitor.Measuring --> HRS_performance_monitor.Done && Q_clock <= Ex_Ub +PrepT
Verification/kernel/elapsed time used: 0,016s / 0s / 0,015s.
Resident/virtual memory usage peaks: 7 452KB / 26 852KB.
Property is satisfied.
```

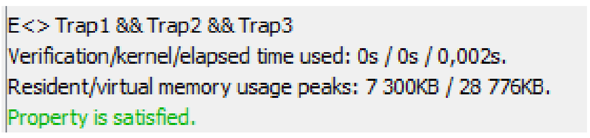
F.10 Test generation statistics of a) Aspect model b) full augmented model

a)



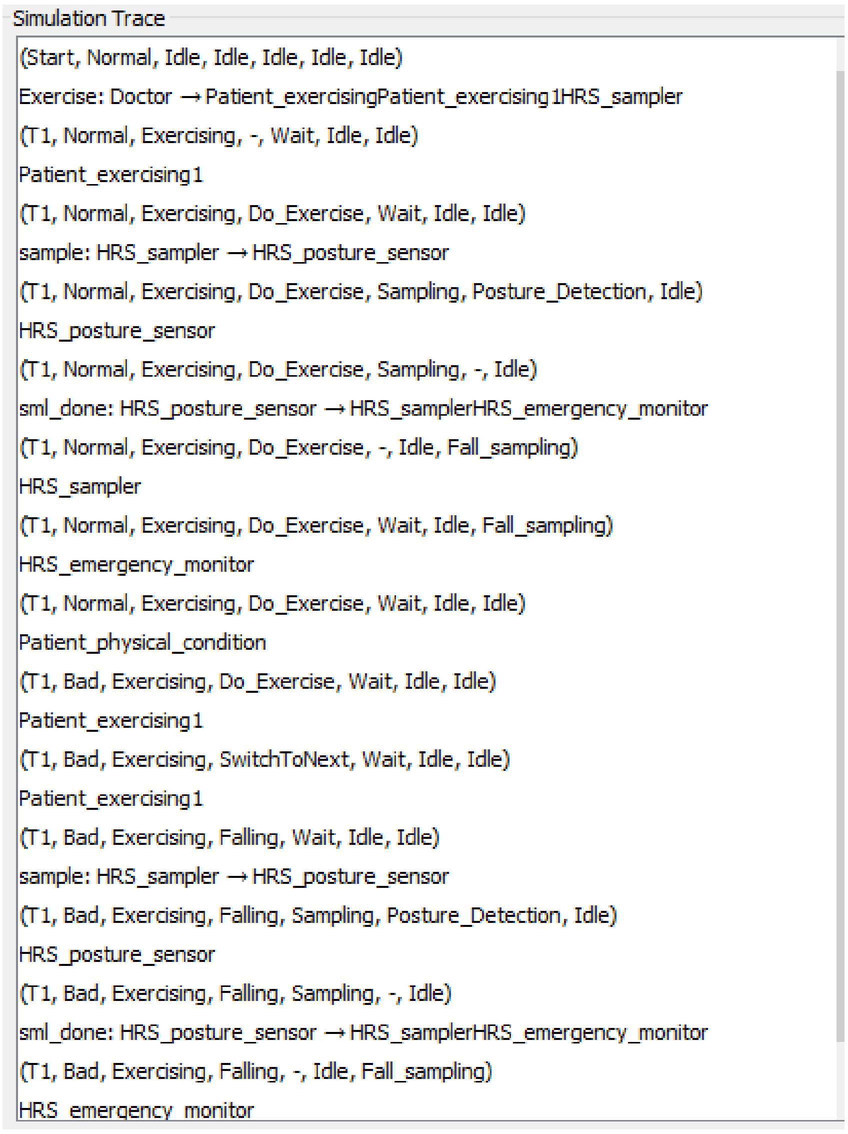
```
E<> Trap1 && Trap2 && Trap3
Verification/kernel/elapsed time used: 0s / 0s / 0s.
Resident/virtual memory usage peaks: 7 576KB / 27 420KB.
Property is satisfied.
```

b)



```
E<> Trap1 && Trap2 && Trap3
Verification/kernel/elapsed time used: 0s / 0s / 0,002s.
Resident/virtual memory usage peaks: 7 300KB / 28 776KB.
Property is satisfied.
```

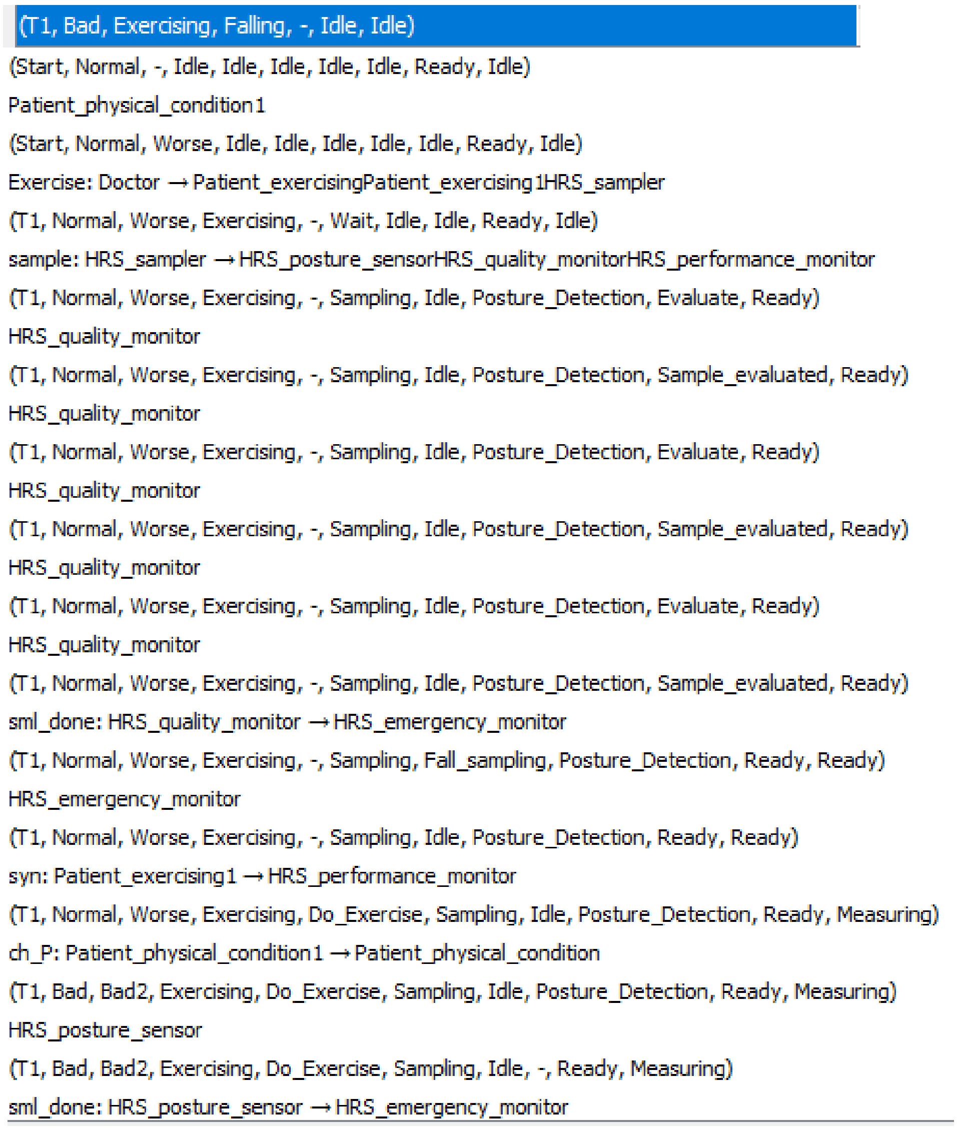
F.11 Simulation trace of coverage criteria Trap1 && Trap2 && Trap3



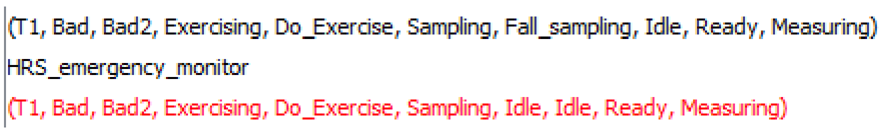
Simulation Trace

```
(Start, Normal, Idle, Idle, Idle, Idle, Idle)
Exercise: Doctor → Patient_exercisingPatient_exercising1HRS_sampler
(T1, Normal, Exercising, -, Wait, Idle, Idle)
Patient_exercising1
(T1, Normal, Exercising, Do_Exercise, Wait, Idle, Idle)
sample: HRS_sampler → HRS_posture_sensor
(T1, Normal, Exercising, Do_Exercise, Sampling, Posture_Detection, Idle)
HRS_posture_sensor
(T1, Normal, Exercising, Do_Exercise, Sampling, -, Idle)
sml_done: HRS_posture_sensor → HRS_samplerHRS_emergency_monitor
(T1, Normal, Exercising, Do_Exercise, -, Idle, Fall_sampling)
HRS_sampler
(T1, Normal, Exercising, Do_Exercise, Wait, Idle, Fall_sampling)
HRS_emergency_monitor
(T1, Normal, Exercising, Do_Exercise, Wait, Idle, Idle)
Patient_physical_condition
(T1, Bad, Exercising, Do_Exercise, Wait, Idle, Idle)
Patient_exercising1
(T1, Bad, Exercising, SwitchToNext, Wait, Idle, Idle)
Patient_exercising1
(T1, Bad, Exercising, Falling, Wait, Idle, Idle)
sample: HRS_sampler → HRS_posture_sensor
(T1, Bad, Exercising, Falling, Sampling, Posture_Detection, Idle)
HRS_posture_sensor
(T1, Bad, Exercising, Falling, Sampling, -, Idle)
sml_done: HRS_posture_sensor → HRS_samplerHRS_emergency_monitor
(T1, Bad, Exercising, Falling, -, Idle, Fall_sampling)
HRS_emergency_monitor
```

F.12 Simulation trace of coverage criteria Trap1 && Trap2 && Trap3 of non-AO model



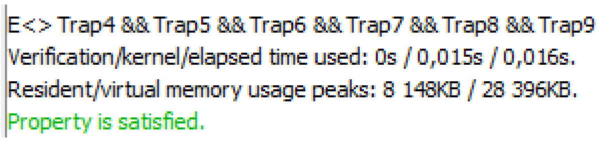
```
(T1, Bad, Exercising, Falling, -, Idle, Idle)
(Start, Normal, -, Idle, Idle, Idle, Idle, Idle, Ready, Idle)
Patient_physical_condition1
(Start, Normal, Worse, Idle, Idle, Idle, Idle, Idle, Ready, Idle)
Exercise: Doctor → Patient_exercisingPatient_exercising1HRS_sampler
(T1, Normal, Worse, Exercising, -, Wait, Idle, Idle, Ready, Idle)
sample: HRS_sampler → HRS_posture_sensorHRS_quality_monitorHRS_performance_monitor
(T1, Normal, Worse, Exercising, -, Sampling, Idle, Posture_Detection, Evaluate, Ready)
HRS_quality_monitor
(T1, Normal, Worse, Exercising, -, Sampling, Idle, Posture_Detection, Sample_evaluated, Ready)
HRS_quality_monitor
(T1, Normal, Worse, Exercising, -, Sampling, Idle, Posture_Detection, Evaluate, Ready)
HRS_quality_monitor
(T1, Normal, Worse, Exercising, -, Sampling, Idle, Posture_Detection, Sample_evaluated, Ready)
HRS_quality_monitor
(T1, Normal, Worse, Exercising, -, Sampling, Idle, Posture_Detection, Evaluate, Ready)
HRS_quality_monitor
(T1, Normal, Worse, Exercising, -, Sampling, Idle, Posture_Detection, Sample_evaluated, Ready)
sml_done: HRS_quality_monitor → HRS_emergency_monitor
(T1, Normal, Worse, Exercising, -, Sampling, Fall_sampling, Posture_Detection, Ready, Ready)
HRS_emergency_monitor
(T1, Normal, Worse, Exercising, -, Sampling, Idle, Posture_Detection, Ready, Ready)
syn: Patient_exercising1 → HRS_performance_monitor
(T1, Normal, Worse, Exercising, Do_Exercise, Sampling, Idle, Posture_Detection, Ready, Measuring)
ch_P: Patient_physical_condition1 → Patient_physical_condition
(T1, Bad, Bad2, Exercising, Do_Exercise, Sampling, Idle, Posture_Detection, Ready, Measuring)
HRS_posture_sensor
(T1, Bad, Bad2, Exercising, Do_Exercise, Sampling, Idle, -, Ready, Measuring)
sml_done: HRS_posture_sensor → HRS_emergency_monitor
```



```
(T1, Bad, Bad2, Exercising, Do_Exercise, Sampling, Fall_sampling, Idle, Ready, Measuring)
HRS_emergency_monitor
(T1, Bad, Bad2, Exercising, Do_Exercise, Sampling, Idle, Idle, Ready, Measuring)
```

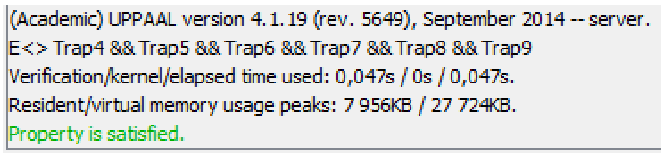
F.13 Trace generation statistics of a) aspect model b) full model

a)



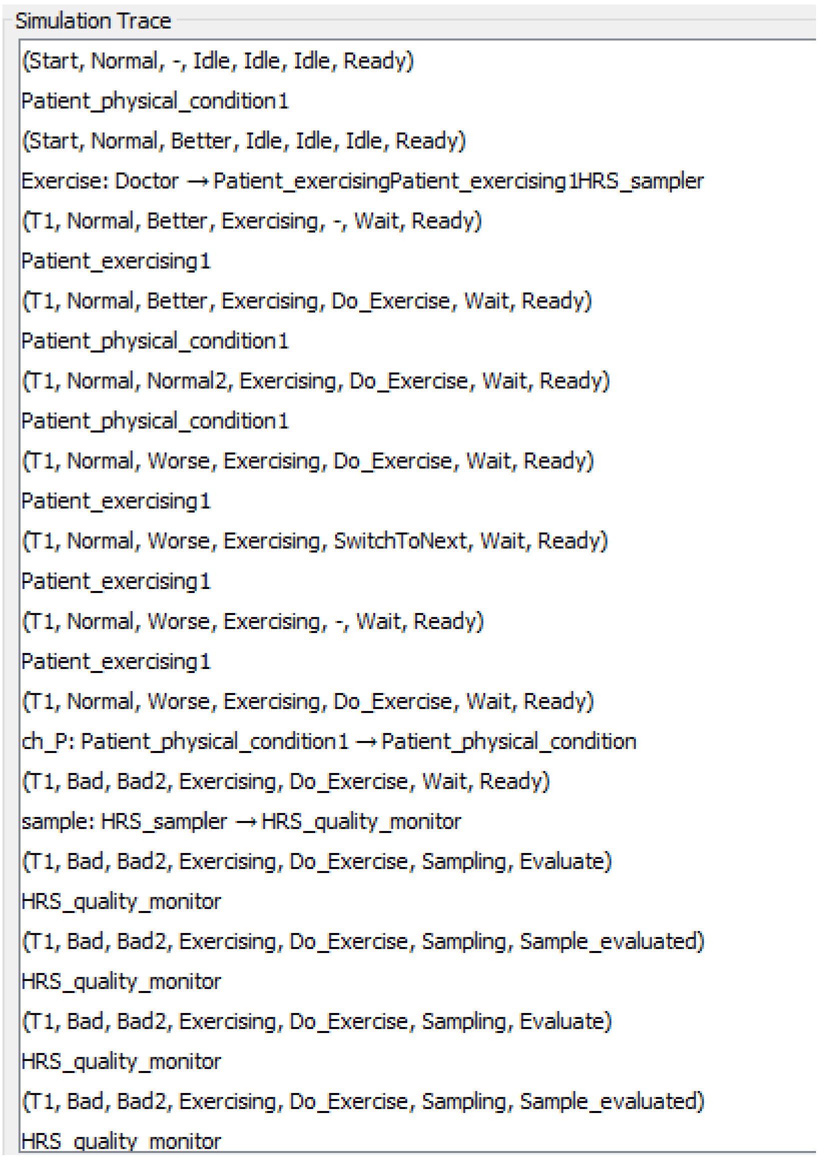
```
E<> Trap4 && Trap5 && Trap6 && Trap7 && Trap8 && Trap9
Verification/kernel/elapsed time used: 0s / 0,015s / 0,016s.
Resident/virtual memory usage peaks: 8 148KB / 28 396KB.
Property is satisfied.
```

b)



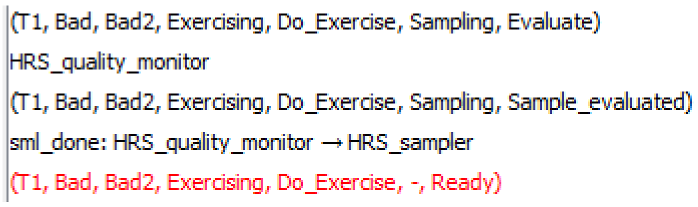
```
(Academic) UPPAAL version 4.1.19 (rev. 5649), September 2014 -- server.
E<> Trap4 && Trap5 && Trap6 && Trap7 && Trap8 && Trap9
Verification/kernel/elapsed time used: 0,047s / 0s / 0,047s.
Resident/virtual memory usage peaks: 7 956KB / 27 724KB.
Property is satisfied.
```

F.14 AO simulation trace of coverage criteria Trap4 && …&& Trap9



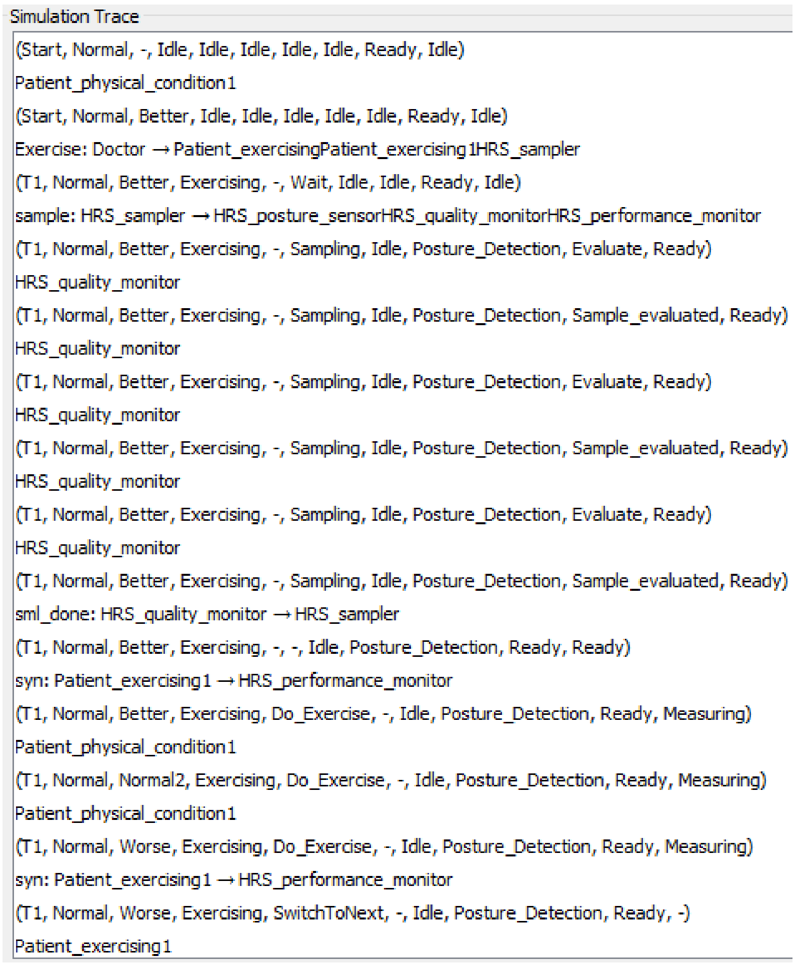
Simulation Trace

```
(Start, Normal, -, Idle, Idle, Idle, Ready)
Patient_physical_condition1
(Start, Normal, Better, Idle, Idle, Idle, Ready)
Exercise: Doctor → Patient_exercisingPatient_exercising1HRS_sampler
(T1, Normal, Better, Exercising, -, Wait, Ready)
Patient_exercising1
(T1, Normal, Better, Exercising, Do_Exercise, Wait, Ready)
Patient_physical_condition1
(T1, Normal, Normal2, Exercising, Do_Exercise, Wait, Ready)
Patient_physical_condition1
(T1, Normal, Worse, Exercising, Do_Exercise, Wait, Ready)
Patient_exercising1
(T1, Normal, Worse, Exercising, SwitchToNext, Wait, Ready)
Patient_exercising1
(T1, Normal, Worse, Exercising, -, Wait, Ready)
Patient_exercising1
(T1, Normal, Worse, Exercising, Do_Exercise, Wait, Ready)
ch_P: Patient_physical_condition1 → Patient_physical_condition
(T1, Bad, Bad2, Exercising, Do_Exercise, Wait, Ready)
sample: HRS_sampler → HRS_quality_monitor
(T1, Bad, Bad2, Exercising, Do_Exercise, Sampling, Evaluate)
HRS_quality_monitor
(T1, Bad, Bad2, Exercising, Do_Exercise, Sampling, Sample_evaluated)
HRS_quality_monitor
(T1, Bad, Bad2, Exercising, Do_Exercise, Sampling, Evaluate)
HRS_quality_monitor
(T1, Bad, Bad2, Exercising, Do_Exercise, Sampling, Sample_evaluated)
HRS_quality_monitor
```



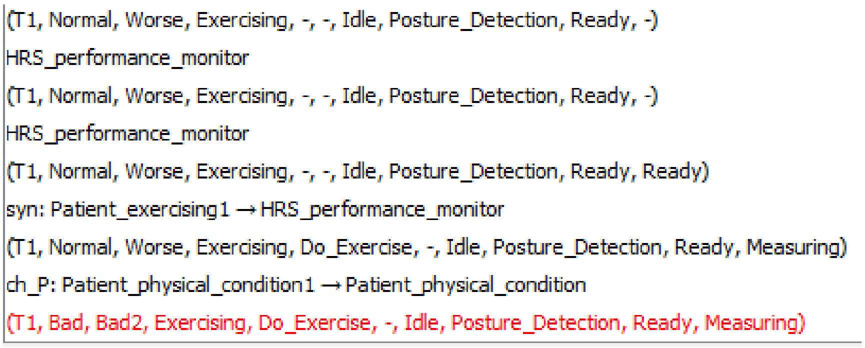
```
(T1, Bad, Bad2, Exercising, Do_Exercise, Sampling, Evaluate)
HRS_quality_monitor
(T1, Bad, Bad2, Exercising, Do_Exercise, Sampling, Sample_evaluated)
sml_done: HRS_quality_monitor → HRS_sampler
(T1, Bad, Bad2, Exercising, Do_Exercise, -, Ready)
```

F.15 Non-AO simulation trace of coverage criteria Trap4 && … && Trap9



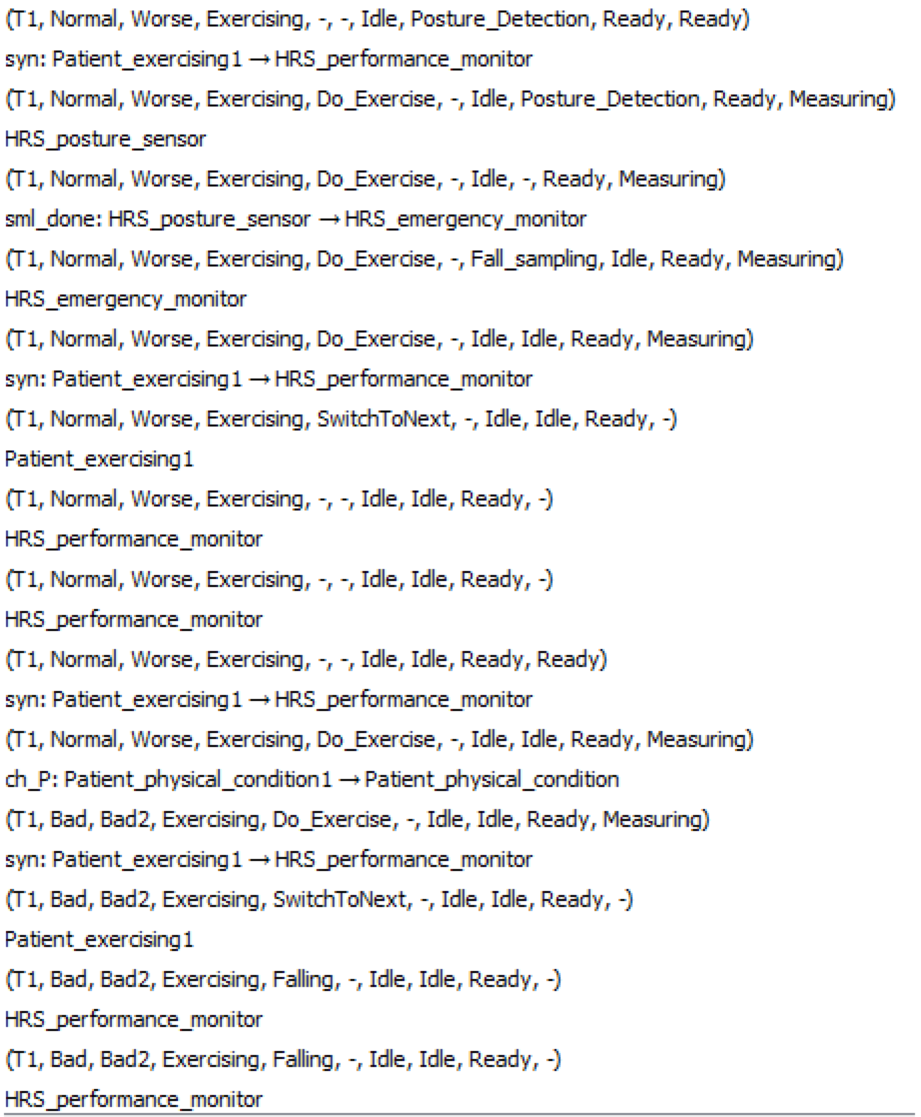
Simulation Trace

```
(Start, Normal, -, Idle, Idle, Idle, Idle, Idle, Ready, Idle)
Patient_physical_condition1
(Start, Normal, Better, Idle, Idle, Idle, Idle, Idle, Ready, Idle)
Exercise: Doctor → Patient_exercisingPatient_exercising1HRS_sampler
(T1, Normal, Better, Exercising, -, Wait, Idle, Idle, Ready, Idle)
sample: HRS_sampler → HRS_posture_sensorHRS_quality_monitorHRS_performance_monitor
(T1, Normal, Better, Exercising, -, Sampling, Idle, Posture_Detection, Evaluate, Ready)
HRS_quality_monitor
(T1, Normal, Better, Exercising, -, Sampling, Idle, Posture_Detection, Sample_evaluated, Ready)
HRS_quality_monitor
(T1, Normal, Better, Exercising, -, Sampling, Idle, Posture_Detection, Evaluate, Ready)
HRS_quality_monitor
(T1, Normal, Better, Exercising, -, Sampling, Idle, Posture_Detection, Sample_evaluated, Ready)
HRS_quality_monitor
(T1, Normal, Better, Exercising, -, Sampling, Idle, Posture_Detection, Evaluate, Ready)
HRS_quality_monitor
(T1, Normal, Better, Exercising, -, Sampling, Idle, Posture_Detection, Sample_evaluated, Ready)
sml_done: HRS_quality_monitor → HRS_sampler
(T1, Normal, Better, Exercising, -, -, Idle, Posture_Detection, Ready, Ready)
syn: Patient_exercising1 → HRS_performance_monitor
(T1, Normal, Better, Exercising, Do_Exercise, -, Idle, Posture_Detection, Ready, Measuring)
Patient_physical_condition1
(T1, Normal, Normal2, Exercising, Do_Exercise, -, Idle, Posture_Detection, Ready, Measuring)
Patient_physical_condition1
(T1, Normal, Worse, Exercising, Do_Exercise, -, Idle, Posture_Detection, Ready, Measuring)
syn: Patient_exercising1 → HRS_performance_monitor
(T1, Normal, Worse, Exercising, SwitchToNext, -, Idle, Posture_Detection, Ready, -)
Patient_exercising1
```

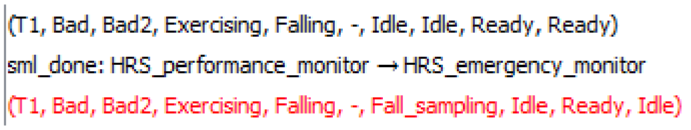


```
(T1, Normal, Worse, Exercising, -, -, Idle, Posture_Detection, Ready, -)
HRS_performance_monitor
(T1, Normal, Worse, Exercising, -, -, Idle, Posture_Detection, Ready, -)
HRS_performance_monitor
(T1, Normal, Worse, Exercising, -, -, Idle, Posture_Detection, Ready, Ready)
syn: Patient_exercising1 → HRS_performance_monitor
(T1, Normal, Worse, Exercising, Do_Exercise, -, Idle, Posture_Detection, Ready, Measuring)
ch_P: Patient_physical_condition1 → Patient_physical_condition
(T1, Bad, Bad2, Exercising, Do_Exercise, -, Idle, Posture_Detection, Ready, Measuring)
```

F.15 Non-AO simulation trace of coverage criteria Trap10 && …&& Trap13 (continuation)



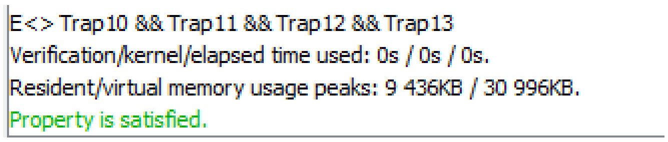
```
(T1, Normal, Worse, Exercising, -, -, Idle, Posture_Detection, Ready, Ready)
syn: Patient_exercising1 → HRS_performance_monitor
(T1, Normal, Worse, Exercising, Do_Exercise, -, Idle, Posture_Detection, Ready, Measuring)
HRS_posture_sensor
(T1, Normal, Worse, Exercising, Do_Exercise, -, Idle, -, Ready, Measuring)
sml_done: HRS_posture_sensor → HRS_emergency_monitor
(T1, Normal, Worse, Exercising, Do_Exercise, -, Fall_sampling, Idle, Ready, Measuring)
HRS_emergency_monitor
(T1, Normal, Worse, Exercising, Do_Exercise, -, Idle, Idle, Ready, Measuring)
syn: Patient_exercising1 → HRS_performance_monitor
(T1, Normal, Worse, Exercising, SwitchToNext, -, Idle, Idle, Ready, -)
Patient_exercising1
(T1, Normal, Worse, Exercising, -, -, Idle, Idle, Ready, -)
HRS_performance_monitor
(T1, Normal, Worse, Exercising, -, -, Idle, Idle, Ready, -)
HRS_performance_monitor
(T1, Normal, Worse, Exercising, -, -, Idle, Idle, Ready, Ready)
syn: Patient_exercising1 → HRS_performance_monitor
(T1, Normal, Worse, Exercising, Do_Exercise, -, Idle, Idle, Ready, Measuring)
ch_P: Patient_physical_condition1 → Patient_physical_condition
(T1, Bad, Bad2, Exercising, Do_Exercise, -, Idle, Idle, Ready, Measuring)
syn: Patient_exercising1 → HRS_performance_monitor
(T1, Bad, Bad2, Exercising, SwitchToNext, -, Idle, Idle, Ready, -)
Patient_exercising1
(T1, Bad, Bad2, Exercising, Falling, -, Idle, Idle, Ready, -)
HRS_performance_monitor
(T1, Bad, Bad2, Exercising, Falling, -, Idle, Idle, Ready, -)
HRS_performance_monitor
```



```
(T1, Bad, Bad2, Exercising, Falling, -, Idle, Idle, Ready, Ready)
sml_done: HRS_performance_monitor → HRS_emergency_monitor
(T1, Bad, Bad2, Exercising, Falling, -, Fall_sampling, Idle, Ready, Idle)
```

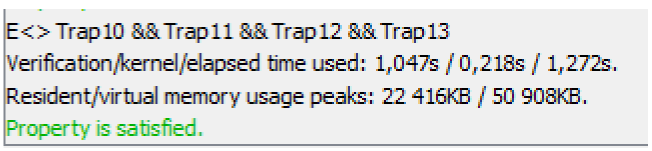
F.16 Trace generation statistics of a) aspect model b) full model

a)



```
E<> Trap10 && Trap11 && Trap12 && Trap13
Verification/kernel/elapsed time used: 0s / 0s / 0s.
Resident/virtual memory usage peaks: 9 436KB / 30 996KB.
Property is satisfied.
```

b)



```
E<> Trap10 && Trap11 && Trap12 && Trap13
Verification/kernel/elapsed time used: 1,047s / 0,218s / 1,272s.
Resident/virtual memory usage peaks: 22 416KB / 50 908KB.
Property is satisfied.
```

CURRICULUM VITAE

Personal data

Name:	Külli Sarna
Date of birth:	22.06.1971
Place of birth:	Tallinn, Estonia
Citizenship:	Estonian

Contact data

E-mail:	kylli.sarna@gmail.com

Education

2009 – 2014	Tallinn University of Technology (TalTech), PhD studies
2005 – 2007	TUT, MSc in Engineering, Informatics, Computer Science
1999 – 2005	TUT, Diploma Engineer, Networking and Intelligent systems
1978 – 1989	Tallinn Nõmme Upper Secondary School

Language competence

Estonian	native language
English	fluent
German	average
Russian	average

Professional employment

2017 – 2018	Register OÜ, IT analyst
2014 – 2016	Percival Software OÜ, IT analyst – quality assurance manager
2013 – 2014	Marie Curie Research Fellowship in Technical University Graz
2008 – 2014	Eliko Competence Centre, domain manager of Smart Environment Applications and researcher
1996 – 2008	AS EMT, various positions: test engineer, programmer, and information systems analyst.

ELULOOKIRJELDUS

Isikuandmed

Nimi:	Külli Sarna
Sünniaeg:	22.06.1971
Sünnikoht:	Tallinn, Eesti
Kodakondsus:	eestlane

Kontaktandmed

E-mail:	kylli.sarna@gmail.com

Hariduskäik

2009 – 2014	Tallinna Tehnikaülikool (TTÜ), doktoriõpe
2005 – 2007	TTÜ, tehnikateaduse magister, Informaatika
1999 – 2005	TTÜ, diplomiinsener, Võrgutarkvara
1978 – 1989	Tallinna Nõmme Gümnaasium

Keelteoskus

Eesti keel	emakeel
Inglise keel	kõrgtase
Saksa keel	keskmine tase
Vene keel	keskmine tase

Teenistuskäik

2017 – 2018	Register OÜ, IT analüütik
2014 – 2016	Percival Software OÜ, IT analüütik – kvaliteedijuht
2013 – 2014	Marie Curie Fellow stipendiumiga Grazi Tehnikaülikoolis külalisdoktorant
2008 – 2014	Eliko OÜ, targa keskkonna rakendusvaldkonna juht ja teadur
1996 – 2008	AS EMT, erinevad ametikohad: testiinsener, programmeerija, infosüsteemide analüütik.